Flashbots report: System requirements, existing and new solutions, and their efficiency.

Orestis Alpos[1], Bernardo David[1,2], Nikolas Kamarinakis[1], and Dionysis Zindros[1,3]

[1] Common Prefix
[2] IT University of Copenhagen (ITU)
[3] Stanford University

Last update: June 20, 2024

1 Introduction

Censorship resistance [17, 28, 29] with short-term inclusion guarantees is an important feature of decentralized systems, missing from many state-of-the-art and even deployed consensus protocols. In leader-based protocols the leader arbitrarily selects the transactions to be included in the new block, and so does a block builder in permissionless protocols, such as Bitcoin and Ethereum.

In a different line of work, the redundancy of consensus for implementing simple payment systems has been recently recognized [16]. Since then, consensusless protocols have been described in theory [10, 22] and deployed in the real world [3], resulting in more efficient blockchain and payment systems.

In this report we review existing consensus and consensusless protocols (Section 2) with regard to their censorship-resistance, efficiency, and other properties relevant for Flashbots. Moreover, we present new constructions with these properties in mind, building on existing leader-based protocols (Section 3) and exploring an efficient consensusless approach (Section 4). In section 5 we approach censorship-resistance based on rational and incentive-based arguments, and in Section 7 we summarize our findings and make our recommendations.

1.1 Model

Model. We distinguish two types of parties, n *replicas* (also called *validators* in other works) and an unlimited number of *clients*. Replicas run the protocol and clients interact with it using its interface. We call a party – replica or client – *honest*, if it follows the protocol, and *malicious* otherwise. Notice that malicious parties can arbitrarily deviate from protocol specifications.

Network. We review protocols modelled in the synchronous, partially synchronous, and asynchronous model. In the synchronous model, we assume that every message is delivered within a *known* finite delay of Δ rounds. In the partially synchronous model, we assume that messages are delivered within a known finite delay of Δ rounds only *after* an unknown but finite period of time called the Global Stabilization Time (GST). Messages sent before the GST may be delivered only Δ rounds after the GST. Finally, in the asynchronous model, we assume that messages are delivered within an *unknown* but finite delay. We denote by δ the actual (but unknown) average network delay between two honest replicas. In synchrony it holds that $\delta \leq \Delta$.

Regarding protocol implementation, Δ can be hardcoded in the protocol or dynamically discovered from the behavior of the network. Protocols that proceed in the actual network speed without waiting for Δ-timeouts are called *responsive*.

1.2 Problem statement

We want to use a protocol, which we refer to as 'transaction layer' in this section, that gets as input user transactions and outputs user transactions, possibly but not necessarily totally-ordered. The protocol does not execute transactions. The required are, in a high level, the following.

Short-term censorship resistance: Malicious replicas cannot censor honest clients. Particularly, they cannot do so even for a short term, meaning that, if the system makes progress and creates an output, then transactions submitted by honest users will appear in it.

Finality: We envision applications such as the following: an auction is run by a trusted party, or in a Trusted Execution Environment, and retrieves its bids from the transaction-layer protocol. We require that any other client reading the transaction layer at the same time obtains the same set of transactions, i.e., the input to the auction is finalized and well-defined. This property, together with the assumption on a trusted auctioneer, will allow for fair and verifiable auction outcome.

Light-client friendly: There exists a succinct way to verify the state of the transaction-layer protocol. Succinctness in this context means that the size of the state description and verification time are independent of the total number of transactions.

Economic incentives: The nodes that maintain this transaction layer have rational incentives to do so. These can be incentive based, for example using transaction fees, or punishment based, based on accountability and punishment conditions.

2 Existing protocols

Censorship resistance vs. transaction duplication. BFT protocols face the following trade-off, stemming from the fact that up to f parties can be malicious. In order to avoid censorship, a client has to send its transaction to at least $f + 1$ replicas, which, depending on the design of the protocol, may lead to request duplication. As recognized in the literature [19, 26], the problem is exacerbated in protocols that feature parallel leaders: it is not straightforward how to ensure that the leaders, who propose blocks in parallel, do not include the same transactions in them. In this section we explore how existing protocols handle this trade-off.

2.1 Comparison

In this section we review and compare existing protocols. We present a summary on Table 1.

By *happy path* we refer to executions in synchronous periods and without faults. For execution in the happy path, we report the following: *Proposal latency* measures the time (in number of communication steps) from the proposal of a transaction (by one or more replicas) until it becomes committed by $2f + 1$ replicas. *Proposal period* measures the time between consecutive proposals. *Max tx censorship* reports the maximum time the adversary can delay a *specific* transaction, assuming the client sent it to all replicas, until it appears in a proposal, and while the rest of the system makes progress. For *communication complexity*, under *best* we refer to happy-path executions and under *worst* to all possible executions. Under *benchmark* we report the throughput and latency for WAN deployments under no faults, as reported on the cited papers. We denote by s the transaction size.

Table 1: Summary and comparison of various existing protocols and our proposals. See Section 2.1 for an explanation of the reported metrics. *IL* stands for the inclusion lists constructions, which are applied on top of any leader-based protocol. For IL constructions, metrics that start with + indicate an *incremental* overhead to a leader-based protocol – for explanation of parameters refer to corresponding section. For leader-based protocols, *max tx censorship* equals the proposal period times f, as up to f successive leaders can produce blocks without the censored transaction. For the leader-based add-on solutions it equals 0, as the leader is limited and cannot censor transactions, or it equals the transaction propagation time, which must be completed before a proposal is created. For DAG-based protocols it equals 0, as any transaction sent to sufficiently many replicas will be included when the protocol commits the next round (or wave, in their terminology).

Protocol	**Theoretical efficiency**					**Bench. (WAN, no faults)**			
	Proposal latency	**Proposal period**	**Max tx censorship**	**Com. compl. best**	**Com. compl. worst**	**#rep**	**thr. (tps)**	**lat. (sec)**	**ref.**
Tendermint [6]	3Δ	3Δ	$3\Delta f$	$O(n^2)$	$O(n^2)$	32	520	2.83	[9, Fig.6]
						100	500	1–9	[11]
HotStuff/DiemBFT [30]	7δ	2δ	$2\delta f$	$O(n)$	$O(n^2)$	20	$50K$	1.7	[15, Fig.5]
Jolteon [15]	5δ	2δ	$2\delta f$	$O(n)$	$O(n^2)$	20	$50K$	2	[15, Fig.5]
Ditto [15]	5δ	2δ	$2\delta f$	$O(n)$	$O(n^2)$	20	$50K$	1.5	[15, Fig.5]
HotStuff-2 [18]	5δ	2δ	$2\delta f$	$O(n)$	$O(n^2)$				
MoonShot [13]	5δ	δ	δf	$O(n^2)$	$O(n^2)$				
IL (Sec.3.1)	+0	+0	0	$+O(n^2\|L\|)$					
IL w. DA (Sec.3.2)	$+t_{\text{ret}}$	+0	$t_{\text{disp}} + t_{\text{ret}}$	$+c_{\text{da}}$					
IL w. b/cast (Sec.3.3)	+0	+0	2δ	$+O(n^2 s)$					
IL w. gossip (Sec.3.4)	+0	+0	t_{prop}	$+O(nc_{\text{gos}}s)$					
IL local (Sec.3.5)	+0	+0	0	$+O(n\|L\|)$					
Narwhal/HotStuff [12]	8δ	4δ	0	$O(n^2)$	$O(n^2)$	20	$125K$	1.8	[12, Fig.6]
						50	$135K$	1.8	[12, Fig.6]
Narwhal/Tusk [12]	6δ	4δ	0	$O(n^2)$	$O(n^2)$	20	$160K$	3.2	[12, Fig.6]
						50	$160K$	3.2	[12, Fig.6]
psync-BullShark [23]	4δ	4δ	0	$O(n^2)$	$O(n^2)$	20	$110K$	2.5	[23, Fig.2]
						50	$130K$	2.2	[23, Fig.2]
Mysticeti [2]	3δ	3δ	0	$O(n^2)$	$O(n^2)$	10	$300K$	< 1	[2, Fig.4]
						50	$100K$	< 1	[2, Fig.4]

2.2 Single-leader protocols

Tendermint [6,9] is a partially synchronous protocol that uses two rounds of voting and an all-to-all communication pattern. In the happy path, where no faults occur and the network is synchronous, it has a proposal latency of 3δ. The proposal period is 3δ. The leader is rotated after every epoch.

HotStuff [30] is a partially synchronous protocol that uses three rounds of voting and an all-to-leader communication pattern. In the happy path, and assuming an implementation with threshold signatures, it achieves linear communication. The proposal latency to 7δ, as every voting round contains an all-to-leader and a leader-to-all communication step. A module of the protocol called *pacemaker* is responsible for synchronizing the views of the replicas. It maintains the current round and, in case of asynchrony or failures, it sends timeout messages to all replicas. Hence, the worst-case communication complexity is quadratic, because of the all-to-all timeout messages sent by the pacemaker. The protocol makes no progress while in asynchrony. Using the technique of *pipelining*, a new proposal can be sent in every round, hence the proposal period is 2δ. HotStuff achieves *optimistic responsiveness*, meaning it can make progress at network

speed (i.e., $O(\delta)$) after GST with an honest leader. The three-round version of HotStuff is also known as DiemBFT [27].

Jolteon [15] is two-round version of HotStuff, achieving a proposal latency of 5δ and linear communication in the happy path. This is achieved at the cost of a quadratic view-change procedure (after asynchrony or a malicious leader, each replica has to send a message of size $O(n)$ to the next leader). As the pacemaker is already quadratic, this does not affect the worst-case communication of the protocol. Ditto [15] is another two-round version of HotStuff, that, like Jolteon, also comes with a quadratic view-change procedure, but replaces the pacemaker with an asynchronous fallback protocol. This allows the protocol to make progress in case of asynchrony. The combination of Jolteon over a Narwhal [12] network is known as HotStuff-over-Narwhal and Narwhal/HotStuff. Finally, HotStuff-2 [18] is also a two-round version of HotStuff, achieving a proposal latency of 5δ in the happy path, at the cost of losing optimistic responsiveness.

MoonShot [13] builds further on HotStuff-2, adding the idea of *optimistic proposals*, where a leader can send a new proposal before receiving enough votes for the previous one. This drops the proposal period to δ in the happy path at the cost of a quadratic communication complexity and a more complicated protocol logic. Finally, HotShot[4] is a Proof-of-Stake version of HotStuff.

2.3 Parallel-blocks approach

Several works use the following idea: in each epoch replicas create blocks or 'mini-blocks' in parallel and the protocol outputs a subset of them. The goal is to limit the leader as much as possible, so it has no other option than produce a correct block, or remain silent.

In HoneyBadger [19] replicas collect user transactions in local buffers. In each epoch they first create and broadcast blocks in parallel, and then agree on a subset of at least $n - f$ correctly broadcast blocks, which are all output by the protocol. The authors observe a trade-off between censorship resistance and protocol throughput. Regarding the censorship resilience vs. transaction duplication dilemma, the authors propose that replicas include in their block a small number of transactions from their local view, chosen at random, and that transactions are threshold-encrypted by the clients. Threshold encryption requires a threshold setup, which either has to be performed by a trusted party or necessitates the implementation of a Distributed Key Generation (DKG) protocol. Moreover, it incurs additional computational and communication cost.

DispersedLedger [29] builds on this idea, but instead of broadcast it uses a Data Availability (formally, Verifiable Information Dispersal, VID) protocol, thus allowing replicas to vote for a transaction without locally downloading it. The protocol guarantees that all blocks proposed by honest replicas will be delivered. Transaction duplication is not resolved – in order to make sure it will not be censored, a client has to send a transaction to $f + 1$ replicas, and all of them may include it in their proposed block.

BigDipper [28] is a system that combines a broadcast, a Data Availability (DA), and a leader-based consensus protocol to build a censorship resistant leader-based consensus protocol. The 'mini blocks idea' is integrated into their DA protocol: Replicas collect transactions from clients and batch them into a 'mini block'. The leader receives mini-blocks from replicas, encodes them appropriately, and disperses the resulting block. The DA protocol employs 2-dimensional polynomial commitments and Reed-Solomon codes (similar to state-of-the-art Information Dispersal protocols [1, 20]) and BLS signatures,

[4] `https://github.com/EspressoSystems/HotShot`

and consists of two rounds of leader-to-all communication. It achieves the property that, if a client sends a transaction tx to at least $n - f$ replicas, then tx will be included in the next block produced [28, Table 3]. The protocol does not handle transaction deduplication, hence fast transaction inclusion comes at the cost of storing the same transaction multiple time on the DA layer. The authors show how it can be integrated into HotStuff-2 [18], but no implementation or benchmark is provided.

Mir-BFT [26] features parallel leaders, each running a standard leader-based protocol, such as PBFT. Regarding transaction duplication, the authors propose partitioning the transaction assignment among the replicas (that is, based on the hash of a transaction, there is one leader responsible for it) and periodically rotating this assignment. This, however, does not differ much from a single-leader protocol, as far as censorship is concerned, as, in the worst case, a transaction will be assigned to an honest leader after f such rotations.

Conclusion. A leader in consensus protocols becomes a temporary point of centralization, and this contributes to censorship. The aforementioned works aim to completely remove or limit the power of the leader. On the other hand, employing a leader is a common technique for efficient consensus (at least, efficient on the so-called 'happy path', where the leader is honest and the network is good), employed by some state-of-the-art protocols [6, 13, 18, 30]. All aforementioned works achieve censorship resilience at the cost of duplicating transactions, hence wasting computation, communication, and storage. In Section 3 we present constructions that achieve censorship resistance using the parallel-blocks idea, can be employed with minimal modification on existing consensus protocols, and achieve increasingly better transaction deduplication.

2.4 DAG-based protocols

The so-called 'DAG-based' protocols observe that the separation of data dissemination and ordering logic improves the efficiency of consensus protocols. Assuming that clients send transactions to 'enough' (explained in the next paragraph) replicas, DAG-based protocols achieve short-term censorship resistance by construction, as blocks are created by all replicas in parallel. This comes, unavoidably, with transaction duplication.

In the asynchronous Narwhal/Tusk [12] the blocks of up to f honest but slow replicas can be arbitrarily delayed (even gargabe-collected, hence never delivered). With up to f replicas being malicious, in order to achieve short-term censorship resistance transactions have to be sent to $2f + 1$ replicas. The partially synchronous BullShark [23, 24] protocol guarantees that, after GST, the blocks of all honest replicas will become delivered. Hence, assuming being in a synchronous period, clients can send transactions to $f + 1$ replicas.

Mysticeti [2] achieves very low latency if there are no Byzantine faults, which the authors argue is the most common case in practice. The improvement comes mainly from using an uncertified DAG, where blocks are multicast and not broadcast. This allows blocks to be sent and committed within three network trips, hitting the lower bound for consensus. Multicasting also allows validators to equivocate, by sending two different blocks. If that happens, either only one of them will be committed, or none will be committed for that epoch. Malicious behavior like this and asynchrony lead to a less efficient fallback 'indirect decision rule'. Mysticeti also provides built-in support for fast-path transactions, that is transactions that do not need to be totally ordered (see Section 2.5).

Regarding practical efficiency, as shown in Table 1, Narwhal/Tusk achieves the highest throughput (160K with 50 replicas), but also the highest latency. The partially synchronous version of BullShark [23, 24], maintains comparable throughput (130K with 50 replicas) and a better latency, but still over 2 sec. Narwhal/HotStuff [12] achieves similar throughput (135K with 50 replicas) and the lowest latency, approx. 1.8 seconds. Mysticeti [2] achieves a throughput of around 300K tps in a deployment with 10 replicas, and 100K tps in a deployment with 50 replicas, while maintaining sub-second latency.

A significant advantage of DAG-based, compared to leader-based, protocols is their better resilience to crash faults. This is because they do not employ view-changes. For example, in benchmarks with ten replicas, BullShark achieves a throughput of $70K$ tps when three replicas crash, while its latency becomes approx. 6 seconds [23, Fig. 4]. In the same experiment, Narwhal/HotStuff [12] also achieves a throughput of $70K$ tps with a latency or approx. 10 seconds [12, Fig. 8], while HotStuff achieves a throughput of $10K$ tps with approx. 14 seconds latency [12, Fig. 8].

Conclusion. DAG-based protocols outperform leader-based protocols in terms of throughput, while exhibiting comparable latency, in the best case approx. 2 seconds. In order to achieve sub-second latency, they have been combined in production systems [4] with consensusless protocols [3].

2.5 Consensusless protocols

Recent literature has recognized the redundancy of consensus for implementing asset-transfer systems [16]. Such schemes have been described in theory [10, 22] and deployed in the real world [3].

The insight that is total order is not required in the case that each account is controlled by one client. Instead, it is sufficient to guarantee that cheating clients cannot equivocate, that is, send different transactions to different replicas. This property is guaranteed by broadcast protocols. These protocols have a similar architecture: the broadcast of transactions is initiated by clients, who either drive the whole broadcast instance [3] or outsource it to trusted replicas [10]. Therefore, a cheating client might lose liveness [3, 22], but equivocating is not possible.

Specifically, in FastPay [3] a client sends its transaction (a payment to some recipient) to all replicas, waits for $n - f$ signatures on it, and forms a certificate with them. The certificate is enough for the sender and the recipient to consider the payment finalized, because it proves that no conflicting transaction can ever be accepted by the replicas. The replicas update the balance of the sender and the recipient when they receive the certificate from the client. A necessary component in the construction is a sequence number maintained by each client: transactions submitted by a client must have consecutive sequence numbers, and no transaction may be pending (a transaction is *pending* when a replica has signed it but not received the certificate for it) when the client submits the next one. The sequence number is exactly what provides safety for payments: clients cannot equivocate (e.g., double-spend) because they can submit at most one transaction per sequence number, and all transactions submitted by a client are ordered. Malicious client, trying to send conflicting transactions for a sequence number, may lose liveness by not being able to form a certificate for any of them.

Astro [10] generalizes the sequence number to an *xlog*, an append-only log that contains all outgoing payments from each account, maintained by the single owner of that account. Only the account owner can broadcast updates to it xlog, hence Astro guarantees total order within each xlog and achieves safety for payments. ABC [22]

is similarly based on reliable broadcast [5]. In addition to transactions, replicas can broadcast *votes* for transactions they have seen, and the votes are weighted by the replica's stake. This enables the system to also work in permissionless settings.

FastPay [3] reaches throughput of $140K$ tps with a latency of approx. 200 ms in a WAN deployment with four replicas. Astro [10] achieves a throughput of $5K$ tps with latency of approx. 200 ms [10, Fig.4, Astro II] in a WAN deployment with 100 replicas. ABC [22] provides no implementation or benchmarks.

Conclusion. To the best of our knowledge, the only consensusless system that has been used in production and supports both payments and arbitrary objects (data declared in smart contracts) is FastPay in the Sui Blockchain [4]. However, that same work observes that consensusless protocols cannot offer checkpointing and are prone to losing liveness even for honest clients, due, for example, to clients' misconfigurations. For this reason, the consensusless protocol is combined with a DAG-based consensus protocol [4]. Transactions that do not need total order can be executed as long as the client broadcasts them, but *all* transactions eventually go through the consensus protocol. Hence, the system offers significantly better latency, but a consensus protocol is still required, and it must be able to handle the total workload of the system. Since different replicas hold different state at any moment in a consensusless protocol, the combination with a consensus protocol also allows light clients to deterministically read a consistent state from the system.

Moreover, all these protocols are tailor-made for payment systems and cannot be used for general distributed applications. They employ, directly or indirectly, sequence numbers in order to achieve total-order for transactions sent by each client, a property that is required [16] for payment systems but not for other use cases, such as auctions.

2.6 Separating block builders and proposers

Chop chop [8] introduces a new layer, called the *brokers*, between clients and replicas running a consensus protocol. Brokers are responsible for building blocks of transactions in a way that minimizes the transaction metadata (such as client signatures) in a block. This allows blocks to contain a larger number of transactions resulting in a system with higher throughput, compared to the underlying consensus protocol. On the other hand, brokers engage in interactive protocols with the clients and the replicas, hence increasing the time needed for a transaction to get committed. The system can support multiple brokers, but each of them runs a non-distributed protocol. Hence, fairness and censorship resistance are not achieved. Encrypting client transactions would not be enough – the brokers need to know the client behind each transaction because they engage in an interactive multi-signature protocol, hence they can censor specific clients.

3 Leader-based protocols with inclusion lists

In this section we present solutions that can be plugged in any leader-based protocol. They change the way a proposal is created and voted for. The underlying protocol proceeds in *epochs* and each epoch has a unique *leader*. We adopt the network model of the underlying protocol (partially synchronous in case of Tendermint [6] and HotStuff [30]).

3.1 The base protocol

The protocol. Clients submit transactions to replicas. On every epoch of the protocol each replica creates an *inclusion list*, signs it and sends it to the leader of that epoch.

The leader waits for $n - f$ inclusion lists from distinct replicas and creates a block that contains *only* the $n - f$ inclusion lists and no other transactions. Upon receiving the proposal, each replica sends a vote if it considers the block valid, according to the underlying protocol. The validity condition now additionally requires that the block contains at least $n - f$ inclusion lists, each signed by a different replica. Note that role of the leader now only consists in choosing which $n - f$ (or more) inclusion lists will be used in the new block.

Properties. The protocol achieves the same safety and liveness properties as the underlying leader-based protocol, and the additional *short-term censorship resistance* property. Note that, as in the underlying protocol, liveness can be attacked by malicious leaders (e.g., by remaining silent and not producing any block), but selective censorship is not possible. We present arguments to incentivize leaders to produce blocks on section 5.

Censorship resistance comes from the fact that the leader can only ignore up to f inclusion lists. If it ignores more, no honest replica will vote for the proposal. Hence, the client needs to ensure that $f + 1$ honest replicas have received its transaction. This can be achieved by sending it to $2f + 1$ replicas.

Special cases.

- It can be the case that not all transactions in the $n - f$ inclusion lists fit in the next block. To maintain fairness, a deterministic rule is needed for the leader to choose which transactions to add. One option is to have the leader add transactions by frequency of appearance in the inclusion lists. A second is to require that transactions are ordered in the inclusion lists and the leader selects the first x transactions from each inclusion list, such that x is as large as possible given the block size.
- Contradictory transactions may exist in the inclusion lists, such as two transactions from a client who can only pay the fees for one of them. We can again break the tie in a deterministic way, for example by keeping the transaction from the inclusion list of the replica with the lowest identifier.

Advantages and drawbacks. For an overview of the construction we refer to Table 1. Our modification can be implemented by having every replica send its inclusion list together with the last vote message of the previous epoch. For example, if implemented on Tendermint [6], the inclusion lists can be sent using ABCI++, piggybacked on vote messages. The proposal latency and proposal period hence remain unchanged. Assuming the leader does not remain silent and none of the special conditions explained above applies, an honest client's transaction will be included in the next block, that is, with a transaction delay of 0. However, similar to the protocols presented in Section 2.3, the construction leads to transaction duplication, as a transaction may appear in multiple inclusion lists. We denote this in Table 1 as an $O(n^2 \cdot |L|)$ additional communication cost, as the leader has to include to its proposal $O(n)$ inclusion lists of average size $|L|$. We present mitigations in the following sections.

3.2 Using a Data Availability layer

In this version, the inclusion lists contain *references* to transactions. The full transactions are submitted by the client to a Data Availability (DA) layer. We abstract the DA layer as follows.

Definition 1 (Data Availability (DA) scheme [20]). *A* DA *scheme is run among* clients *and* storage servers*. It exposes the following algorithms, which are initiated by a client and by the client and all storage nodes.*

- *disperse(*tx*)* $\rightarrow P$[5]*: It takes as input a transaction* tx *and returns a* certificate of retrievability P*.*
- *retrieve(*P*)* $\rightarrow \mathsf{tx}$*: It takes as input a certificate of retrievability* P *and returns a transaction* tx *or* $\perp$*.*

If an honest client invokes disperse(tx*), then it will obtain a certificate of retrievability* P*, such that, if an honest (and possibly different) client invokes retrieve(*P*), then the second client will obtain* tx*. Moreover, all calls to retrieve() return the same value to all honest clients, except with negligible probability, even if the client that initiated disperse() was malicious (in which case retrieve() may return* $\perp$*).*

The protocol. The client firsts submits transaction tx to the DA layer. Once it obtains the certificate of availability P, it sends it to all replicas. Upon receiving P, if *retrieve(*P*)* $\neq \perp$ then a replica appends P to its inclusion list, which is forwards to the leader. The leader waits for $n - f$ valid inclusion lists, where an inclusion list is valid if, for all certificates of availability P it contains, it holds that *retrieve(*P*)* $\neq \perp$. The leader creates a block that contains all transactions retrieved from the $n - f$ valid inclusion lists. The leader sends a proposal with the new block and the $n - f$ signed inclusion lists to all replicas. The proposal is valid if it contains $n - f$ inclusion lists and the block contains all corresponding transactions.

Let t_{disp} denote the average time of *disperse()* and t_{ret} that of *retrieve()*. This construction increases the proposal latency by t_{ret}, because a replica has to run *retrieve()* before voting for a proposal, and the minimum transaction delay becomes $t_{\text{disp}} + t_{\text{ret}}$, because a client needs to disperse tx and the leader checks that it can be retrieved. Assuming the leader will produce some block, the maximum transaction delay is $t_{\text{disp}} + t_{\text{ret}}$ as well. Finally, communication complexity increases by a factor of c_{da}, depending on the implementation of the DA layer. We show these on Table 1.

Advantages and drawbacks. The inclusion list can now contain pointers to transactions, while the actual payload exists only in the DA layer. On the other hand, the DA layer adds latency to the protocol.

Optimizations.

- In order to further reduce the output size, the leader can write the certificates of availability – instead of the corresponding transactions – in the block. This comes at the cost or requiring clients to query the DA layer and retrieve it.
- We can allow clients to submit invalid certificates of availability, i.e., P for which *retrieve(*P*)* $= \perp$. This works because, by the properties of the DA scheme, clients that read the output of our protocol will agree on the output of *retrieve(*P*)*. The drawback of this is that the output can contain garbage transactions.

We remark that the proposed construction is similar to BigDipper [28], with the following differences. First, the DA layer is here decoupled from the consensus layer, and it is the client's responsibility to disperse the transaction. Second, our protocol achieves transaction deduplication, as the leader includes each transaction only once in the proposed block.

[5] We abstract the commitment C from [20] inside P.

3.3 Using reliable broadcast

Instead of using a separate Data Availability layer, in this section we have the client broadcast tx to the replicas.

The protocol. The client sends a transaction tx using a version of reliable broadcast [5]. The broadcast algorithm consists of three communication steps. On the first, the client sends tx to all replicas. The other two consist of all-to-all communication among the replicas. When a replica delivers tx, it adds to its inclusion list the hash of tx. By properties of reliable broadcast, if an honest replica delivers tx, then all honest will eventually deliver tx. Hence, for every inclusion list of an honest replica, the leader will eventually deliver all included transactions. The leader includes in the new block the first $n - f$ inclusion lists whose transactions are delivered in the broadcast layer.

As shown on Table 1, when implemented on top of a leader-based protocol, this construction results in a minimum and maximum transaction delay of 2δ, (for the maximum we assume the leader will not remain silent), because reliable broadcast requires two additional communication rounds. Proposal latency and proposal period remain unchanged, while communication complexity increases by a factor of $O(n^2 \cdot s)$, where s is the average transaction size.

Advantages and drawbacks. The inclusion lists, appended to the new block, can now contain hashes of transactions, and not the transactions themselves, thus reducing the size of the block. On the other hand, the protocol adds two all-to-all communication rounds to the underlying consensus protocol.

Notice that, different to DAG-based approaches, broadcasts in this construction are initiated by the clients, and they are performed on transaction and not block level. Hence, we can avoid transaction duplication.

3.4 Using a gossip layer

Instead of a broadcast primitive, we can use a gossip layer to make transactions available to all parties.

The protocol. The only difference from the previous section is that replicas do not broadcast the transactions received from clients, but they gossip them to each other. The inclusion lists contain again pointers to transactions. Since there are at least $n - f$ honest parties, and assuming the gossip layer has been instantiated correctly to allow propagation of transactions to all replicas, the leader will eventually receive $n - f$ inclusion lists, such that it has received the corresponding transactions via the gossip layer.

Let t_{prop} denote the average propagation time and c_{gos} the number of replicas each replica connects to in the gossip-layer implementation, and s the average transaction size. As shown on Table 1, when implemented on top of a leader-based protocol, this construction results in a minimum and maximum transaction delay of t_{prop}, assuming the leader will not remain silent. Proposal latency and proposal period remain unchanged, while communication complexity increases by a factor of $O(n \cdot c_{\text{gos}} \cdot s)$.

Advantages and drawbacks. Compared to the broadcast based, this solution does not require two additional rounds of communication for every transaction. Moreover, replicas need to maintain fewer network connections, as there is no all-to-all communication.

3.5 A protocol without writing the inclusion lists on the block

We now present a modification to the protocol in Section 3.1 which does not require the leader to append the $2f + 1$ used inclusion lists in the proposal message.

The protocol. Similar to Section 3.1 each replica sends its inclusion list to the leader. The leader chooses $n - f$ and creates a block with their transactions. In the proposal message the leader includes the *lists-used* field, a list with the identifiers of the replicas whose inclusion lists it used. Replicas vote for a proposal only if contains a *lists-used* field of size at least $2f + 1$. Additionally, a replica whose identifier is in the *lists-used* field verifies whether its inclusion list is indeed in the transactions of the new block.

On Table 1 we summarize the trade-offs of this solution. Compared to the protocol in Section 3.1, this only incurs an additional communication cost of $O(n \cdot |L|)$, where $|L|$ is the average size of inclusion lists, as each replica sends one inclusion list to the leader.

Design choices and correctness. Note that the leader must send the proposal and consider the votes from all the replicas. If it sent it only to the $2f+1$ whose inclusion lists it used, or counted the votes only from them, then a single malicious replica among these $2f + 1$ would be able to harm liveness. In other words, the f replicas whose inclusion list was not used by the leader have to vote for the proposal, without being able to verify whether the leader actually included all the transactions from the *lists-used* field. Observe that these might be honest replicas. Moreover, f votes can come from malicious replicas, hence the leader needs only one vote from an honest replica in the *lists-used* field. This means that the leader only needs to actually use *one* inclusion list sent by an honest replica, when it claims to have used $2f + 1$.

Censorship resistance. In this protocol the leader can ignore up to $2f$ inclusion lists from honest replicas. Hence, the client needs to ensure that $2f + 1$ honest replicas have received its transaction. This can be achieved by sending it to all replicas. We comment on Section 5 on how this translates to worse censorship resistance, compared to the rest of the protocols in this section.

4 The Partially Ordered Dataset (POD) Construction

In this section we describe a approach based on *Partially Ordered Dataset (POD)*[6], a solution that allows censorship-resistant recording of arbitrary messages. POD can be used as the base for a wide range of consensus-less applications, such as payment systems, a shared mempool layer for decentralized sequencers, auctions with non-omission guarantees, and storing feeds for decentralized social media. By giving up on total order we come up with a high-performant layer in terms of both throughput and latency.

POD associates transactions with *timestamps*. It does not guarantee full ordering, as classical consensus mechanisms do, but instead results in a 'fluctuating order'. That is, honest clients may observe different timestamps for each transaction, yet the timestamps will be within a restricted and well-defined interval.

[6] `https://commonprefix.notion.site/Pod-4842ce5f7ccd47ff90e1b4b447519498`

4.1 Modeling POD

We assume that time proceeds in *rounds* and that parties have loosely synchronized clocks allowing them to determine the current round, and the local round counter is used to assign timestamps to transactions.

Definition 2 (Partially Ordered Dataset (POD)). *A* Partially Ordered Dataset (POD) *is a finite sequence of pairs,* $D = \{(r, T), \ldots, (r', T')\}$*, where* r *is a round and* T *is a set of transactions. For an entry* (r, T)*, notation* $D[r]$ *returns* T*.*

Definition 3 (Past-perfect round). *We say that a round* r_{perf} becomes past perfect *if no new transactions can appear with recorded round* $r_{rec} \leq r_{perf}$*.*

Definition 4 (POD protocol). *A POD protocol exposes the following methods.*

- *input event write(*tx*)*
- *output event write_return(*tx*,* π*)*
- *input event read_perfect()*
- *output event read_perfect_return(*r*,* D*,* Π*)*
- *input event read_all()*
- *output event read_all_return(*D*,* Π*)*
- *identify(*π*,* Π*)* $\rightarrow P' \subset P$

Clients call write(tx*) to write a transaction* tx*. Upon completion, the protocol outputs write_return(*tx*,* π*), where* π *is a* record certificate*. A transaction, for which an honest client has obtained a record certificate, is called* recorded*. Clients call read_perfect() to read the transactions in the bulletin. Upon completion, the protocol outputs read_perfect_return(*r*,* D*,* Π*), where* r *is a round, called the* past perfect *round,* L *is a set of transactions,* D *is a POD, and* Π *is a* past-perfect certificate*. For each entry* (r', T) *in* D*, we say that transactions in* T *became* finalized *at round* r'*. Operation read_all() is similar, but it returns all transactions up to the current round without past-perfection guarantees, hence it can return faster than read_perfect(). Clients call identify(*π*,* Π*)* $\rightarrow P' \subset P$ *to identify the set* P' *of parties who vouched for the finalization of a* tx*, where* Π *is a POD and* π *is the certificate returned by write_return(*tx*,* π*).*

A POD satisfies the following properties.

1. Liveness
   - **Acknowledged within** u**:** Assume an honest client calls *write(*tx*)* at round r. Then the protocol will output *write_return(*tx*,* π*)* by round $\mathsf{r} + u$.
   - **Included within** v **with accountability:** Assume the protocol outputs *write_return(*tx*,* π*)* to an honest client at round r_1, for some transaction tx. Then, if an honest client calls *read_perfect()* at round $\mathsf{r}_2 \geq \mathsf{r}_1 + v$, one of the following holds: (1) The protocol will output *read_perfect_return(*r*,* $\cdot$*,* Π*)*, for some $\mathsf{r} \leq \mathsf{r}_2$, such that D contains an entry $(\mathsf{r}', \mathsf{T})$ with $\mathsf{r}' \leq \mathsf{r}$ and $\mathsf{tx} \in \mathsf{T}$. (2) The POD D and certificates π and Π identify a set of replicas P' that have misbehaved.
2. Safety
   - **Transaction safety:** Assume *read_perfect()* returns $(\cdot, D_1, \cdot)$ to an honest client and $(\cdot, D_2, \cdot)$ to another. If a transaction tx satisfies $\mathsf{tx} \in D_1[\mathsf{r}]$ and $\mathsf{tx} \in D_2[\mathsf{r}']$, for some r and r', then $|\mathsf{r} - \mathsf{r}'| < W$, where W is a parameter we call *temperature*.
   - **Past-perfection accountable safety:** Assume *read_perfect()* returns $(\mathsf{r}_1, D_1, \Pi_1)$ and $(\mathsf{r}_2, D_2, \Pi_2)$ to two honest clients, for $\mathsf{r}_2 \geq \mathsf{r}_1 + \delta$. If $\mathsf{tx} \in D_2[\mathsf{r}]$, for some round $\mathsf{r} < \mathsf{r}_1$, then $\mathsf{tx} \in D_1[\mathsf{r}']$, for some round $\mathsf{r}' \leq r_1$. Otherwise, certificates Π_1 and Π_2 identify a set of replicas P' that have misbehaved.

Fair punishment No honest replica gets punished as a result of malicious operation. If *identify*(π, Π) $\rightarrow P'$, where π is a record certificate for transaction tx and Π is a past-perfect certificate for a POD D, can only be created if all parties in P' sign tx and D.

In Figure 1 we show an example. The timestamp of tx_1 fluctuates within some *temperature*, and transaction tx_2 may or may not be returned by the first read operation. Similarly, the second read operation may include any subset of $\mathsf{tx}_4, \mathsf{tx}_5$, but it will include all previously written transactions. Concurrent transactions, such as tx_3 and tx_4, may be observed by clients in any order. When a timestamp becomes past-perfect, honest clients are guaranteed that no more transactions will ever be output with a timestamp smaller than the past-perfect. This is illustrated in Figure 3, where the output of the protocol contains a 'finalized' part and an evolving part.

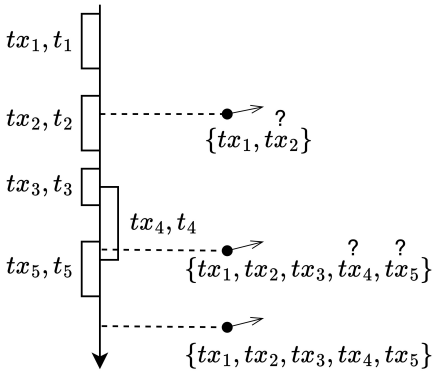

Fig. 1: Example, POD assigning timestamps to transactions

4.2 A construction for POD

We present an overview of the construction and a sketch of the security arguments. In Figure 2 and Figure 3 we show the *write()* and *read_perfect()* operations, respectively, in an implementation with five replicas.

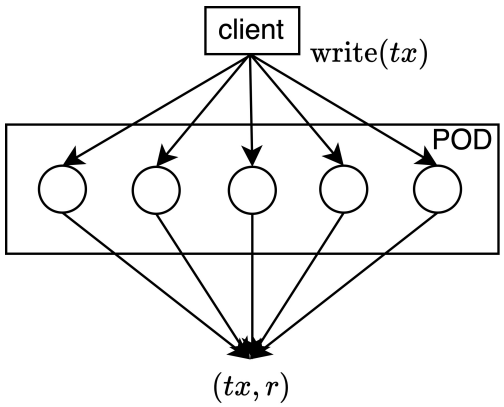

Fig. 2: POD, write operation

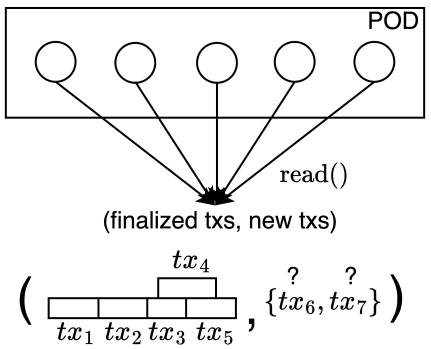

Fig. 3: POD, read operation

Writing transactions. In order to write a transaction a client sends it to all replicas. Upon receiving a write request, a replica assigns it its local round number r_{loc} and returns it to the client. Upon receiving $n - f$ write responses, a client considers the request *recorded*. If R is the set with the $n - f$ local rounds received by the client, then the *recorded round* of the transaction is $\mathsf{r}_{\text{rec}} = med(R)$.

Reading transactions. Following recent techniques from the literature [21], we allow clients to make their own timing assumptions when reading the state for the system, resulting in two ways to decide when a round becomes past-perfect. Clients that believe in synchrony can use the following protocol. A client sends a *read_all()* request to the replicas, to which a replica responds with (r, D), where r is its local round and D contains all the pairs $(\mathsf{tx}', \mathsf{r}')$ it possesses. The client waits for $n-f$ read responses. If not interested in past-perfection guarantees (i.e., for a *read_all()*), the client can output the returned transactions with the median as the recorded round. Otherwise, for a *read_perfect()*, it does the following. First, it sets r_{perf} to be the *minimum* of the received r values. Second, it sets $\mathsf{T}_{\text{final}}$ to be the union of all transactions in the responses. Then the client uses the *client-gossip technique* [25], hence making sure that r_{perf} will become a past-perfect round after δ time and that $\mathsf{T}_{\text{final}}$ contains *all* the transactions with recorded round $\mathsf{r}_{\text{rec}} \leq \mathsf{r}_{\text{perf}}$.

Client gossip. [25] When clients observe that replicas miss some transactions, they rewrite them back to the replicas using *write()*. If writing takes time T, this technique ensures the client that a round becomes past perfect after T, as all replicas will return the rewritten transactions to subsequent read operations. In periods of synchrony we can set $T = \delta$.

Consensus layer. The system contains a second component, a lightweight consensus layer, which replicas also use to agree on past-perfect rounds. The output of this component can be seen as a prefix of the synchronous reading protocol. Clients that require total-order safety can use the consensus component. It is lightweight in the sense that its input and output are round numbers – specifically, it does not involve the transactions themselves. An existing finality gadget, such as Casper FFG [7], can be used as well. This consensus layer, as discussed in Sections 2.4 and 2.5, can also be used for checkpointing and pruning the state of the system.

Light clients and optimization. The system can support light clients without additional effort, using the provided *read_all()* and *read_perfect()* operations. An additional argument r_{min} can be used to instruct replicas to only return transactions newer that r_{min}. Moreover, the replicas can be lightweight (they do not need a hard drive) if 'secondaries' are used (akin to DBMS secondaries). Finally, the requirement for a client to know the addresses and to write to all replicas can be lifted by employing 'helper nodes' that are responsible for writing. Observe that these nodes cannot harm safety for the client, who still is the one that signs the transactions.

5 Economic arguments

The economic-censoring model. When reasoning about the censorship resistance of a protocol, we work with two models. The first is the *honest-malicious* setting, where *honest* replicas follow the protocol, and hence do not censor any transactions they have received, and *malicious* replicas can behave arbitrarily. The second is the *economic-censoring* model, which is the same as the honest-malicious, but replicas (both honest and malicious) have one additional choice: for each received client transaction, they decide whether to ignore it or not. They base this choice on economic criteria, which we abstract in the notion of a *bribery*. A bribery for a transaction tx is an amount of money greater than the reward a replica would get for including that transaction. If a replica is bribed to censor tx, then it will censor it, and if it is not bribed, then it will not censor it.

Censorship resistance of the protocol in Sections 3.1–3.4. In order for the adversary to delay the inclusion of a transaction for one epoch, it would have to bribe the leader of that epoch and $2f$ replicas.

Censorship resistance of the protocol in Section 3.5 . In order for the adversary to delay the inclusion of a transaction for one epoch, it would have to bribe the leader of that epoch and f replicas.

6 Existing implementations

In this section we provide references to implementations of the protocols mentioned throughout the report.

The 3-round version of HotStuff [30] (see HotStuff/DiemBFT in Table 1 and Section 2.2) has been implemented[7] in Rust, but the authors state it is not production ready. The same protocol is available[8] as part of Diem's codebase, again in Rust. A modular, academic implementation[9] exists in Go, a prototype implementation[10] as part of the Bamboo [14] framework also exists in Go, while the academic prototype[11] for the original paper was written in C++.

The 2-round version of HotStuff [15] (see Jolteon in Table 1 and Section 2.2) has been implemented[12][13][14] in some of the aforementioned repositories. Ditto, the 2-round version of HotStuff with an asynchronous fallback protocol, has also been implemented[15] in Rust. A prototype implementation[16] of HotStuff-over-Narwhal is also available.

Regarding DAG-based protocols, Narwhal/Tusk [12], that is, the asynchronous Tusk consensus protocol over Narwhal, is available[17] in Rust. Narwhal/Bullshark, that is, the partially-synchronous BullShark consensus protocol over Narwhal, has also been implemented[18] in Rust. A prototype implementation of Mysticeti [2] is also available[19]. Mysten labs provides implementations of Narwhahl[20], as well as Narwhal/Tusk and Narwhal/Bullshark [21].

The Tendermint consensus algorithm [6], also known as CometBFT, has been implemented[22] in Go and in Rust[23]. FastPay has been implemented[24] by Facebook in Rust.

[7] https://github.com/asonnino/hotstuff/tree/3-chain
[8] https://github.com/diem/diem/tree/latest/consensus
[9] https://github.com/relab/hotstuff/tree/master/consensus/chainedhotstuff
[10] https://github.com/gitferry/bamboo/tree/master/hotstuff
[11] https://github.com/hot-stuff/libhotstuff
[12] https://github.com/asonnino/hotstuff/
[13] https://github.com/relab/hotstuff/tree/master/consensus/fasthotstuff
[14] https://github.com/gitferry/bamboo/tree/master/fasthostuff
[15] https://github.com/danielxiangzl/Ditto
[16] https://github.com/facebookresearch/narwhal/tree/narwhal-hs
[17] https://github.com/asonnino/narwhal
[18] https://github.com/asonnino/narwhal/tree/bullshark
[19] https://github.com/MystenLabs/mysticeti
[20] https://github.com/MystenLabs/sui/tree/main/narwhal
[21] https://github.com/MystenLabs/narwhal
[22] https://github.com/cometbft/cometbft
[23] https://github.com/informalsystems/tendermint-rs
[24] https://github.com/novifinancial/fastpay

7 Conclusion and recommendations

In this report we evaluate existing leader-based protocols (Section 2.2) and propose censorship-resistance solutions that can be implemented on top of them (Section 3). We also explore existing parallel-leader (Section 2.3) and DAG-based (Section 2.4) protocols, that achieve censorship resistance by construction. Yet, the parallel-leader protocols pay the price of transaction duplication, are not efficient, and feature no production implementation. The DAG-based protocols, albeit reaching comparatively high throughput, they still suffer from high latency. This is the reason why in production they have been combined with consensusless protocols [4], resulting in sub-second latency. We review consensusless protocol in Section 2.5.

For the requirements of Flashbots, there exist several options.

Leader-based. A leader-based protocol with a censorship-resistance add-on component (Section 3) would achieve the desired definition of short-term censorship resistance. The constructions in Sections 3.2–3.5 avoid transaction duplication. The one in Section 3.5 does not require additional communication rounds. As shown in Section 5, the constructions in Sections 3.1–3.4 achieve, in a rational setting, better censorship resistance. Of advantage here is the existence of production implementations, in particular of Tendermint, but also of HotStuff. The drawback with this approach is the low throughput and high – for the requirements of Flashbots – latency of single-leader protocols, as well as the performance deterioration in presence of crash faults.

DAG-based. DAG-based protocols also achieve short-term censorship resistance, but with duplicate (specifically, up to $2f + 1$ copies) transactions in the output of the protocol. An option would be to develop (or build on existing codebases of) or Narwhal/Bullshark [23], aiming for latency in the order of 2 seconds, or Mysticeti [2], aiming for sub-second latency. We observe, however, that, after removing duplicate transactions, the throughput of DAG-based protocols is not expected to differ much from single-leader protocols.

Broadcast-based POD construction. Consensus-based protocols, either leader-based, with multiple leaders, or DAG-based, are not fast enough for the requirements of Flashbots. We believe that a solution sufficient for the requirements of Flashbots must be based on a consensusless protocol. For these reasons we have proposed POD in Section 4, and we provide in Section 8 the implementation path we are planning to follow for POD. As we reason in Section 2.4, systems based solely on consensusless primitives cannot offer attributes such as checkpointing. For this reason we have explored how we can combine POD with a lightweight consensus layer, or with an existing consensus network, in order to enable the complete functionality.

8 Implementation plan for POD

We now provide details for an MVP implementation, leaving some parts as future work.

Stage 1: MVP with POD core functionality

– This includes the *write()* and synchronous *read_perfect()* protocols that we have presented.

– The resulting protocol satisfies the liveness and safety properties as presented in Section 4.1, that is, the **Past-perfection safety** property holds assuming synchrony. Specifically, some client c_2, performing a *read_perfect()* after client c_1, may observe a transaction that c_1 has not observed, if the two reads are too close to each other.
– Accountability-related properties are not satisfied.
– We do not need to implement an execution engine or heavy machinery, like Cosmos SDK does, as we are building a lazy ledger.
– The implementation can be based on the FastPay or Google's Certificate Transparency code-base. Certificate Transparency, in particular, offers built-in accountability, which our implementation could benefit from in the next steps.

Stage 2: POD Core + consensus layer

The consensus layer can be used by clients who wish to *read_perfect()* who wish to obtain total-order guarantees. Additionally, the consensus layer will allow the system to use checkpoints and safely prune old transactions.

Stage 3: POD Core + consensus layer + fee mechanism

Additionally, this will enable incentivizing, slashing, complaining, and hence accountability. The fee and slashing logic can also be implemented on an existing platform, such as a smart contract on Ethereum, without a consensus layer over POD.

References

1. N. Alhaddad, L. Reyzin, and M. Varia. Committing avid with partial retrieval and optimal storage. Cryptology ePrint Archive, Paper 2024/685, 2024. `https://eprint.iacr.org/2024/685`.
2. K. Babel, A. Chursin, G. Danezis, L. Kokoris-Kogias, and A. Sonnino. Mysticeti: Low-latency DAG consensus with fast commit path. *CoRR*, abs/2310.14821, 2023.
3. M. Baudet, G. Danezis, and A. Sonnino. Fastpay: High-performance byzantine fault tolerant settlement. In *AFT*, pages 163–177. ACM, 2020.
4. S. Blackshear, A. Chursin, G. Danezis, A. Kichidis, L. Kokoris-Kogias, X. Li, M. Logan, A. Menon, T. Nowacki, A. Sonnino, B. Williams, and L. Zhang. Sui lutris: A blockchain combining broadcast and consensus. *CoRR*, abs/2310.18042, 2023.
5. G. Bracha. Asynchronous byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, 1987.
6. E. Buchman, J. Kwon, and Z. Milosevic. The latest gossip on BFT consensus. *CoRR*, abs/1807.04938, 2018.
7. V. Buterin and V. Griffith. Casper the friendly finality gadget. *CoRR*, abs/1710.09437, 2017.
8. M. Camaioni, R. Guerraoui, M. Monti, P. Roman, M. Vidigueira, and G. Voron. Chop chop: Byzantine atomic broadcast to the network limit. *CoRR*, abs/2304.07081, 2023.
9. D. Cason, E. Fynn, N. Milosevic, Z. Milosevic, E. Buchman, and F. Pedone. The design, architecture and performance of the tendermint blockchain network. In *SRDS*, pages 23–33. IEEE, 2021.
10. D. Collins, R. Guerraoui, J. Komatovic, P. Kuznetsov, M. Monti, M. Pavlovic, Y. Pignolet, D. Seredinschi, A. Tonkikh, and A. Xygkis. Online payments by merely broadcasting messages. In *DSN*, pages 26–38. IEEE, 2020.
11. CometBFT Team. CometBFT QA Results v0.37.x. `https://docs.cometbft.com/v0.37/qa/cometbft-qa-37`, 2024.
12. G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman. Narwhal and tusk: a dag-based mempool and efficient BFT consensus. In *EuroSys*, pages 34–50. ACM, 2022.
13. I. Doidge, R. Ramesh, N. Shrestha, and J. Tobkin. Moonshot: Optimizing chain-based rotating leader BFT via optimistic proposals. *CoRR*, abs/2401.01791, 2024.
14. F. Gai, A. Farahbakhsh, J. Niu, C. Feng, I. Beschastnikh, and H. Duan. Dissecting the performance of chained-bft. In *ICDCS*, pages 595–606. IEEE, 2021.
15. R. Gelashvili, L. Kokoris-Kogias, A. Sonnino, A. Spiegelman, and Z. Xiang. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. In *Financial Cryptography*, volume 13411 of *Lecture Notes in Computer Science*, pages 296–315. Springer, 2022.
16. R. Guerraoui, P. Kuznetsov, M. Monti, M. Pavlovic, and D. Seredinschi. The consensus number of a cryptocurrency. *Distributed Comput.*, 35(1):1–15, 2022.
17. M. Kelkar, S. Deb, S. Long, A. Juels, and S. Kannan. Themis: Fast, strong order-fairness in byzantine consensus. In *CCS*, pages 475–489. ACM, 2023.
18. D. Malkhi and K. Nayak. Extended abstract: Hotstuff-2: Optimal two-phase responsive BFT. *IACR Cryptol. ePrint Arch.*, page 397, 2023.
19. A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song. The honey badger of BFT protocols. In *CCS*, pages 31–42. ACM, 2016.
20. K. Nazirkhanova, J. Neu, and D. Tse. Information dispersal with provable retrievability for rollups. In *AFT*, pages 180–197. ACM, 2022.
21. J. Neu, E. N. Tas, and D. Tse. Ebb-and-flow protocols: A resolution of the availability-finality dilemma. In *SP*, pages 446–465. IEEE, 2021.
22. J. Sliwinski and R. Wattenhofer. ABC: asynchronous blockchain without consensus. *CoRR*, abs/1909.10926, 2019.
23. A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias. Bullshark: DAG BFT protocols made practical. In *CCS*, pages 2705–2718. ACM, 2022.
24. A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias. Bullshark: The partially synchronous version. *CoRR*, abs/2209.05633, 2022.
25. S. Sridhar, D. Zindros, and D. Tse. Better safe than sorry: Recovering after adversarial majority. *IACR Cryptol. ePrint Arch.*, page 1556, 2023.
26. C. Stathakopoulou, T. David, and M. Vukolic. Mir-bft: High-throughput BFT for blockchains. *CoRR*, abs/1906.05552, 2019.
27. The Diem Team. DiemBFT v4: State Machine Replication in the Diem Blockchain. `https://developers.diem.com/papers/diem-consensus-state-machine-replication-in-the-diem-blockchain/2021-08-17.pdf`, 2021.
28. B. Xue, S. Deb, and S. Kannan. Bigdipper: A hyperscale BFT system with short term censorship resistance. *CoRR*, abs/2307.10185, 2023.

29. L. Yang, S. J. Park, M. Alizadeh, S. Kannan, and D. Tse. Dispersedledger: High-throughput byzantine consensus on variable bandwidth networks. In *NSDI*, pages 493–512. USENIX Association, 2022.
30. M. Yin, D. Malkhi, M. K. Reiter, G. Golan-Gueta, and I. Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *PODC*, pages 347–356. ACM, 2019.

About Common Prefix

Common Prefix is a blockchain research, development, and consulting company consisting of a small number of scientists and engineers specializing in many aspects of blockchain science. We work with industry partners who are looking to advance the state-of-the-art in our field to help them analyze and design simple but rigorous protocols from first principles, with provable security in mind.

Our consulting and audits pertain to theoretical cryptographic protocol analyses as well as the pragmatic auditing of implementations in both core consensus technologies and application layer smart contracts.