Autobahn: Seamless high speed BFT

#26

Abstract

Today's practical, high performance Byzantine Fault Tolerant (BFT) consensus protocols operate in the partial synchrony model. However, existing protocols are inefficient when deployments are indeed *partially* synchronous. They deliver either low latency during fault-free, synchronous periods (*good intervals*) or robust recovery from events that interrupt progress (*blips*). At one end, traditional, view-based BFT protocols optimize for latency during good intervals, but, when blips occur, can suffer from performance degradation (*hangovers*) that can last beyond the return of a good interval. At the other end, modern DAG-based BFT protocols recover more gracefully from blips, but exhibit lackluster latency during good intervals. To close the gap, this work presents Autobahn, a novel high-throughput BFT protocol that offers both low latency and *seamless* recovery from blips. By combining a highly parallel asynchronous data dissemination layer with a low-latency, partially synchronous consensus mechanism, Autobahn (*i*) avoids the hangovers incurred by traditional BFT protocols and (*ii*) matches the throughput of state of the art DAG-based BFT protocols while cutting their latency in half, matching the latency of traditional BFT protocols.

1 Introduction

This work presents Autobahn, a Byzantine Fault Tolerant (BFT) state machine replication (SMR) protocol that sidesteps current tradeoffs between low latency, high throughput, and robustness to faults.

BFT SMR offers the appealing abstraction of a centralized, trusted, and always available server, even as some replicas misbehave. Consensus protocols implementing this abstraction [22, 52, 59] are at the core of recent interest in decentralized systems such as blockchains, central-bank endorsed digital currencies [2, 3], and distributed trust systems. CCF, a TEE-enabled BFT system [10], for instance, is used to power Azure SQL's integrity functionality [9]. These protocols must offer (*i*) high throughput, (*ii*) low end-to-end latency, and (*iii*) robustness to both faults and changing network conditions; all whilst being simple enough to build and deploy.

Today's popular (and actually deployed) BFT consensus protocols operate primarily in the *partial synchrony model* [28], introduced to get around the impossibility of a safe and live solution to consensus in an asynchronous system [31]. Partially synchronous protocols are always safe, but provide no liveness or performance guarantees before some "magic" *Global Stabilization Time* (GST), after which synchrony (all messages are received within a timebound Δ) is expected to hold forever on; they optimize for latency *after* GST holds. Deployed systems simulate this model using exponentially increasing timeouts–pragmatically, they declare GST attained and Δ found once timeouts are large enough that they are not being violated: in this regime, user requests (transactions) are guaranteed to commit.

In practice, however, this happy regime only holds intermittently: GST is an elegant fiction, as real deployments are subject to unpredictable replica and network failures, network adversaries (ddos/route hijacking), latency fluctuations, link asymmetries, etc. that can cause periods in which progress stalls (*blips*): these begin when a timeout is violated and end once timeouts are again met (perhaps by adjusting the timeout length, or removing a presumed faulty leader). Unlike theoretical partial synchrony, which promises endless bliss after an initial (asynchronous) storm, *real* partial synchrony is instead piece-wise: periods where timeouts are met are separated by blips, during which progress is stalled.

Unfortunately, we find that existing partially synchronous protocols are not actually efficient under real partial synchrony: they have to choose between achieving low-latency when timeouts are met or remaining robust to blips.

Traditional BFT [22, 36, 43, 59] protocols and their multi-leader variants [38, 54, 55] optimize for performance after GST (*i.e.,* in the absence of timeout events). This design choice allows these protocols to minimize message exchanges in the common (synchronous) case. Unfortunately, we find that, after even a brief blip, these protocols cannot guarantee a graceful resumption of operations once synchrony returns; instead, they experience what we characterize as a *hangover*. Specifically, under sustained high load, the lack of progress caused by a blip (and the consequent loss of throughput) can generate large request backlogs that can cause a degradation in end-to-end latency that persists well after the consensus blip has ended (Figs. 1,7,8). Thus, while on paper traditional BFT protocols enjoy low consensus latency, their poor resilience to blips can cause them to underperform under real-life partial synchrony.

DAG-based BFT, a newly popular class of consensus protocols [16, 24, 41, 46, 51, 52] take an entirely different approach, steeped in an *asynchronous* system model. Asynchronous protocols optimize for worst-case message arrivals, and leverage randomness to guarantee progress without relying on timeouts. DAG systems employ as backbone a high throughput asynchronous data dissemination layer which, through a series of structured, tightly synchronized rounds, forms a DAG of temporally related data proposals. Replicas eventually converge on the same DAG and deterministically *interpret* their local view to establish a consistent total order. This approach yields excellent throughput and reduces the effects of blips on the system. Unfortunately, these benefits come at the cost of prohibitive latency; consequently, these protocols see little practical use. In pursuit of lower latency, recent (deployed) DAG protocols [51, 53] switch to partial synchrony and reintroduce timeouts, but still require several rounds of Reliable Broadcast communication. For instance, Bullshark [52], the state of the art partially synchronous DAG protocol at the core of Sui [5] and Aptos [57], still requires up to 12 message delays (md) to commit transactions.

This paper shows that, thankfully, these tradeoffs are not fundamental. We propose Autobahn, a new consensus protocol that minimizes hangovers without sacrificing low latency during synchronous periods. Autobahn's architecture is inspired by recent DAG protocols: it constructs a highly parallel data dissemination layer that continues to make progress at the pace of the network (*i.e.,* at the lowest rate of transfer between correct nodes) even during blips. Atop, it carefully layers a traditional-style partially synchronous consensus mechanism that orders the stream of data proposals by committing concise state cuts. Like any partially synchronous protocol, consensus may fail to make progress during blips – however, upon return of progress, Autobahn can instantaneously commit the *entire* data backlog (with complexity independent from its size), mimizing the effect of hangovers.

The aim of decoupling consensus from data dissemination is not unique to Autobahn [15, 22, 24]. However, Autobahn more cleanly separates the precise responsibility of each layer and carefully regulates their interaction. This is key to coping with blips without experiencing hangovers while, at the same time, ensuring low latency in the common case.

Our results are promising (§6): Autobahn matches the throughput of Bullshark [52] while reducing its latency by more than half; and it matches the latency of Hotstuff [59] without suffering from Hotstuff's hangovers.

This paper makes three core contributions:

- It formalizes two new notions – *hangovers* and *seamless* partial synchrony (§2) – to characterize the performance of partially synchronous protocols in practice.
- It revisits the set of system properties required to design a BFT consensus protocol that avoids protocol-induced hangovers (*i.e.* is seamless).
- It presents the design of Autobahn, a novel *seamless* partially synchronous BFT consensus protocol with high throughput and low latency.

2 Partial Synchrony: Theory meets Practice

The partial synchrony model is an appealing theoretical framework for reasoning about liveness in spite of the seminal FLP impossibility result [31]. Real systems can in principle approximate this model using timeouts: they can exponentially increase the value of the timeout until they enter a regime where timeout violations subside, and progress becomes possible, as if the network had indeed reached its Global Stabilization Time. Unfortunately, timeouts are notoriously hard to set [44, 51, 58]): too low of a timeout, and even common network fluctuations stop progress; too high, and the protocol is slow to respond to faults.

In practice, timeout violations are thus simply inevitable. WAN latencies are unpredictable and can vary by order of magnitudes [39], and many RPCs have been showed to suffer from high tail latency [25, 49]. The operating context for today's blockchain systems is even more challenging. Participants in these systems can have highly asymmetric network and hardware resources, and widely different deployment experience, all factors that can greatly impact how well they can keep up with requests. These asymmetries complicate timeout configurations and result in frequent timeout events [6].

More fundamentally, accounting for the possibility of repeated timeout violations is simply part of the job description for any system that aims to be resilient to Byzantine faults. Malicious replicas may, for example, intentionally fail to send a required message, or a network adversary may target correct participants via ddos/route hijacking attacks. Indeed, recent work, for instance, demonstrates that targeted network attacks on leaders are surprisingly easy to mount and drastically hurt performance [34].

Whatever the cause may be, the corrective actions that BFT SMR protocols take in response to a timeout violation (e.g., electing a new consensus leader) can lead to *blips*, *i.e.,* periods during which progress stalls. Practical deployments are thus truly *partially* synchronous. Messages will not trigger timeouts for a while, but these periods of synchrony will be regularly interrupted by blips. To properly assess the performance of a partially synchronous system, we should consider how it behaves *at all times*, during periods of synchrony, but also after recovering from asynchrony blips. To this effect, we introduce the dual notions of *hangovers* and *seamlessness*. Informally, hangovers consist of performance degradations triggered by a blip that persist once the blip is over; we say that a partially synchronous protocol is seamless if it does not suffer from *protocol-induced* hangovers (defined in §2.1 below).

2.1 Hangovers & Seamlessness

Following Aardvark [23] and Abraham et al. [13] we say that an interval is *good* if the system is synchronous (w.r.t. some implementation-dependent timeout on message delay), and the consensus process is led by a correct replica. Intuitively, good intervals capture the periods during which progress is guaranteed; all non-good intervals are blips. Additionally, we consider an interval to be *gracious* [23] if it is good, and *all* replicas are correct. This distinction is helpful to reason about specific optimizations, such as so-called *fast paths* through the protocol. Given this, we define:

Definition 1 *A* ***hangover*** *is any performance degradation caused by a blip that persists beyond the return of a good interval.*

Hangovers impact both throughput and latency. We characterize a throughput hangover as the duration required to commit all outstanding transactions issued before the start of the next good interval. If the provided load is below the system's steady-state throughput, the system will eventually recover all throughput "lost" during the blip.[1] Delayed throughput recovery in turn increases latency, even for transactions issued *during* a good interval that follows a blip. Before these transactions can commit, they must first wait for the backlog formed during the blip to commit. Hangovers can not only cause immediate latency spikes, but also make the system more susceptible to future blips. Good intervals do not last forever, and unresolved hangovers can add up—and continue indefinitely.

In contrast, in a hangover-free system, all submitted transactions whose progress is interrupted by a blip commit instantaneously when progress resumes; further, these are the only transactions that experience a hike in latency, and for no more than the remaining duration of the blip.

Some hangovers are unavoidable, *e.g.,* those due to insufficient network bandwidth or to message delays. No consensus protocol, not even asynchronous ones can provide progress beyond the pace of the network: without data, there isn't anything to agree on! Consider, for instance, the case of a network partition that causes ten thousand 1MB transactions to accumulate; it will take at least 80 seconds, on a 1Gbit link, for them to be propagated once the partition resolves.

Other hangovers, however, are the result of suboptimal system design, where protocol logic (timeouts, commit rule, etc) introduces unnecessary delays. We refer to these hangovers are *protocol-induced.* As we describe further in §2.2, the most common type of protocol-induced hangovers for consensus is the artificial coupling of data dissemination with the protocol's ordering logic.

Thankfully, protocol-induced hangover are not inevitable and can be side-stepped through careful protocol design. Perhaps obviously, a protocol should accomplish this *without* making itself more susceptible to blips; a protocol that introduces *more blips* may implicitly introduce hangovers, or lose progress entirely.

We say that a consensus protocol is *seamless* if it behaves optimally in the face of blips. Formally:

Definition 2 *A partially synchronous system is* ***seamless*** *if (i) it experiences no protocol-induced hangovers, and (ii) it does not introduce any mechanisms*[2] *that make the protocol newly susceptible to blips.*

Seamlessness is a desirable property for any system, but especially relevant to BFT systems. A Byzantine leader can easily cause a blip: it can simply fail to propose. If this blip causes a hangover, it will continue to impact the performance of the system even after the leader has been replaced. This is clearly undesirable, especially because, after all, BFT protocols should be robust to Byzantine faults [23].

2.2 Existing Approaches

Traditional BFT protocols[22, 59] are not seamless. They tightly couple data dissemination and ordering: for each batch of transactions on which it seeks to achieve consensus, the protocol's current leader jointly disseminates the data associated with the batch and proposes how this batch should be ordered relative to other batches. Consensus is then reached on each batch independently. If the consensus logic stalls,*e.g.,* because of a Byzantine leader triggering a timeout, so does data dissemination. When a good interval resumes, the time necessary to commit (transmit, verify and order) the backlog of transactions accumulated during the blip will thus be on the order of the backlog's size.

Take for instance, the behavior of Hotstuff (HS) [59], a state of the art BFT protocol at the core of the former Diem blockchain [17]. We use a 1s (doubling) timeout (the default in many companies [6]), and trigger a ca. three second blip by simulating a single leader failure; because HS pipelines rounds and eagerly rotates leaders, a single failure can cause two timeouts to trigger (§6). After the blip ends, HS, burdened with individually disseminating and processing the transaction batches accumulated in its backlog, suffers from a hangover that is 30% longer than the original blip itself! (Figure 1).

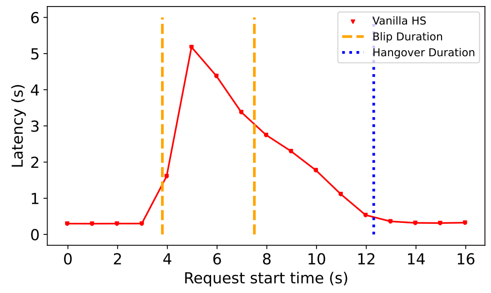

Figure 1. Latency Hangover in Hotstuff [59].

Simply proposing larger batches is not a viable option for reducing the number of batches in the backlog, as it comes at the cost of increased latency in the common case. A naive decoupling of data dissemination and consensus ordering logic (by proposing only a batch digest) [15, 22, 24] can help scale throughput, but is not robust: (*i*) malicious proposers may overwhelm the mempool, and (*ii*) replicas that are out of sync must fetch missing data (data synchronization) before voting, which is on the timeout-critical path of consensus. We demonstrate empirically in §6 that such synchronization effects are common even in absence of blips, and restrict throughput scalability.

Recent partially-synchronous DAG-based protocols [24, 52], in contrast, come *close* to seamlessness. These protocols decouple data dissemination from the consensus ordering logic,

[1]If the load is equal or greater to the throughput, then the lost committed throughput (or *goodput*) can never be recovered.
[2]Beyond the timeouts that are, of course, necessary for liveness.

and construct an asynchronously growing DAG consisting of batches of transactions chained together. The construction leverages Reliable Broadcast (RB) to generate proofs that transitively vouch for the availability of those batches and of all topologically preceding batches in the DAG. The consensus ordering logic can then commit entire backlogs with constant cost by proposing only a single batch. Commit complexity is thus *independent* of blip duration.

Current DAG protocols allow data dissemination to proceed at the pace of the network, rather than that of consensus, but do so at the price of slowing down consensus, and thus fall short of seamlessness. In particular, they still place data synchronization on the timeout-critical path (before voting), which may introduce protocol-generated blips; further, their need to recursively traverse the DAG backwards to infer all topologically preceding data may require fetching data for multiple sequential rounds, delaying commit.[3]

Besides, existing DAG protocols exhibit, during *good* intervals, up to twice the latency of traditional BFT protocols. Consensus safety is hardwired into the structure of the DAG, and establishing it requires up to four rounds of RB (each comprising three message delays)–for a total of 12 message delays.

This paper asks: is it possible to develop a low-latency and high-throughput *seamless* protocol? To answer this question, we propose Autobahn, a novel BFT protocol. Autobahn instantly commits all backlogged proposals upon returning from a blip, and without making consensus more susceptible to blips (thus achieving seamlessness).

3 Autobahn: Overview

Autobahn comprises two logically distinct layers: a horizontally scalable data dissemination layer that always makes progress at the pace of the network; and a low-latency, partially synchronous consensus layer that tries to reach agreement on snapshots of the data layer. Decoupling consensus from data dissemination is not unique to Autobahn [15, 22, 24]; its core innovation lies in more cleanly separating the responsibility of each layer while carefully orchestrating their interaction. Seamlessness requires two key properties:

- *Responsive transaction dissemination.* The data dissemination layer should proceed at the pace of the network. We say that a transaction has been successfully disseminated once it is delivered by one correct replica.
- *Streamlined Commit.* During good intervals, all successfully disseminated proposals should be committed in bounded time and with cost and latency independent of the number of transactions to commit.

Implementing a responsive data dissemination layer may appear straightforward: simply broadcast all transactions to all replicas in parallel. This naive strategy unfortunately makes it extremely challenging to implement a consensus layer that streamlines commit. The consensus layer would have to individually propose (references to) all disseminated transactions, which would either require replicas to fetch and verify locally missing transactions (data synchronization) prior to voting in consensus, or risk breaking liveness in the presence of Byzantine proposers. Achieving seamlessness and low latency instead requires injecting the right amount of coordination in the data layer's design: too little and the consensus layer cannot be designed to streamline commit; too much, and the good-case latency may spike. We find that the dissemination layer must specifically support *instant referencing*: it must allow the consensus layer to uniquely identify, and prove the availability of, the set of disseminated transactions. This must be done in constant time and space (w.r.t to blip length), and without necessarily being in possession of all data. Note that the bounded time requirement implicitly requires that there be no data synchronization on the critical (voting) path of consensus, as this could lead to timeout violations, resulting in further blips (*non-blocking sync*). Moreover, any data syncing should finish by the time consensus commits to avoid delays that may cause hangovers (*timely sync*).

These principles shape Autobahn's data dissemination and consensus layers.

Data dissemination. All replicas in Autobahn act as proposers, responsible for disseminating transactions received by clients. Each replica, in parallel, broadcasts batches of transactions (data proposals) and constructs a local chain (a *data lane*) that implicitly assigns an ordering to all of its data proposals. This structure allows us to transitively prove the availability of all data proposals in a chain in constant time, thus guaranteeing *instant referencing*. Data lanes grow independently of one another, subject only to load and resource capacity, and are agnostic to blips (guaranteeing responsiveness). Finally, Autobahn guarantees that all successfully disseminated data proposals will commit during good intervals (*reliable inclusion*); this holds true even for Byzantine proposers, and ensures that replicas cannot "wastefully" disseminate transactions without intending to commit them.

Consensus. The consensus layer exploits the structure of data lanes to, in a single shot, commit arbitrarily large data lane state. Autobahn efficiently summarizes data lane state as a cut containing only each replica's latest proposal (its *tip*), using instant referencing to uniquely identify a lane's history. Consensus itself follows a classical (latency optimal) PBFT-style coordination pattern [22, 33] consisting of two round-trips; during gracious intervals, Autobahn's *fast path* commits in only a single round-trip. After committing to a tip cut, a replica can infer the respective chain-histories of each tip and synchronize on any locally missing data (some transactions in the cut may not have been replicated to all nodes). To ensure that data synchronization doesn't stall consensus progress, Autobahn carefully constructs voting rules in the

[3]In practice, DAGs minimize worst-case synchronization by forcing (a supermajority of) replicas to advance through rounds in lock-step.

data dissemination layer to allow synchronization to complete in parallel with agreement, and in a single round-trip.

Once a replica is in possession of the data included in each tip's history data, it deterministically interleaves the n data lanes to construct a single total order.

This design allows a single consensus proposal to (*i*) reach throughput that scales in the number of replicas, (*ii*) reach agreement with cost independent of the size of the data lanes, and (*iii*) preserve the latency-optimality of traditional BFT protocols. We describe the full Autobahn protocol in §5.

4 Model

Autobahn adopts the standard assumptions of prior BFT work [22, 52, 59], positing $n = 3f + 1$ replicas, at most f of which are faulty. We consider a participant (client or replica) correct if it adheres to the protocol specification; a participant that deviates is considered faulty (or Byzantine). We make no assumptions about the number of faulty clients. We assume the existence of a strong, yet static adversary, that can corrupt and coordinate all faulty participants' actions; it cannot however, break standard cryptographic primitives. Participants communicate through reliable, authenticated, point-to-point channels. We use $\langle m \rangle_r$ to denote a message m signed by replica r. We expect that all signatures and quorums are validated, and that external data validity predicates are enforced; we omit explicit mention of this in protocol descriptions.

Autobahn operates under partial synchrony [28]: it makes no synchrony assumptions for safety, but guarantees liveness only during periods of synchrony [31].

5 Autobahn: The Protocol

To describe Autobahn more thoroughly, we first focus on the data dissemination layer (§5.1), which guarantees responsive data dissemination, even during blips. We then focus on how the consensus layer (§5.2.1) leverages the data layer's lane structure to efficiently commit arbitrarily long data lanes with cost and latency independent of their length, thus seamlessly recovering from blips.

5.1 Data dissemination

Autobahn's data dissemination layer offers the necessary structure for the consensus layer to ensure a streamlined commit. It precisely provides to consensus the three properties of instant referencing, non-blocking sync and timely-sync. Crucially, it provides no stronger guarantee, as additional properties may be subject to increased tail latency.

Lanes and Cars. In Autobahn, every replica acts as a proposer, continuously batching and disseminating incoming transactions. Each replica proposes new batches of these transactions (data proposals) at its own rate, and fully independently of other replicas — we say that each replica operates in its own *lane*, and as fast as it can.

Within its lane, each replica leverages a simple Propose and Vote message patterns: the proposer broadcasts data proposals to all other nodes, and replicas vote to acknowledge delivery. $f + 1$ vote replies constitute a Proof of Availability (PoA) that guarantees at least one correct replica is in possession of the data proposal and can forward it to others if necessary. We call this simple protocol pattern a *car* (Certification of Available Request), illustrated in Figure 2. Note that, Autobahn, unlike most DAG-BFT protocols, does not force all data proposals to go through a round of *reliable broadcast* [24, 52] that must process $n - f$ votes in order to achieve non-equivocation. Reliable, non-equivocating broadcast is not necessary in Autobahn to achieve the desired data layer properties (§5.2).

A lane is made up of a series of cars that are chained together (Fig. 3). A proposer, when proposing a new data proposal, must include a reference to its previous car's data proposal. Similarly, replicas will vote for a car at position i if and only if its proposal references a proposal for car $i-1$ that the replica has already received and voted on. This construction ensures that a successful car for block i transitively proves the availability of a car for all blocks 0 to $i - 1$ in this proposer's lane. Validating the head of the lane is thus sufficient to reference arbitrary lane state (enabling instant referencing) and confirm that it has been disseminated (necessary for non-blocking sync). Requiring replicas to always vote *in-order* further ensures that at least one correct replica is in possession of all proposals in this lane, allowing the consensus layer to (§5.2.2) to experience timely-sync.

Protocol specification. Each replica maintains a local view of all lanes. We call the latest proposal of a lane a tip; a tip is certified if its car has completed (a matching PoA exists).

Algorithm 1 Data layer cheat sheet.

```
1: car ▷ Propose & Vote pattern
2: PoA ▷ $f$+1 matching VOTES for proposal
3: pass ▷ passenger: (proposal, Option(PoA))
4: pos ▷ car position in a lane (proposal seq no)
5: lane ▷ map: [pos → passenger]
6: lanes ▷ map: [replica → lane]
7: tip ▷ last passenger in a lane ($pos = lane.length$)
```

1: P → R: Replica P broadcasts a new data proposal PROP.

P assembles a batch B of transactions, and creates a new data proposal for its lane l. It broadcasts a message PROP := $(\langle pos, B, parent \rangle_P, cert)$ where $pos := l.length+1$ (the position of the proposal in the lane), $parent := h(l.tip.prop)$ (the hash of previous lane tip), and $cert := l.tip.$PoA (the availability proof of the previous proposal).

2: R → P: Replica R processes PROP and votes.

R checks whether the received PROP is valid: it checks that (*i*) it has already voted for the parent of this proposal (*H(lanes[P][pos-1].prop) == parent)* and (*ii*) it has not voted for this lane position before. R then stores the parent's PoA received as part of the message as the latest certified tip

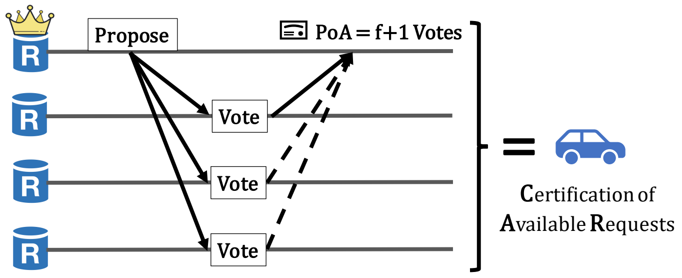

Figure 2. Car protocol pattern.

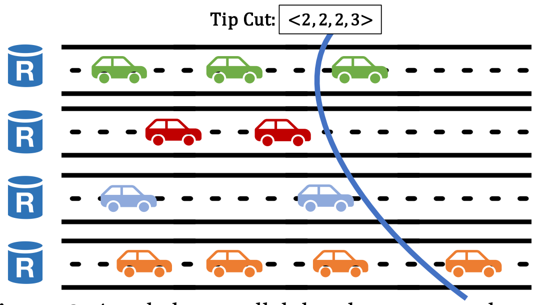

Figure 3. Autobahn parallel data lanes; example cut.

(*lanes[P][pos-1].PoA = cert*), and the current proposal as latest optimistic tip in *lanes[P][pos].prop=*Prop. Finally, it replies with a Vote := $\langle dig = h(\text{Prop}), pos\rangle_R$. If R has not yet received the parent proposal for Prop, it buffers the request and waits.

3: P → R: Replica P assembles Votes and creates a PoA.

The proposer aggregates $f+1$ distinct matching Vote messages into a PoA := $(dig, pos, \{\sigma_r\})$, which it will include in the next car; if no new batch is ready then P can choose to broadcast the PoA immediately.

Cars enforce only light (but sufficient) structure on the data layer. The number and timing of cars in each lane may differ; they depend solely on the lane owner's incoming request load and connectivity (bandwidth, point-latencies). Cars also do not preclude equivocation. While lanes governed by correct replicas will only ever consist of a single chain, Byzantine lanes may fork arbitrarily. This is by design: non-equivocation is not a property that needs to be enforced at the data layer and causes unnecessary increase in tail latency.

5.2 From Lanes to Consensus

As stated, Autobahn's data dissemination layer guarantees responsive data dissemination, instance referencing, and non-blocking, timely sync, but intentionally provides no consistency guarantee on the state observed by correct replicas. For instance, different replicas may observe inconsistent lane states (neither a prefix of another).

The role of the consensus layer in Autobahn is then to reconcile these views and to totally order all certified data proposals. To ensure seamless recovery from blips, the commit process should be *streamlined*: the consensus layer should be able to commit, in a single shot, an arbitrarily large number of certified data proposals. Unlike traditional BFT consensus protocols, that take as input a batch of transactions, a *consensus proposal* in Autobahn consists of a snapshot *cut* of all data lanes. Specifically, a consensus proposal contains a vector of n *certified* tip references, as shown in Figure 3.

This design achieves two goals: (*i*) committing a cut of all the lane states allows Autobahn to achieve consensus throughput that scales horizontally with the number of lanes (n total), while (*ii*) proposing certified lane tips allows Autobahn to commit arbitrarily large backlogs with complexity independent of its size. We discuss next how Autobahn reaches consensus on a cut (§5.2.1), and how it carefully exploits the lane structure to achieve seamlessness (§5.2.2).

5.2.1 Core consensus. Autobahn's consensus protocol follows a classic two-round (linear) PBFT-style [22] agreement pattern, augmented with a fast path that reduces latency to a single round (3md) in gracious intervals. The protocol progresses in a series of slots: each slot s is assigned to a leader that can start proposing a new lane cut once slot $s-1$ commits. Within each slot, Autobahn follows a classical view based structure [22] – when a slot's consensus instance fails to make timely progress, replicas revolt, triggering a *View Change* and electing a new leader.

Each view consists of two phases: *Prepare* and *Confirm* (5md total). The Prepare phase tries to achieve agreement within a view (non-equivocation) while the Confirm phase enforces durability across views (Fig. 4). During gracious intervals the *Confirm* phase can be omitted (Fast Path), reducing consensus latency down to a single phase (3md).

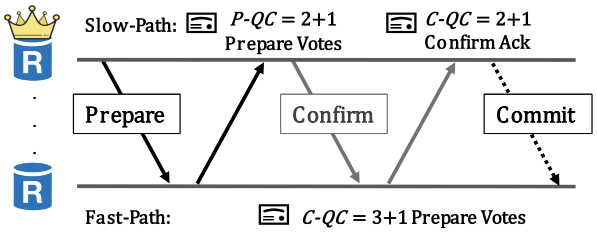

Figure 4. Core Consensus coordination pattern.

For simplicity, we first present our protocol for a single slot. We assume that all replicas start in view $v = 0$. We explain Autobahn's view change logic in §5.3, and discuss in §5.4 how to orchestrate agreement for slots in parallel to reduce end-to-end latency. We defer formal proofs of correctness and seamlessness to our supplementary material.

Prepare Phase (P1).

1: L → R: Leader L broadcasts Prepare.

A leader L for slot s begins processing (view $v = 0$) once it has received a ticket $T :=$ CommitQC$_{s-1}$ (Alg.2), and "sufficiently" many new data proposals to propose a cut that advances the frontier committed in $s-1$; we coin this requirement *lane coverage*, and defer discussion of it to §5.2.3.

The leader broadcasts a message Prepare := $(\langle P\rangle_L, T)$ that contains the *consensus proposal* P as well as the necessary

Algorithm 2 Consensus layer cheat sheet.

```
1: $CommitQC_s$ ▷ Commit proof for slot $s$
2: $TC_{s,v}$ ▷ View change proof for slot $s$, view $v$
3: Ticket $T_{s,v}$ ▷ $\text{COMMITQC}_{s-1}$ or $\text{TC}_{s,v-1}$
4: Proposal $P_{s,v}$ ▷ Proposed lane cut. Requires $T_{s,v}$
5: $prop$ ▷ map: slot ⟶ $P_{s,v}$
6: $conf$ ▷ map: slot ⟶ $\text{PREPAREQC}_{s,v}$
7: $last\text{-}commit$ ▷ map: lane ⟶ pos
8: $log$ ▷ total order of all user requests
```

ticket T. The proposal P itself consists of the slot number s, the view v, and a cut of the latest certified lane tips observed by L: P := *<s, v, [lanes[1].tip.PoA, ..., lanes[n].tip.PoA]>*.

2: R ⟶ L: Replica R processes PREPARE and votes.

R first confirms the PREPARE's validity: it checks that (*i*) the ticket is for the right leader/slot), and (*ii*) R has not yet voted for this slot. R then stores a copy of the proposal ($prop[s] = P$), and responds with a PREP-VOTE := $\langle dig = h(P)\rangle_R$. Finally, if it has not locally received all tips (and their histories) included in P, R begins to asynchronously fetch all missing data (§5.2.2). This is possible thanks to the data dissemination layer's non-blocking sync guarantee: R knows that the data will be available, and thus need not sync before voting.

3: L: Leader L assembles quorum of PREP-VOTE messages.

L waits for at least $n - f = 2f + 1$ matching PREP-VOTE messages and aggregates them into a *Prepare Quorum Certificate (QC)* PREPAREQC := (s, v, dig, {PREP-VOTE}). This ensures agreement *within* a view. No two PREPAREQCs with different dig can exist as all PREPAREQCs must intersect in at least one correct node, who will never vote more than once per view.

Fast Path. In the general case, the Prepare phase is insufficient to ensure that a particular proposal will be persisted across views. During a view change, a leader may not see sufficiently many PREP-VOTE messages to repropose this set of operations. In gracious intervals, however, the leader L can receive n such votes by waiting for a small timeout beyond the first $n - f$. A quorum of n votes directly guarantees durability across views (any subsequent leader will observe at least one PREP-VOTE for this proposal). No second phase is thus necessary. In this scenario, L upgrades the QC into a *fast commit quorum certificate* and proceeds directly to the Commit step, skipping the confirm phase): it broadcasts the COMMITQC, and commits locally.

4 (Fast): L ⟶ R: L upgrades to COMMITQC and commits.

Confirm Phase (P2). In the slow path, the leader L instead moves on to the Confirm phase.

1: L ⟶ R: Leader L broadcasts CONFIRM.

L forwards the $\text{PREPAREQC}_{s,v}$ by broadcasting a message CONFIRM := $\langle PrepareQC_{s,v}\rangle$ to all replicas.

2: R ⟶ L: Replica R receives and acknowledges CONFIRM.

R returns an acknowledgment CONFIRM-ACK := $\langle dig\rangle_R$, and buffers the PREPAREQC locally ($conf[s] = \text{PREPAREQC}$) as it may have to include this message in a later view change.

3: L ⟶ R: Leader L assembles COMMITQC and commits.

L aggregates $2f + 1$ matching CONFIRM-ACK messages into a *Commit Quorum Certificate* COMMITQC := (s, v, dig, {CONFIRM-ACK}), and broadcasts it to all replicas.

5.2.2 Processing committed cuts.

Recall that consensus proposals in Autobahn are cuts of data proposal, rather than simple batches of transactions. Upon committing, the replicas must then establish a (consistent) total order of proposals to correctly execute transactions. To help with this process, each replica maintains a log of all ordered transactions, and a map *last-commit* which tracks, for each lane, the position of the latest committed data proposal (Alg. 2).

Seamless data synchronization. To process a cut, a replica must first ensure that, for each lane, it is in possession of all data proposals that are transitively referenced by the respective tip (*tip history* for short). Otherwise, it must first acquire (*synchronize*) the missing data; in the worst-case, this may be the entire history. During a partial partition, for instance, lanes may grow without all correct replicas being aware, requiring them to later synchronize on missing data.

Autobahn exploits the lane structure to synchronize in parallel with consensus, and in just a single message exchange, no matter the size of the history. Firstly, certified tips allow replicas to transitively prove the availability of a lane history *without* directly observing the history. This allows replicas with locally inconsistent lane states to vote for consensus proposals without blocking (*non-blocking sync*), and moves synchronization off the timeout-critical path. If Autobahn employed only best-effort (non-certified) lanes, replicas would have to synchronize *before* voting on a consensus proposal to assert whether a tip (and its history) indeed exists. Such blocking increases the risk of timeouts undermining seamlessness. Second, in-order voting in the data layer enables *timely sync* by guaranteeing that there exists at least one correct replica that is in possession of *all* data proposals in a certified tip's history.

To synchronize, a replica requests (from the replicas that voted to certify the tip) to SYNC all proposals in the tip's history with positions greater than its locally last committed lane proposal (up to $last\text{-}commit[tip.lane].pos + 1$). By design, lane positions must be gap free, and proposals chained together; the requesting replica thus knows (*i*) whether a SYNC-REPLY contains the correct number of requested proposals, and (*ii*) whether the received proposals form a correct (suffix of the) history. By correctness of consensus, all correct replicas agree on the last committed lane proposal, and thus synchronize on the same suffix.

Creating a Total Order. Once a replica has fully synchronized the cut ($\langle tips \rangle$) committed in slot s, and all previous slots have been committed and processed, it tries to establish a total order across all "new" data proposals subsumed by the lane tips. A replica first identifies, for each lane l, the oldest ancestor $start_l$ of $tips[l]$ with a position exceeding $last_l := last\text{-}commit[l]$ (i.e. $start_l.pos == last_l + 1$). Next, a replica updates the latest committed position for each lane ($last\text{-}commit[l] = tips[l].pos$), and appends to the log all new proposals (lane proposals from $start_l$ to $tips[l]$) using some deterministic zipping function.

We note that the proposal committed at $last_l$ may not (and need not) be the parent of $start_l$; for instance, the replica governing l may have equivocated and sent multiple proposals for the same position. This does not affect safety as we simply ignore and garbage collect ancestors of $start_l$.

5.2.3 Governing Progress. Correct leaders in Autobahn only initiate a new consensus proposal once data lanes have made "ample" progress. We coin this requirement *lane coverage* – it is a tunable hyperparameter that governs the pace of consensus. Intuitively, there is no need for consensus if there is nothing new to agree on! Choosing the threshold above which there are sufficiently many new data proposals to agree on requires balancing latency, efficiently and fairness. A system with heterogeneous lane capacities, for instance, may opt to minimize latency by starting consensus for every new car; a resource conscious system, in contrast, may opt to wait for multiple new data proposals to be certified. In Autobahn, we set coverage, as default, to require at least $n-f$ new tips. This ensures that at least half of the included tips belong to correct replicas.

Reliable Inclusion. Importantly, lane coverage does not cause Autobahn to discriminate against slow lanes. Unlike contemporary DAG-BFT protocols [24, 52] that are driven by the fastest $n-f$ replicas, and may ignore proposals of the f slowest, proposal cuts in Autobahn include *all* n lanes. Since correct replicas' lanes never fork, all of their proposals are guaranteed to commit in a timely fashion (as soon as a correct leader receives a *PoA* that subsumes them).

Notably, Autobahn's consensus layer ensures that *all* certified proposals–including those of Byzantine replicas–are reliably committed. Since lane growth requires that at least one correct replica be in possession of a *PoA* for history of the latest car, Byzantine proposers cannot continue to disseminate transactions without intending to commit them; thus effectively bounding the maximum amount of "waste" that correct replicas may receive from Byzantine actors. We defer a formal analysis to our supplementary material (A.4).

5.3 View change

In the presence of a faulty leader, progress may stall. As is standard, Autobahn relies on timeouts in these situations to elect a new leader: replicas that fail to see progress revolt, and create a special *Timeout Certificate* (TC) ticket to trigger a *View Change* [22, 43].[4] Each view v (for a slot s) is mapped to one designated leader (e.g. using round-robin). To prove its tenure a leader carries a ticket $T_{s,v}$, corresponding to either (in $v = 0$) a COMMITQC$_{s-1}$,[5] or (in $v > 0$) a quorum of mutineers $TC_{s,v-1}$ from view $v-1$.

PREPARE and CONFIRM messages contain the view v associated with the sending leader's tenure. Each replica maintains a *current-view*$_s$, and ignores all messages with smaller views; if it receives a valid message in a higher view v', it buffers it locally, and reprocesses it once it reaches view v'. Per view, a replica sends at most one PREP-VOTE and CONFIRM-ACK.

A replica starts a timer for view v upon first observing a ticket $T_{s,v}$. Seamlessness alleviates the need to set timeouts aggressively to respond quickly to failures. Autobahn favors conservative timers for smooth progress in good intervals as the data layer continues progressing even during blips. The replica cancels the timer for v upon locally committing slot s or observing a ticket $T_{s,v+1}$. Upon timing out, it broadcasts a complaint message TIMEOUT and includes any locally observed proposals that could have committed.

1: R → R: Replica R broadcasts TIMEOUT.

A replica R, whose timer t_v expires, broadcasts a message TIMEOUT $:= \langle s, v, highQC, highProp \rangle_R$. *highQC* and *highProp* are, respectively, the PREPAREQC$_{s,v'}$ ($conf[s]$) and proposal$_{s,v''}$ ($prop[s]$) with the highest views locally observed by R.

2: R Replica R forms TC and advances view.

A replica R accepts a TIMEOUT message if it has not yet advanced to a higher view ($current\text{-}view_R \leq$ TIMEOUT$.v$) or already received a COMMITQC for that slot. In the latter case, it simply forwards the COMMITQC to the sender of TIMEOUT.

Replica R joins the mutiny once it has accepted at least $f+1$ TIMEOUT messages (indicating that at least one correct node has failed to see progress) and broadcasts its own TIMEOUT message. This ensures that if one correct replica locally forms a TC, all correct replicas eventually will. Upon accepting $2f+1$ distinct timeouts for v a replica assembles a *Timeout Certificate* TC $:= (s, v, \{\text{TIMEOUT}\})$, and advances its local view $current\text{-}view = v+1$ and starts a timer for $v+1$. If R is the leader for $v+1$, it additionally begins a new Prepare phase, using the TC as ticket.

A leader L that uses a TC as ticket must recover the latest proposal that *could have* been committed at some correct replica. To do so, the leader replica chooses, as *winning proposal*, the greater of (*i*) the highest *highQC* contained in TC, and (*ii*) the highest proposal present $f+1$ times in TC; in a tie, precedence is given to the *highQC*. This two-pronged structure is mandated by the existence of both a fast path and a slow path in Autobahn. If a proposal appears $f+1$

[4] We adopt and modify the View Synchronizer proposed in Jolteon [33].
[5] The view associated with a COMMITQC ticket is immaterial; by safety of consensus, all COMMITQC's in a slot s must have the same value.

times in TC, then this proposal may have gone fast-path as going fast-path requires n votes. Any later $n-f$ quorums is guaranteed to see at least $f+1$ such votes. Similarly, if a CommitQC formed on the slow path (from 2f+1 Confirm-Acks of a PrepareQC), at least one such PrepareQC will be included in the TC (by quorum intersection). Finally, the leader reproposes the chosen consensus proposal.

Put together, this logic guarantees that if in some view v a CommitQC was formed for a proposal P, all consecutive views will only re-propose P.

3: L → R: Leader L sends Prepare_{v+1}.

Finally, the leader L broadcasts a message Prepare $:= (\langle v+1, tips\rangle_L), \text{TC})$, where *tips* is the winning proposal derived from TC; or $L's$ local certified lane cut if no winning proposal exists. Replicas that receive such a Prepare message validate TC, and the correctness of *tips*, and otherwise proceed with Prepare as usual.

5.4 Parallel Multi-Slot Agreement

Until now, we have described Autobahn as proceeding through a series of sequential slot instances. Unfortunately, lane proposals that narrowly "miss the bus" and are not included in the current slots proposed consensus cut may, in the worst case, experience the sequential latency of up to two consensus instances (the latency of the consensus instance they missed in addition to the one necessary to commit).

To circumvent sequential wait-times, Autobahn, inspired by PBFT [22], allows the next slot instance to begin in parallel with the current one, without waiting for it to commit. We modify a slot ticket for s to no longer require a CommitQC_{s-1}: the leader of slot s can begin consensus for slot s as soon as it receives the first Prepare_{s-1} message (the view is immaterial), and has observed sufficiently many new tips to satisfy lane coverage.

Parallel consensus can avoid sequential delays entirely when instantiated with a stable leader (the same leader proposes multiple slots). When instantiated with rotating leaders, there remains a single message delay: the time to receive Prepare_{s-1}. We adjust Autobahn as follows.

Managing concurrent instances. Autobahn maintains independent state for all ongoing consensus slot instances. Each instance has its own consensus messages, and is agnostic to all other processing. Once a slot has committed, it waits until all preceding slot instances have already executed to execute itself. Similarly, when ordering a slots proposal (a cut of tips), Autobahn replicas identify and order only proposals that are new. Old tip positions are simply ignored. This filtering step is necessary as consecutive slots may propose non-monotonic cuts.

Adjusting view synchronization. To account for parallel, overlapping instances, replicas no longer begin a timer for slot s upon reception of a CommitQC_{s-1}, but upon reception of the first $\text{Prepare}_{(s-1,v)}$ message. Moreover, we modify the leader election scheme to ensure that different slots follow a different leader election schedule. Specifically, we offset each election schedule by f. Without this modification, the worst case number of sequential view changes to commit k successive slots would be $\frac{k*(f+1)}{2}$, as each slot s would have to rotate through the same faulty leaders to generate a ticket for $s+1$.

Bounding parallel instances During gracious intervals, the number of concurrent consensus instances will be small. As each consensus instance incurs less than five message delays, there should, in practice, be no more than three to four concurrent instances at a time. During blips, however, this number can grow arbitrarily large: new instances may keep starting (as they only need to observe a relevant Prepare message). Continuously starting (but not finishing) parallel consensus instances is wasteful; it would be preferable to wait to enter a good interval, as a single new cut can subsume all past proposals. Autobahn thus bounds the maximum number of ongoing consensus instances to some k by (re-) introducing CommitQC tickets. To begin slot instance s, one must include a ticket CommitQC_{s-k}. When committing slot s we can garbage collect the CommitQC for slot $s-k$ as $CommitQC_s$ transitively certifies commitment for slot $s-k$.

Discussion. Autobahn's parallel proposals departs from modern chained-BFT protocols [33, 35, 59] which pipeline consensus phases (e.g. piggyback Prepare_{s+1} on Confirm_s). We decide against this design for three reasons: (*i*) pipelining phases is not latency optimal as it requires at least two message delays between proposals, (*ii*) it introduces liveness concerns that require additional logic to circumvent [35], and (*iii*) it artificially couples coverage between successive slots, i.e. coverage for slot 2 affects progress of slot 1.

5.5 Optimizations

Finally, we discuss a number of optimizations, that, while not central to Autobahn's ethos, can improve performance.

5.5.1 Further reducing latency.

By default, Autobahn consensus pessimistically only proposes *certified* tips. This allows voting in the consensus layer to proceed without blocking, which is crucial for seamlessness (§5.2). Assembling and sharing certificates, however, imposes a latency overhead of three message exchanges in addition to the unavoidable consensus latency. This results in a best-case end-to-end latency of six message delays on the fast path, and eight otherwise. Autobahn employs two optimizations to reduce the latency between the dissemination of a data proposal and it being proposed as part of consensus (inclusion latency).

Leader tips. First, we allow a leader to include a reference to the latest proposal (*dig,pos*) it has broadcast as its own lane's tip, even as the tip has not yet been certified. If a Byzantine leader intentionally does not disseminate data, yet includes the proposal as a tip, it is primarily hurting itself. Correct nodes will, at most, make a single extra data sync request before requesting a view change. Moreover, if the

Byz leader ever wants to propose additional data, it will have to backfill the data for this proposal (in-order voting).

Optimistic tips. Second, Autobahn may choose to relax seamlessness and allow leaders to optimistically propose non-certified tip references (dig, pos) for lanes that have, historically, remained reliable. In gracious intervals, this optimization lowers inclusion latency to a single message exchange, which is optimal. However, this comes at the risk of incurring blocking synchronization and triggering additional view changes. Replicas that have not received a tip locally hold off from voting, and request the tip from the leader directly. We note, that critical-path synchronization is only necessary for the tip itself (and thus has constant cost) – by design, all ancestors must be certified, and thus can be synchronized asynchronously.

5.5.2 Other optional modifications. Autobahn can, if desired, be further augmented with several standard techniques. We discuss them briefly for completeness.

Signature aggregation. Autobahn, like others [36, 52, 59] can be instantiated with threshold (TH) signatures [18, 50] that aggregate matching vote signature of any Quorum Certificate (PoA, PrepareQC, CommitQC) into a single signature. This reduces Autobahn's complexity per car and consensus instance (during good intervals) to $O(n)$; view changes require $O(n^2)$ messages [59].

All-to-all communication. Autobahn follows a linear-PBFT communication pattern [36] that allows it to achieve linear complexity (during good intervals) when instantiated with aggregate signatures. Autobahn can, of course, also be instantiated with all-to-all communication for better latency (but no linearity). In this regime, consensus consists of only 3 message exchanges (2md on FastPath): replicas broadcast Prepare-Votes and Confirm-Acks, and respectively assemble $PrepareQC$s and $ConfirmQC$s locally.

6 Evaluation

Our evaluation seeks to answer three questions:

1. **Performance**: How well does Autobahn perform in gracious intervals? (§6.1)
2. **Scaling** How well does it scale as we increase n? (§6.1)
3. **Blip Tolerance**: Does Autobahn make good on its promise of seamlessness? (§6.2)

We implement a prototype of Autobahn in Rust, starting from the open source implementation of Narwhal and Bullshark [30]. We use Tokio's TCP [8] for networking, and RocksDB [7] for persistent storage. Finally, we use ed25519-dalek [1] signatures for authentication.

Baseline systems. We compare against open-source prototypes of Hotstuff [59] and Bullshark [52, 53] which, respectively, represent today's most popular traditional and DAG-based BFT protocol. We use the "batched" Hotstuff (BatchedHS) prototype [29] that simulates an upper bound on achievable performance by naively separating dissemination and consensus. Replicas optimistically stream batches, and consensus leaders propose hashes of any received batches. This design is not robust in real settings: (*i*) Byzantine replicas and blips may exhaust memory. There is no guarantee that batches will be committed (no *reliable inclusion*) requiring the system to eventually drop transactions. (*ii*) Replicas must synchronize on missing batches before voting, which can cause blips (no *non-blocking, timely sync*). We also evaluate a standard version of Hotstuff (VanillaHS) in which we modify BatchedHS to send batches only alongside consensus proposals of the issuing replica.

RTT	us-east1	us-east5	us-west1	us-west4
us-east1	0.5	19	64	55
us-east5	19	0.5	50	57
us-west1	64	50	0.5	28
us-west4	55	57	28	0.5

Table 1. RTTs between regions (ms)

Both Bullshark [30] and Autobahn adopt the horizontally scalable worker layer proposed by Narwhal [24]. For a fair comparison with Hotstuff we run only with a single, co-located worker; in this setup the Reliable Broadcast used by workers is unnecessary, so we remove it to save latency.

Experimental setup. We evaluate all systems on Google Cloud Platform (GCP), using an intra-US configuration with nodes evenly distributed in `us-west1`, `us-west4`, `us-east1` and `us-east5`. We summarize RTTs between regions in Table 1. We use machine type `t2d-standard-16`[4] with 20GB of SSD. A client machine (co-located in the same region as the replica) issues a constant stream of transactions (tx), consisting of 512 random bytes [24, 52]. We measure latency as the time between transaction arrival at a replica and the time it is ready to execute; throughput is measured as execution-ready transactions per second. Each experiments runs for 60 seconds. We set a batch size of 500KB (1000 transactions) across all experiments and systems, but allow consensus proposals to include/reference more than one batch if available; this *mini-batching* design [29, 30] allows replicas to organically reach larger effective batch sizes with reduced latency trade-off. All systems are run with rotating leaders; we set view timers to 1s.

6.1 Performance in ideal conditions

Figure 5 shows the performance in a fault-free, synchronous setting, using $n = 4$ nodes. Autobahn's throughput matches that of Bullshark (ca. 234k tx/s), but reduces latency by a factor of 2.1x (from 592ms to 280ms), edging out even BatchedHS (333ms) and VanillaHS (365ms). Hotstuff and Bullshark require, respectively 7md and up to 12 md (10.5 average) to commit a tx, while Autobahn reduces coordination to 4md on the Fast Path, and 6md otherwise (with optimistic tips). Both Autobahn and Bullshark significantly outperform both Hotstuff variants for throughput by a factor

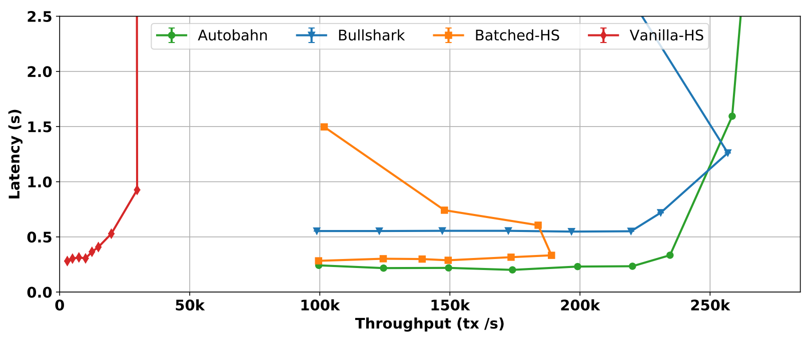

Figure 5. Throughput and Latency under increasing load.

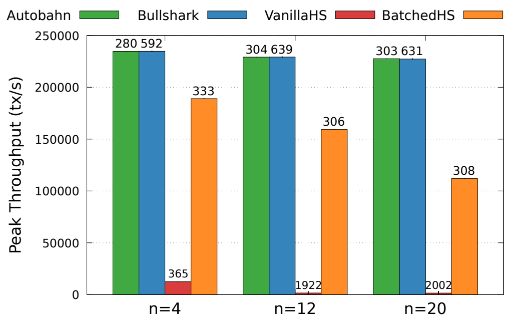

Figure 6. Peak throughput for varying n. Numbers atop bars show measured latency (ms).

of 1.23x (BatchedHS) and 15.6x (VanillaHS), and are bottlenecked on the cost of deserializing and storing data on disk. VanillaHS offers very low throughput (ca. 15k tx/s before latency noticeably rises) as it requires that replicas disseminate their own batches alongside consensus proposals. The system quickly bottlenecks on the (network and processing) bandwidth of one broadcast. BatchedHS scales significantly better (189k tx/s) than VanillaHS as it amortizes broadcasting cost. At high load, however, we find that it incurs a significant amount of data synchronization (resulting at times even in view changes) causing it to bottleneck; this corroborates findings in Narwhal [24] and demonstrates that this design is not robust in practice.

Fast Path/Optimistic Tips. To better understand the performance benefits of the fast path and optimistic tip optimizations, we evaluate Autobahn with the fast path deactivated and without the optimistic tip optimization. We omit the graph for space constraints. We observe a 40ms increase in latency when forced to always go "slow". This increase is smaller than a cross-country RT: the fast path incurs higher tail latency due to the larger CommitQC ($|n|$) and because the leader ends up contacting non-local replicas, including those located across coasts. Similarly, we record an additional 33ms increase when proposing only certified tips. This follows directly from the need to contact $f+1$ replicas to form a PoA.

Scaling number of replicas. Figure 6 shows the peak throughput achievable as we increase the number of replicas. Both Autobahn and Bullshark scale gracefully as n increases; throughput remains bottlenecked on data processing, while latency is unaffected. In our setup, the total load remains constant, but each replica proposes a smaller fraction as n grows.[6] BatchedHS, in contrast, experiences a sizeable throughput degradation with growing n (-16% at $n = 12$, and -41% at $n = 20$) as replicas are increasingly likely to be out of sync and thus are forced to perform additional data synchronization. Finally, we observe that VanillaHS suffers a significant throughput-latency tradeoff: because each replica proposes unique batches, and batches are only proposed when a replica becomes the leader, latency grows proportionally to n. We highlight this tension by bounding latency to be at most 2s. In this setup, VanillaHS throughput diminishes sharply as n rises; it drops from 15k to only 1.5k tx/s as latency surges rapidly. VanillaHS is 1.3 times slower than Autobahn at n=4, but over 6.6 times slower at n=20.

[6]If replicas were to instead propose at constant rates, throughput would increase with n (up to the data processing bottleneck).

6.2 Operating in *partial* synchrony

The previous section highlighted Autobahn's good performance in the failure-free scenario: Autobahn achieves the best of both worlds. It matches Bullshark's throughput while preserving Hotstuff's low latency. We now investigate whether Autobahn remains robust to blips. We use $n=4$.

Faulty Leader Blips. We first consider blips that result from a single leader failure. Bullshark and BatchedHS exhibit the same blip behavior as Autobahn, and we thus omit it for clarity. Figure 7 plots latency over time. Data points are averages over second-long windows. For the first two blips, we use a standard timeout of 1s [6]. Hotstuff's pipelined, rotating leader design, can, in the presence of a faulty leader, trigger not one but two timeouts. Because the votes for a proposal issued by a leader $L1$ are eagerly forwarded only to $L2$, a single failure of $L2$ can cause two timeouts to trigger: one for its preceding proposal, and one for its own. When instantiated with stable leaders, it can only generate a single blip. We thus consider both scenarios (respectively called Dbl, and 1s). VanillaHS (under 15k tx/s load) suffers from hangovers that persist for respectively 1.6x, 1.3x, and 1.2x of the blip duration beyond the good interval resuming. The system remains bottlenecked by data dissemination and delivery when trying to work off the transaction backlog. For the Dbl (3s) blip, for instance, the latency penalty not only exceeds the blip duration, for a maximum transaction latency of 6.2s, but remains as high as 1.3s after three seconds post-blip, and 700ms at four seconds post-blip (respectively 3.5x and 1.9x increase over the steady state latency of 365 ms). Autobahn (under 220k tx/s load), in contrast, immediately recovers from the hangover. This behavior continues to hold true for larger timeout values. For a timeout as high 5s (still realistic, Diem, Facebook's former blockchain used a timeout of 30s [6, 17]), VanillaHS's hangover persisted for almost 7 seconds after the blip ended.

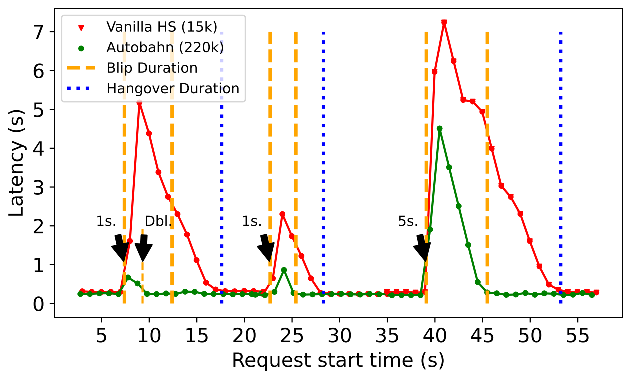

Figure 7. Leader failures.

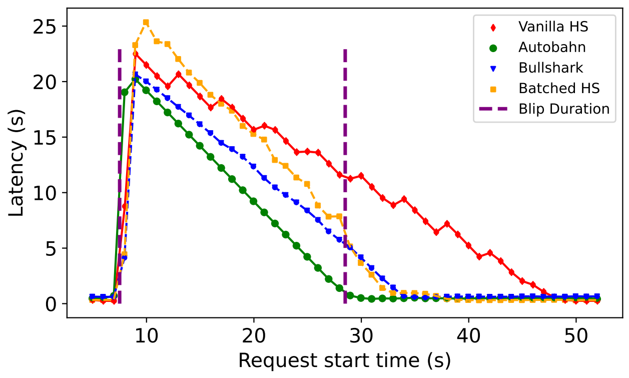

Figure 8. Partial partition.

Partition Blips. Next, we consider a blip caused by a temporary network partition (20s) that isolates replicas into two halves (Fig. 8). The load is 15k tx/s (the max for VanillaHS). Autobahn experiences only a small hangover because it continues to disseminate data (replicas can reach $f+1$ other replicas, enough to grow their lanes). Once the partition resolves, a single slot instantly commits the entire lane backlog, and replicas identify and request (in a single step) all missing data. It takes, respectively, about 1s to transfer and process all missing data (we are bottlenecked by bandwidth and delivery); this illustrates an unavoidable, non protocol-induced hangover. Transactions submitted during the partition period experience a latency penalty close to the remaining blip duration. Bullshark (ca. 6s) and BatchedHS (ca. 8s) require slightly longer to recover. BatchedHS optimistically continues to disseminate during the partition, but upon resolution must enforce a cap on mini-batch references per proposal to avoid excessive synchronization on the timeout-critical path. Bullshark cannot continue to disseminate during the partition because it must reach $2f+1$ nodes to advance the DAG; after it resolves, however, its efficient dissemination can quickly recover the backlog. VanillaHS, in contrast, exhibits a large hangover, directly proportional to the blip duration.

7 Related Work

Autobahn draws inspiration from several existing works. Scalog [26] demonstrates how to separate data dissemination from ordering to scale throughput at low latencies (for crash failures). Narwhal [24] discusses how to implement a scalable and BFT robust asynchronous broadcast layer, a core tenet for achieving seamlessness. Autobahn's consensus itself is closely rooted in traditional, latency-optimal BFT-protocols, most notably PBFT's core consensus logic and parallelism [22], Zyzzya's fast path [36, 43], and Aardvark's doctrine of robustness [23]. Autobahn's architecture, by design, isn't tied to a specific consensus mechanism, and thus can easily accommodate future algorithmic improvements.

Traditional BFT protocols such as PBFT [22], Hotsuff [59], and their respective variants [36, 43][33, 40] optimize for good intervals: they minimize latency in the common case, but guarantee neither progress during periods of asynchrony, nor resilience to hangovers. Multi-log frameworks [38, 54, 55] partition the request space across multiple proposers to sidestep a perceived leader bottleneck, and intertwine the sharded logs to arrive at a single, total order. They inherit both the good and the bad of their BFT building blocks: consensus has low latency, but remains prone to hangovers.

DAG-BFT, as previously discussed [16, 24, 41, 46, 52] originates in the asynchronous model and enjoy both excellent throughput and network resilience; recent designs [42, 51, 53] make use of partial synchrony to improve consensus latency. These (certified) DAGs are close to achieving seamlessness (data synchronization on the timeout-critical path aside), but require several rounds of Reliable Broadcast (RB) communication. Bullshark [53], the state of the art, requires 12md to commit (9md when optimized by Shoal [51]). Recent work proposes (but does not empirically evaluate) uncertified DAGs [42, 45]. These designs aim to improve latency by replacing the RB in each round with best-effort-broadcast (BEB), but forgo seamlessness because they must synchronize (possibly recursively) on missing data before voting.

DAG vs Lanes. Lanes in Autobahn loosely resemble the DAG structure, but minimize quorum sizes and eschew rigid dependencies across proposers. Notably, Autobahn proposes a cut of all n replica lanes; in contrast, DAGs can reliably advance proposals simultaneously from only $n-f$ replicas, and are thus subject to ignoring some proposals [52].

Asynchronous BFT consensus. Asynchronous BFT [12, 24, 32, 33, 37, 44, 47] protocols sidestep the FLP impossibility result [31] and guarantee liveness even during asynchrony by introducing randomization. Unfortunately, these protocols require expensive cryptography and tens of rounds of message exchanges, resulting in impractical latency.

8 Conclusion

This work presents Autobahn, a novel partially synchronous BFT consensus protocol that is actually robust *partial synchrony*. Autobahn achieves seamlessness, while simultaneously matching the throughput of DAG-based BFT protocols and the latency of traditional BFT protocols. Drive safe!

References

[1] [n. d.]. Dalek elliptic curve cryptography. https://github.com/dalek-cryptography/ed25519-dalek, last accessed on 04/10/24.

[2] [n. d.]. Digital Euro. https://www.ecb.europa.eu/euro/digital_euro/html/index.en.html, last accessed on 04/10/24.

[3] [n. d.]. Digital euro explained: Is the digital euro based on blockchain? https://cointelegraph.com/learn/the-digital-euro-project-how-will-it-impact-the-current-financial-system, last accessed on 04/10/24.

[4] [n. d.]. Google Cloud Platform (GCP) - general purpose machines. https://cloud.google.com/compute/docs/general-purpose-machines#t2d_machines, last accessed on 04/10/24.

[5] [n. d.]. Mysten Labs. https://mystenlabs.com/, last accessed on 04/10/24.

[6] [n. d.]. Private communications with researchers at two of the leading DAG-based blockchain companies (Aptos, Mysten), and formerly of Facebook Novi. March 2024.

[7] [n. d.]. RocksDB, version 0.16.0. https://rocksdb.org/, last accessed on 04/10/24.

[8] [n. d.]. Tokio, version 1.5.0. https://tokio.rs/, last accessed on 04/10/24.

[9] 04/2024. Azure Confidential Ledger. https://azure.microsoft.com/en-gb/products/azure-confidential-ledger, last accessed on 04/10/24.

[10] 04/2024. Confidential Consortium Framework, Microsoft. https://ccf.microsoft.com/, last accessed on 04/10/24.

[11] Ittai Abraham, Naama Ben-David, and Sravya Yandamuri. 2022. Efficient and Adaptively Secure Asynchronous Binary Agreement via Binding Crusader Agreement. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*. 381–391.

[12] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. 2019. Asymptotically Optimal Validated Asynchronous Byzantine Agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (Toronto ON, Canada) *(PODC '19)*. Association for Computing Machinery, New York, NY, USA, 337–346. https://doi.org/10.1145/3293611.3331612

[13] Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. 2021. Good-Case Latency of Byzantine Broadcast: A Complete Categorization. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing* (Virtual Event, Italy) *(PODC'21)*. Association for Computing Machinery, New York, NY, USA, 331–341. https://doi.org/10.1145/3465084.3467899

[14] Ittai Abraham and Alexander Spiegelman. 2022. Provable Broadcast. https://decentralizedthoughts.github.io/2022-09-10-provable-broadcast/, last accessed on 04/10/24.

[15] Vivek Bagaria, Sreeram Kannan, David Tse, Giulia Fanti, and Pramod Viswanath. 2019. Prism: Deconstructing the blockchain to approach physical limits. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 585–602.

[16] Leemon Baird. 2016. The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance. *Swirlds Tech Reports SWIRLDS-TR-2016-01, Tech. Rep* 34 (2016), 9–11.

[17] Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, François Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, and Alberto Sonnino. 2019. State machine replication in the Libra Blockchain. *The Libra Association Technical Report* (2019).

[18] Alexandra Boldyreva. 2002. Threshold signatures, multisignatures and blind signatures based on the gap-Diffie-Hellman-group signature scheme. In *International Workshop on Public Key Cryptography*. Springer, 31–46.

[19] Gabriel Bracha. 1987. Asynchronous Byzantine agreement protocols. *Information and Computation* 75, 2 (1987), 130–143.

[20] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. 2001. Secure and efficient asynchronous broadcast protocols. In *Annual International Cryptology Conference*. Springer, 524–541.

[21] Christian Cachin, Klaus Kursawe, and Victor Shoup. 2000. Random oracles in constantipole: practical asynchronous byzantine agreement using cryptography. In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*. 123–132.

[22] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (New Orleans, Louisiana, USA) *(OSDI '99)*. USENIX Association, USA, 173–186.

[23] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. 2009. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation* (Boston, Massachusetts) *(NSDI'09)*. USENIX Association, USA, 153–168.

[24] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. 2022. Narwhal and Tusk: a DAG-based mempool and efficient BFT consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 34–50.

[25] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.

[26] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert Van Renesse. 2020. Scalog: Seamless reconfiguration and total order in a scalable shared log. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 325–338.

[27] Sisi Duan, Michael K. Reiter, and Haibin Zhang. 2018. BEAT: Asynchronous BFT Made Practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2028–2041. https://doi.org/10.1145/3243734.3243812

[28] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)* 35, 2 (1988), 288–323.

[29] Novi Facebook Research. [n.d.]. Hotstuff Implementation. https://github.com/asonnino/hotstuff/commit/d771d4868db301bcb5e3deaa915b5017220463f6, last accessed on 04/10/24.

[30] Novi Facebook Research. [n.d.]. Narwahl and Bullshark implementation. https://github.com/asonnino/narwhal, last accessed on 04/10/24.

[31] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. 1985. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)* 32, 2 (1985), 374–382.

[32] Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. 2022. Dumbo-ng: Fast asynchronous bft consensus with throughput-oblivious latency. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1187–1201.

[33] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. 2022. Jolteon and Ditto: Network-Adaptive Efficient Consensus with Asynchronous Fallback. In *Financial Cryptography and Data Security: 26th International Conference, FC 2022, Grenada, May 2–6, 2022, Revised Selected Papers* (Grenada, Grenada). Springer-Verlag, Berlin, Heidelberg, 296–315. https://doi.org/10.1007/978-3-031-18283-9_14

[34] Marc Frei Fabio Streun Lefteris Kokoris-Kogias Adrian Perrig Giacomo Giuliari, Alberto Sonnino. 2024. An Empirical Study of Consensus Protocols' DoS Resilience. In *ACM ASIA Conference on Computer and Communications Security (AsiaCCS)*.

[35] Neil Giridharan, Florian Suri-Payer, Matthew Ding, Heidi Howard, Ittai Abraham, and Natacha Crooks. 2023. BeeGees: stayin'alive in chained BFT. In *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing*. 233–243.

[36] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. 2019. SBFT: A Scalable and Decentralized Trust Infrastructure. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, USA, 568–580. https://doi.org/10.1109/DSN.2019.00063

[37] Bingyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. 2020. Dumbo: Faster asynchronous bft protocols. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 803–818.

[38] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. 2021. RCC: resilient concurrent consensus for high-throughput secure transaction processing. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1392–1403.

[39] Toke Høiland-Jørgensen, Bengt Ahlgren, Per Hurtig, and Anna Brunstrom. 2016. Measuring latency variation in the internet. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*. 473–480.

[40] Mohammad M. Jalalzai, Jianyu Niu, and Chen Feng. 2020. Fast-HotStuff: A Fast and Resilient HotStuff Protocol. *CoRR* abs/2010.11454 (2020). arXiv:2010.11454 https://arxiv.org/abs/2010.11454

[41] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. 2021. All you need is dag. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*. 165–175.

[42] Idit Keidar, Oded Naor, Ouri Poupko, and Ehud Shapiro. 2023. Cordial Miners: Fast and Efficient Consensus for Every Eventuality. In *37th International Symposium on Distributed Computing (DISC 2023)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[43] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2010. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Transactions on Computer Systems (TOCS)* 27, 4, Article 7 (Jan. 2010), 39 pages. https://doi.org/10.1145/1658357.1658358

[44] Shengyun Liu, Wenbo Xu, Chen Shan, Xiaofeng Yan, Tianjing Xu, Bo Wang, Lei Fan, Fuxi Deng, Ying Yan, and Hui Zhang. 2023. Flexible Advancement in Asynchronous BFT Consensus. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 264–280.

[45] Dahlia Malkhi, Chrysoula Stathakopoulou, and Maofan Yin. 2024. BBCA-CHAIN: Low Latency, High Throughput BFT Consensus on a DAG. *Financial Cryptography and Data Security (FC'24)* (2024).

[46] Dahlia Malkhi and Pawel Szalachowski. 2022. Maximal Extractable Value (MEV) Protection on a DAG. *arXiv preprint arXiv:2208.00940* (2022).

[47] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The Honey Badger of BFT Protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 31–42. https://doi.org/10.1145/2976749.2978399

[48] Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. 2014. Signature-free asynchronous Byzantine consensus with t< n/3 and O (n2) messages. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*. 2–9.

[49] Korakit Seemakhupt, Brent E Stephens, Samira Khan, Sihang Liu, Hassan Wassel, Soheil Hassas Yeganeh, Alex C Snoeren, Arvind Krishnamurthy, David E Culler, and Henry M Levy. 2023. A Cloud-Scale Characterization of Remote Procedure Calls. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 498–514.

[50] Victor Shoup. 2000. Practical threshold signatures. In *Advances in Cryptology—EUROCRYPT 2000: International Conference on the Theory and Application of Cryptographic Techniques Bruges, Belgium, May 14–18, 2000 Proceedings 19*. Springer, 207–220.

[51] Alexander Spiegelman, Balaji Aurn, Rati Gelashvili, and Zekun Li. 2024. Shoal: Improving DAG-BFT Latency And Robustness. *Financial Cryptography and Data Security (FC'24)* (2024).

[52] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2022. Bullshark: DAG BFT Protocols Made Practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 2705–2718. https://doi.org/10.1145/3548606.3559361

[53] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2022. Bullshark: The Partially Synchronous Version. *arXiv preprint arXiv:2209.05633* (2022).

[54] Chrysoula Stathakopoulou, Tudor David, Matej Pavlovic, and Marko Vukolić. 2019. Mir-bft: High-throughput robust bft for decentralized networks. *arXiv preprint arXiv:1906.05552* (2019).

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

[56] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. 2021. Basil: Breaking up BFT with ACID (transactions). In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 1–17.

[57] Aptos Team. [n.d.]. Aptos homepage. https://aptoslabs.com.

[58] Pasindu Tennage, Cristina Basescu, Lefteris Kokoris-Kogias, Ewa Syta, Philipp Jovanovic, Vero Estrada-Galinanes, and Bryan Ford. 2023. QuePaxa: Escaping the tyranny of timeouts in consensus. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 281–297.

[59] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (Toronto ON, Canada) *(PODC '19)*. Association for Computing Machinery, New York, NY, USA, 347–356. https://doi.org/10.1145/3293611.3331591