Motorway: Seamless high speed BFT

Neil Giridharan[†,*] Florian Suri-Payer[‡*],
Ittai Abraham, Lorenzo Alvisi[‡], Natacha Crooks[†]
[†] *UC Berkeley,* [‡] *Cornell University, Intel Labs*

Abstract

Today's practical, high performance Byzantine Fault Tolerant (BFT) consensus protocols operate in the partial synchrony model. However, existing protocols are often inefficient when networks are indeed *partially* synchronous. They obtain either low latency during synchrony or robust recovery from periods of asynchrony. At one end, traditional, view-based BFT protocols optimize for latency in the sunny-network case, but when faced with periods of asynchrony are subject to performance degradations (*hangovers*) that can last beyond the return to synchrony. At the other end, modern DAG-based BFT protocols recover gracefully from asynchrony, but exhibit lackluster latency during synchronous intervals. To close the gap, this work presents Motorway, a novel high-throughput BFT protocol that offers both low latency and *seamless* recovery from periods of asynchrony. Motorway combines a highly parallel asynchronous data dissemination layer with a low-latency, partially synchronous consensus mechanism to construct an efficient consensus protocol for *partial* synchrony. Motorway (*i*) avoids the hangovers incurred by traditional BFT protocols and (*ii*) matches the throughput of state of the art DAG-BFT protocols while reducing latency by 2.1x, matching the latency of traditional BFT protocols.

1 Introduction

This work presents Motorway, a high-throughput Byzantine Fault Tolerant (BFT) protocol that sidesteps the tradeoff between low latency and robustness that burdens current BFT protocols under realistic network conditions. Motorway achieves low end-to-end latency while simultaneously streamlining recovery from periods of asynchrony.

BFT state machine replication (SMR) offers the appealing abstraction of a centralized, trusted, and always available server, even as some replicas misbehave. Consensus protocols implementing this abstraction [16, 45, 51] are at the core of recent interest in decentralized systems such as blockchains.

To aspire to mainstream adoption a BFT consensus protocol must offer (*i*) high throughput, (*ii*) low end-to-end latency, and (*iii*) remain robust to changing network conditions (*i.e.*, achieve *network robustness*); all whilst being simple enough to build and deploy. Unfortunately, existing BFT systems must choose between low latency and robustness. This tension, we submit, arises from an oversimplified network model that is at odds with realistic network behavior.

Today's popular (and actually deployed) BFT consensus protocols operate primarily in the *partial synchrony model* [24], introduced to get around the impossibility of a safe and live solution to consensus in an asynchronous system [27]. Partially synchronous protocols are always safe, but provide no liveness or performance guarantees before some "magic" *Global Stabilization Time* (GST), after which synchrony is expected to hold forever on; they optimize for latency *after* GST holds. The partial synchrony that real network experience, however, looks quite different: it manifests in constants fluctuations in latency, where periods of approximate synchrony interleave with asynchronous periods. We posit that, despite their name, today's partially synchronous BFT protocols are not efficient under actual partial synchrony.

Broadly speaking, there are two popular approaches to BFT SMR, offering different tradeoffs:

Traditional BFT [16, 31, 36, 51] protocols and their multi-leader variants [32, 47, 48] optimize for sunny-day networks, that is, for performance after GST. This allows such protocols to minimize the number of message exchanges (hereafter consensus latency) in the common (synchronous) case. Unfortunately, we find that, after even a brief period of asynchrony, such protocols do not gracefully resume operation processing once synchrony returns; we characterize such behavior as *asynchrony hangover*.

Under continuous high load, lack of progress during periods of asynchrony (and a consequent loss of throughput) can cause large request backlogs that cannot be made up for quickly (if at all) in the common synchronous case. This causes a degradation in end-to-end latency that persists well after synchrony returns (Figs. 1,8). While traditional BFT protocols enjoy low consenus latencies on paper, their network robustness is poor. Consequently, they underperform under *partially* synchronous network condidtions.

DAG-based BFT, a newly popular class of consensus protocols [10, 20, 34, 39, 44, 45], originate in the *asynchronous* model and take an entirely different approach. Asynchronous protocols optimize for worst-case message arrivals, and leverage randomness to guarantee progress no matter the circumstance. DAG BFT protocols employ as backbone a high throughput asynchronous data dissemination layer that proceeds through a series of structured, tightly synchronized rounds, forming a DAG of temporally related data proposals. Replicas deterministically *interpret* their local view of the DAG to establish a consistent total order. This approach enjoys both excellent throughput and network resilience. However, if in asynchronous protocols progress is guaranteed, it comes at a higher, often prohibitive latency; consequently, these protocols see little practical use. Recent DAG protocols [44, 46] leverage partial synchrony to improve consensus

[*]Joint first authors, alphabetical.

latency, yet consensus still requires several rounds of Reliable Broadcast communication, with Bullshark [45], the state of the art, exhibiting consensus latency of up to 12 message delays (henceforth *md*). DAG-based BFT protocols tolerate periods of asynchrony gracefully (or *seamlessly*), but underperform during periods of synchrony.

This work discusses how to soften the tension between low consensus latency and robustness and arrive at an efficient design under *partial* synchrony. Motorway demonstrates how to selectively leverage asynchrony and partial synchrony across different protocol components, to construct an efficient partially synchronous protocol.

At a high level, Motorway's architecture is inspired by recent DAG protocols: it constructs a highly parallel data dissemination layer that continues to make progress even under asynchrony. 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 asynchronous periods – however, upon synchrony's return, Motorway can instantaneously commit (with constant complexity) the *entire* data backlog that was disseminated asynchronously.

Our results are promising (§6): Motorway matches the throughput of Bullshark [45] while reducing its latency by 2.1x; it even edges out slightly the latency of Hotstuff [51] while sidestepping Hotstuff's asynchrony hangovers.

This paper makes three core contributions:

- It introduces the notions of asynchrony *hangovers*, and *seamless* partial synchrony (§2) to characterize the performance of partially synchronous protocols in practice.
- It presents the design of Motorway, a novel partially synchronous BFT consensus protocol with low latency and high throughput that tolerates asynchrony hangovers.
- It evaluates the performance of a Motorway prototype, and compares it with Hotstuff [51] (traditional BFT) and Bullshark [45] (DAG-BFT).

2 Partial Synchrony: Theory meets Practice

This work is premised on a simple observation: partially synchronous protocols today do not *practically* tolerate periods of asynchrony. Partial synchrony is an appealing theoretical model that captures when consensus protocols can still achieve liveness, in spite of the seminal FLP impossibility result [27]. All partially synchronous protocols are live when the partially synchronous network reaches the Global Stabilization Time (GST), from which point all messages all be delivered within Δ time [24]. While useful for proving liveness, this model does not accurately describe how real networks work. Message delays due to intermittent failures or packet queuing are frequent; most RPCs for instance suffer from high tail latency [21, 42]. Setting accurate timeouts is similarly a notoriously hard challenge [37, 44, 50]. In short, modern networks are truly partially synchronous: most messages will arrive within Δ most of the times, but not always. There will be periods of asynchrony in which the network behaves poorly. To properly assess the performance of a partially synchronous system, we should consider how it behaves *at all times*, during periods of full synchrony, but also *after* periods of asynchrony.

While Aadvark introduced the concept of synchronous intervals [17] when describing the need for robust BFT systems, there does not currently exist a precise way to capture the performance of partially synchronous protocols during synchronous intervals *after* experiencing periods of asynchrony. To this effect, we introduce the dual notions of asynchrony hangovers and seamless partially synchronous protocols. Informally, a partially synchronous protocol is seamless if it does not suffer from protocol-induced hangovers.

Asynchrony Hangovers & Seamlessness. All deterministic partially synchronous protocols require a coordinator (or *leader*) to propose an input and drive progress; "leaderless" protocols are a misnomer, and typically refer to either (*i*) the use of multiple coordinators driving independent instances in parallel [32, 47], or (*ii*) the use of a client as coordinator [49]. Byzantine leaders may establish "progress", but intentionally limit performance; such Byzantine behavior is indistinguishable from asynchrony. Following Aardvark [17] and Abraham et al. [7] 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; we say it is *gracious* if it is synchronous and all replicas are correct [17]. Given this, we define:

Definition 1 *An* ***asynchrony hangover*** *is any performance degradation that is caused by a period of asynchrony, and persists beyond the return of a good interval.*

Asynchrony hangovers impact both throughput and latency. We characterize a throughput hangover as the duration required to commit all outstanding requests issued before the start of a good interval. If the provided load is below the system's steady-state throughput, the system will eventually recover all throughput "lost" during asynchrony.[1] The longer the hangover lasts, the more severe it is. Delayed throughput recovery can, in turn, materialize as increased latency even for commands issued *during* a good interval. Assuming FIFO priority, such commands experience the latency required to first commit both the backlog formed during asynchrony, and the perpetuated backlog induced by new waiting commands. Hangovers can cause not only immediate headaches, but also make the system more susceptible to future asynchronous bumps in the road. Good intervals may not last forever, and unresolved hangovers can add up—and continue indefinitely.

In a hangover-free system, all submitted requests whose progress is interrupted by asynchrony commit instantaneously when progress resumes; further, these are the only requests

[1] If the load is equal/greater to the throughput, then the lost committed throughput (or *goodput*) can never be recovered.

that experience a hike in latency, and for no more than the remaining duration of the asynchronous period.

Careful protocol design cannot of course eliminate all hangovers, *e.g.*, those due to insufficient network bandwidth or to bottlenecks caused by message deliveries, but it can avoid introducing unforced ones. Thus, we say that a partially synchronous protocol behaves optimally (w.r.t to periods of asynchrony), or *seamlessly*, if it experiences no asynchrony hangovers solely due to protocol design choices.

Definition 2 *A partially synchronous system is* ***seamless*** *if it is free of protocol mechanisms subject to asynchrony hangovers.*

Analysis of existing approaches. Traditional BFT protocols[16, 51] are not seamless. They disseminate batches of requests and reach consensus on each batch independently; the number of consensus instances required to commit all requests is directly proportional to the input rate of requests the batches' size. Consequently, the time required by these protocols to process the backlog accumulated during progress-free (asynchronous) intervals is in the order of the backlog's size.[2]

We demonstrate empirically in Fig. 1 that Hotstuff [51], for instance, suffers a latency hangover because it is bottlenecked by the (processing) bandwidth of each proposal (§6).

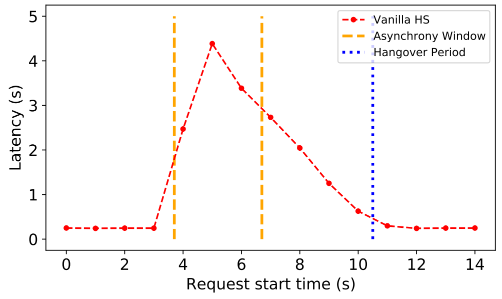

Figure 1: Latency Hangover (nearly 4 seconds) in Hotstuff [51] following an asynchroneous blip of 3 seconds.

Simply proposing larger batches is not a viable option for reducing the number of required agreement instances as it comes at the cost of increased latency in the common case. Similarly, a straightforward decoupling of dissemination and consensus (proposing only a batch digest) [9, 20] can alleviate data transfers and request verification on the critical path of consensus, but does not address the principled concern.

A class of recent DAG-based BFT protocols [20, 45], in contrast, is seamless. These protocols decouple data dissemination and consensus, and construct an asynchronously growing DAG consisting of data batches chained together. The construction leverages Reliable Broadcast to generate proofs that transitively vouch for data availability and request verification of *all* topologically preceding batches in the DAG. The consensus mechanism can commit entire backlogs with constant cost by proposing only a single batch, and thus experiences complexity *independent* of asynchronous durations.

[2]We assume there is continuous client load during asynchrony.

Unfortunately, existing seamless DAG-BFT protocols exhibit, during *good* intervals, up to twice the latency of traditional BFT protocols. To bridge the gap, we propose Motorway, a novel high throughput BFT protocol that delivers low latency during synchrony and handles *partial* synchrony seamlessly.

3 Motorway Overview

The high level architecture of Motorway comprises of two logically distinct layers: a horizontally scalable data dissemination layer that always makes progress, even during asynchrony; and a low-latency, partially synchronous consensus layer that tries to reach agreement on snapshots of the data layer. While the aim of decoupling consensus from data dissemination is not unique to Motorway, it more carefully separates their precise responsibilities to achieve better latency and robustness to asynchrony.

Data dissemination. To enable seamless consensus, Motorway's data dissemination layer must offer two guarantees:

- *Asynchronous Growth.* Data (transaction) dissemination will continue during periods of asynchrony.
- *Instant Referencing.* The set of proposed transactions will be identifiable in constant time and with constant space.

All replicas in Motorway act as proposers, and take responsibility for disseminating transaction requests received by clients. Each replica, in parallel, broadcasts batches of requests (data proposals), and constructs a local chain that implicitly assigns an ordering to all of its proposals. This structure allows us to transitively prove the availability of all data in a chain in constant time, thus guaranteeing *instant referencing*. We refer to this local chain as a *data lane*. Data lanes grow independently of one another, subject only to load and resource capacity, and are agnostic to periods of asynchrony (offering *asynchronous growth*). Finally, Motorway guarantees that any growth is committed in a timely fashion (*reliable inclusion)*; while this is not core to seamlessness, it is vital to ensure robustness to Byzantine broadcasters in practice.

Consensus. The consensus layer exploits the structure of data lanes to, in a single shot, commit arbitrarily large data lane state. We refer to this property as *constant commit*; it is key to ensure seamlessness. To satisfy this requirement, Motorway 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 [16, 29] consisting of two round-trips; during gracious intervals, Motorway employs a *fast path* allowing consensus to commit within only a single round-trip. Upon reaching agreement on a tip cut, a replica infers the respective chain-histories of each tip, synchronizes on any missing data, and deterministically intertwines the n chain segments to construct a single total order. Importantly, for seamlessness, there must never be any data synchronizing on the critical path of consensus (*non-blocking sync*), and any data fetch must happen in constant time (*bounded sync*). Motorway achieves this through a careful selection of voting rules in the data dissemination layer.

This simple 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*) to preserve the latency-optimality in gracious executions of traditional BFT protocols.

We describe the full Motorway protocol Section §5, and discuss in detail what other design considerations are required to achieve seamlessness, and low end-to-end latency.

4 Model

Motorway adopts the standard assumptions of prior partially synchronous BFT work [16, 45, 51], 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 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 in protocol descriptions.

Motorway operates under partially synchrony [24]: it makes no synchrony assumptions for safety, but guarantees liveness only during periods of synchrony [27].

5 Introducing Motorway

To describe Motorway more thoroughly, we first focus on the data dissemination layer (§5.1), which ensures constant progress during asynchrony. We then focus on how the consensus layer (§5.2.1) leverages the data layer's lane structure to efficiently, and in constant time, commit arbitrary long data lanes, thus seamlessly recovering from asynchrony.

5.1 Data dissemination

Motorway's data dissemination layer precisely provides the three properties of asynchronous growth, instant referencing, and reliable inclusion. Crucially, it provides no additional guarantee, as any additional property can only be achieved at the cost of increased tail latency.

Lanes and Cars. In Motorway, every replica acts as a proposer, continuously batching and disseminating incoming user requests. Each replica proposes new batches (data proposals) at its own rate, and fully independently of other replicas' proposal rates — 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 pattern to disseminate its data proposals: the proposer broadcasts them 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, Motorway, unlike most DAG-BFT protocols, does not force all data proposals to go through a round of *reliable broadcast* [20, 45] that must process $n-f$ votes in order to achieve non-equivocation. Reliable, non-equivocating broadcast is not necessary in Motorway to achieve the desired data layer properties thanks to its tip-cut consensus (§5.2).

A lane is made up of a series of cars that are chained together (Fig. 3). A proposer, when proposing a new batch, must include a reference to its previous car's data proposal. Similarly, replicas will vote in a car at position i if and only if its proposal references a proposal for car $i-1$ that it has already received and voted on. This construction ensures that a successful car for block i transitively proves the existence of a car for all blocks 0 to $i-1$ in this proposer's lane, and that a single correct replica is in possession of all proposals in this lane. We refer to this property as *FIFO voting*; it is necessary to ensure instant referencing when recovering from asynchrony (§5.2.2).

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 user requests (transactions) into a batch B, 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 preceding proposal, the previous tip of the lane), and $cert := l.tip.$PoA (the proof of availability of the previous proposal).

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

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

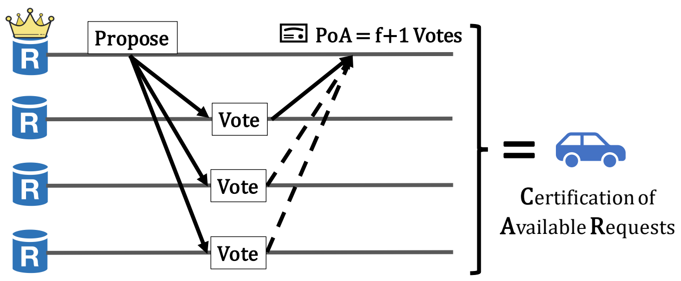

Figure 2: Car protocol pattern.

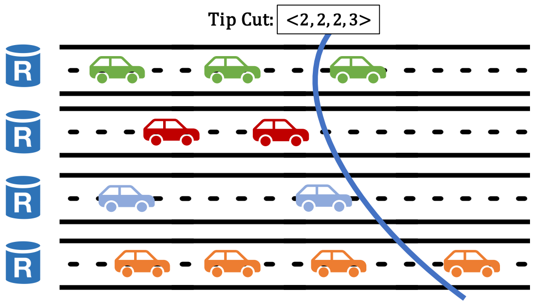

Figure 3: Motorway parallel data lanes; example cut.

VOTE $:= \langle dig = h(\text{PROP}), pos\rangle_R$. If, in contrast, R has not yet received the parent proposal for PROP, it simply waits.

3: P → R: Replica P assembles VOTEs and creates a POA.

The proposer aggregates $f+1$ matching VOTE messages from distinct replicas into a POA $:= (dig, pos, \{\sigma_r\})$. It includes the POA on the next car's proposal; if no new batch is ready then P may decide to broadcast the POA immediately.

Cars 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. By limiting the batch size per car, however, Motorway forces Byzantine replicas to finish certifying a proposal to disseminate more data. Consequently, at least one correct replica will eventually propose all certified proposals for consensus (§5.2), allowing correct replicas to naturally bound the maximum receivable "waste" (*reliable inclusion*). Finally, we note that 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).

5.2 From Lanes to Consensus

Motorway's data dissemination layer (*i*) disseminates batches of requests (*data proposals*), and (*ii*) succinctly summarizes which proposals are reliably stored. It intentionally provides no consistency guarantee on the state observed by any two correct replicas. For instance, different replicas may observe inconsistent lane states (neither a prefix of another).

The role of the consensus layer in Motorway is to reconcile these views and to totally order all certified proposals. To ensure seamless recovery, the consensus layer should be able to commit, in a single shot, an arbitrary large number of certified data proposals (*constant-commit*); while synchronizing inconsistent replica states off the critical path (*non-blocking sync*), and in constant time (*bounded sync*). Without these properties, the protocol will necessarily be subject to asynchrony hangovers. Unlike traditional BFT consensus protocols, that take as input a batch of requests to commit, a *consensus proposal* in Motorway 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 the all lane states allows Motorway to achieve consensus throughput that scales horizontally with the number of lanes (*n* total) and their volume, while (*ii*) proposing certified lane tips allows Motorway to commit arbitrarily large backlogs with complexity independent of its size. We discuss in turn how Motorway 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

Motorway's consensus protocol follows a classic two-round (linear) PBFT-style [16] agreement pattern, augmented with a fast path that reduces latency to a single round (three message delays) in gracious intervals. The protocol progresses in a series of slots: leaders are assigned to a slot s and can start proposing a new lane cut upon observing committment of slot $s-1$, and sufficiently many new tips to ensure the proposal is doing useful work. Within each slot, Motorway follows a classical view based structure [16] – when a slot's consensus instance fails to make timely progress, replicas arm themselves and rebel, triggering a *View Change* and electing a new leader.

Each view consists of two phases (5md total), the *Prepare* and the *Confirm* phases. The Prepare 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).

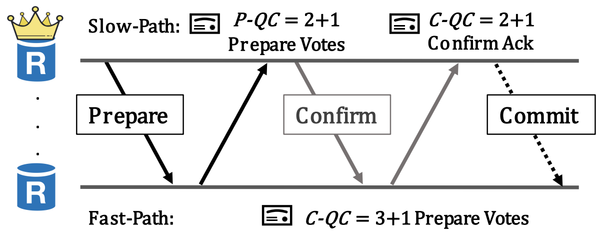

Figure 4: Core Consensus coordination pattern. On the Fast-Path the Confirm phase is skipped.

For simplicity, we first present our protocol for a single slot and view. We assume that all replicas start in view $v=0$. We explain Motorway'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 a formal discussion of correctness to the Appendix A.

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 $\rightarrow P_{s,v}$
6: $conf$              ▷ map: slot $\rightarrow \text{PREPAREQC}_{s,v}$
7: $last\text{-}commit$ ▷ map: lane $\rightarrow$ pos
8: $log$               ▷ total order of all user requests
```

Prepare Phase (P1).

1: L $\rightarrow$ R: Leader L broadcasts PREPARE.

A leader L for slot s begins processing (view $v = 0$) once it has received a ticket $T := \text{COMMITQC}_{s-1}$, 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 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: $P := <s, v, [lanes[1].tip.\text{POA}, ..., lanes[n].tip.\text{POA}]>$.

2: R $\rightarrow$ L: Replica R processes PREPARE and votes.

R first confirms the PREPARE's validity: it checks that (*i*) the ticket endorses the leader's tenure ($s = T.slot + 1$) and $Leader(s) = L$), and (*ii*) R has not yet voted for this slot. If it is valid, R stores a copy of the proposal ($prop[s] = P$), and returns a vote PREP-VOTE $:= \langle dig = h(P) \rangle_R$ for the digest of the proposal. Finally, if it has not locally received all tips (and their histories) included in P, R begins asynchronously fetching all missing data (§5.2.2). Unlike prior work, Motorway's data layer FIFO voting guarantee ensures that the missing data can be fetched in a single round-trip.

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

L waits for at least $n - f = 2f + 1$ matching PREP-VOTE messages from distinct replicas and aggregates them into a *Prepare Quorum Certificate (QC)* PREPAREQC $:= (s, v, dig, \{\text{PREP-VOTE}\})$. 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, and this correct node will never vote more than once per view. To make the decision durable across views as well (the safety requirement to commit), L initiates the *Confirm phase*.

Fast Path. 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. 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 $\rightarrow$ 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 $\rightarrow$ 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 $\rightarrow$ 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 $\rightarrow$ R: Leader L assembles COMMITQC and commits.

L aggregates $2f + 1$ matching CONFIRM-ACK messages from distinct replicas into a *Commit Quorum Certificate* COMMITQC $:= (s, v, dig, \{\text{CONFIRM-ACK}\})$, and broadcasts it to all replicas. Replicas that observe this COMMITQC commit the associated proposal. Recall however, that consensus proposals in Motorway are cuts of data proposal, rather than simple batches of commands. Upon committing, the replicas must then establish a (consistent) total order of proposals to correctly execute operations.

5.2.2 Processing committed cuts

Each replica maintains a log of all ordered requests, and a map *last-commit* which tracks, for each lane, the position of the latest committed 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). If it does not then it must first acquire (*synchronize*) the missing data; in the worst-case, this may be the entire history. A Byzantine replica G could leverage a period of asynchrony to certify a long lane, yet broadcast data proposals to only $f + 1$ replicas, requiring all other replicas to sync on an arbitrarily long history. If done naively, a replica may need to recursively fetch each data proposal in a tip's history, requiring sequential message exchanges in the order of a lane's length ($\sim$length of asynchrony).

Motorway exploits the lane structure to synchronize in parallel with consensus, and in just a single message exchange, no matter the size of the history (subject to data bandwidth constraints). This ensures smooth consensus progress, and allows replicas to successfully synchronize on all required tip histories by the time the proposed cut commits. This property is unique to Motorway and not currently offered by other BFT protocols.

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, and moves synchronization off the critical path. If lanes, in contrast, were not certified (e.g. Motorway employed only "best-effort" dissemination) then replicas would have to synchronize *before* voting on a consensus proposal to assert whether a tip (and its history) indeed exists. This blocking increases the risk of timeouts (and view changes), especially after long periods of asynchrony, undermining seamlessness.

Second, FIFO voting in turn guarantees that *there exists at least one* correct replica that is in possession of *all* data in a certified tip's history. To synchronize, a replica requests (from the replicas that voted to certify the tip) to sync on 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 sync on the same suffix.

Creating a Total Order. Once a replica has fully synchronized on a 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" 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. $start.pos == last_l + 1$). We note that the proposal committed at $last_l$ may not (and need not) be the parent of $start$; 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 $start$.

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 (all lane proposals between, and including, $start_l$ and $tips[l]$) using some deterministic zipping function. For instance, we may zip all lanes new proposals (from oldest to newest) in round-robin fashion (to provide fairness).

5.2.3 Lane Coverage

Correct leaders in Motorway 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 and pace, 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 Motorway, we set coverage, as default, to at least $n-f$ new tips (at least $n-f$ lanes with at least one new proposal each). This ensures that at least half of the included tips belong to correct replicas.

Lane coverage in Motorway is best-effort and not enforced. From a performance perspective, a Byzantine leader that disregards coverage is indistinguishable from a crashed one; from a fairness perspective, it is equivalent to leaders in traditional BFT protocols that assemble batches at will. The next correct leader (in a good interval) will make up for any lost coverage.

Importantly, lane coverage does not cause Motorway to discriminate against slow lanes. Unlike contemporary DAG-BFT protocols that are driven by the fastest $n-f$ replicas, and are subject to *orphaning* proposals of the f slowest, proposal cuts in Motorway include *all n* lanes. Consequently, all (correct) replicas' proposals are guaranteed to commit in a timely fashion (as soon as a correct leader receives them).

This property is a direct consequence of Motorway's careful separation of concern between the data layer and the consensus layer. The data layer is *exclusively* responsible for disseminating data and making it available to the consensus layer. The consensus layer is instead responsible for committing cuts of *all the data*. As such, the $n-f$ fastest replicas still drive consensus, but necessarily include all data that has been received, without penalizing slow nodes.

5.3 View change

In the presence of a faulty leader the protocol is not guaranteed to create the COMMITQC necessary to commit. As is standard, Motorway relies on timeouts in these situations to elect a new leader: replicas that fail to see progress rebel, and create a special *Timeout Certificate* (TC) ticket in order to force a *View Change* [16, 36].[3]

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},[4] 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 $current\text{-}view_s$, 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. Motorway favors conservative timer for smooth progress in good intervals as the data layer continues progressing even in asynchrony. It cancels a 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.

[3] We adopt and modify the View Synchronizer proposed in [29].

[4] 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.

1: R $\rightarrow$ 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 rebellion 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-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 Motorway. If a proposal appears $f+1$ 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 $\rightarrow$ 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

Motorway consensus, by default, proceeds through a series of *sequential* slots instances. Unfortunately, lane proposals that narrowly "miss the bus" and are not included in the current slot's 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 consensus instance necessary to commit).

Parallel Consensus. To circumvent sequential wait-times, Motorway, inspired by PBFT [16], allows for a new slot instance to begin in parallel with ongoing slots rather than systematically waiting for the previous slot 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 first PREPARE$_{s-1}$ message (the view is immaterial), and has sufficient lane coverage is satisfied (w.r.t to PREPARE$_{s-1}$). While this encourages leaders to minimize redundant work, lane coverage is strictly best effort: it cannot be enforced as it is not guaranteed that PREPARE$_{s-1}$ represents the proposal that is ultimately committed in slot $s-1$ (e.g. due to a View Change).

Allowing for parallel consensus instances minimises the articial wait-time imposed by sequential consensus. With a stable leader (the same leader proposes multiple slots), this delay completely disappears. With rotating leaders (a different leader drives each slot), the delay reduces to a single message delay; the time to receive PREPARE$_{s-1}$.

We adjust Motorway as follows.

Managing concurrent instances. Motorway maintains independent state for all ongoing consensus slot instances. Each instance has its own consensus messages, and is agnostic to consensus processing on all concurrent slots. Slots that commit wait to execute until all preceding slot instances have already executed. Similarly, when ordering a slot's proposal (a cut of tips), Motorway replicas identify and order only proposals that are new. Old tip positions are simply ignored. This filtering step is necessary as there is no guarantee that consecutive slots will propose 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 PREPARE$_{(s-1,v)}$ message.. Finally, 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 four message delays, there should, in practice, be no more than three to four concurrent instances at a time. During periods of asynchrony (or in presence of faulty leaders) 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 gracious interval, as a single new cut can subsume all past proposals. Motorway 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. Motorway's parallel proposals departs from modern chained-BFT protocols [29, 30, 51] 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 [30], and (*iii*) it artificially couples coverage between successive slots, i.e. coverage for slot 2 affect progress of slot 1.

5.5 Optimizations

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

5.5.1 Further reducing latency

By default, Motorway consensus pessimistically only proposes *certified* tips. This allows voting in the consensus layer to proceed without blocking, a crucial requirement to achieve 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. Motorway employs two optimizations to reduce the latency between the dissemination of a data proposal and its inclusion in a consensus proposal (*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 (FIFO voting).

Optimistic tips. Second, Motorway allows leaders to optimistically propose non-certified tip references (dig,pos) for lanes that have, historically, remained reliable. 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 – by design, all ancestors must be certified, and thus can be synchronized asynchronously. Moreover, synchronization is only necessary in either view $v=0$ or in subsequent views that did not select a winning proposal from a prior view. In all other cases, the selected proposal P included in $\text{PREPARE}_{v'>0}$ is, by design, implicitly certified ($\geq f+1$ replicas voted for P).

In gracious intervals, this optimization lowers inclusion latency to a single message exchange, which is optimal. When the triangle inequality holds all replicas will locally deliver all proposed tips histories before receiving a PREPARE; when it does not, replicas may over-eagerly request unnecessary synchronization. During less gracious intervals proposing optimistic tips poses a risk to a leader's tenure. For instance, a Byzantine proposer may forward its batch only to the leader, burdening it with synchronization, and potentially causing its view to time out. To improve the predictability of unwanted synchronization, Motorway can be augmented with a simple (replica-local) reputation mechanism. A leader forced to synchronize on some lane l's tip downgrades it's local perception of l's reputability. Below a threshold, the leader stops proposing optimistic tips for l, and simply falls back to proposing a certified tip. Each replica is thus responsible for warranting optimism for its own lane and consequently its own data proposals' latency. Reputation can be regained over time, e.g. for each committed car.

5.5.2 Ride sharing

Consensus phases and cars share the same basic message pattern [8] and can, in theory, be sent and signed together (*ride-sharing*) to approach the "zero-message" consensus overhead advertised by recent DAG-BFT protocols [20, 45].[5] Ride-sharing, however, comes at a slight latency tradeoff, as consensus messages have to wait for cars to be available. We find that in practice that, even for small n, consensus coordination is not a throughput bottleneck (data processing is, §6), and thus we opt to defer in-depth discussion to a future TR.

5.5.3 Other optional modifications

Motorway can be further augmented with several standard techniques. Our prototype does not implement these, but we discuss them briefly for completeness.

Signature aggregation. Motorway, like many related works [31, 45, 51] can be instantiated with threshold (TH) signatures [12, 43] that aggregate matching vote signature of any Quorum Certificate (PoA, PrepareQC, CommitQC) into a single signature. This reduces Motorway's complexity per car and consensus instance (during good intervals) to $O(n)$; view changes require $O(n^2)$ messages [51].

All-to-all communication. Motorway follows a linear-PBFT communication pattern [31] that allows it to achieve linear complexity (during good intervals) when instantiated with aggregate signatures. Motorway 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.

[5] We note that the relative consensus overheads decrease linearly as n scales, and are thus already exceedingly small for $n >> k$ (max #instances).

6 Evaluation

Our evaluation seeks to answer three questions:

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

We implement a prototype of Motorway in Rust, starting from the open source implementation of Narwahl [26]. We use Tokio's TCP [4] for networking, and RocksDB [3] for persistent storage. Finally, we use ed25519-dalek [2] signatures for authentication.

Baseline systems. We compare against open-source prototypes of Hotstuff [51] and Bullshark [45, 46] (both implemented in the same codebase as Motorway) which, respectively, represent today's most popular traditional and DAG-based BFT protocol. Concretely, we use the "batched" Hotstuff (BatchedHS) prototype [25] used in Narwhal [20] that simulates an upper bound on achievable performance by optimistically separating dissemination and consensus. Replicas broadcast batches asynchronously in advance, and consensus leaders propose hashes of any received batches. This design is not robust in real settings: Byzantine replicas (or periods of asynchrony) may exhaust the mempool as there is no guarantee that batches will be committed (no *reliable inclusion*) requiring the system to eventually drop transactions. This design may also force leaders to block on the critical path for consensus by synchronizing on batches that were not yet actually disseminated (no *non-blocking sync*, nor *bounded sync*). We evaluate also a standard version of Hotstuff (VanillaHS) in which we modify BatchedHS to send batches only alongside consensus proposals of the issuing replica.

Both the Bullshark [26] and Motorway prototypes adopt the horizontally scalable worker layer proposed by Narwhal [20] that streams batches in advance (akin to BatchedHS) using Reliable Broadcast (RB), and includes only batch references in consensus proposals. For a fair comparison with Hotstuff we run only with a single, co-located worker; in this setup the RB 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`[1] with 20GB of SSD. We co-locate one client machine per replica in the same region. Each such machine issues a constant stream of transactions (tx). All transactions consist of 512 random bytes (consistent with the experiments in Narwhal [20] and Bullshark [46]). We measure latency as the time between tx arrival at a replica and the time it is assigned a position in the totally ordered log (often referred to as finality); throughput is measured as finalized 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 [25, 26] 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.

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	0.5	28

Table 1: RTTs between regions (ms)

6.1 Performance in ideal conditions

Figure 5 shows the performance in a fault-free, synchronous setting, using $n=4$ nodes. Motorway'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 Motorway reduces coordination to 4md on the Fast Path, and 6md otherwise (when instantiated with optimistic tips).

To better understand the performance benefits of the fast path and optimistic tip optimizations, we evaluate Motorway 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 as a result of its larger CommitQC ($|n|$) and must contact non-local replicas, including those located across the coasts. Similarly, we record a 33ms increase when proposing only certified tips. This follows directly from the need to contact $f+1$ replicas in a car to form a PoA. In our setup, one of these replicas will necessarily be located in a non-local datacenter on the same coast. Both Motorway and Bullshark significantly outperform both Hotstuff variants for throughput by a factor of 1.23x (BatchedHS) and 18.8 (VanillaHS), and are bottlenecked on the cost of deserializing and storing data on disk. VanillaHS offers very low throughput (12.4k tx/s) 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 by proposing only digests. 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 [20] and demonstrates that this design is not robust in practice.

Scaling number of replicas: Figure 6 shows the peak throughput achievable as we increase the number of replicas. Each bar includes the respective latency (in ms). Both Motorway and Bullshark scale gracefully as n increases; throughput remains bottlenecked on data processing (disk

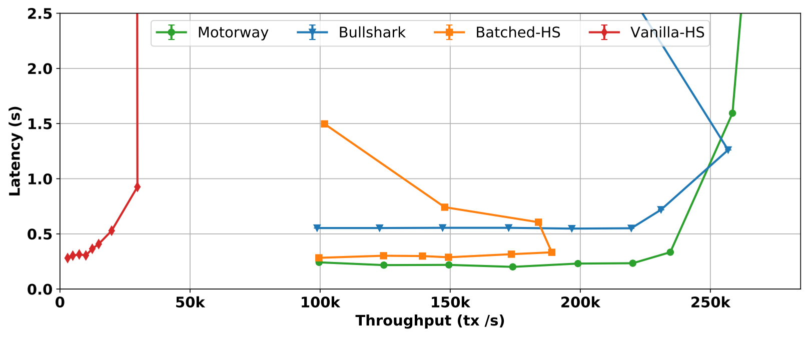

Figure 5: Throughput and Latency under increasing load.

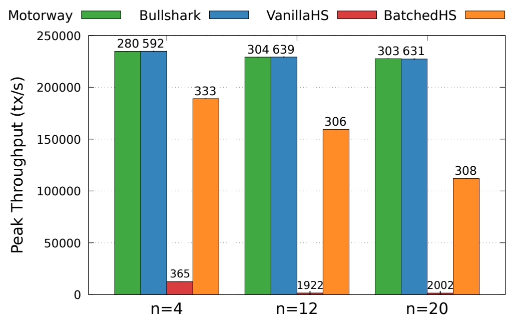

Figure 6: Peak throughput for varying n. Numbers atop each bar describe measured latency (ms).

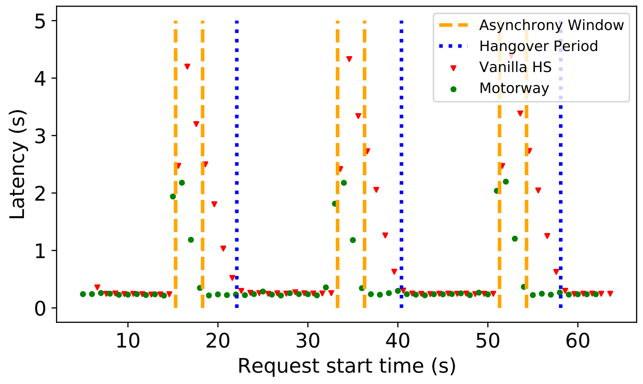

Figure 7: Asynchrony hangovers.

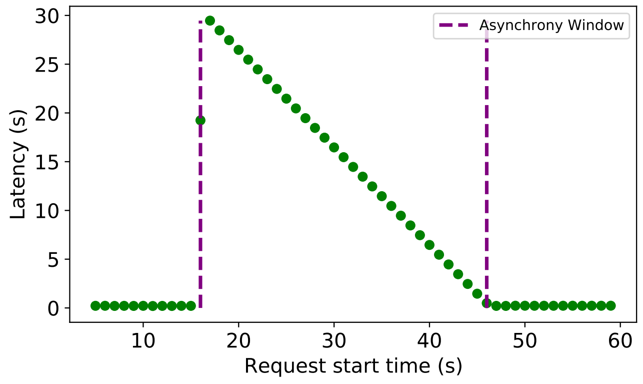

Figure 8: Motorway seamlessness.

and data ser/deser), while latency is unaffected.[6] BatchedHS, in contrast, experiences a sizeable throughput degradation (-16% at $n = 12$, and -41% at $n = 20$) when growing n as replicas are increasingly likely to be out of sync and thus be forced to perform additional data synchronization on the critical path. Finally, we observe that VanillaHS suffers a stark 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 12.4k to only 1.5k tx/s as latency surges rapidly. VanillaHS is 1.3 times slower than Motorway at n=4, but over 6.6 times slower at n=20. .

6.2 Operating in *partial* synchrony

The previous section highlighted Motorway's good performance in the failure-free scenario: Motorway achieves the best of both worlds. It matches Bullshark's throughput while preserving Hotstuff's low latency. We now investigate whether Motorway remains robust to periods of asynchrony.

We first demonstrate that VanillaHS is subject to hangovers in the presence of even short, asynchronous blips (3 seconds). Figure 7 shows three blips and the resulting latency hangovers for VanillaHS (under 15k tx/s load, $n = 4$); Motorway (at 210k tx/s load) experiences no hangovers. We plot latency over time. Data points are averages over second-long windows. VanillaHS suffers hangovers that persists for up to 3.6 seconds beyond the resumption of a good interval as it remains bottlenecked by data dissemination and delivery when trying to work off the async backlog. We observe that the latency penalty not only exceeds the asynchronous duration, for a maximum single-tx latency of 5.1 seconds, but remains, respectively, as high as 1.1 seconds at 2 seconds, and 700ms at 3 seconds into the hangover (a respective 3x and 1.9x increase over the steady state 365ms). Fig. 1 shows the third blip and hangover close up.

To validate that Motorway is seamless and agnostic asynchrony duration we simulate a temporary network partition by forcing a leader failure for 30 seconds, ca. 15 seconds into the experiment, under 100k tx/s load and $n = 4$ (Fig. 8). Motorway experiences no hangover because it instantly commits the entire backlog as soon as synchrony returns. Transactions submitted during the asynchronous period only experience a latency penalty proportional to the asynchrony duration, which is optimal.

7 Related Work

Motorway draws inspiration from several existing works. Scalog [22] demonstrates how to separate data dissemination from ordering to scale throughput at low latencies (for crash

[6] In our setup, the total load remains constant, but each replica proposes a smaller fraction as n grows. If replicas were to instead propose at constant rates, throughput would increase with n (up to the data processing bottleneck).

failures); akin to Motorway's lanes, Scalog disseminates data, and reaches consensus on cuts of independent log segments. Narwhal [20] discusses how to implement a scalable and BFT robust asynchronous broadcast layer; a core tenant for achieving seamlessness. Motorway shares the spirit of layering consensus over asynchronous dissemination, but improves efficiency at both levels. Motorway's consensus itself is closely rooted in traditional, latency-optimal BFT-protocols, most notably PBFT's core consensus logic and parallelism [16], Zyzzya's fast path [31, 36], and Aardvark's doctrine of robustness [17]. We note that Motorway's architecture, by design, isn't tied to a specific consensus mechanism, and thus can easily accommodate future algorithmic improvements (including advances in asynchronous protocols). Finally, Motorway draws on Shoal [44] and Carousel [18] for inspiration on reputation-schemes that can be used to support robust optimistc tips.

7.1 Partially synchronous BFT consensus

Traditional BFT protocols such as PBFT [16], Hotsuff [51], and their respective variants [31, 36][29, 33] optimize for good intervals: they minimize latency in the common (synchronous) case, but guarantee neither progress during periods of asynchrony, nor resilience to asynchrony hangovers.

Protocols in this class employ a dedicated (single) leader that disseminates data and drives agreement in the classic PREPARE/CONFIRM pattern adopted by Motorway. PBFT [16] rotates leaders only in presence of faults (or asynchrony), and allows a stable leader to drive consensus for multiple slots in parallel. Hotstuff [51] rotates leaders for every proposal, but pipelines consecutive consensus phases to reduce latency and share signature costs. Motorway incorporates features of both approaches: it allows parallel proposals for minimal latency, rotates leaders eagerly for fairness and robustness [17], and is amenable to ride sharing for optimal resource efficiency.

Multi-log frameworks [32, 47, 48] see a single leader as a throughput bottleneck and partition the request space across multiple proposers. They carefully orchestrate traditional BFT instances in parallel, and intertwine their logs to arrive at a single, total order. Multi-log frameworks inherit both the good and the bad of their BFT building blocks: consensus has low latency, but remains prone to hangovers.

DAG-BFT, a new class of BFT protocols [10, 20, 34, 39, 45] originate in the asynchronous model and enjoy both excellent throughput and network resilience; recent designs [35, 44, 46] make use of partial synchrony to improve consensus latency. DAG protocols employ a highly parallel asynchronous data dissemination layer that proceeds through a series of structured, tightly synchronized rounds. Each proposal references $n-f$ proposals in the prior round, forming a DAG of causally related, data proposals. Consensus uses this structure as a common clock, and is embedded with no message overhead: in every other round, the consensus protocol *interprets* the DAG structure to identify a leader node, and constructs an ordered log by committing and flattening the leaders topological (or *causal*) history in the DAG. *Certified* DAGs achieve seamlessness but require several rounds of Reliable Broadcast (RB) communication; Bullshark [46], the state of the art, requires between 3 and 4 rounds of RB (3 when optimized [44][7]).

A few un-certified DAGs have been proposed recently; these designs improve latency, but forgo seamlessness. Cordial Miners [35] replaces the RB in each round with best-effort-broadcast, allowing it to match PBFT's latency in gracious intervals; to guarantee progress, however, it must wait in each round (up to a timer) for a message from the leader. BBCA-chain [38][7] modifies Bullshark by weakening non-leader proposals to BEB, and strengthening leader proposals to single-shot PBFT instances; it chains these instances via a ticket-based design akin to Motorway but is subject to up to 2x sequential consensus latency (§5.4).

DAG vs Lanes. Lanes in Motorway loosely resemble the DAG structure, but minimize quorum sizes and eschew rigid dependencies across proposers. DAGs must operate at the proposal rate of the slowest correct replica, while Motorway's independent lanes allow for flexible proposal rates (each replica can propose at their load and resource capability); this is desirable in modern permissioned blockchains, where replicas might be operated by heterogeneous entities (with varying resource capacities). Notably, Motorway 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 orphaning some proposals [45].

7.2 Asynchronous BFT consensus

Asynchronous BFT [6, 20, 29, 37, 40] protocols sidestep the FLP impossibility result [27] and guarantee liveness even during asynchrony by introducing randomization. They typically use a combination of reliable broadcast (RBC) [13] and asynchronous Byzantine Agreement, either Binary BBA [5, 13, 15, 41] or Multi-Valued (MVBA) [6, 14], to implement asynchronous Byzantine atomic broadcast (BAB) [23, 28, 40]. Unfortunately, these protocols require expensive cryptography and tens of rounds of message exchanges, resulting in impractical latency.

The asynchronous systems closest related to Motorway are Honeybadger [40] and Dumbo-NG [28]. Honeybadger [40] achieves high throughput by operating n instances of Reliable Broadcast (RB) in parallel, and trades off latency for large batch sizes; it uses n BBA instances to select $n-f$ proposals that may commit. Dumbo-NG [28] separates data dissemination from consensus to ameliorate batching/latency trade-offs. It constructs parallel chains of RB and layers a sequential multi-shot MVBA protocol atop; this architecture closely resembles the spirit of Motorway, but incurs significantly higher latency.

Tusk [20] avoids blackbox BBA/MVBA protocols, and instead constructs a DAG using Narwhal [20] whose structure

[7]Concurrent work with Motorway.

it interprets to derive a common coin; this requires an expected 4.5 rounds of RB.

Ditto [29] and Bullshark [45] improve the common-case performance of asynchronous protocols by, respectively, augmenting a 2-round-Hotstuff variant to use MVBA only as a fallback, and introducing a partially synchronous fast path (2 rounds of RB) to Tusk. However, these protocols must explicitly switch between modes of operation (to ensure that at most one commits), and may not gracefully respond to changing network conditions. Abraxas [11] and ParBFT [19] improve the performance of Ditto by eagerly running both a partially synchronous fast path and asynchronous slow path in parallel.

MyTumbler [37][7] targets low latency for heterogeneous participants by leveraging a combination of Timestamp-based parallel consensus, and BA with a fast path for good intervals (SuperMA). Like Motorway, it allows for flexible proposal speeds, but requires trading off latency and throughput by tuning the pace at which consensus terminates; notably, each proposal requires its own consensus instance, while Motorway requires only one per cut. Motorway could, like SuperMA, be augmented with an asynchronous fallback—we leave such exploration to future work.

8 Conclusion

This work presents Motorway, a novel partially synchronous BFT consensus protocol that is efficient under *partial synchrony*. We introduce the dual notions of an *asynchrony hangover*, and a *seamless* partially synchronous protocol to characterize the performance of partially synchronous protocols in practice. Empiric evaluation shows that Motorway achieves seamlessness, while simultaneously matching the throughput of DAG-based BFT protocols and reducing latency to that of traditional BFT protocols. Drive safe!

Acknowledgement

We thank Alexander Spiegelman for an insightful discussion on leveraging reputation schemes to reduce latency.

References

[1] [n. d.]. https://cloud.google.com/compute/docs/general-purpose-machines#t2d_machines

[2] [n. d.]. Dalek elliptic curve cryptography. https://github.com/dalek-cryptography/ed25519-dalek.

[3] [n. d.]. RocksDB, version 0.16.0. https://rocksdb.org/.

[4] [n. d.]. Tokio, version 1.5.0. https://tokio.rs/.

[5] 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.

[6] 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

[7] 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

[8] Ittai Abraham and Alexander Spiegelman. 2022. Provable Broadcast. https://decentralizedthoughts.github.io/2022-09-10-provable-broadcast/.

[9] 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.

[10] 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.

[11] Erica Blum, Jonathan Katz, Julian Loss, Kartik Nayak, and Simon Ochsenreither. 2023. Abraxas: Throughput-Efficient Hybrid Asynchronous Consensus. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (<conf-loc>, <city>Copenhagen</city>, <country>Denmark</country>, </conf-loc>) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 519–533. https://doi.org/10.1145/3576915.3623191

[12] 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.

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

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

[15] 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.

[16] 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.

[17] 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.

[18] Shir Cohen, Rati Gelashvili, Lefteris Kokoris Kogias, Zekun Li, Dahlia Malkhi, Alberto Sonnino, and Alexander Spiegelman. 2022. Be Aware of Your Leaders. 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, 279–295. https://doi.org/10.1007/978-3-031-18283-9_13

[19] Xiaohai Dai, Bolin Zhang, Hai Jin, and Ling Ren. 2023. ParBFT: Faster Asynchronous BFT Consensus with a Parallel Optimistic Path. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (<conf-loc>, <city>Copenhagen</city>, <country>Denmark</country>, </conf-loc>) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 504–518. https://doi.org/10.1145/3576915.3623101

[20] 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.

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

[22] 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.

[23] 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

[24] 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.

[25] Novi Facebook Research. [n.d.]. Hotstuff Implementation. https://github.com/asonnino/hotstuff/commit/d771d4868db301bcb5e3deaa915b5017220463f6.

[26] Novi Facebook Research. [n.d.]. Narwahl and Bullshark implementation. https://github.com/asonnino/narwhal.

[27] 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.

[28] 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.

[29] 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

[30] 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.

[31] 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

[32] 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.

[33] 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

[34] 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.

[35] 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.

[36] 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

[37] 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.

[38] Dahlia Malkhi, Chrysoula Stathakopoulou, and Maofan Yin. 2023. BBCA-CHAIN: One-Message, Low Latency BFT Consensus on a DAG. *arXiv preprint arXiv:2310.06335* (2023).

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

[40] 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

[41] 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.

[42] 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.

[43] 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.

[44] Alexander Spiegelman, Balaji Aurn, Rati Gelashvili, and Zekun Li. 2023. Shoal: Improving DAG-BFT Latency And Robustness. *arXiv preprint arXiv:2306.03058* (2023).

[45] 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

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

[47] 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).

[48] 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.

[49] 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.

[50] 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.

[51] 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

A Proofs

A.1 Safety

We show that all correct replicas commit, for every slot, the same proposal. It necessarily follows, that all correct replicas compute a consistent log. In the following, we prove agreement for a given slot s, and drop the slot number for convenience.

We first prove that, within a view v, no two conflicting proposals can be committed.

Lemma 1. *If there exists a* PREPAREQC *or* COMMITQC *for consensus proposal p in view v, then there cannot exist a* PREPAREQC *or a* COMMITQC *for consensus proposal $p' \neq p$ in view v.*

Proof. Suppose for the sake of contradiction there exists a PREPAREQC or COMMITQC for consensus proposal $p' \neq p$ in view v. There must exist at least $n-f$ replicas who sent a PREP-VOTE for p' in view v. At least $n-f$ replicas sent a PREP-VOTE for proposal p in view v. These two quorums intersect in at least 1 correct replica, a contradiction since an correct replica only sends one PREP-VOTE message in a view. ☐

Lemma 2. *If an correct replica commits a (consensus) proposal p in view v, the no correct replica commits a proposal $p' \neq p$ in view v.*

Proof. Suppose for the sake of contradiction an correct replica commits proposal $p' \neq p$ in view v. There must exist a fast COMMITQC or a PREPAREQC for p' in view v. By lemma 1, however, any PREPAREQC or fast COMMITQC in view v must be for proposal p. ☐

Next, we show that if a proposal committed in some view v, no other proposal can be committed in a future view.

Lemma 3. *If an correct replica commits a consensus proposal p in view v, then any valid* PREPARE *for view $v' > v$ must contain proposal p.*

Proof. We prove by induction on view v'.

Base case: Let $v' = v+1$. Suppose for the sake of contradiction there exists a valid PREPARE for a proposal $p' \neq p$ in view v'. There are two cases: either 1) p was committed in view v on the fast path or 2) p was committed in view v on the slow path.

1) Fast path. Since p was committed on the fast path there must exist n PREP-VOTE messages for p in view v, of which at least $n-f$ correct replicas stored a PREPARE for proposal p locally. Any TC in view v contains $n-f$ TIMEOUT messages, and by quorum intersection at least $n-2f$ of these TIMEOUT messages must contain a PREPARE message for proposal p. Since a TIMEOUT message can contain at most 1 PREPARE message, there cannot exist $n-2f$ PREPARE messages for p' in view v. By lemma 1, any PREPAREQC or COMMITQC in view v must be for p. Therefore any PREPAREQC, COMMITQC, or set of $n-2f$ PREPARE messages for p' must be in a view $< v$. Therefore, the winning proposal for any TC in view v must be p, a contradiction since by the assumption there was a valid PREPARE containing a valid TC for proposal p'.

2) Slow path. Since p was committed on the slow path there must exist $n-f$ CONFIRM-ACK messages (COMMITQC) for p in view v, of which at least $n-2f$ correct replicas stored a PREPAREQC for proposal p locally. Any TC in view v contains $n-f$ TIMEOUT messages, and by quorum intersection at least 1 of these TIMEOUT messages must contain a PREPAREQC for proposal p. By lemma 1, any prepare or COMMITQC in view v must be for p. Therefore any PREPAREQC or COMMITQC for p' must be in a view $< v$. Therefore, the winning proposal for any TC in view v must be p since even if there does exist $f+1$ PREPARE for p' in view v, a prepare/COMMITQC takes precedence. This, however, is a contradiction since by the assumption there was a valid PREPARE in view v' containing a valid TC with a winning proposal p'.

Induction Step: We assume the lemma holds for all views $v'-1 > v$, and now prove it holds for view v'. Suppose for the sake of contradiction there exists a valid PREPARE for a proposal $p' \neq p$ in view v'. Since $v' > 0$, this PREPARE message must contain a TC for view $v'-1$. This TC consists of $n-f$ TIMEOUT messages, for which the winning proposal is p'. This means that the highest (by view) PREPAREQC or set of $n-2f$ matching PREPARE messages must be for p'. By the base case and induction step any valid PREPARE (and thereby PREPAREQC) in views $> v$ must be for proposal p. There are two cases: either p was committed on the fast path or on the slow path in view v.

Fast Path. Since p was committed on the fast path, the only PREPARE message all correct replicas voted for in view v was for proposal p, so there can exist at most f TIMEOUT messages containing a PREPARE in view v for proposal p'. Thus, any PREPAREQC or set of $n-2f$ PREPARE for proposal p' must be in a view $< v$. Since p, was committed on the fast path, there exists at least $n-f$ replicas who updated their highest PREPARE to be PREPARE for proposal p in view v. By quorum intersection, any TC must contain at least $n-2f$ of these messages, a contradiction since the winning proposal is for p'.

Slow Path. By lemma 1, any PREPAREQC in view v must be for proposal p. Thus, any PREPAREQC must be in a view $< v$. Since p was committed on the slow path, there must exist at least $n-f$ replicas who voted for a PREPAREQC for proposal p in view v.By quorum intersection, any TC must contain at least 1 TIMEOUT message from this set of replicas, which will contain a PREPAREQC with view at least v. By the induction assumption any PREPARE message with view $> v$ must be for proposal p. Any ties in the view between a PREPAREQC and a set of $f+1$ PREPARE is given to the PREPAREQC, so the winning proposal must be p for any TC, a contradiction. □

Lemma 4. *If an correct replica commits a consensus proposal in slot s, then no correct replica commits a different consensus proposal in slot s.*

Proof. Let view v be the earliest view in which an correct replica commits a proposal p. Suppose for the sake of contradiction an correct replica commits a proposal $p' \neq p$ in view v'. There are two cases: 1) $v' = v$ and 2) $v' > v$. **Case 1.** By lemma 2 no correct replica will commit a proposal $p' \neq p$ in view v, a contradiction. **Case 2.** Since p' is committed by an correct replica, there must exist a COMMITQC or a fast COMMITQC for p' in view v', which implies there must exist a valid PREPARE message for p' in view v'. By lemma 3 any valid PREPARE message in view $v' > v$ must be for proposal p, a contradiction. □

Finally, we show that all correct replicas arrive at a consistent log.

Theorem 1. *All correct replicas commit the same requests in the same order.*

Proof. Correct replicas order all consensus proposals by increasing slot numbers. By lemma 4, for any slot s, correct replicas commit the same consensus proposal in slot s. By the collision resistance property of hash functions, all correct replicas will agree on the same history for each tip within a consensus proposal. And since all correct replicas use the same deterministic function that resolves forks in Byzantine lanes, zips the data, and constructs an ordering within a slot, all correct replicas will order the same blocks in the same order. □

A.2 Liveness

Next, we prove that all correct clients' requests will eventually be committed. We assume a continuous input stream of client requests to all correct replicas (and thus that all data lanes progress). We first prove that Motorway consensus itself is guaranteed to make progress given synchrony, and then show that every correct replica's data proposal will eventually be committed.

To do so, we lay some brief groundwork. We define GST to be the (unknown) time at which the network becomes synchronous, and Δ to be the (known) upper bound on message delivery time after GST. For simplicity we assume synchrony lasts forever after GST[8]. We set a replica's local consensus timeout value for any slot s to 10Δ; this, we show, suffices to guarantee commitment in good intervals.

We begin by showing that every correct replica will eventually commit a slot s

To do so, we first prove that after GST, a correct leader will take a bounded amount of time to send a PREPARE message. For any view $v > 0$, this is trivial but for $v = 0$ it may take

[8]In practice, synchrony is of course finite, and only needs to be "sufficiently" long.

time for a correct leader to satisfy the coverage rule. We show this takes at most 2Δ after GST.

Lemma 5. *If slot s, view v, starts after GST and is led by an correct leader, it will take at most* 2Δ *time after receiving a ticket to send a* PREPARE *message in slot s, view v.*

Proof. There are two cases: 1) $v=0$ or 2) $v>0$.

Case 1. For view 0, a correct leader must have received a PREPARE message in slot $s-1$ containing a proposal with n tips, of which at least $n-f$ are from correct replicas. A proof of availability for correct tips will take at most Δ time to form after GST, and the next tip at the next greater height will take at most Δ time to reach all replicas. Therefore, by 2Δ, the leader will receive at least $n-f$ new tips, satisfying the coverage rule to propose a new cut.

Case 2. For view $v>0$, a correct leader must have received a TC for view $v-1$ in slot s. If the winning proposal for a TC is $\neq \perp$, then the leader will immediately send a PREPARE message for slot s, view v, containing the winning proposal (and the TC). Otherwise if the winning proposal is $\perp$, the leader will immediately send a PREPARE message containing its own local set of n tips as the consensus proposal. Therefore, it takes at most 2Δ for a correct leader to send a PREPARE message in slot s, view v. □

Next, we show that all correct replicas will reliably enter a common view, in bounded time. This ensures that all correct replicas will accept messages from the leader, and respond accordingly.

Lemma 6. *For any slot s, let v be a view after GST with an correct leader, and t be the time the first correct replica enters v. All correct replicas will enter v by time* $t+4\Delta$*.*

Proof. The earliest correct replica to enter view v in slot s must have received a PREPARE message in slot $s-1$ or a TC in slot s, view $v-1$ (ticket). It will then forward this ticket to the leader of slot s, view v, which will arrive at the leader within Δ time. If $v=0$, by lemma 5 the leader's proposal will take at most 2Δ time to form. Otherwise if $v>0$, then the leader And it then sends a PREPARE message for slot s, which arrives at all correct replicas within Δ time, after which all correct replicas enter view v. □

Recall, that replicas only vote for a consensus proposal (PREPARE) if they can locally assert to the availability of all data. This is trivial when proposing only cerified tips (no synchronization is necessary), but when proposing optimistic tips, replicas may need to first fetch missing data. We show that this takes only a bounded amount of time.

Lemma 7. *If an correct replica receives a* PREPARE *message in slot s, view v, from an correct leader after GST, it will take at most* 2Δ *before it sends a* PREP-VOTE *message.*

Proof. An correct replica will only vote for a PREPARE message if it can assert the availability of every tip (and it's history) in the consensus proposal. If the tip is certified (POA), no further work is necessary, and the replica can vote immediately. If it instead is not certified (optimistic) a replica must ensure that it has already received the associated data proposal. If it has, it may vote immediately. Otherwise, a replica will send to the leader a SYNC message, requesting all missing optimistic tips, which will arrive at the leader by Δ time. A correct leader, by design, will only propose optimistic tips for which it has all the associated data proposal; thus it will reply to a sync request with the missing data proposal (batch + parent POA), which takes at most Δ time to arrive. Notably, no sync is necessary for ancestors of the respective tips, as the parent POA transitively proves it's availability. □

Next, we show that all correct replicas will commit in a view led by a correct leader.

Lemma 8. *If there is some view v in slot s that is led by a correct leader after GST, then all correct replicas commit the correct leader's proposal in view v.*

Proof. By lemma 6, all correct replicas will receive a PREPARE message in slot s, view v', and enter view v' within 4Δ of the first correct replica entering v'. By lemma 7, all correct replicas will take at most 2Δ time to send a PREP-VOTE to the leader, who will receive these votes by Δ time and form a PREPAREQC from these $n-f$ votes (or a fast COMMITQC if it receives n votes. The leader will then send a CONFIRM message if it formed a PREPAREQC or a COMMIT message if it formed a fast COMMITQC to all replicas, which will arrive within Δ time. If an correct replica receives a valid fast COMMITQC, it will commit. Otherwise, correct replicas will send a CONF-VOTE to the leader, which will arrive at the leader by Δ time. The leader will form a COMMITQC from $n-f$ CONF-VOTE messages, and then send a COMMIT message containing the COMMITQC to all correct replicas, who will receive it by Δ time. Upon receiving a COMMIT message, an correct replica will commit. Since the local timeout is 10Δ, all correct replicas will commit in view v' before timing out. □

Using the above lemma, we show that every correct replica will eventually commit in slot s.

Lemma 9. *Let view v in slot s start after GST. Every correct replica eventually commits a proposal in a view* $v' \geq v$ *in slot s.*

Proof. Since the number of Byzantine replicas is bounded by f, there exists some view $v' \geq v$ in slot s, such that v' is led by an correct leader. By lemma 8, all correct replicas will commit the correct leader's proposal in view v'. □

To construct the log, a replica must be able to synchronize on all data proposals subsumed by a consensus proposal cut. We show that it is not only guaranteed to be able to fetch all data, but will do so in constant time, and before it locally commits s.

Lemma 10. *For any committed proposal, an correct replica will be able to retrieve all data contained in that proposal.*

Proof. For an optimistic tip, an correct replica does not vote unless it receives the data locally for that tip. Since an correct replica only commits upon receiving n PREP-VOTE messages (fast COMMITQC) or $n - f$ CONFIRM-ACK messages (COMMITQC), there must exist at least $f+1$ correct replicas who have the corresponding data stored for an optimistic tip. Certified tips and cars have a proof of availability, consisting of at least $f+1$ VOTE messages, of which at least 1 must have been from an correct replica. An correct replica respects FIFO voting, therefore it must have stored and voted for all cars in a certified tip's history. Thus, for an optimistic tip, a replica can ask to synchronize with the quorum of replicas who formed a COMMITQC to retrieve the data. Upon receiving the data from an optimistic tip it can synchronize with replicas who certified the parent of the optimistic tip to retrieve all history (and therefore a subset for the deterministic ordering function). For certified tips, the replica can directly synchronize with the quorum of replicas who voted for the tip to retrieve the entire history (and therefore a subset for the deterministic ordering function). In both cases, all data is retrieved. □

Lemma 11. *For any committed proposal, an correct replica will be able to synchronize on missing data in constant time, and before (by the time) it commits the proposal after GST.*

Proof. If an correct replica is missing data for an optimistic tip, then by lemma 7 it will take at most 2Δ to retrieve the data. If it is missing data for a certified tip or any ancestor of an optimistic tip, then it will send a SYNC message to quorum of the $f+1$ replicas who voted to create the proof of availability (PoA), of which at least 1 must be correct. This SYNC message will arrive at this correct replica Δ time afterwards. Since correct replicas respect FIFO voting, that is they don't vote for a data lane proposal until they voted for all ancestors of the data lane proposal, this correct replica must have the data for all ancestors for the missing tip. It will then send a SYNC-REPLY message to the requesting replica containing all data for the missing tip and its ancestors. This message will arrive at the correct replica within another Δ time, taking in total 2Δ for an correct replica to synchronize. Since an correct replica starts synchronizing upon receiving a PREPARE message, and commits at the earliest (via fast path) after 2 more message delays, all correct replicas will synchronize on the data by the time consensus finishes.

□

Finally, we show that all data proposals from correct data lanes are eventually committed, and thus all correct client requests are eventually committed:

Lemma 12. *If a correct replica receives a client request, all correct replicas will eventually add a tip containing the client request to its local set of tips.*

Proof. When a correct replica receives a client request it will add it to the new tip of its data lane. Before proposing this tip, a correct replica must wait for a proof of availability (PoA) to form for the parent of this tip. A correct replica's tip will eventually reach all correct replica, $f+1$ correct replicas will vote for it, which will eventually be received, forming a PoA. Upon the formation of a PoA, a correct replica will send its latest tip (along with the PoA) to all replicas, which will eventually arrive at all correct replicas, at which point a correct replica will add this tip to its local set of tips. □

Lemma 13. *Given a continuous stream of client requests, a correct replica's lane will grow infinitely.*

Proof. From lemma 12, correct replicas will disseminate tips and proofs of availability (PoA) whenever they receive client requests. Since we assume client requests to be sent infinitely often, correct replicas' lanes will continuously grow infinitely often. □

Lemma 14. *Any correct replica's lane will be committed infinitely often.*

Proof. If an correct replica is the leader of view 0 for a given slot, it will propose tips for all n replicas. Since there exists an infinite number of slots that have not begun, where correct replicas are the leader of view 0 for that slot after GST, and a correct leader's proposal will be committed after GST by lemma 8, any correct replica's lane will be committed infinitely often. □

Theorem 2. *Eventually any client request will be ordered.*

Proof. We assume that each client request will be resent to a different replica if it has not received a response after some timeout. Eventually, some correct replica, h, will receive the client request. By lemma 12, all correct replicas will eventually receive this tip. Since an correct replica never forks, any future tip in an correct replica's lane must extend this car. By lemma 14, this lane will eventually be committed in some slot s, thereby committing all cars including the one that contains the client request. Again by lemma 9, eventually a proposal will be committed in all slots smaller than s, finalizing slot s. Applying lemma 10 for every slot from 1 to s, the data for every proposal (including all ancestors of a tip) will be retrieved, processed (forks resolved), and ordered, including the client request. □