Morpheus Consensus: Excelling on trails and autobahns

Andrew Lewis-Pye
London School of Economics, UK

Ehud Shapiro
Weizmann Institute of Science, Israel

Abstract

Recent research in consensus has often focussed on protocols for State-Machine-Replication (SMR) that can handle high throughputs. Such state-of-the-art protocols (generally DAG-based) induce undue overhead when the needed throughput is low, or else exhibit unnecessarily-poor latency and communication complexity during periods of low throughput.

Here we present Morpheus Consensus, which naturally morphs from a quiescent low-throughput leaderless blockchain protocol to a high-throughput leader-based DAG protocol and back, excelling in latency and complexity in both settings. During high-throughout, Morpheus pars with state-of-the-art DAG-based protocols, including Autobahn [15]. During low-throughput, Morpheus exhibits competitive complexity and lower latency than standard protocols such as PBFT [10] and Tendermint [8, 9], which in turn do not perform well during high-throughput.

The key idea of Morpheus is that as long as blocks do not conflict (due to Byzantine behaviour, network delays, or high-throughput simultaneous production) it produces a forkless blockchain, promptly finalizing each block upon arrival. It assigns a leader only if one is needed to resolve conflicts, in a manner and with performance not unlike Autobahn.

2012 ACM Subject Classification Computer systems organization → Dependable and fault-tolerant systems and networks

Keywords and phrases Distributed computing, consensus, quiescence

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2025.35

Related Version *Full Version*: `https://arxiv.org/abs/2502.08465`

1 Introduction

Significant investment in blockchain technology has recently led to renewed interest in research on consensus protocols. Much of this research is focussed on developing protocols that operate efficiently 'at scale'. In concrete terms, this means looking to design protocols that can handle a high throughput (i.e. high rate of incoming transactions) with low latency (i.e. quick transaction finalization), even when the number of processes (validators) carrying out the protocol is large.

Dealing efficiently with low and high throughput. While blockchains may often need to handle high throughputs, it is not the case that *all* blockchains need to deal with high throughput *all* of the time. For example, various 'subnets' or 'subchains' may only have to deal with high throughputs infrequently, and should ideally be optimised to deal also with periods of low throughput. The motivation for the present paper therefore stems from a real-world need for consensus protocols that deal efficiently with both high and low throughputs. Specifically, we are interested in a setting where:

29th International Conference on Principles of Distributed Systems (OPODIS 2025).
Editors: Andrei Arusoaie, Emanuel Onica, Michael Spear, and Sara Tucci-Piergiovanni; Article No. 35; pp. 35:1–35:20
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1. The processes/validators may be few, but could be up to a few hundred in number.
2. The protocol should be able to handle periods of asynchrony, i.e. should operate efficiently in the partially synchronous setting.
3. The protocol is required to have optimal resilience against Byzantine adversaries, i.e., should be live and consistent so long as less than 1/3 of processes display Byzantine faults, but should be optimised to deal with the 'normal case' that processes are not carrying out Byzantine attacks and that faults are benign (crash or omission failures).
4. There are expected to be some periods of high throughput, meaning that the protocol should ideally match the state-of-the-art during such periods.
5. Often, however, throughput will be low. This means the protocol should also be optimised to give the lowest possible latency during periods of low throughput.
6. Ideally, the protocol should be 'leaderless' during periods of low throughput: the use of leaders is to be avoided if possible, since, even without malicious action, leaders who are offline/faulty may cause significant increases in latency.
7. Ideally, the protocol should also be 'quiescent', i.e., there should be no need for the sending and storing of new messages when new transactions are not being produced.
8. Transactions may come from *clients* (not belonging to the list of processes/validators), but will generally be produced by the processes themselves.

The main contribution of this paper. We introduce and analyse the Morpheus protocol, which is designed for the setting described above. The protocol is quiescent and has the following properties during periods of low throughput:

- It is leaderless, in the sense that transactions are finalized without the requirement for involvement by leaders.
- Transactions are finalized in time 3δ, where δ is the actual (and unknown) bound on message delays after GST.[1] This more than halves the latency of existing DAG-based protocols and variants such as Autobahn [15] for the low throughput case, and even decreases latency by at least δ when compared with protocols such as PBFT (and even if we suppose leaders for those protocols are non-faulty), since the leaderless property of our protocol negates the need to send transactions to a leader before they can be included in a block.[2]
- A further advantage over protocols such as PBFT and Tendermint is that crash failures by leaders are not able to impact latency during periods of low throughput.

During periods of high throughput, Morpheus is very similar to Autobahn, and so inherits the benefits of that protocol. In particular:

- It has the same capability to deal with high throughput as DAG-based protocols and variants such as Autobahn, and has the same ability to recover quickly from periods of asynchrony ('seamless recover' in the language of Autobahn).
- It has the same latency as Autobahn during high throughput, matching the latency of Sailfish [24], which is the most competitive existing DAG-based protocol in terms of latency.

[1] The partially synchronous setting and associated notions such as GST are formally defined in Section 2.
[2] See the online version of the paper at `https://arxiv.org/abs/2502.08465` for a detailed analysis of latency and complexity considerations.

- Morpheus has the same advantages as Autobahn in terms of communication complexity when compared to DAG-based protocols such as Sailfish, DAG-Rider [17], Cordial Miners [19], Mysticeti [3] or Shoal [26].

Of course, much of the complexity in designing a protocol that operates efficiently in both low and high throughput settings is to ensure a smooth transition and consistency between the different modes of operation that the two settings necessitate.

Further contributions of the paper. In Section 3, we also formalise the task of *Extractable SMR*, as an attempt to make explicit certain implicit assumptions that are often made by papers in the area. While State-Machine-Replication (SMR) requires correct processes to finalize *logs* (sequences of transactions) in such a way that *consistency* and *liveness* are satisfied, it is well understood in the community that some papers describing protocols for SMR specify protocols that *do not* actually aim to explicitly ensure all correct processes receive all finalized blocks (required for liveness). Roughly, the protocol instructions suffice instead to ensure *data availability* (that each finalized block is received by at least one correct process), and then the protocol is required to establish a total ordering on transactions that can be extracted via further message exchange, given data availability. Liveness is therefore only achieved after further message exchange (and via some unspecified method), which (while a trivial addition if one does not consider communication complexity) is not generally taken into account when calculating message complexity.

In Hotstuff [29], for example, one of the principal aims is to ensure linear message complexity within *views*. Since this precludes all-to-all communication within views, a Byzantine leader may finalize a block of transactions in a given view without certain correct processes even receiving the block. Those correct processes must eventually receive the block for liveness to be satisfied, but the protocol instructions do not explicitly stipulate the mechanism by which this should be achieved. While ensuring that all correct processes receive the block is trivial if one is not concerned with communication complexity (e.g., just have each correct process broadcast each finalized block they observe, together with a quorum certificate verifying that the block is finalized), the messages required to do so are not counted when analyzing message complexity. The obvious methods of ensuring that the block is propagated to all correct processes will require more than linear communication complexity, which undermines the very point of the Hotstuff protocol.

The question arises, "what precisely is the task being achieved by such protocols if they do not satisfy liveness without further message exchange (and so actually fail to achieve the task of SMR with the communication complexity computed)". We assert that the task of Extractable SMR is an appropriate formalisation of the task being achieved, and hope that the introduction of this notion is a contribution of independent interest.

The structure of the paper. The paper structure is as follows: Section 2 describes the basic model and definitions; Section 3 formalises the task of Extractable SMR; Section 4 gives the intuition behind the Morpheus protocol; Section 5 gives the formal specification of the protocol; Appendix A formally establishes consistency and liveness; Appendix B discusses related work. See the online version of the paper at `https://arxiv.org/abs/2502.08465` for a detailed analysis of latency and complexity considerations.

2 The setup

We consider a set $\Pi = \{p_0, \ldots, p_{n-1}\}$ of n processes. Each process p_i is told i as part of its input. We consider an adaptive adversary, which chooses a set of at most f processes to corrupt during the execution, where f is the largest integer less than $n/3$. A process that is corrupted by the adversary is referred to as *Byzantine* and may behave arbitrarily, subject to our cryptographic assumptions (stated below). Processes that are not Byzantine are *correct*.

Cryptographic assumptions. Our cryptographic assumptions are standard for papers in distributed computing. Processes communicate by point-to-point authenticated channels. We use a cryptographic signature scheme, a public key infrastructure (PKI) to validate signatures, a threshold signature scheme [6, 23], and a cryptographic hash function H. The threshold signature scheme is used to create a compact signature of m-of-n processes, as in other consensus protocols [30]. In this paper, $m = n - f$ or $m = f + 1$. The size of a threshold signature is $O(\kappa)$, where κ is a security parameter, and does not depend on m or n. We assume a computationally bounded adversary. Following a common standard in distributed computing and for simplicity of presentation (to avoid the analysis of negligible error probabilities), we assume these cryptographic schemes are perfect, i.e., we restrict attention to executions in which the adversary is unable to break these cryptographic schemes. Hash values are thus assumed to be unique.

Message delays. We consider a discrete sequence of timeslots $t \in \mathbb{N}_{\geq 0}$ in the partially synchronous setting: for some known bound Δ and unknown *Global Stabilization Time* (GST), a message sent at time t must arrive by time $\max\{\text{GST}, t\} + \Delta$. The adversary chooses GST and also message delivery times, subject to the constraints already specified. We write δ to denote the actual (unknown) bound on message delays after GST, noting that δ may be significantly less than the known bound Δ.

Clock synchronization. We do not suppose that the clocks of correct processes are synchronized. For the sake of simplicity, however, we do suppose that the clocks of correct processes all proceed in real time, i.e. if $t' > t$ then the local clock of correct p at time t' is $t' - t$ in advance of its value at time t. This assumption is made only for the sake of simplicity, and our arguments are easily adapted to deal with a setting in which there is a known upper bound on the difference between the clock speeds of correct processes after GST. We suppose all correct processes begin the protocol execution before GST. A correct process may begin the protocol execution with its local clock set to any value.

Transactions. Transactions are messages of a distinguished form. For the sake of simplicity, we consider a setup in which each process produces their own transactions, but one could also adapt the presentation to a setup in which transactions are produced by clients who may pass transactions to multiple processes.

3 Extractable SMR

Informal discussion. State-Machine-Replication (SMR) requires correct processes to finalize *logs* (sequences of transactions) in such a way that *consistency* and *liveness* are satisfied. As noted in Section 1, however, for many papers describing protocols for SMR, the explicit instructions of the protocol *do not* actually suffice to ensure liveness without further message exchange, potentially impacting calculations of message complexity and other measures. Roughly, the protocol instructions do not explicitly ensure that all correct processes receive all finalized blocks, but rather ensure *data availability* (that each finalized block is received by at least one correct process), and then the protocol is required to establish a total ordering on transactions that can be extracted via further message exchange, given data availability. Although it is clear that the protocol can be used to solve SMR given some (as yet unspecified) mechanism for message exchange, the protocol itself does not solve SMR.

So, what exactly is the task that the protocol solves?

Extractable SMR (formal definition). If σ and τ are strings, we write $\sigma \subseteq \tau$ to denote that σ is a prefix of τ. We say σ and τ are *compatible* if $\sigma \subseteq \tau$ or $\tau \subseteq \sigma$. If two strings are not compatible, they are *incompatible*. If σ is a sequence of transactions, we write $\texttt{tr} \in \sigma$ to denote that the transaction $\texttt{tr}$ belongs to the sequence σ.

If $\mathcal{P}$ is a protocol for *extractable SMR*, then it must specify a function $\mathcal{F}$ that maps any set of messages to a sequence of transactions. Let M^* be the set of all messages that are received by at least one (potentially Byzantine) process during the execution. For any timeslot t, let $M(t)$ be the set of all messages that are received by at least one correct process at a timeslot $\leq t$. We require the following conditions to hold:

Consistency. For any M_1 and M_2, if $M_1 \subseteq M_2 \subseteq M^*$, then $\mathcal{F}(M_1) \subseteq \mathcal{F}(M_2)$.

Liveness. If correct p produces the transaction $\texttt{tr}$, there must exist t such that $\texttt{tr} \in \mathcal{F}(M(t))$.

Note that consistency suffices to ensure that, for arbitrary $M_1, M_2 \subseteq M^*$, $\mathcal{F}(M_1)$ and $\mathcal{F}(M_2)$ are compatible. To see this, note that, by consistency, $\mathcal{F}(M_1) \subseteq \mathcal{F}(M_1 \cup M_2)$ and $\mathcal{F}(M_2) \subseteq \mathcal{F}(M_1 \cup M_2)$.

Converting protocols for Extractable SMR to protocols for SMR. In this paper, we focus on the task of Extractable SMR. One way to convert a protocol for Extractable SMR into a protocol for SMR is to assume the existence of a gossip network, in which each process has some (appropriately chosen) constant number of neighbors. Using standard results from graph theory ([5] Chapter 7), one can assume correct processes form a connected component: this assumption requires classifying some small number of disconnected processes that would otherwise be correct as Byzantine. If each correct process gossips each 'relevant' protocol message, then all such messages will eventually be received by all correct processes. Overall, this induces an extra communication cost per message which is only linear in n. Of course, other approaches are also possible, and in this paper we will remain agnostic as to the precise process by which SMR is achieved from Extractable SMR.

4 Morpheus: The intuition

In this section, we informally describe the intuition behind the protocol. The protocol may be described as 'DAG-based', in the sense that each block may *point to* more than one previous block via the use of hash pointers. The blocks *observed* by any block b are b and all those blocks observed by blocks that b points to. The set of blocks observed by b is denoted by $[b]$. If neither of b and b' observe each other, then these two blocks are said to *conflict*. Blocks will be of three kinds: there exists a unique *genesis* block b_g (which observes only itself), and all other blocks are either *transaction* blocks or *leader* blocks.

The operation during low throughput. Roughly, by the 'low throughput mode', we mean a setting in which processes produce blocks of transactions infrequently enough that correct processes agree on the order in which they are received, meaning that transaction blocks can be finalized individually upon arrival. Our aim is to describe a protocol that finalizes transaction blocks with low latency in this setting, and without the use of a leader: the use of leaders is to be avoided if possible, since leaders who are offline/faulty may cause significant increases in latency. The way in which Morpheus operates in this setting is simple:

1. Upon having a new transaction block b to issue, a process p_i will send b to all processes.
2. If they have not seen any blocks conflicting with b, other processes then send a *1-vote* for b to all processes.

3. Upon receiving $n-f$ 1-votes for b, and if they still have not seen any block conflicting with b, each correct process will send a *2-vote* for b to all others.
4. Upon receiving $n-f$ 2-votes for b, a process regards b as finalized.

Recall that δ is the actual (unknown) bound on message delays after GST. If the new transaction block b is created at time $t >$GST, then the procedure above causes all correct processes to regard b as finalized by time $t+3\delta$.

Which blocks should a new transaction block b point to? For the sake of concreteness, let us specify that if there is a *sole tip* amongst the blocks received by p_i, i.e., if there exists a unique block b' amongst those received by p_i which observes all other blocks received by p_i, then p_i should have b point to b'. To integrate with our approach to the 'high throughput mode', we also require that b should point to the last transaction block created by p_i. Generally, we will only require transaction blocks to point to at most two previous blocks. This avoids the downside of many DAG-based protocols that all blocks require $O(n)$ pointers to previous blocks.

Moving to high throughput. When conflicting transaction blocks are produced, we need a method for ordering them. The approach we take is to use *leaders*, who produce a second type of block, called *leader* blocks. These leader blocks are used to specify the required total ordering.

Views. In more detail, the instructions for the protocol are divided into *views*, each with a distinct leader. If a particular view is operating in 'low throughput' mode and conflicting blocks are produced, then some time may pass during which a new transaction block fails to be finalized. In this case, correct processes will complain, by sending messages indicating that they wish to move to the next view. Once processes enter the next view, the leader of that view will then continue to produce leader blocks so long as the protocol remains in high throughput mode. Each of these leader blocks will point to all *tips* (i.e. all blocks which are not observed by any others) seen by the leader, and will suffice to specify a total ordering on the blocks they observe.

The two phases of a view. Each view is thus of potentially unbounded length and consists of two phases. During the first phase, the protocol is in high throughput mode, and is essentially the same as Autobahn.[3] Processes produce transaction blocks, each of which just points to their last produced transaction block. Processes do not send 1 or 2-votes for transaction blocks during this phase, but rather vote for leader blocks, which, when finalized, suffice to specify the required total ordering on transactions. Leader blocks are finalized as in PBFT, after two rounds of voting. If a time is reached after which transaction blocks arrive infrequently enough that leader blocks are no longer required, then the view enters a second phase, during which processes vote on transaction blocks and attempt to finalize them without the use of a leader.

How to produce the total ordering. For protocols in which each block points to a single precedessor, the total ordering of transactions specified by a finalized block b is clear: the ordering on transactions is just that inherited by the sequence of blocks below b and the transactions they contain. In a context where each block may point to multiple others, however, we have extra work to do to specify the required total ordering on transactions.

[3] We note that Autobahn includes the option of various optimisations (with corresponding tradeoffs) that can be used to further reduce latency in certain 'good' scenarios (where all processes act correctly, for example). For the sake of simplicity we do not include these optimisations in our formal description of Morpheus in Section 5, but the online version of this paper discusses those options.

The approach we take is similar to many DAG-based protocols (e.g. [19]). Given any sequence of blocks S, we let $\mathtt{Tr}(S)$ be the corresponding sequence of transactions, i.e. if $b_1, \dots, b_k$ is the subsequence of S consisting of the transaction blocks in S, then $\mathtt{Tr}(S)$ is $b_1.\mathrm{Tr} * b_2.\mathrm{Tr} * \dots * b_k.\mathrm{Tr}$, where $*$ denotes concatenation, and where $b.\mathrm{Tr}$ is the sequence of transactions in b. We suppose given $\tau^\dagger$ such that, for any set of blocks B, $\tau^\dagger(B)$ is a sequence of blocks that contains each block in B precisely once, and which respects the observes relation: if $b, b' \in B$ and b' observes b, then b appears before b' in $\tau^\dagger(B)$. Each transaction/leader block b will contain q which is a 1-Quorum-Certificate (1-QC), i.e., a threshold signature formed from $n-f$ 1-votes, for some previous block: this will be recorded as the value b.1-QC $= q$, while, if q is a 1-QC for b', then we set $q.\mathrm{b} = b'$. QCs are ordered first by the view of the block to which they correspond, then by the type of the block (leader or transaction, with the latter being greater), and then by the height of the block. We then define $\tau(b)$ by recursion:

- $\tau(b_g) = b_g$.
- If $b \neq b_g$, then let $q = b$.1-QC and set $b' = q.\mathrm{b}$. Then $\tau(b) = \tau(b') * \tau^\dagger([b] - [b'])$.

Given any set of messages M, let M' be the largest set of blocks in M that is *downward closed*, i.e. such that if $b \in M'$ and b observes b', then $b' \in M'$. Let q be a maximal 2-QC in M such that $q.\mathrm{b} \in M'$, and set $b = q.\mathrm{b}$, or if there is no such 2-QC in M, set $b = b_g$. We define $\mathcal{F}(M)$ to be $\mathtt{Tr}(\tau(b))$.

Maintaining consistency. Consistency is formally established in Appendix A, and uses a combination of techniques from PBFT, Tendermint, and previous DAG-based protocols. Roughly, the argument is as follows. When the protocol moves to a new view, consistency will be maintained using the same technique as in PBFT. Upon entering the view, each process sends a 'new-view' message to the leader, specifying the greatest 1-QC they have seen. Upon producing a first leader block b for the view, the leader must then justify the choice of b.1-QC by listing new-view messages signed by $n-f$ distinct processes in Π. The value b.1-QC must be greater than or equal to all 1-QCs specified in those new-view messages. If any previous block b' has received a 2-QC, then at least $f+1$ correct processes must have seen a 1-QC for b', meaning that b.1-QC must be greater than or equal to that 1-QC. Subsequent leader blocks b'' for the view just set b''.1-QC to be a 1-QC for the previous leader block.

To maintain consistency between finalized transaction blocks and between leader and transaction blocks within a single view, we also have each transaction block specify q which is 1-QC for some previous block. Correct processes will not vote for the transaction block unless q is greater than or equal to any 1-QC they have previously received.

Overall, the result of these considerations is that, if two blocks b and b' receive 2-QCs q and q' respectively, with q greater than q', then the iteration specifying $\tau(b)$ (as detailed above) proceeds via b', so that $\tau(b)$ extends $\tau(b')$.

0-votes. While operating in low throughput, a 1-QC for a block b suffices to ensure both data availability, i.e. that some correct process has received the block, and *non-equivocation*, i.e. two conflicting blocks cannot both receive 1-QCs. When operating in high throughput, however, transaction blocks will not receive 1 or 2-votes. In this context, we still wish to ensure data availability. It is also useful to ensure that each individual process does not produce transaction blocks that conflict with each other, so as to bound the number of tips that may be created. To this end, we make use of *0-votes*, which may be regarded as weaker than standard votes for a block:

1. Upon having a new transaction block b to issue, a process p_i will send b to all processes.

2. If the block is properly formed, and if other processes have not seen p_i produce any transaction blocks conflicting with b, then they will send a *0-vote* for b back to p_i. Note that 0-votes are sent only to the block creator, rather than to all processes.
3. Upon receiving $n - f$ 0-votes for b, p_i will then form a 0-QC for b and send this to all processes.

When a block b' wishes to point to b, it will include a z-QC for b (for some $z \in \{0, 1, 2\}$). As a consequence, any process will be able to check that b' is valid/properly formed without actually receiving the blocks that b' points to: the existence of QCs for those blocks suffices to ensure that they are properly formed (and that at least one correct process has those blocks), and other requirements for the validity of b' can be checked by direct inspection. For this to work, votes (and QCs) must specify certain properties of the block beyond its hash, such as the height of the block and the block creator. The details are given in Section 5.

5 Morpheus: the formal specification

The pseudocode uses a number of local variables, functions, objects and procedures, detailed below. In what follows, we suppose that, when a correct process sends a message to 'all processes', it regards that message as immediately received by itself. All messages are signed by the sender. For any variable x, we write $x\downarrow$ to denote that x is defined, and $x\uparrow$ to denote that x is undefined. Table 1 lists all message types.

Message type	Description
Blocks	
Genesis block	Unique block of height 0
Transaction blocks	Contain transactions
Leader blocks	Used to totally order transaction blocks
Votes and QCs	
0-votes	Guarantee data availability and non-equivocation in high throughput
1-votes	Sent during 1st round of voting on a block
2-votes	Sent during 2nd round of voting on a block
z-QC, $z \in \{0, 1, 2\}$	formed from $n - f$ z-votes
View messages	
End-view messages	Indicate wish to enter next view
$(v + 1)$-certificate	Formed from $f + 1$ end-view v messages
View v message	Sent to the leader at start of view v

Table 1 Message types.

The genesis block. There exists a unique *genesis block*, denoted b_g. For any block b, b.type specifies the type of the block b, b.view is the *view* corresponding to the block, b.h specifies the *height* of the block, b.auth is the block creator, and b.slot specifies the *slot* corresponding to the block. For b_g, we set:

- b_g.type = gen, b_g.view = -1, b_g.h = 0, b_g.auth = $\perp$, b_g.slot = 0.

A comment on the use of slots. Each block will either be the genesis block, a *transaction* block, or a *leader* block. If $p_i \in \Pi$ is correct then, for $s \in \mathbb{N}_{\geq 0}$, p_i will produce a single transaction block b with b.slot $= s$ before producing any transaction block b' with b'.slot $= s+1$. Similarly, if $p_i \in \Pi$ is correct then, for $s \in \mathbb{N}_{\geq 0}$, p_i will produce a single leader block b with b.slot $= s$ before producing any leader block b' with b'.slot $= s+1$.

z-votes. For $z \in \{0,1,2\}$, a *z-vote* for the block b is a message of the form $(z, b.\text{type}, b.\text{view}, b.\text{h}, b.\text{auth}, b.\text{slot}, H(b))$, signed by some process in Π. The reason votes include more information than just the hash of the block is explained in Section 4. A *z-quorum* for b is a set of $n-f$ z-votes for b, each signed by a different process in Π. A *z-QC* for b is the message $m = (z, b.\text{type}, b.\text{view}, b.\text{h}, b.\text{auth}, b.\text{slot}, H(b))$ together with a threshold signature for m, formed from a z-quorum for b using the threshold signature scheme.

QCs. By a *QC* for the block b, we mean a z-QC for b, for some $z \in \{0,1,2\}$. If q is a z-QC for b, then we set q.b $= b$, q.z $= z$, q.type $= b$.type, q.view $= b$.view, q.h $= b$.h, q.auth $= b$.auth, q.slot $= b$.slot. We define a preordering $\leq$ on QCs as follows: QCs are preordered first by view, then by type with lead $<$ Tr, and then by height.[4]

The variable M_i. Each process p_i maintains a local variable M_i, which is automatically updated and specifies the set of all received messages. Initially, M_i contains b_g and a 1-QC for b_g.

Transaction blocks. Each *transaction block* b is entirely specified by the following values:

- b.type $=$ Tr, b.view $= v \in \mathbb{N}_{\geq 0}$, b.h $= h \in \mathbb{N}_{>0}$, b.slot $= s \in \mathbb{N}_{\geq 0}$.
- b.auth $\in \Pi$: the block creator.
- b.Tr: a sequence of transactions.
- b.prev: a non-empty set of QCs for blocks of height $< h$.
- b.1-QC: a 1-QC for a block of height $< h$.

If b.prev contains a QC for b', then we say that b *points to* b'. For b to be *valid*, we require that it is of the form above and:

1. b is signed by b.auth.
2. If $s > 0$, b points to b' with b'.type $=$ Tr, b'.auth $= b$.auth and b'.slot $= s-1$.
3. If b points to b', then b'.view $\leq b$.view.
4. If $h' = \max\{b'.\text{h} : \ b \text{ points to } b'\}$, then $h = h'+1$.

We suppose correct processes ignore transaction blocks that are not valid. In what follows we therefore adopt the convention that, by a ‘transaction block’, we mean a ‘valid transaction block’.

[4] For the sake of completeness, if q.view $= q'$.view, q.type $= q'$.type, and q.h $= q'$.h, then we set $q \leq q'$ and $q' \leq q$. We will show that, in this case, q.b $= q'$.b.

> *A comment on transaction blocks.* During periods of high throughput, a transaction block produced by p_i for slot s will just point to p_i's transaction block for slot $s-1$. During periods of low throughput, if there is a unique block b' received by p_i that does not conflict with any other block received by p_i, any transaction block b produced by p_i will also point to b' (so that b does not conflict with b').
> The use of b.1-QC is as follows: once correct p_i sees a 1-QC q, it will not vote for any transaction block b unless b.1-QC is greater than or equal to q. Ultimately, this will be used to argue that consistency is satisfied.

When blocks observe each other. The genesis block *observes* only itself. Any other block b observes itself and all those blocks observed by blocks that b points to. If two blocks do not observe each other, then they *conflict*. We write $[b]$ to denote the set of all blocks observed by b.

The leader of view v. The leader of view v, denoted $\texttt{lead}(v)$, is process p_i, where $i = v \bmod n$.

End-view messages. If process p_i sees insufficient progress during view v, it may send an end-view v message of the form (v), signed by p_i. By a *quorum* of end-view v messages, we mean a set of $f+1$ end-view v messages, each signed by a different process in Π. If p_i receives a quorum of end-view v messages before entering view $v+1$, it will combine them (using the threshold signature scheme) to form a $(v+1)$-certificate. Upon first seeing a $(v+1)$-certificate, p_i will send this certificate to all processes and enter view $v+1$. This ensures that, if some correct process is the first to enter view $v+1$ after GST, all correct processes enter that view (or a later view) within time Δ.

View v messages. When p_i enters view v, it will send to $\texttt{lead}(v)$ a view v message of the form (v, q), signed by p_i, where q is a maximal amongst 1-QCs seen by p_i. We say that q is the 1-QC *corresponding* to the view v message (v, q).

> *A comment on view v messages.* The use of view v messages is to carry out view changes in the same manner as PBFT. When producing the first leader block b of the view, the leader must include a set of $n-f$ view v messages, which act as a *justification* for the block proposal: the value b.1-QC must be greater than or equal all 1-QCs corresponding to those $n-f$ view v messages. For each subsequent leader block b' produced in the view, b'.1-QC must be a 1-QC for the previous leader block (i.e., that for the previous slot). The argument for consistency will thus employ some of the same methods as are used to argue consistency for PBFT.

Leader blocks. Each *leader block* b is entirely specified by the following values:

- b.type = lead, b.view $= v \in \mathbb{N}_{\geq 0}$, b.h $= h \in \mathbb{N}_{>0}$, b.slot $= s \in \mathbb{N}_{\geq 0}$.
- b.auth $\in \Pi$: the block creator.
- b.prev: a non-empty set of QCs for blocks of height $< h$.
- b.1-QC: a 1-QC for a block of height $< h$.
- b.just: a (possibly empty) set of view v messages.

As for transaction blocks, if b.prev contains a QC for b', then we say that b *points to* b'. For b to be *valid*, we require that it is of the form described above and:

1. b is signed by b.auth and b.auth $= \texttt{lead}(v)$.
2. If b points to b', then b'.view $\leq b$.view.

3. If $h' = \max\{b'.\text{h} : \ b \text{ points to } b'\}$, then $h = h' + 1$.
4. If $s > 0$, b points to a unique b^* with $b^*.\text{type} = \text{lead}$, $b^*.\text{auth} = b.\text{auth}$ and $b^*.\text{slot} = s - 1$.
5. If $s = 0$ or $b^*.\text{view} < v$, then $b.\text{just}$ contains $n - f$ view v messages, each signed by a different process in Π. This set of messages is called a *justification* for the block.
6. If $s = 0$ or $b^*.\text{view} < v$, then $b.$1-QC is greater than or equal to all 1-QCs corresponding to view v messages in $b.\text{just}$.
7. If $s > 0$ and $b^*.\text{view} = v$, then $b.$1-QC is a 1-QC for b^*.

As with transaction blocks, we suppose correct processes ignore leader blocks that are not valid. In what follows we therefore adopt the convention that, by a 'leader block', we mean a 'valid leader block'.

> *A comment on leader blocks.* The conditions for validity above are just those required to carry out a PBFT-style approach to view changes (as discussed previously). The first leader block of the view must include a justification for the block proposal (to guarantee consistency). Subsequent leader blocks in the view simply include a 1-QC for the previous leader block (i.e., that for the previous slot).

The variable Q_i. Each process p_i maintains a local variable Q_i, which is automatically updated and, for each $z \in \{0,1,2\}$, stores at most one z-QC for each block: For $z \in \{0,1,2\}$, if p_i receives[5] a z-quorum or a z-QC for b, and if Q_i does not contain a z-QC for b, then p_i automatically enumerates a z-QC for b into Q_i (either the z-QC received, or one formed from the z-quorum received).

We define the 'observes' relation $\succeq$ on Q_i to be the minimal preordering satisfying (transitivity and):

- If $q, q' \in Q_i$, $q.\text{type} = q'.\text{type}$, $q.\text{auth} = q'.\text{auth}$ and $q.\text{slot} > q'.\text{slot}$, then $q \succeq q'$.
- If $q, q' \in Q_i$, $q.\text{type} = q'.\text{type}$, $q.\text{auth} = q'.\text{auth}$, $q.\text{slot} = q'.\text{slot}$, and $q.\text{z} \geq q'.\text{z}$, then $q \succeq q'$.
- If $q, q' \in Q_i$, $q.\text{b} = b$, $q'.\text{b} = b'$, $b \in M_i$ and b points to b', then $q \succeq q'$.

We note that the observes relation $\succeq$ depends on Q_i and M_i, and is stronger than the preordering $\geq$ we defined on z-QCs previously, in the following sense: if q and q' are z-QCs with $q \succeq q'$, then $q \geq q'$, while the converse may not hold. When we refer to the 'greatest' QC in a given set, or a 'maximal' QC in a given set, this is with reference to the $\geq$ preordering, unless explicitly stated otherwise. If $q.\text{type} = q'.\text{type}$, $q.\text{auth} = q'.\text{auth}$ and $q.\text{slot} = q'.\text{slot}$, then it will follow that $q.\text{b} = q'.\text{b}$.

> *A comment on the observes relation on Q_i.* When p_i receives $q, q' \in Q_i$, it may not be immediately apparent whether $q.\text{b}$ observes $q'.\text{b}$. The observes relation defined on Q_i above is essentially that part of the observes relation on blocks that p_i can testify to, given the messages it has received (while also distinguishing the 'level' of the QC).

The tips of Q_i. The *tips* of Q_i are those $q \in Q_i$ such that there does not exist $q' \in Q_i$ with $q' \succ q$ (i.e. $q' \succeq q$ and $q \not\succeq q'$). The protocol ensures that Q_i never contains more than $2n$ tips: The factor 2 here comes from the fact that leader blocks produced by correct p_i need not observe all transaction blocks produced by p_i (and vice versa).

[5] Here, we include the possibility that p_i receives the z-QC inside a message, such as in $b'.\text{prev}$ for a received block b'

Single tips. We say $q \in Q_i$ is a *single tip of* Q_i if $q \succeq q'$ for all $q' \in Q_i$. We say $b \in M_i$ is a *single tip of* M_i if there exists q which is a single tip of Q_i and b is the unique block in M_i pointing to q.b.

> *A comment on single tips.* When a transaction block is a single tip of M_i, this will enable p_i to send a 1-vote for the block. Leader blocks do not have to be single tips for correct processes to vote for them.

The `voted` function. For each $i, j, s, z \in \{0, 1, 2\}$ and $x \in \{\text{lead}, \text{Tr}\}$, the value $\texttt{voted}_i(z, x, s, p_j)$ is initially 0. When p_i sends a z-vote for a block b with b.type $= x$, b.auth $= p_j$, and b.slot $= s$, it sets $\texttt{voted}_i(z, x, s, p_j) := 1$. Once this value is set to 1, p_i will not send a z-vote for any block b' with b'.type $= x$, b'.auth $= p_j$, and b'.slot $= s$.

The phase during the view. For each i and v, the value $\texttt{phase}_i(v)$ is initially 0. Once p_i votes for a transaction block during view v, it will set $\texttt{phase}_i(v) := 1$, and will then not vote for leader blocks within view v.

> *A comment on the phase during a view.* As noted previously, each view can be thought of as consisting of two phases. Initially, the leader is responsible for finalizing transactions. If, after some time, the protocol enters a period of low throughput, then the leader will stop producing leader blocks, and transactions blocks can then be finalized directly. Once a process votes for a transaction block, it may be considered as having entered the low throughput phase of the view. The requirement that it should not then vote for subsequent leader blocks in the view is made so as to ensure consistency between finalized leader blocks and transaction blocks within the view.

When blocks are final. Process p_i regards $q \in Q_i$ (and q.b) as *final* if there exists $q' \in Q_i$ such that $q' \succeq q$ and q' is a 2-QC (for any block).

The function $\mathcal{F}$. This is defined exactly as specified in Section 4.

The variables $\texttt{view}_i$ and $\texttt{slot}_i(x)$ for $x \in \{\text{lead}, \text{Tr}\}$. These record the present view and slot numbers for p_i.

The $\texttt{PayloadReady}_i$ function. We remain agnostic as to how frequently processes should produce transaction blocks, i.e. as to whether processes should produce transaction blocks immediately upon having new transactions to process, or wait until they have a set of new transactions of at least a certain size. We suppose simply that:

- Extraneous to the explicit instructions of the protocol, $\texttt{PayloadReady}_i$ may be set to 1 at some timeslots of the execution.
- If $\texttt{PayloadReady}_i = 1$ and $\texttt{slot}_i(\text{Tr}) = s > 0$, then there exists $q \in Q_i$ with q.auth $= p_i$, q.type $=$ Tr and q.slot $= s - 1$.

> *A comment on the* $\texttt{PayloadReady}_i$ *function.* The second requirement above is required so that p_i can ensure that the new transaction block it forms can point to its transaction block for the previous slot.

The procedure $\texttt{MakeTrBlock}_i$. When p_i wishes to form a new transaction block b, it will run this procedure, by executing the following instructions:

1. Set b.type := Tr, b.auth := p_i, b.view := $\texttt{view}_i$, b.slot := $\texttt{slot}_i(\text{Tr})$.

2. Let $s := \texttt{slot}_i(\text{Tr})$. If $s > 0$, then let $q_1 \in Q_i$ be such that q_1.auth $= p_i$, q_1.type $=$ Tr and q_1.slot $= s - 1$. If $s = 0$, let q_1 be a 1-QC for b_g. Initially, set b.prev $:= \{q_1\}$.
3. If there exists $q_2 \in Q_i$ which is a single tip of Q_i, then enumerate q_2 into b.prev.
4. If $h' = \max\{q.\text{h} : \ q \in b.\text{prev}\}$, then set b.h $:= h' + 1$.
5. Let q be the greatest 1-QC in Q_i. Set b.1-QC $:= q$.
6. Sign b with the values specified above, and send this block to all processes.
7. Set $\texttt{slot}_i(\text{Tr}) := \texttt{slot}_i(\text{Tr}) + 1$;

The boolean $\texttt{LeaderReady}_i$. At any time, this boolean is equal to 1 iff either of the following conditions are satisfied, setting $v = \texttt{view}_i$:

1. Process p_i has not yet produced a block b with b.view $= v$ and b.type $=$ lead, and both:
   a. Process p_i has received view v messages signed by at least $n - f$ processes in Π.
   b. $\texttt{slot}_i(\text{lead}) = 0$ or Q_i contains q with q.auth $= p_i$, q.type $=$ lead, q.slot $= \texttt{slot}_i(\text{lead}) - 1$.
2. Process p_i has previously produced a block b with b.view $= v$ and b.type $=$ lead, and Q_i contains a 1-QC for b' with b'.auth $= p_i$, b'.type $=$ lead, b'.slot $= \texttt{slot}_i(\text{lead}) - 1$.

> *A comment on the boolean* $\texttt{LeaderReady}_i$. If p_i is the leader for view v, then before producing the first leader block of the view, it must receive view v messages from $n - f$ different processes, and must also receive a QC for the last leader block it produced (if any). Before producing any subsequent leader block in the view, it must receive a 1-QC for the previous leader block.

The procedure $\texttt{MakeLeaderBlock}_i$. When p_i wishes to form a new leader block b, it will run this procedure, by executing the following instructions:

1. Set b.type $:=$ lead, b.auth $:= p_i$, b.view $:= \texttt{view}_i$, b.slot $:= \texttt{slot}_i(\text{lead})$.
2. Initially, set b.prev to be the tips of Q_i.
3. Set $s := \texttt{slot}_i(\text{Tr})$ and $v := \texttt{view}_i$. If $s > 0$, then let $q \in Q_i$ be such that q.auth $= p_i$, q.type $=$ lead and q.slot $= s - 1$. If b.prev does not already contain q, add q to this set.
4. If $h' = \max\{q.\text{h} : \ q \in b.\text{prev}\}$, then set b.h $:= h' + 1$.
5. If p_i has not yet produced a block b with b.view $= \texttt{view}_i$ and b.type $=$ lead then:
   a. Set b.just to be a set of view v messages signed by $n - f$ processes in Π.
   b. Set b.1-QC to be a 1-QC in Q_i greater than or equal to all 1-QCs corresponding to messages in b.just.
6. If p_i has previously produced a block b with b.view $= \texttt{view}_i$ and b.type $=$ lead then let $q' \in Q_i$ be a 1-QC with q'.auth $= p_i$, q'.type $=$ lead and q'.slot $= s - 1$. Set b.1-QC $:= q'$ and set b.just to be the empty set.
7. Sign b with the values specified above, and send this block to all processes.
8. Set $\texttt{slot}_i(\text{lead}) := \texttt{slot}_i(\text{lead}) + 1$;

The pseudocode. The pseudocode appears in Algorithm 1 (with local variables described first, and the main code appearing later). Section 5.1 gives a 'pseudocode walk-through'.

Algorithm 1 Morpheus: local variables for p_i

Local variables
M_i, initially contains b_g and a 1-QC-certificate for b_g ▷ Automatically updated
Q_i, initially contains 1-QC-certificate for b_g ▷ Automatically updated
$\texttt{view}_i$, initially 0 ▷ The present view
$\texttt{slot}_i(x)$ for $x \in \{\text{lead}, \text{Tr}\}$, initially 0 ▷ Present slot
$\texttt{voted}_i(z, x, s, p_j)$ for $z \in \{0, 1, 2\}$, $x \in \{\text{lead}, \text{Tr}\}$, $s \in \mathbb{N}_{\geq 0}$, $p_j \in \Pi$, initially 0
$\texttt{phase}_i(v)$ for $v \in \mathbb{N}_{\geq 0}$, initially 0 ▷ The phase within the view
Other procedures and functions
$\texttt{lead}(v)$ ▷ Leader of view v
$\texttt{PayloadReady}_i$ ▷ Set to 1 when ready to produce transaction block
$\texttt{MakeTrBlock}_i$ ▷ Sends a new transaction block to all
$\texttt{LeaderReady}_i$ ▷ Indicates whether ready to produce leader block
$\texttt{MakeLeaderBlock}_i$ ▷ Sends a new leader block to all

5.1 Pseudocode walk-through

Lines 16-22: These lines are responsible for view changes. If p_i has received a quorum of end-view v messages for some greatest v greater than or equal to its present view, then it will use those to form a $(v+1)$-certificate and will send that certificate to all processes (immediately regarding that certificate as received and belonging to M_i). Upon seeing that it has received a v-certificate for some greatest view v greater than its present view, p_i will: (i) enter view v, (ii) send that v-certificate to all processes, and (iii) send a view v message to the leader of view v, along with any tips of Q_i corresponding to its own blocks. Process p_i will also do the same upon seeing q with q.view greater than its present view: the latter action ensures that any block b produced by p_i during view v does not point to any b' with b'.view $> b$.view.

Lines 24-28. These lines are responsible for the production of 0-QCs. Upon producing any block, p_i sends it to all processes. Providing p_i is correct, meaning that the block is correctly formed etc, other processes will then send back a 0-vote for the block to p_i, who will form a 0-QC and send it to all processes.

Lines 30 and 31. These lines are responsible for producing new transaction blocks. Line 30 checks to see whether p_i is ready to produce a new transaction block, before line 31 produces the new block: $\texttt{PayloadReady}_i$ and $\texttt{MakeTrBlock}_i$ are specified in Section 5.

Lines 33 and 34. These lines are responsible for producing new leader blocks. Line 33 ensures that only the leader is asked to produce leader blocks, that it will only do so once ready (having received QCs for previous leader blocks, as required), and only when required to (only if Q_i does not have a single tip and if still in the first phase of the view). $\texttt{LeaderReady}_i$ and $\texttt{MakeLeaderBlock}_i$ are specified in Section 5.

Lines 36-47. These lines are responsible for determining when correct processes produce 1 and 2-votes for transaction blocks. Lines 36 and 37 dictate that no correct process produces 1 or 2-votes for transaction blocks while in view v until at least one leader block for the view has been finalized (according to the messages they have received), and only if there do not exist unfinalized leader blocks for the view. Given these conditions, p_i will produce a 1-vote for any transaction block b that is a single tip of M_i, so long as b.1-QC is greater than or equal to any 1-QC it has seen. It will produce a 2-vote for a transaction block b if there exists q with q.b $= b$ which is a single tip of Q_i and if p_i has not seen any block of greater height. The latter condition is required to ensure that p_i cannot produce a 1-vote for some b' of greater height than b, and then produce a 2-vote for b (this fact is used in the proof of Theorem 2). After producing any 1 or 2-vote for a transaction block while in view v, p_i enters the second phase of the view and will no longer produce 1 or 2-votes for leader blocks while in view v.

Algorithm 1 Morpheus: The instructions for p_i

Process p_i executes the following transitions at timeslot t (according to its local clock), until no further transitions apply. If multiple transitions apply simultaneously, then p_i executes the first that applies, before checking whether further transitions apply, and so on.

▷ Update view

If there exists greatest $v \geq \mathtt{view}_i$ s.t. M_i contains at least $f+1$ end-view v messages **then**:
  Form a $(v+1)$-certificate and send it to all processes;
If there exists some greatest $v > \mathtt{view}_i$ such that either:
  (i) M_i contains a v-certificate q, or (ii) Q_i contains q with q.view $= v$, **then**:
    Set $\mathtt{view}_i := v$; Send (either) q to all processes;
    Send all tips q' of Q_i such that q'.auth $= p_i$ to $\mathtt{lead}(v)$;
    Send (v, q') signed by p_i to $\mathtt{lead}(v)$, where q' is a maximal amongst 1-QCs seen by p_i

▷ Send 0-votes and 0-QCs

If M_i contains some b s.t. $\mathtt{voted}_i(0, b.\text{type}, b.\text{slot}, b.\text{auth}) = 0$:
  Send a 0-vote for b (signed by p_i) to b.auth; Set $\mathtt{voted}_i(0, b.\text{type}, b.\text{slot}, b.\text{auth}) := 1$;
If M_i contains a 0-quorum for some b s.t.:
  (i) b.auth $= p_i$, **and** (ii) p_i has not previously sent a 0-QC for b to other processors, **then**:
    Send a 0-QC for b to all processes;

▷ Send out a new transaction block

If $\mathtt{PayloadReady}_i = 1$ **then**:
  $\mathtt{MakeTrBlock}_i$;

▷ Send out a new leader block

If $p_i = \mathtt{lead}(\mathtt{view}_i)$, $\mathtt{LeaderReady}_i = 1$, $\mathtt{phase}_i(\mathtt{view}_i) = 0$ **and** Q_i does not have a single tip:
  $\mathtt{MakeLeaderBlock}_i$;

▷ Send 1 and 2-votes for transaction blocks

If there exists $b \in M_i$ with b.type $=$ lead and b.view $= \mathtt{view}_i$ **and**
there does not exist unfinalized $b \in M_i$ with b.type $=$ lead and b.view $= \mathtt{view}_i$ **then**:
  If there exists $b \in M_i$ with b.type $=$ Tr, b.view $= \mathtt{view}_i$ and which is a single tip of M_i s.t.:
    (i) b.1-QC is greater than or equal to every 1-QC in Q_i and;
    (ii) $\mathtt{voted}_i(1, \text{Tr}, b.\text{slot}, b.\text{auth}) = 0$, **then**:
      Send a 1-vote for b to all processes; Set $\mathtt{phase}_i(\mathtt{view}_i) := 1$;
      Set $\mathtt{voted}_i(1, \text{Tr}, b.\text{slot}, b.\text{auth}) := 1$;
  If there exists a 1-QC $q \in Q_i$ which is a single tip of Q_i s.t.:
    (i) q.type $=$ Tr **and** (ii) $\mathtt{voted}_i(2, \text{Tr}, q.\text{slot}, q.\text{auth}) = 0$, **then**:
      If there does not exist $b \in M_i$ of height greater than q.h:
        Send a 2-vote for q.b to all processes; Set $\mathtt{phase}_i(\mathtt{view}_i) := 1$;
        Set $\mathtt{voted}_i(2, \text{Tr}, q.\text{slot}, q.\text{auth}) := 1$;

▷ Vote for a leader block

If $\mathtt{phase}(\mathtt{view}_i) = 0$:
  If $\exists b \in M_i$ with b.type $=$ lead, b.view $= \mathtt{view}_i$, $\mathtt{voted}_i(1, \text{lead}, b.\text{slot}, b.\text{auth}) = 0$ **then**:
    Send a 1-vote for b to all processes; Set $\mathtt{voted}_i(1, \text{lead}, b.\text{slot}, b.\text{auth}) := 1$;
  If $\exists q \in Q_i$ which is a 1-QC with $\mathtt{voted}_i(2, \text{lead}, q.\text{slot}, q.\text{auth}) = 0$, q.type $=$ lead,
  q.view $= \mathtt{view}_i$, **then**:
    Send a 2-vote for q.b to all processes; Set $\mathtt{voted}_i(2, \text{lead}, q.\text{slot}, q.\text{auth}) := 1$;

▷ Complain

If $\exists q \in Q_i$ which is maximal according to $\succeq$ amongst those that have not been finalized for time 6Δ since entering view $\mathtt{view}_i$:
  Send q to $\mathtt{lead}(\mathtt{view}_i)$ if not previously sent;
If $\exists q \in Q_i$ which has not been finalized for time 12Δ since entering view $\mathtt{view}_i$:
  Send the end-view message $(\mathtt{view}_i)$ signed by p_i to all processes;

Lines 49-54. These lines are responsible for determining when correct processes produce 1 and 2-votes for leader blocks. Correct processes will only produce such votes while in the first phase of the view.

Lines 56-59. These lines are responsible for the production of new-view messages. The proof of Theorem 3 justifies the choice of 6Δ and 12Δ.

Proofs of consistency and liveness appear in Appendix A. A detailed analysis of latency and complexity appears in the online version of the paper, which can be found at `https://arxiv.org/abs/2502.08465`. Related work appears in Appendix B.

6 Final Comments

We have presented Morpheus Consensus, a protocol that dynamically adapts its structure—shifting from a quiescent leaderless blockchain to an active leader-based DAG—while maintaining strong latency and complexity properties throughout. In high-throughput regimes, Morpheus demonstrates comparable performance to state-of-the-art DAG solutions like Autobahn. In low-throughput conditions, it achieves better latency and equivalent complexity versus established protocols such as PBFT and Tendermint, which fail to scale effectively under high load.

References

1. Nicolas Alhaddad, Sourav Das, Sisi Duan, Ling Ren, Mayank Varia, Zhuolun Xiang, and Haibin Zhang. Balanced byzantine reliable broadcast with near-optimal communication and improved computation. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, pages 399–417, 2022.
2. Balaji Arun, Zekun Li, Florian Suri-Payer, Sourav Das, and Alexander Spiegelman. Shoal++: High throughput dag bft can be fast! *arXiv preprint arXiv:2405.20488*, 2024.
3. Kushal Babel, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Arun Koshy, Alberto Sonnino, and Mingwei Tian. Mysticeti: Reaching the limits of latency with uncertified dags. *arXiv preprint arXiv:2310.14821*, 2023.
4. Leemon Baird. The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance. *Swirlds Tech Reports SWIRLDS-TR-2016-01, Tech. Rep*, 34:9–11, 2016.
5. Béla Bollobás. *Modern graph theory*, volume 184. Springer Science & Business Media, 2013.
6. Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *International conference on the theory and application of cryptology and information security*, pages 514–532. Springer, 2001.
7. Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.
8. Ethan Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, 2016.
9. Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on bft consensus. *arXiv preprint arXiv:1807.04938*, 2018.
10. Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.
11. Xiaohai Dai, Guanxiong Wang, Jiang Xiao, Zhengxuan Guo, Rui Hao, Xia Xie, and Hai Jin. Lightdag: A low-latency dag-based bft consensus through lightweight broadcast. *Cryptology ePrint Archive*, 2024.
12. Xiaohai Dai, Zhaonan Zhang, Jiang Xiao, Jingtao Yue, Xia Xie, and Hai Jin. Gradeddag: An asynchronous dag-based bft consensus with lower latency. In *2023 42nd International Symposium on Reliable Distributed Systems (SRDS)*, pages 107–117. IEEE, 2023.

13 George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: a dag-based mempool and efficient bft consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 34–50, 2022.

14 Adam Gągol, Damian Leśniak, Damian Straszak, and Michał Świętek. Aleph: Efficient atomic broadcast in asynchronous networks with byzantine nodes. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pages 214–228, 2019.

15 Neil Giridharan, Florian Suri-Payer, Ittai Abraham, Lorenzo Alvisi, and Natacha Crooks. Autobahn: Seamless high speed bft. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 1–23, 2024.

16 Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. Sbft: A scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pages 568–580. IEEE, 2019.

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

18 Idit Keidar, Andrew Lewis-Pye, and Ehud Shapiro. Grassroots consensus. *arXiv preprint arXiv:2505.19216*, 2025.

19 Idit Keidar, Oded Naor, Ouri Poupko, and Ehud Shapiro. Cordial miners: Fast and efficient consensus for every eventuality. *arXiv preprint arXiv:2205.09174*, 2022.

20 Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 45–58, 2007.

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

22 Kartik Nayak, Ling Ren, Elaine Shi, Nitin H Vaidya, and Zhuolun Xiang. Improved extension protocols for byzantine broadcast and agreement. *arXiv preprint arXiv:2002.11321*, 2020.

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

24 Nibesh Shrestha, Rohan Shrothrium, Aniket Kate, and Kartik Nayak. Sailfish: Towards improving latency of dag-based bft. *Cryptology ePrint Archive*, 2024.

25 Yonatan Sompolinsky, Yoad Lewenberg, and Aviv Zohar. Spectre: A fast and scalable cryptocurrency protocol. *Cryptology ePrint Archive*, 2016.

26 Alexander Spiegelman, Balaji Arun, Rati Gelashvili, and Zekun Li. Shoal: Improving dag-bft latency and robustness. *arXiv preprint arXiv:2306.03058*, 2023.

27 Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: Dag bft protocols made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2705–2718, 2022.

28 TK Srikanth and Sam Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2(2):80–94, 1987.

29 Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus in the lens of blockchain. *arXiv preprint arXiv:1803.05069*, 2018.

30 Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.

A Establishing consistency and liveness

Let M^* be the set of all messages received by any process during the execution. Towards establishing consistency, we first prove the following lemma.

▶ **Lemma 1.** *If $q, q' \in M^*$ are 1-QCs with $q \leq q'$ and $q' \leq q$, then $q.b = q'.b$.*

Proof. Suppose q.view $= q'$.view, q.type $= q'$.type, and q.h $= q'$.h. Consider first the case that q.b and q'.b are both leader blocks for the same view. If q.slot $= q'$.slot, but q.b $\neq q'$.b, then no correct process can produce 1-votes for both blocks. This gives an immediate contradiction, since two subsets of Π of size $n - f$ must have a correct process in the intersection, meaning that 1-QCs cannot be produced for both blocks. So, suppose that q'.slot $> q$.slot. Since each leader block b with b.slot $= s > 0$ must point to a leader block b' with b'.auth $= b$.auth and b'.slot $= s - 1$, it follows that q'.h $> q$.h, which also gives a contradiction.

So, consider next the case that q.b and q'.b are distinct transaction blocks. Since both blocks are of the same height, and since any correct process only votes for a block when it is a sole tip of its local value M_i, no correct process can vote for both blocks. Once again, this gives the required contradiction. ◀

Note that Lemma 1 also suffices to establish a similar result for 2-QCs, since no block can receive a 2-QC without first receiving a 1-QC: No correct process produces a 2-vote for any block without first receiving a 1-QC for the block.

Lemma 1 suffices to show that we can think of all 1-QCs $q \in M^*$ as belonging to a hierarchy, ordered by q.view, then by q.type, and then by q.h, such that if q and q' belong to the same level of this hierarchy then q.b $= q'$.b.

▶ **Theorem 2.** *The Morpheus protocol satisfies consistency.*

Proof. Given the definition of $\mathcal{F}$ from Section 4, let us say $b' \to b$ iff:

- $b' = b$, or;
- $b' \neq b_g$ and $b'' \to b$, where $q = b'$.1-QC and $b'' = q$.b.

To establish consistency it suffices to show the following:

(†): If b has a 1-QC $q_1 \in M^*$ and also a 2-QC $q_2 \in M^*$, then for any 1-QC $q \in M^*$ such that $q \geq q_1$, q.b $\to b$.

Given (†), suppose $M_1 \subseteq M_2 \subseteq M^*$. For each $i \in \{1, 2\}$, let M_i' be the largest set of blocks in M_i that is downward closed (in the sense specified in Section 4). Let q_i' be a maximal 2-QC in M_i such that q_i'.b $\in M_i'$, and set $b_i^* = q_i'$.b, or if there is no such 2-QC in M_i, set $b_i^* = b_g$. Let the sequence $b_k, \ldots, b_1 = b_g$ be such that $b_k = b_2^*$, and, for each $j < k$, if $q = b_{j+1}$.1-QC, then q.b $= b_j$. From (†) it follows that b_1^* belongs to the sequence $b_k, \ldots, b_1$, so that $\mathcal{F}(M_2) \supseteq \mathcal{F}(M_1)$.

We establish (†) by induction on the level of the hierarchy to which q belongs. If $q \leq q_1$ (and $q_1 \leq q$) then the result follows from Lemma 1.

For the induction step, suppose that $q > q_1$ and suppose first that q.type $=$ lead. Let $s = q$.slot, $v = q$.view. By validity of q.b, if $s > 0$, q.b points to a unique b^* with b^*.type $=$ lead, b^*.auth $= q$.auth and b^*.slot $= s - 1$. If $s = 0$ or b^*.view $< v$, then q.just (i.e. (q.b).just) contains $n - f$ view v messages, each signed by a different process in Π. Note that, in this case, any correct process that produces a 2-vote for b must do so before sending a view v message. It follows that, in this case, q.1-QC (i.e. (q.b).1-QC) belongs to a level of the hierarchy strictly below q and greater than or equal to that of q_1. The result therefore follows by the induction hypothesis. If $s > 0$ and b^*.view $= v$, then q.1-QC is a 1-QC-certificate for

b^*. Once again, q.1-QC therefore belongs to a level of the hierarchy strictly below q and greater than or equal to that of q_1, so that the result follows by the induction hypothesis.

So, suppose next that q.type = Tr. Note that, in this case, any correct process that produces a 2-vote for b must do so before sending a 1-vote for q.b. If q.view $>$ b.view this follows immediately, because a correct process p_i only sends 1 or 2-votes for any block b' while $\texttt{view}_i = b'$.view. If q.view $= b$.view and b.type = lead, this follows because no correct process sends 1 or 2-votes for a leader block after having voted for a transaction block within the same view. If q.view $= b$.view and b.type = Tr, this follows because any correct process only sends a 2-vote for b so long as there does not exist $b' \in M_i$ of height greater than b. Also, any correct process that produces a 2-vote for b will not vote for q.b unless q.1-QC is greater than or equal to any 1-QC it has received. It follows that q.1-QC belongs to a level of the hierarchy strictly below q and greater than or equal to that of q_1. Once again, the result follows by the induction hypothesis. ◀

▶ **Theorem 3.** *The Morpheus protocol satisfies liveness.*

Proof. Towards a contradiction, suppose that correct p_i produces a transaction block b, which never becomes finalized (according to the messages received by p_i). Note that all correct processes eventually send 0-votes for b to p_i, meaning that p_i forms a 0-certificate for b, which is eventually received by all correct processes. Since correct processes send end-view messages if a some QC is not finalized for sufficiently long within any given view (see line 58), correct processes must therefore enter infinitely many views. Let v be a view with a correct leader, such that the first correct process p_j to enter view v does so at some timeslot after GST, and after p_i produces b. Process p_j sends a v-certificate to all processes upon entering the view, meaning that all correct processes enter the view within time Δ of p_j doing so. Upon entering view v, at time t say, note that p_i will send a QC for a transaction block b' that it has produced to the leader. This block b' has a slot number greater than or equal to that of b. The leader will produce a leader block observing b' by time $t + 3\Delta$, which will be finalized (according to the messages received by p_i) by time $t + 6\Delta$. ◀

B Related Work

Morpheus uses a PBFT [10] style approach to view changes, while consistency between finalised transaction blocks within the same view uses an approach similar to Tendermint [8, 9] and Hotstuff [30]. Hotstuff's approach of relaying all messages via the leader could be used by Morpheus during low throughput to decrease communication complexity, but this is unlikely to lead to a decrease in 'real' latency (i.e. actual finalisation times). The optimistic 'fast commit' of Zyzzyva [16, 20] can also be applied as a further optimisation. The recent paper [18] shows how to implement player reconfiguration for a form of the Morpheus protocol.

Morpheus transitions between being a leaderless 'linear' blockchain during low throughput to a leader-based DAG-protocol during high throughput. DAG protocols have been studied for a number of years, Hashgraph [4] being an early example. Hashgraph builds an unstructured DAG and suffers from latency exponential in the number of processes. Spectre was another early DAG protocol, designed for the 'permissionless" setting [25], with proof-of-work as the mechanism for sybil resistance. The protocol implements a 'payment system', but does not totally order transactions. Aleph [14] is more similar to most recent DAG protocols in that it builds a structured DAG in which each process proceeds to the next 'round' after receiving blocks from $2f + 1$ processes corresponding to the previous round, but still has greater latency than modern DAG protocols.

More recent DAG protocols use a variety of approaches to consensus. Narwhal [13] builds a DAG for the purpose of ensuring data availability, from which (one option is that) a protocol like Hotstuff or PBFT can then be used to efficiently establish a total ordering on transactions. DAG-Rider [17], on the other hand, builds the DAG in such a way that a total ordering can be extracted from the structure of the DAG, with zero further communication cost. The protocol proceeds in 'waves', where each wave consists of four rounds, each round building one 'layer' of the DAG. In each round, each process uses an instance of Reliable Broadcast (RBC) to disseminate their block for the round. Each wave has a leader and an expected six rounds (6 sequential RBCs) are required to finalise the leader's block for the first round of the wave. This finalises all blocks observed by that leader block, but other blocks (such as those in the same round as the leader block) may have signicantly greater latency. Tusk [13] is an implementation based on DAG-Rider.

Given the ability of DAG-Rider to handle significantly higher throughput in many settings, when compared to protocols like PBFT that build a linear blockchain, much subsequent work has taken a similar approach, while looking to improve on latency. While DAG-Rider functions in asynchrony, Bullshark [27] is designed to achieve lower latency in the partially synchronous setting. GradedDAG [12] and LightDAG [11] function in asynchrony, but look to improve latency by replacing RBC [7] with weaker primitives, such as consistent broadcast [28]. This means that those protocols solve Extractable SMR (as defined in Section 3), rather than SMR, and that further communication may be required to ensure full block dissemination in executions with faulty processes. Cordial Miners [19] has versions for both partial synchrony and asynchrony and further decreases latency by using the DAG structure (rather than any primitive such as Consistent or Reliable Broadcast) for equivocation exclusion. Mysticeti [3] builds on Cordial Miners and establishes a mechanism to accommodate multiple leaders within a single round. Shoal [26] and Shoal++ [2] extend Bullshark by establishing a 'pipelining approach' that implements simultaneous instances of Bullshark with a leader in each round. This reduces latency in the good case because one is required to wait less time before reaching a round in which a leader block is finalised. Both of these papers, however, use a 'reputation' system to select leaders, which comes with its own trade-offs. Sailfish [24] similarly describes a mechanism where each round has a leader, but does not make use of a reputation system. As noted previously, the protocol most similar to Morpheus during high throughput is Autobahn [15]. One of the major distinctions between Autobahn and those previously discussed, is that most blocks are only required to point to a single parent. This significantly decreases communication complexity when the number of processes is large and allows one to achieve linear ammortised communction complexity without the use of erasure coding [1, 22] or batching [21].