The Pipes Model for Latency and Throughput Analysis

ANDREW LEWIS-PYE, London School of Economics and Political Science, United Kingdom
KARTIK NAYAK, Duke University, USA
NIBESH SHRESTHA, Supra Research, USA

In much of the distributed computing literature, latency is expressed as the number of communication rounds or network delays; such metrics do not directly account for the amount of data sent or the dependencies among parties sending the data. Throughput is also normally evaluated empirically. As a consequence, the standard models used for analysing state machine replication (SMR) protocols do not by themselves provide an end-to-end analytical comparison of practical latency and throughput as functions of per-processor bandwidth and transaction arrival rate.

In this paper, we present Pipes, a model for analysing latency and throughput in SMR protocols. The Pipes model captures the effect of processor bandwidth S, transaction arrival rate D, and network delays, enabling us to explicitly specify the throughput bottleneck and the latency of a protocol. Using Pipes, we derive good-case latency expressions and throughput bottlenecks for concrete multi-shot protocols implementing Best-effort Broadcast, Consistent Broadcast, Reliable Broadcast, and Reliable Broadcast with erasure coding and pipelining. We consider both single-sender and multi-sender approaches, and extend the analysis to SMR protocols including DispersedSimplex, Sailfish, Tendermint, and HotStuff. Our comparisons reveal the trade-off between the lower latency achievable through pipelining and erasure coding in single-sender protocols and the higher throughput bottlenecks achievable by multi-sender DAG-based protocols. We also quantify when the additional communication phases of HotStuff are preferable to the fixed wait in a Tendermint-style protocol.

Beyond these protocol-specific latency calculations, we establish tight Best-effort Broadcast latency bounds in both the single-sender and multi-sender settings. The analysed protocols are optimal when each recipient receives a full block directly from the transaction's source, the same blocks are sent to everyone, and outstanding upload work is bounded. We give a second tight bound assuming only direct receipt, attained by a protocol that staggers block boundaries across recipients. For protocols using AVID-style erasure coding, we prove latency lower bounds that combine block formation, reconstruction and fragment forwarding.

Finally, we implement and evaluate variants of DispersedSimplex and Sailfish, comparing measured latency with the predictions of the Pipes model.

1 INTRODUCTION

The common timing assumptions considered in Distributed Computing are the synchronous, asynchronous, and partially synchronous settings. In the synchronous setting, messages are delivered within a known bound Δ. In the asynchronous setting [32], messages are eventually delivered but with no time bound. In the partially synchronous setting [27], there is an unknown global-stabilisation-time (GST) after which messages are always delivered within the known bound Δ.

Common to all these models is the fact that message sizes are unbounded: for example, in the synchronous setting, a message of *any size* sent at time t is guaranteed to be delivered by $t + \Delta$. Of course, part of the motivation for this assumption is to simplify analysis, and consideration of message sizes may not be critical when establishing fundamental properties of protocols such as consistency and liveness. However, a consequence of their indifference to message sizes is that these standard timing models, by themselves, give limited ability to analyse some other important properties of a blockchain protocol, such as the maximum throughput it can handle.

These models also give limited ability to analyse (real-world) *latency*. In the case of protocols for SMR, this can be informally defined as the time between the first point at which a transaction is received by a correct (non-faulty) processor and the first time at which that transaction is *finalised* by all correct processors. In the standard models,

'latency' is upper-bounded simply by counting the number of rounds of communication required: if 10 rounds of communication are required in the synchronous model, then 'latency' is 10Δ. Since there is a clear understanding that this is unrealistic (sending larger amounts of data will generally take longer than sending a smaller amount), we are then committed to an awkward dance in which we also consider *communication complexity*, e.g., the number of bits of data that must be sent to achieve finalisation in the case of SMR protocols, with no formal way to weigh these metrics. If a protocol has lower 'latency' (round complexity) but higher communication complexity than another, what does this mean in terms of real-world latency? Generally (but not always), it is the real-world latency that one actually cares about, rather than the round complexity or communication complexity: while potentially of interest in their own right, the latter metrics are generally used as proxies for the former.

Recent research on multi-sender protocols. These considerations are brought to a head by recent research that looks to develop SMR protocols that can handle high throughputs. Typical protocols, such as PBFT [18] (henceforth called 'single-sender' protocols), have a single designated (potentially rotating) 'leader' who is responsible for making and disseminating proposals (blocks of transactions proposed for addition to the finalised log) to others. In a context where processors do have limited upload/download speeds, there is an estimation by many in the community that the leader should act as a bottleneck, limiting the ability to deal with high throughput. A significant amount of research has therefore investigated the use of 'multi-sender' protocols in which many processors distribute proposals in parallel. Examples include DAG-based protocols such as DAG-Rider [42], Bullshark [73], Shoal [7], Sailfish [69], Cordial Miners [43], Mysticeti [8], and variants such as Autobahn [39]. Some of these protocols serve as the consensus mechanism for major blockchains with market caps in the billions of dollars.[1] However, the standard timing and communication-complexity models used in distributed computing do not directly support the per-processor, workload-sensitive analysis needed to determine when their better use of network bandwidth should lead to improved performance (i.e., lower real-world latency for various throughputs, or greater achievable throughputs) when compared with single-sender protocols such as PBFT, Tendermint [13], and HotStuff [78].

The goals of this paper. The first goal of this paper is to describe a basic model that allows one to calculate latency as a function of network bandwidth, and thereby to compare the performance of protocols using the single-sender and multi-sender approaches (as an example application). According to this model, formally described in Section 2, each processor has a bandwidth (upload/download capacity) of S (constant size) 'data parcels' per timeslot, while the network receives incoming transactions at a rate of D data parcels per timeslot. For any given protocol, one can then calculate latency as a function of S, D, network delays, the number of processors n, and some other system parameters.[2] It is also easily observed that each protocol has a *throughput bottleneck*: this is an incoming transaction rate at which latency becomes unbounded over the protocol execution, i.e., a maximum throughput that the protocol can handle without unbounded latency.

With the model in place, our second goal is then to understand how latency varies with system parameters such as the desired throughput D and the number of processors n. A significant part of this analysis is to understand how the throughput bottleneck (i.e., maximum possible throughput) depends on these parameters, but we are also interested in how latency varies below the bottleneck. In particular, we aim to answer the following question:

[1] For example, see https://coinmarketcap.com/currencies/sui/.

[2] We note that some existing papers (e.g., [9, 44, 59]) model network capacity, but these are tailored to gossip-based protocols and don't enable granular comparison of single vs. multi-sender approaches. See Section 9 for a more detailed comparison.

If each processor has a bandwidth of S parcels per time slot, if the incoming data rate is D parcels per time slot, and if there is a delay of δ time slots between processors, can we analytically compute the trade-off between throughput and latency for a given protocol and a given number of processors?

Simplifications made by the analysis in this paper. A simplification which is a feature of our analysis here, but which is not inherent in the model described, is that we generally focus on the 'good case' that message delivery is reliable and processors act correctly. There are at least two reasons for this focus:

(1) Real-world experience (see, for example, [15]) shows that the good case is actually the common case, i.e., in real-world applications, message delivery is reliable more often than not, and processors normally behave correctly. The intention of developers, therefore, is generally to build SMR protocols that remain consistent during periods of asynchrony and under substantial adversarial action, but which are optimised to give low latency in the good case, since this is the case that applies most of the time.
(2) Unsurprisingly, using the new model makes latency analysis more complex. By focusing on the good-case scenario for latency, we can highlight the results that researchers and practitioners are likely most interested in, while keeping our analysis reasonably simple.

Our model is also designed with the aim of being as straightforward as possible, while still effectively capturing key factors involved in comparing latency for different protocol design approaches. To this end, we certainly over-simplify some significant considerations that will impact latency in practice. In particular, we make the following simplifications and believe it is important that future work should consider more elaborate models:

- We do not consider the time taken to carry out computationally expensive tasks such as signature generation/verification and erasure coding.
- We suppose upload and download speeds are equal and constant.
- We suppose all processors have the same bandwidth S.
- We suppose message delay is the same between all pairs of processors.

These simplifications mean that the model captures an idealised scenario, assuming (amongst other things) zero computational cost. Although we expect that future models may benefit from further elaborating on these considerations, we believe it is beneficial to start with a model that is as simple as possible, while effectively addressing critical elements necessary for evaluating the latency/throughput trade-off.

The benefits of a model. By introducing this model, we aim to take a step towards the development of a formal theory for latency analysis, the establishment of lower bounds, and the ability to predict and analyse a protocol's latency/throughput trade-off. A further practical benefit is that such a model provides a useful stage in protocol design *before* large-scale experimentation. Running experiments with many processors distributed across geographically separated data centres can be expensive and time-consuming. In the absence of an analytical model, one may design a protocol with little ability to predict its real-world latency or maximum sustainable throughput, and then discover its performance characteristics only by implementing it and carrying out costly experiments. Once the relevant communication schedule has been specified, calculations in the Pipes model are generally quick and inexpensive: they can identify likely bandwidth bottlenecks, estimate maximum throughput, show how latency should vary with system parameters, and allow candidate protocol designs to be compared before substantial resources are committed to experimental evaluation.

The purpose of the model is not to replace experiments. Real implementations involve effects that our deliberately simplified model omits, including computation costs, heterogeneous network conditions, congestion-control behaviour, and other implementation details. Rather, modelling and experiments play complementary roles. Analytical calculations can be used first to rule out unattractive designs, identify promising optimisations and parameter regimes, and determine which comparisons are worth testing experimentally; experiments can then test the model's assumptions and quantify effects that it leaves out. A model is also useful when interpreting experimental results, since it gives a common baseline against which implementation-specific effects can be understood.

A high-level overview of results. With the aim of building to an analysis for state-of-the-art SMR protocols, we begin by considering protocols for simpler primitives: Best-effort Broadcast (BeB), Consistent Broadcast (CB), Reliable Broadcast (RB), and RB with erasure coding. In each case, we consider the 'multi-shot' version, since SMR protocols that use these primitives do so in the multi-shot setting.

Best-effort Broadcast. For Best-effort Broadcast, we establish two tight lower bounds, in both the single-sender and multi-sender settings. The first proves optimality of the simple-broadcast protocols described in the main text: recipients receive full blocks directly from their source, the same blocks are sent to everyone, and outstanding upload work is bounded. The second requires only direct receipt of a full block from the transaction's source. A protocol using different, staggered block boundaries for different recipients attains this broader bound. The distinction identifies a latency cost of requiring common blocks, even without erasure coding or gossip. In both cases, there is a factor n difference between the single-sender and multi-sender throughput bottlenecks.

Consistent Broadcast, Reliable Broadcast and Reliable Broadcast with erasure coding. Apart from the lower bounds in Sections 3 and 5.4, our results analyse specific protocol variants, with the stated sending rules and block sizes chosen to minimise latency under the model. The precise latency expressions therefore depend on the protocols being analysed, rather than constituting lower bounds for the corresponding primitives. For the protocols we consider, the picture for stronger broadcast primitives is more nuanced. In particular, for Reliable Broadcast the factor n difference in the throughput bottleneck disappears: maximum throughput for the single-sender and multi-sender approaches differs only by a constant factor, while multi-sender approaches will generally have latency that grows more quickly with n. The constant factor depends on the precise dissemination protocol, including whether pipelining and erasure coding are used.

Lower bounds for erasure-coded broadcast. Section 5.4 establishes latency lower bounds for protocols using AVID-style erasure coding, in which blocks and their Merkle roots are fixed before fragment transmission begins. The first bound depends only on block formation, fragment sizes and the number of complete authenticated fragments needed for reconstruction. When each nonleader receives only its designated fragment directly from the leader and obtains the other reconstruction fragments by forwarding, we obtain a stronger bound accounting for the two propagation delays traversed by those fragments.

SMR protocols. We then use these primitive analyses to study representative SMR protocols. With respect to DAG-based protocols, we focus on 'certified' protocols such as DAG-Rider, Bullshark, Shoal, and Sailfish, which use some form of Consistent or Reliable Broadcast as the underlying method of block propagation: the term 'certified' [61] stems from the fact that such protocols start by producing a certificate for each block, prior to the process of reaching consensus on total ordering. Our motivation in doing so is principally to limit the length of the paper, and we leave to future work the analysis of uncertified protocols such as Cordial Miners and Mysticeti.

Once the underlying broadcast primitive has been analysed, the good-case latency calculation for a certified DAG protocol largely amounts to determining how many such dissemination stages are required before a block is finalised. To make things concrete, we focus on Sailfish [69] and analyse the version of Sailfish that uses Consistent Broadcast for block propagation.[3]

In the case of single-sender protocols, the use of erasure coding can make a significant difference to maximum throughput. Without erasure coding, the leader sends a full copy of each block to every processor; with erasure coding, it can send smaller fragments instead, reducing the amount of data it uploads. For the single-sender setting *with erasure coding*, we use DispersedSimplex [65] as our principal example. DispersedSimplex is particularly useful for our purposes because it spells out explicitly how a single leader can use erasure coding and pipelining to prevent the leader from becoming the bandwidth bottleneck. Its stable-leader variant permits successive dispersals to overlap. The baseline variant analysed in detail in this paper has a data expansion factor of approximately 3.[4]

These analyses identify two broad principles. For the version of Sailfish analysed here, the throughput bottleneck is at $D = S$. This is not peculiar to Sailfish: for a protocol in which block production and dissemination do not introduce another limiting communication step, fixed metadata costs can be made negligible by taking blocks sufficiently large, and the maximum transaction-data rate approaches S. For the stable-leader pipelined version of DispersedSimplex we analyse, the corresponding bottleneck is at $D = S/3$. This reflects its data expansion factor of approximately 3. More generally, when the payload data are expanded by a factor ρ and no other communication step imposes a stricter limit, the leading bandwidth constraint is of the form $D < S/\rho$.

For the versions of Sailfish and DispersedSimplex analysed here, we obtain a clear trade-off. DispersedSimplex has substantially lower latency over a wide range of throughputs, particularly with a stable leader and pipelining, while Sailfish can sustain a higher input rate before reaching its throughput bottleneck. Figure 1 gives one concrete illustration. For this example, processors have upload/download capacity $S = 1$ Gbps, transactions are 2500 bits, and the network delay is 0.2 seconds. The other parameters, including the sizes of metadata and votes, are specified in Section 7. The left graph fixes the incoming rate at 10^5 transactions per second and varies n; the right graph fixes $n = 400$ and varies the incoming transaction rate. The detailed analysis and the precise assumptions underlying these curves are given later in the paper.

Conference version. A preliminary version of this paper was accepted for publication at the IEEE Symposium on Security and Privacy [49]. The present version includes the full Best-effort Broadcast lower-bound proof, the layered-DAG calculation, and the additional erasure-coded broadcast and SMR analyses omitted from that version. It also gives a tight Best-effort Broadcast bound assuming only direct receipt of full blocks, together with a protocol attaining that bound; gives shared timing lemmas for the broadcast analyses; and proves reconstruction and forwarding lower bounds for protocols using AVID-style erasure coding. We further distinguish actual network delay from fixed waits in the Tendermint/HotStuff comparison. The experimental data are unchanged.

[3] Using Reliable Broadcast gives stronger data-availability guarantees, but for the protocol considered here yields a throughput bottleneck of order S/n, thereby defeating much of the throughput advantage of the multi-sender architecture; with Consistent Broadcast the bottleneck is of order S. Other DAG protocols use different dissemination rules. For example, Starfish [61] explores different dissemination and pipelining techniques, while BlueBottle's BB-Core [75] obtains a two-delay fast path under the stronger resilience assumption $n \geq 5f + 1$, where f is the maximum number of Byzantine core processors tolerated. The same bandwidth and communication-delay considerations apply to these protocols once their sending schedules are specified.

[4] Here the data expansion factor ρ is the ratio between the aggregate size of the erasure-coded fragments and the size of the uncoded data being encoded. So, $\rho = 3$ means that B units of uncoded data give rise to approximately $3B$ units of coded fragment data, ignoring fixed metadata and lower-order terms. The more recent version of DispersedSimplex reduces this factor to approximately 1.5 in favourable executions, while retaining its optimistic latency properties [65]. Subsequent work in *The Carnot Bound* [50] shows more generally that protocols can optimistically use data expansion rates below 1.5 when fewer processors are faulty and fall back to a data expansion rate of 1.5 when necessary.

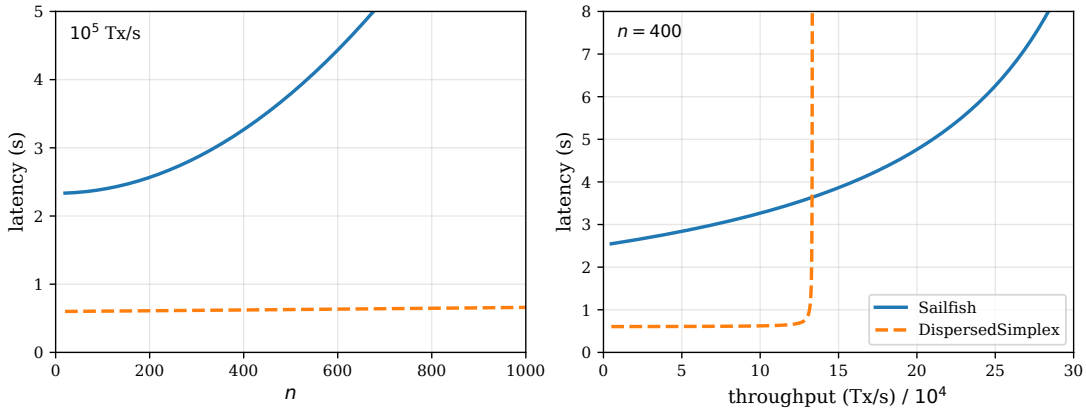

Fig. 1. Latency for Sailfish and DispersedSimplex. DispersedSimplex has substantially lower latency over much of the range shown, while Sailfish has a higher throughput bottleneck.

Paper structure. Section 2 defines the model. Section 3 considers Best-effort Broadcast. Section 4 considers Consistent Broadcast and Reliable Broadcast, and Section 5 considers Reliable Broadcast with erasure coding and pipelining. Section 6 considers SMR protocols, while Section 7 compares the resulting latencies. Section 8 describes the experiments. Sections 9 and 10 describe related work and have some final comments. The Best-effort Broadcast lower-bound proofs and the construction showing those bounds to be tight, the layered-DAG analysis, and the additional DispersedSimplex calculations appear in Appendices A–C. The coded lower-bound proof appears in Appendix E.

2 MODEL

The system of processors. We consider a system of n processors $\{p_1, \ldots, p_n\}$, where each processor maintains a direct connection with every other processor. Processors communicate with each other using reliable delivery-in-order channels. In practice, TCP provides such a functionality. We assume all communication happens in the form of information parcels, *each of which is some fixed (constant) number of bits.*[5] Processors may be *correct*, meaning that they execute protocol instructions as prescribed, or may display various faults (e.g. be *Byzantine*), but we will focus on the performance of protocols when processors are correct in what follows.

Cryptographic assumptions. Our cryptographic assumptions are standard for papers in distributed computing. Some of the protocols considered use a cryptographic signature scheme, a public key infrastructure (PKI) to validate signatures, a threshold signature scheme, a scheme for erasure coding, and/or collision-resistant hash functions. We assume a computationally bounded adversary. Following a common standard in distributed computing and for simplicity of presentation (to avoid the analysis of certain 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.

Message delays. We write Δ for the known upper bound on network delay after GST, and δ for the actual, unknown upper bound after GST, where $0 < \delta \leq \Delta$. A protocol may use Δ, but is not assumed to know δ.

[5]For simplicity, one can just think of each parcel as a single bit, but it is sometimes useful to have some flexibility with regard to the number of bits in each parcel.

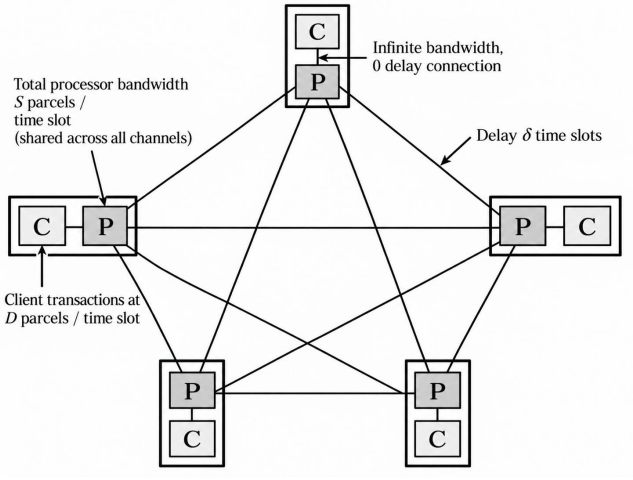

Fig. 2. Depicting a 5-processor system. Each processor involves a client processor (denoted C) and consensus processor (denoted P). The bandwidth S is shared across all channels of a consensus processor.

For simplicity, we suppose time is divided into discrete slots[6] and that the message delay between any two processors is always δ time slots in the executions considered in this paper.[7] We note that a protocol executed by the system of processors may function in partial synchrony or asynchrony [27], where the assumptions above may not always hold. However, our goal is to analyse throughput and latency under optimistic conditions — so, the synchronous assumption is reasonable for the purposes of our analysis.

The upload and download buffers. Intuitively, each processor has a 'bandwidth' of S parcels per time slot, i.e., it is capable of uploading S parcels and downloading S parcels in each time slot. To formalise this, we consider each processor to have a single upload buffer and a single download buffer, from which it can upload and download parcels of data. When a processor executes an instruction to send a message along any of its channels at time slot t, the corresponding information parcels are immediately added to its upload buffer. Each parcel added to the upload buffer of p_i has an intended recipient p_j. At the end of timeslot t, S parcels (or x parcels if the buffer presently only holds $x < S$ parcels) are removed from the upload buffer in a FIFO manner, and then appear on the download buffers of their intended recipients at $t + \delta$. We say that a parcel is *uploaded* when it is removed from the upload buffer. The bandwidth is S for the whole upload buffer and, separately, S for the whole download buffer, not S for each channel.

At the beginning of a time slot, (up to) S parcels are removed from the download buffer in a FIFO manner, and those parcels are then 'received' by the processor, together with information specifying the sender of each parcel .

Modelling connections to clients. In a state machine replication protocol, the n processors receive transactions from external clients. Thus, the processors utilise their bandwidth to also receive transactions from these clients. To

[6]We think of each slot as sufficiently short that a one-slot rounding error is not significant. Events assigned to the same slot need not be processed atomically: if a rule is enabled after receipt of q among several same-slot messages, a processor may act after the first q are processed, without waiting for the others. Equivalently, this is the limit under arbitrarily small perturbations of otherwise simultaneous events. Such perturbations may determine which threshold is triggered but add no latency term; the permitted same-slot orderings are included among the executions over which worst-case latency is taken.

[7]The delay between processors is not uniform for practical instantiations in a geo-distributed setting, e.g., two processors located on the east coast of the US have a smaller delay than that between a processor in the US and Asia. Nevertheless, we make this simplifying uniformity assumption.

model this cleanly, we suppose that every logical processor consists of two physical processors, a *client* processor and a *consensus* processor, which are connected to each other using an infinite bandwidth and zero delay channel. Communication channels between logical processors are between their respective consensus processors, and the upload and download buffers of those processors impact the rate at which information can be sent along these channels between consensus processors. The clients send data to client processors and we will suppose that the client processors in the system receive transactions at a total (combined) rate of D parcels per time slot.[8]

The transactions received through the download buffer of the client processor are sent to the consensus processor through the infinite bandwidth connection, and thus the consensus processors receive this information at the combined rate of D parcels per time slot.

The single-sender and multi-sender settings. The idea behind DAG-based protocols is that each processor should propose their own blocks of transactions. Typically, this can result in some complications. For instance, different processors may end up transmitting the same data, thus resulting in repetitive work, or incompatible transactions may be transmitted by different processors. In this work, we ignore such concerns and assume that all of the transmitted data is useful. In the multi-sender setting, for example, we suppose that the D transaction parcels arriving at client processors per time slot are (somehow) evenly divided between the client processors, so that each receives D/n transaction parcels per time slot:

> In the *single-sender* setting, we suppose that a designated leader receives D transaction parcels at its client processor at each timeslot, and that other processors do not receive transaction parcels at their client processors. In the *multi-sender* setting, we suppose that each processor receives D/n transaction parcels at its client processor at each timeslot. For the sake of simplicity, we suppose that all transactions are unique.

Latency. We consider a range of protocols with different 'delivery' notions. For example, in the context of Best-effort Broadcast, we will consider a processor to 'deliver' a data parcel upon receiving a full block of transactions to which the data parcel belongs. In the context of a protocol for SMR, we consider a processor to deliver a data parcel upon adding it to its finalised log. The relevant notion of delivery will be made precise in each context.

The latency calculations below start with empty buffers at time slot 0, with all processors correct and with processors synchronised where specified. Transactions arrive at the stated constant rate throughout the execution.[9]

A protocol has latency ℓ_{tx} if ℓ_{tx} is the least value such that the following holds in every protocol execution (consistent with the setting considered) and for all time slots t: *any* data parcel received by a correct processor at t is delivered by all correct processors by $t + \ell_{\text{tx}}$. Since we are concerned with the optimistic case, we consider only synchronous periods. Moreover, the use of the quantifier *any* implies that we are only concerned with the worst-case latency (as opposed to the average-case or common-case latency).

For each fixed choice of the network parameters and incoming transaction rate, we calculate latency with block size chosen to minimise it. How an implementation estimates those parameters or adjusts its block size is a separate question, which we do not analyse here. The known bound Δ applies to network delay; transmission and queueing times are counted separately.

[8] Whether or not the download buffer of *client* processors is rate limited will not meaningfully impact our analysis.
[9] We do not measure the time required to clear a backlog accumulated before GST, or the delay of a final incomplete block after transactions stop arriving.

Rounding and lower order terms. Most of the latency formulas below ignore integer rounding. We calculate transmission times as though parcels could be divided and the buffers were served continuously at rate S; for example, uploading X parcels takes X/S time in this calculation. We refer to this as continuous service. In the discrete model, parcel counts and time slots must instead be integers. A transmission may therefore take an additional slot, and a chosen block size or per-slot allocation may have to be rounded. We retain this rounding in statements explicitly concerning integral slots. Making a time slot shorter reduces time-rounding errors.

For simplicity, the displayed calculations count copies sent by a processor to itself, although an implementation need not send such copies. In some comparison formulas we also replace $n - 1$ or $n + 1$ by n to remove lower order terms. These are approximations for large n. In particular, a simplified throughput bottleneck need not equal the exact threshold for a fixed n: near the bottleneck, even a small change to that threshold can determine whether latency is bounded. We retain the unsimplified condition when making this distinction.

For simplicity, we assume throughout that binary representations of protocol counters and identifiers, such as view and round numbers, block heights, and instance identifiers, have constant bounded length. Strictly speaking, these representations can grow during an infinite execution, but treating them as fixed-size is a reasonable approximation for executions of realistic duration.

3 BEST-EFFORT BROADCAST

As described previously, our ultimate aim is to use our model to compare latency for state-of-the-art SMR protocols. As a stepping stone, we first consider latency for a number of primitives such as Best-effort Broadcast, Consistent Broadcast, and Reliable Broadcast. In each case, we consider the 'multi-shot' version (in which one must reach a sequence of consensus decisions, rather than a single decision), since SMR protocols that use these primitives do so in the multi-shot setting.

In this section, we consider Best-effort Broadcast protocols, either in the single-sender or multi-sender setting. Although these protocols are simple, we spend significant time analysing them: doing so highlights some important principles which are brought to light by the new model, as well as some useful methods.

3.1 Best-effort Broadcast with a single sender

The first primitive we consider is a form of Best-effort Broadcast in the single-sender setting. A designated 'leader' p_ℓ receives D data parcels per time slot at its client processor and sends parcels to all processors. Our goal is to consider the rate at which p_ℓ can push data to all others, in a context where data should be included in 'blocks' that must also contain metadata, and the bound this puts on throughput and latency: to specify latency, processors 'deliver' a transaction when they receive a full block containing that transaction. We initially suppose that p_ℓ sends data to others in a straightforward fashion, without using techniques such as erasure coding.

Data parcel terminology. We use the following terminology:

- Data parcels arriving at the client processor are called *transactions*.
- A block is made up of *block data parcels*: these are either transactions or *metadata parcels*.
- *Addressed parcels* are those in the upload/download buffers of consensus processors. These are either *addressed transactions* or *addressed metadata*.

Protocol overview. Figure 3 describes the protocol.[10] The goal is for the leader to produce a block of transactions and send it to all processors along with some metadata, while minimising latency. This task is complicated by the fact that we must consider two types of metadata:

- *Dependent metadata*, such as p_ℓ's signature on the block, which cannot be determined prior to deciding the transactions included in the block.
- *Independent metadata* that can be determined before deciding which transactions are included in the block. For example, p_ℓ may be able to determine an appropriate hash pointer to a previous block before the transactions are decided.[11]

We denote the sizes of dependent and independent metadata by M^{de} and M^{in} respectively, and let $M = M^{\mathrm{de}} + M^{\mathrm{in}} > 0$ be the total metadata size for each block (we suppose these sizes are given and fixed). As we will see, it is important to distinguish these different forms of metadata because they have different impacts on latency.[12]

To specify the block, the leader maintains a local variable called sendbuffer (not to be confused with the upload buffer), and adds transactions to sendbuffer.[13] A crucial distinction between sendbuffer and the upload buffer is that the latter needs to contain a distinct copy of each block data parcel for each recipient. At each timeslot, p_ℓ therefore makes n copies[14] of some of the parcels in sendbuffer and adds these addressed parcels to its upload buffer (adding parcels to the upload buffer constitutes the decision to 'send' those parcels). Once the number of transactions added to sendbuffer reaches a predetermined size, it computes the dependent metadata for the *current* block and the independent metadata for the *next* block and appends that data to sendbuffer. Note that independent metadata is sent before all other block data, while dependent metadata is placed at the end of the block. On the receiver's end, every processor receives data at each timeslot until it has received a complete block along with the associated metadata.

Understanding the protocol parameterisation. Recall that D transactions arrive at the leader's client processor per time slot. As a first observation, we note a simple upper bound on values of D for which bounded latency is possible: since the leader has a bandwidth of S and since each transaction arriving at the client processor needs to be sent to n processors, if $Dn > S$ then the required data cannot be taken from the upload buffer of the leader as fast it is arriving at its client processor. This means that the latency for subsequent transactions will grow in an unbounded fashion. For the remainder of Section 3.1, we therefore assume that $D \leq S/n$. In particular, suppose $D = \alpha S/n$ for $\alpha \in (0, 1]$.

Observe that if $\alpha = 1$, the leader can indeed disseminate the arriving transactions but does not have additional bandwidth to send any metadata. We will see that, for any $\alpha < 1$, setting the number of transactions included in each block to be $B = \frac{\alpha}{1-\alpha} M$ allows the leader to send metadata along with the transactions, while also minimising latency.

We make the following claim, which we will later show to be optimal in a precise sense:

[10] For simplicity of presentation (and as noted in Section 2), we ignore some issues of integer rounding (such as the fact that S/n may not be an integer) when describing and analysing the protocol.

[11] It will often be convenient to suppose that the *first* block produced by a protocol does not need to include independent metadata. For example, the first block will not generally require pointers to previous blocks.

[12] While we do not consider DAG-based protocols in Section 3.1, such considerations may be especially relevant for such protocols, since they have substantial metadata when n is large.

[13] Throughout the paper, we write $x||y$ to denote x concatenated with y. In particular, the instruction "sendbuffer = sendbuffer$||y$" means to add y to sendbuffer.

[14] For simplicity, we assume that p_ℓ sends data to all processors, including itself.

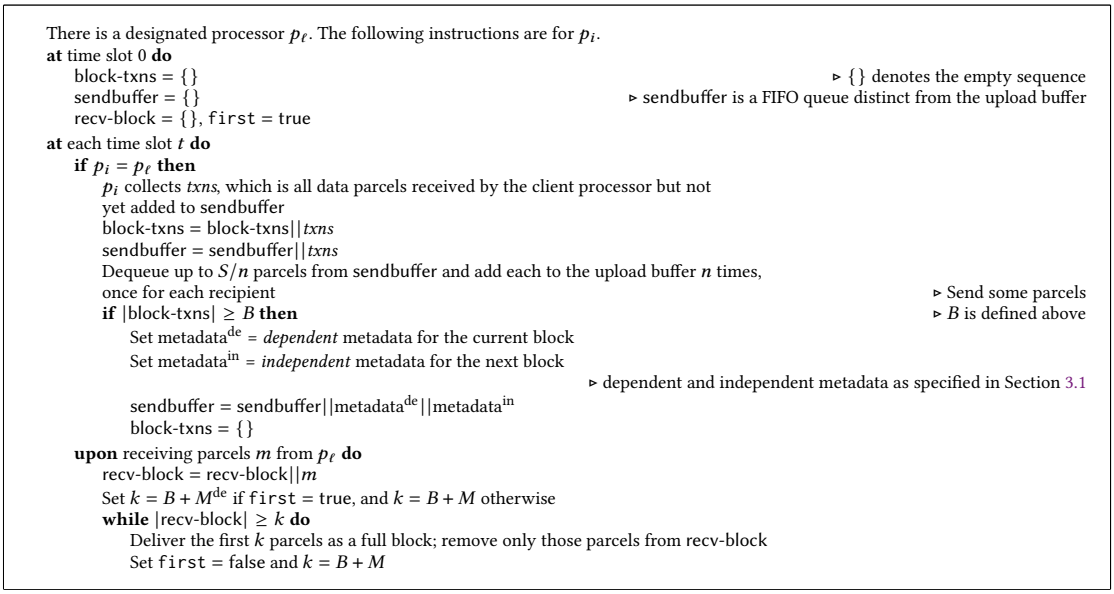

Fig. 3. Best-effort Broadcast by a single designated processor ($B = \alpha M/(1-\alpha)$)

Claim 1. *Suppose all processors are correct. Let M be the total size of the block metadata, while M^{de} is the size of the dependent metadata. If $D = \alpha S/n$ for $\alpha \in (0,1)$, then latency for the protocol described in Figure 3 is:*

$$\frac{(M + (1-\alpha)M^{de})n}{(1-\alpha)S} + \delta \;\; = \frac{(M + (1-(nD/S))M^{de})n}{S - nD} + \delta.$$

Proof. We first show that the transaction data of each block have been uploaded by the time the leader adds its dependent metadata to sendbuffer. We can then calculate latency by adding the time needed to upload that metadata and the network delay δ.

Consider first what happens between timeslot 0 and the first timeslot, t_1 say, at which p_ℓ adds metadata to sendbuffer. At each timeslot in this interval, less than S/n transactions arrive at the client processor of p_ℓ (since $\alpha < 1$), and are immediately added to sendbuffer. Each transaction is then added to p_ℓ's upload buffer n times (once for each recipient), with the upload buffer emptied of addressed transactions by the end of the timeslot. It follows that, by the end of time slot t_1, sendbuffer does not contain any transactions and that the leader's upload buffer does not contain any addressed transactions.

Now suppose that, at some timeslot t, the leader completes a block, i.e., it adds the dependent metadata for some block as well as the independent metadata for the next block to sendbuffer. Suppose (inductively) that, by the end of timeslot t, sendbuffer does not contain any transactions and that the leader's upload buffer does not contain any addressed transactions (i.e. all such parcels added to these buffers have been removed). All earlier metadata precedes these transactions in FIFO order and has therefore also been uploaded. So the only work remaining from time t is the metadata just added. Let b be the next block and let t' be the time at which the leader completes b. Recall that

$B = \alpha M/(1-\alpha)$ and that transactions arrive at rate $D = \alpha S/n$. Thus

$$t' - t = \frac{B}{D} = \frac{Mn}{(1-\alpha)S}.$$

At time t, the leader places M metadata parcels in sendbuffer. It then appends the transactions for b as they arrive. While this buffer is nonempty, the leader removes block data at rate S/n and new transactions arrive at rate D. The backlog therefore decreases at rate $S/n - D$ and is cleared after

$$\frac{M}{S/n - D} = \frac{Mn}{(1-\alpha)S} = \frac{B}{D}$$

time. Each removed parcel gives rise to n addressed copies, which the upload buffer can clear at rate S. Hence, by the end of time t', sendbuffer contains no transactions and the upload buffer contains no addressed transactions. This proves the inductive step.

Recall that the latency is the (longest possible) time gap between a transaction arriving at the client processor of the leader and all correct processors having received a full block containing the transaction. To compute the latency, we have to determine the time between t, when transactions placed in b first start arriving at the client processor of the leader, and the time t'' when all correct processors have received b. Note that, although the leader placed dependent metadata in sendbuffer at t, this was the dependent metadata for the *previous* block. Once the leader completes b at t', to determine t'' we still have to consider the extra time taken to remove the dependent metadata for b from the upload buffer, together with the message delay δ (noting that the download buffers of other processors are never a bottleneck in this analysis). Removing the addressed metadata of size M^{de} for b from the upload buffer requires $\frac{M^{\mathrm{de}}n}{S}$ time slots. As claimed, this gives a latency of:

$$\frac{Mn}{(1-\alpha)S} + \frac{M^{\mathrm{de}}n}{S} + \delta = \frac{(M + (1-\alpha)M^{\mathrm{de}})n}{(1-\alpha)S} + \delta.$$

The same expression bounds delivery of the first block. Its B transaction parcels are uploaded as they arrive, and its dependent metadata requires a further nM^{de}/S time. The receive rule uses length $B + M^{\mathrm{de}}$ for this first block and $B + M$ for later blocks, retaining parcels belonging to subsequent blocks. So it does not wait for the first block to have the independent metadata that was omitted. □

Analysis. In the analysis of this section, we set the number of transactions in each block to be $B = \frac{\alpha}{1-\alpha}M$. Of course, this value may not be an integer, in which case one should use the value $B = \lceil \frac{\alpha}{1-\alpha}M \rceil$, but we subdue such considerations here for the sake of simplicity. We note that, if $B < \frac{\alpha}{1-\alpha}M$, then latency will be unbounded. On the other hand, any $B \geq \frac{\alpha}{1-\alpha}M$ suffices for bounded latency, while $B = \frac{\alpha}{1-\alpha}M$ minimises the latency. For any such B, the interval between block completions is B/D. The preceding block's metadata and the intervening transactions are cleared within that interval since $n(B+M)/S \leq B/D$. The worst transaction latency is therefore $B/D + nM^{\mathrm{de}}/S + \delta$, including the extra waiting for arrivals when B is increased. This expression is increasing in B. As α tends to 1, block size and latency tend to infinity.

When α is fairly small, latency is roughly $(M+M^{\mathrm{de}})n/S+\delta$. If n is small enough or S is large enough that $(M+M^{\mathrm{de}})n/S$ is small compared to δ, then latency is dominated by δ. As n becomes large, the former term dominates and latency becomes roughly linear in n.

We also observe that, while the entire block (including metadata) could be sent in time $\frac{Mn}{(1-\alpha)S}+\delta$, dependent metadata causes latency to be higher. In the steady state, the protocol will transmit the block in the following order: transmit

independent metadata first, followed by the transactions in the block. Then send dependent metadata corresponding to the block. This ensures that the full bandwidth for data transmission is always utilised. Had we instead included all metadata for the block at the end of the block, this would lead to an *increased* latency of:

$$\frac{(2-\alpha)Mn}{(1-\alpha)S} + \delta.$$

A practical modification of the protocol. Since we are only concerned with latency in a setting where D and S are known, we specify the protocol of Figure 3 by fixing block size so as to minimise latency in this setting. An alternative approach, probably more practical in contexts where D is not fixed, would be for the leader to complete a block (i.e., add the corresponding dependent metadata to sendbuffer) whenever it finds that sendbuffer is empty. It is easily verified that, in the setting with fixed D, this will lead to precisely the same behaviour as the protocol of Figure 3 in the steady state.

Clearing times. The proof of Claim 1 introduces two notions that will be useful in later sections:

(a) *The extended clearing time.* Consider a timeslot t at which the protocol of Figure 3 places metadata of size M in sendbuffer (which is empty prior to this addition). If nM is small compared to S then this data will be cleared quickly from sendbuffer. If nM is large compared to S, however, then a backlog of transactions will begin to build up in sendbuffer starting at t. Further transactions arrive as we work to clear this backlog, adding to the time required to clear the backlog and reach a state in which all received transactions have been removed from sendbuffer. According to the analysis given in the proof, the backlog will finally be cleared by time $t + \frac{Mn}{(1-\alpha)S}$. While the analysis in the proof specifically considered metadata of size M, exactly the same analysis holds for *any* set of data of arbitrary size X (say). We may therefore consider the *extended clearing time* for a set of data of size X to be:

$$EC(X) := \frac{Xn}{(1-\alpha)S}$$

(b) *The clearing time.* In the proof of Claim 1 we also had to consider the time $M^{\mathrm{de}}n/S$, which is the time it takes to clear the dependent metadata from sendbuffer. More generally, we can consider the *clearing time* for a set of data of size X to be:

$$C(X) := \frac{Xn}{S}$$

Expressed using these terms, latency for the protocol of Figure 3 is:

$$EC(M) + C(M^{\mathrm{de}}) + \delta.$$

In Section 3.2, we show that this latency is optimal among simple-broadcast protocols.

3.2 Best-effort Broadcast with a single sender: lower bounds

This section is concerned with establishing lower bounds: Readers who wish to skip forward can do so without any impact on their ability to read later sections.

The protocol of Figure 3 suggests that the timing of metadata is important for minimising latency. To establish a precise optimality statement, we first formalise what we mean by protocols that do not use techniques such as erasure coding or a gossip network. We first establish optimality under the three conditions below, and then give a tight bound under the first condition alone, together with a protocol attaining that bound.

Simple-broadcast protocols. We say that a protocol is a *simple-broadcast* protocol if the following conditions hold in the executions considered:

(1) Each correct processor p_i *delivers* a transaction initially received by the client processor of p_j when p_i receives a full block b from p_j containing that transaction amongst its transaction parcels,[15] i.e., p_i must receive every block data parcel m of b from p_j before transactions in the block are delivered.[16]

(2) Correct processors send the same blocks to all processors, i.e., if p_i is correct and sends a block b to p_j then p_i must send b to all processors.

(3) For each fixed choice of the system parameters, the total number of addressed parcels remaining to be uploaded by each correct processor for blocks whose transaction contents have already been fixed is bounded independently of elapsed time.[17]

The first condition specifies how transactions are delivered: every recipient must receive the full block directly from the transaction's source. This excludes erasure coding and the use of a gossip network to improve latency. The second condition requires a source to send the same blocks to everyone, rather than grouping transactions differently for different recipients. The third condition excludes executions in which the amount of data awaiting transmission grows without bound. This is a condition on outstanding work, not just on transaction latency: delayed copies of blocks may repeat transactions that have already been delivered through other blocks.

These conditions include standard approaches to Best-effort Broadcast, such as the leader waiting until a full block is formed and then sending it to all processors. The protocol of Figure 3 is more specific in the way it separately treats independent and dependent metadata, and has lower latency than such approaches.

Claim 2. *Consider the single-sender setting with all processors correct, fixed metadata sizes $M = M^{\mathrm{in}} + M^{\mathrm{de}} > 0$, and $D = \alpha S/n$ for $0 < \alpha < 1$. The latency of any simple-broadcast protocol is at least*

$$f(M, M^{\mathrm{de}}, D, S, n, \delta) := \frac{nM}{S - nD} + \frac{nM^{\mathrm{de}}}{S} + \delta.$$

This bound is attained by the protocol of Figure 3. The statement uses the same suppression of time-slot rounding as Claim 1; Appendix A gives the integral-slot allowances.

The complete proof is given in Appendix A.

A lower bound assuming only condition (1). The first condition alone permits different recipients to receive different blocks, provided that each recipient receives every transaction in a full block directly from its source. The following bound is tight in this larger class, without any assumption on outstanding upload work.

Claim 3. *Consider the single-sender setting with all processors correct, fixed metadata sizes $M = M^{\mathrm{in}} + M^{\mathrm{de}} > 0$, and $0 < nD < S$. Any protocol satisfying condition (1) has transaction latency at least*

$$\frac{nM}{S - nD} + \frac{M^{\mathrm{de}}}{S} + \delta. \tag{1}$$

[15] So that p_i can determine when to deliver a transaction, one may suppose that transactions contain within their data the identity of the client processor to which they are sent.

[16] In particular, it does not suffice that p_i receives information (from p_j or other processors) which suffices to *recover* the block: p_i must actually receive every data parcel in b from p_j before delivering the transactions in b.

[17] The transaction contents of a block are *fixed* when its producer forms the block's dependent metadata; the precise definition is given in Appendix A.1. For this count, a required copy of a parcel for a recipient is included even if that copy has not yet been placed in the upload buffer.

This bound is attained by the protocol described in Appendix A.3. Both the bound and its attainment use the continuous-service convention of Claim 2; the appendix treats time-slot rounding separately.

The protocol attaining the bound uses different block boundaries for different recipients, staggered evenly in time. It transmits transaction parcels before fixing each block's contents, and then sends the dependent metadata for that block to its intended recipient. This reduces the dependent-metadata term from nM^{de}/S to M^{de}/S. The protocol has bounded outstanding upload work, but does not send the same blocks to all processors. So Claim 2 remains the appropriate optimality statement for the protocol in the main text, while Claim 3 identifies what can be achieved when only direct receipt of full blocks is required. The lower-bound proof and the full construction are given in Appendix A.

3.3 Best-effort Broadcast by all processors

We now consider a related primitive where every processor sends parcels to all processors in the multi-sender setting. The dissemination process is thereby parallelised, potentially leading to a more even use of the available bandwidth.

Here each client processor receives D/n transaction parcels per slot. We set $\alpha = D/S$ and use $B = \alpha M/(1-\alpha)$ transaction parcels in each block, with $0 < \alpha < 1$. The metadata sizes M^{in}, M^{de} and M have the same meanings as before.

Initialise
block-txns = {}
sendbuffer = {}
$\forall j$ recv-block$_j$ = {}

Collect txns
Collect *txns*, which is all data parcels received by the client processor but not yet added to sendbuffer
block-txns = block-txns$||$*txns*
sendbuffer = sendbuffer$||$*txns*

Transfer to upload
Dequeue up to S/n parcels from sendbuffer and add each to the upload buffer n times, once for each recipient

Receive blocks
Upon receiving block data parcels m from p_j, set recv-block$_j$ = recv-block$_j||m$
While recv-block$_j$ has a complete block as its first part:
Deliver that block and remove only its parcels from recv-block$_j$

Fig. 4. Some procedures used in Figure 5 (and in later sections)

The block boundaries in Figure 4 are determined by the known block size and metadata format. For the first Best-effort Broadcast block, the independent metadata is absent, exactly as in the single-sender rule. A size test |block-txns| $\geq B$ is interpreted with the rounding convention in Section 2; excess arrivals must not be discarded.

Each processor needs to transmit D/n parcels per timeslot. Our goal is to compute the latency for pushing these parcels to all processors. Similar to the case with a single sender, there is an upper bound on the values of D for which bounded latency is possible. In particular, there is now a throughput bottleneck at $D = S$ (as opposed to S/n). If $D > S$, each processor cannot download/upload data from/to other processors at the rate at which it is arriving.

The following instructions are for p_i.
at time slot 0 **do**
  Initialise ▷ As specified in Figure 4
at each time slot t **do**
  Collect txns ▷ As specified in Figure 4
  Transfer to upload ▷ As specified in Figure 4

  if |block-txns| $\geq \frac{\alpha}{1-\alpha} M$ **then** ▷ M denotes the total size of metadata
    ▷ $\alpha = D/S$
    Set metadata^{de} = *dependent* metadata for the current block
    Set metadata^{in} = *independent* metadata for the next block
    sendbuffer = sendbuffer||metadata^{de}||metadata^{in}
    block-txns = { }
  Receive blocks ▷ As specified in Figure 4

Fig. 5. Best-effort Broadcast by each processor

Claim 4. *Suppose $D = \alpha S$ for $\alpha \in (0, 1)$ and that all processors are correct. Then latency for the protocol in Figure 5 is optimal among simple-broadcast protocols in the multi-sender setting, and is:*

$$\frac{(M + (1-\alpha)M^{de})n}{(1-\alpha)S} + \delta \;\; = \frac{(M + (1-\frac{D}{S})M^{de})n}{S-D} + \delta.$$

Proof. For the lower bound, apply Claim 2 separately to each source at input rate D/n. The third condition in the definition of a simple-broadcast protocol bounds each source's outstanding upload work. Ignoring download contention can only decrease latency, so this gives the claimed lower bound. For attainment, the clearing argument in Claim 1, with input rate D/n, shows that the protocol of Figure 5 has this latency and bounded outstanding upload work. Every processor is now performing dissemination, and is correspondingly also receiving data from every other processor. Since no processor sends more than S/n block data parcels in each time slot, no processor receives more than $n \times S/n$ parcels in a time slot. This means that download buffers are not a bottleneck.

For a larger feasible block size B, the interval between block completions is nB/D, and the same clearing argument gives latency $nB/D + C(M^{\text{de}}) + \delta$. This includes the additional waiting for transactions and is increasing in B. □

If only condition (1) is required, the corresponding tight lower bound is

$$\ell_{\text{tx}} \geq \frac{nM}{S-D} + \frac{M^{\text{de}}}{S} + \delta. \tag{2}$$

The attaining protocol staggers block boundaries at every source and offsets the recipient order between sources so that neither upload nor download capacity is exceeded. The lower-bound proof and this extension of the construction are given in Appendix A.4, under the same continuous-service convention as Claim 3. This does not change the common-block protocol of Figure 5, which we use in the comparisons below.

The approach in which all processors transmit data is commonly used in DAG-based protocols where each processor's block for a given 'layer' references $\Theta(n)$ blocks from the previous layer (e.g. [42]). As described in Appendix B, the need to receive blocks from layer d before producing blocks for layer $d+1$ produces certain complications in the latency analysis. For the sake of simplicity, however, we can first consider a form of DAG-based protocol in which each new block for layer d is just required to point to n blocks from previous layers (the most recently received block from each processor, say). Setting $M^{\text{de}} = \lambda$ and $M^{\text{in}} = n\lambda$ then leads to the following corollary:

Corollary 1. *If $M^{de} = \lambda$, $M^{in} = n\lambda$, and if $D = \alpha S$ for $\alpha \in (0, 1)$, latency for the protocol in Figure 5 is:*

$$\frac{(n + 2 - \alpha)\lambda n}{(1 - \alpha)S} + \delta.$$

Analysis. Consider the setting above, where $M^{\text{de}} = \lambda$, $M^{\text{in}} = n\lambda$, and $D = \alpha S$ for $\alpha \in (0, 1)$ (rather than $D = \alpha S/n$ as in Section 3.1). Note that latency approaches infinity as α tends to 1. When α is fairly small, latency is roughly $\lambda n^2/S + \delta$. If n is small enough or S is large enough that $\lambda n^2/S$ is small compared to δ, then this term is dominated by δ. As n becomes large, the $\lambda n^2/S$ term dominates and latency becomes roughly quadratic in n. Interestingly, existing empirical analyses of DAG-based protocols (e.g., [25]) consider high bandwidth networks (large S) and relatively small-to-moderate n.

3.4 Best-effort Broadcast with a layered DAG

As previously noted, the approach in which all processors transmit data is commonly used in DAG-based protocols where each processor's block for a given 'layer' references $\Theta(n)$ blocks from the previous layer. The need to receive blocks from layer d before producing blocks for layer $d + 1$ produces certain complications in the latency analysis: Appendix B gives the resulting three-regime latency calculation.

3.5 Comparing latencies for Best-effort Broadcast

Of course, the principal difference between the single-sender and multi-sender settings in the context of Best-effort Broadcast (with no erasure coding) is the factor n in the point at which the throughput bottleneck appears. There are, however, some subtle trade-offs which arise due to differing sizes in metadata, and since extra latency may be induced by the need to wait for the blocks of other processors in the case that one is building a structured DAG.

Figure 6 plots the latency expressions for the single-sender protocol of Figure 3, the multi-sender protocol of Figure 5, and the layered-DAG calculation of Appendix B. To illustrate the trade-offs, we suppose processors can upload/download at a rate of 10 Gbps. We suppose transactions are 2500 bits (about 300 bytes, similar to typical Bitcoin transactions). In the single-sender setting, we suppose M is 1000 bits, while in the multi-sender setting, we suppose $M = 500 + 500n$ bits. For the plotted Best-effort Broadcast curves we set $M^{\text{de}} = 500$ bits. We suppose that the incoming transaction rate D is 10000 transactions per second and that $\delta = 0.2$ seconds. Note that these parameters mean that 'Regime C' applies in the 'layered DAG' analysis for the range of values displayed in Figure 6. The left-hand graph in Figure 6 shows the resulting latency (in seconds) as a function of n. The right-hand graph shows the same data but has a smaller scale on the latency axis, so as to emphasise (as a point of academic interest) that there are some regions in which the single-sender protocol of Figure 3 does have (slightly) lower latency than the multisender protocol of Figure 5.

It should be emphasised that these comparisons are not entirely apples-to-apples due to the inherently different assumptions of the single-sender and multi-sender settings. For example, in some contexts one may consider the assumption of the multi-sender setting (that transactions are divided evenly between processors without repetition), as rather generous: how is such a division to be achieved in a decentralised fashion, and will achieving this distribution of transactions induce substantial extra latency? In other contexts, one may feel that such a partition of transactions (or an approximation to it) is likely to arise naturally, while relaying all transactions to a leader may induce extra latency. Also, we have only considered the case that all processors are correct. Faulty processors may impact latency differently for protocols in the single-sender and multi-sender settings.

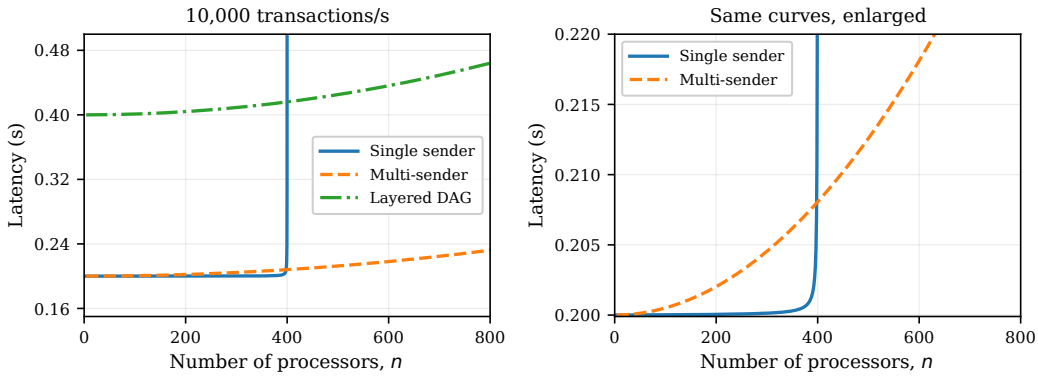

Fig. 6. Latency for BEB: parameters are explained in Section 3.5. The graph to the right is a zoomed-in version of the one to the left to display the difference between single-sender and multi-sender approaches.

3.6 Some simplifications going forward

We make the following simplifications going forward:

(1) In Section 3, we have shown that subtle issues can arise from the distinction between independent and dependent metadata. Of course, these considerations are more important when independent metadata is large. However, to simplify the analysis of subsequent sections, we will henceforth ignore such optimisations and will focus instead on the generic case that all metadata is determined at the end of the block production process. So, we will no longer distinguish between independent and dependent metadata.

(2) In Section 3, we have also given explicit instructions concerning the *receiving* of blocks. In what follows, we will assume that processors receive blocks and other messages sent to them automatically as part of the protocol instructions (protocols will be carefully constructed to ensure that download buffers are never a bottleneck). Explicit instructions will determine when processors *deliver* blocks, thus specifying the appropriate version of latency in each case.

(3) Claims 2 and 4 establish tight lower bounds for simple-broadcast protocols in the single-sender and multi-sender settings. Claim 3 and its multi-sender extension establish the corresponding tight bounds when only direct receipt of a full block from the transaction's source is required. The following sections primarily analyse latency for particular protocols, with block size chosen to minimise latency. Section 5.4 also gives a reconstruction-based lower bound for protocols using AVID-style erasure coding.

Choosing the block size. Several of the arguments below end with the same calculation. We first analyse the messages sent by a particular protocol to determine which block sizes permit bounded transaction latency, and then choose the block size that minimises latency. We describe the common calculation here to avoid repeating it.

Let B be the number of transaction parcels in each block, and let $r > 0$ be the transaction arrival rate at the processor producing those blocks. So, $r = D$ in the single-sender setting and $r = D/n$ at each producer in the symmetric multi-sender setting. In several cases, the analysis gives a condition of the form

$$\frac{B}{r} \geq T_0 + \kappa B,$$

where $T_0 > 0$ and $\kappa \geq 0$ are constants independent of B. Here B/r is the time taken for B transaction parcels to arrive. The expression on the right comes from the time required by the protocol's transmissions and waits, as calculated separately in each case.

Rearranging gives $(1 - r\kappa)B \geq rT_0$. Since $rT_0 > 0$, this inequality has a solution precisely when $r\kappa < 1$. Its smallest solution is

$$B^* = \frac{rT_0}{1 - r\kappa}. \tag{3}$$

Whenever the calculated latency increases with B, choosing this smallest permitted size minimises latency. At this choice,

$$\frac{B^*}{r} = T_0 + \kappa B^* = \frac{T_0}{1 - r\kappa}.$$

Each application must still justify the timing condition and check that the chosen block size gives the claimed behaviour from the initially empty buffers.

For reference, the applications below use the following values of r and κ. The displayed rate condition is $r\kappa < 1$; fixed metadata and delay terms affect the block size selected by (3) and the resulting latency, but not this rate condition. In the Appendix C row, $h = 1$ denotes early fragment forwarding and $h = 2$ denotes waiting for the complete fragment before forwarding.

Analysis	r	κ	Rate condition
CB, single sender	D	n/S	$S > Dn$
CB, multi-sender	D/n	n/S	$S > D$
RB, single sender	D	$2n/S$	$S > 2Dn$
RB, multi-sender	D/n	$n(n+1)/S$	$S > (n+1)D$
Coded RB, single sender	D	$6/S$	$S > 6D$
Coded RB, multi-sender	D/n	$3(n+1)/S$	$S > 3(n+1)D/n$
Pipelined coded RB	D	$3/S$	$S > 3D$
Tendermint/HotStuff	D	n/S	$S > Dn$
Appendix C, parameter h	D	$3h/S$	$S > 3hD$

Common block-production procedure. Starting in Section 4, the ordinary CB and RB protocol figures invoke a procedure called **Produce block**. In a single-sender protocol only the designated sender is a producer; in a multi-sender protocol every p_i is. When a producer executes **Produce block** with send = true, it collects the transactions that have arrived at its client processor and appends them to both block-txns and sendbuffer. If its current block then contains at least B transaction parcels , it forms the metadata for the current depth, appends that metadata after the block transactions already in sendbuffer, clears block-txns, and sets send = false. If p_i is not a producer, or if send = false, the procedure does nothing. A delivery rule that advances the producer's own stream re-enables send for the next depth. The protocol-specific rules in the relevant figure are executed after **Produce block**, and each time slot ends with "Transfer to upload" as in Figure 4. The erasure-coded and SMR protocols later in the paper state their different production rules explicitly.

4 CONSISTENT BROADCAST AND RELIABLE BROADCAST

The requirement for CB is that no two correct processors should deliver different values (blocks from the sender in our case), and that correct processors should deliver the sender's block if the latter is correct. RB has the extra requirement that, if any correct processor delivers a block from the sender, then all correct processors must do so.

We separate two parts of the analysis: the time required by the communication following a block's transaction data, and the effect of that time on the continuing transaction stream. Claim 5 provides a worked example. Lemma 1 then gives the shared argument for the remaining ordinary CB/RB calculations. The specifications are in Appendix D; the communication patterns needed to read the results are stated here.

Throughout this section, M is the block metadata size, λ is the size of a vote, and $C(X) = nX/S$ is the clearing time defined in Section 3.1. We use the model and good-case conventions of Section 2. For correctness, the protocols tolerate at most $f < n/3$ Byzantine processors using authenticated messages. We write $H(b)$ for the hash of a block b. A broadcast instance is specified by its producer and block depth; votes identify these, the protocol phase and the block hash. A correct processor votes at most once in each phase of an instance. Two sets of $n - f$ signers intersect in more than f processors, and therefore in a correct processor.

4.1 Consistent Broadcast

The two communication stages for the CB protocols we consider are: the sender broadcasts its block, and each recipient broadcasts a vote after receiving and checking it. A processor delivers after receiving the block and $n - f$ votes. With n simultaneous producers, each processor broadcasts n votes, one for each producer's block.

A *certificate* for a block consists of $n - f$ matching votes, or a threshold signature formed from those votes. The intersection property above prevents certificates for two different blocks in the same instance. A certificate also establishes that at least $n - 2f$ correct processors have checked the block (but does not by itself retrieve a missing block).[18] Our good-case calculations measure transaction delivery and assume the required blocks arrive through the stated broadcasts, without additional retrieval.

There is a designated processor p_ℓ. The following instructions are for p_i.
at time slot 0 **do**: Initialise; set $d = 1$, send = vote = true
at every time slot t **do**:
  Produce block ▷ As specified in Section 3.6
  If vote = true and p_i has received a first valid block b of depth d from p_ℓ **do**:
    Set sendbuffer = sendbuffer||(VOTE, $d, H(b)$) and vote = false
  If p_i has received (VOTE, $d, H(b')$) from $n - f$ distinct processors and has also received b' **do**:
    Deliver b'; set send = vote = true and $d = d + 1$
  Transfer to upload ▷ As specified in Figure 4

Fig. 7. Consistent Broadcast by a designated sender

4.1.1 *Consistent Broadcast by a single sender.*

In Figure 7, votes have size λ and metadata has size M.

Claim 5. *Suppose all processors are correct. If $S > Dn$ and B is set to minimise latency, then latency for the protocol of Figure 7 is:*

$$\left(\frac{2(M+\lambda)n}{S} + 4\delta\right)\left(1 + \frac{1}{2((S/(Dn)) - 1)}\right)$$

The first factor is the latency approached when Dn is small relative to S: the second factor tends to 1 in this regime. As Dn approaches S, the second factor grows without bound. The expressions derived below often have this form, with a fixed communication cost multiplied by a term that grows as the incoming rate approaches the throughput bottleneck.

[18] In DAG protocols, parent validity can be checked from certificates only when the validity conditions depend on the information certified with them; conditions involving parent contents require those contents. This distinction underlies Extractable SMR [51].

Proof. We first determine a suitable block size, and then check that the protocol repeatedly clears its transaction buffers at that size. Once the leader has finished uploading a block's transactions, it must still upload its metadata. The processors then receive the block, broadcast their votes and receive the votes. These steps take $C(M) + C(\lambda) + 2\delta$ time. Broadcasting the next block's B transactions requires a further $C(B)$ time. During the resulting interval of length $T(B)$, another $DT(B)$ transaction parcels arrive. If this exceeds B, successive blocks cannot retain size B without a growing backlog. Hence repeated operation with blocks of size B requires

$$B \geq DT(B), \qquad T(B) = C(B + M) + C(\lambda) + 2\delta.$$

Using (3) with $r = D$, $T_0 = C(M + \lambda) + 2\delta$ and $\kappa = n/S$ gives the smallest size satisfying this inequality:

$$B^* = \frac{D((M + \lambda)n + 2\delta S)}{S - Dn}. \tag{4}$$

We take $B = B^*$ and verify the repeated behaviour below.

Consider next what happens between time slot 0 and the first timeslot, t_1 say, at which p_ℓ adds metadata to sendbuffer. At each timeslot in this interval, less than S/n transactions arrive at the client processor of p_ℓ (since $S > Dn$), and are immediately added to sendbuffer (via **Produce block**). Each is then added to p_ℓ's upload buffer n times (once for each recipient, via the instruction "Transfer to upload"), with the upload buffer emptied of addressed transactions by the end of the timeslot. It follows that, by the end of time slot t_1, sendbuffer does not contain any transactions and that the leader's upload buffer does not contain any addressed transactions.

Now suppose that, at some timeslot t, the leader completes a block, i.e. while its local value send = true, it adds the metadata for some block to its sendbuffer. Suppose (inductively) that, by the end of timeslot t, its sendbuffer does not contain any transactions and that its upload buffer does not contain any addressed transactions (i.e. all such parcels added to these buffers have been removed). All correct processors will then deliver that block at time:

$$t' := t + C(M) + C(\lambda) + 2\delta = t + B(S - Dn)/DS.$$

In the interval $(t, t']$, the leader p_ℓ will therefore receive $B(S-Dn)/S$ many transaction parcels, which will be immediately added to sendbuffer at t'. To complete the block, the leader requires a further

$$B - B(S - Dn)/S = nBD/S$$

transaction parcels, which will arrive in time nB/S, i.e. $C(B)$, which is exactly the time that p_ℓ requires to clear sendbuffer of all transactions. It therefore holds that when the leader completes the next block at $t'' := t' + nB/S$, its sendbuffer does not contain any transactions and that its upload buffer does not contain any addressed transactions.

To calculate the latency, we have to consider the time from when transactions start building up at t until the next block (which the leader starts sending at t') is delivered by all correct processors. This is:

$$\frac{(B + 2(M + \lambda))n}{S} + 4\delta.$$

At the chosen size, $B/D = C(B) + C(M) + C(\lambda) + 2\delta$, so this latency is also $B/D + C(M) + C(\lambda) + 2\delta$. For any fixed-size operation, the first transaction of a block must wait for the remaining transactions to arrive and then for this final metadata and voting time. Increasing the fixed block size therefore cannot improve on the value attained here. Substituting (4) gives the claimed expression. □

One calculation for streamed blocks. The preceding proof illustrates a calculation that also arises for the multi-sender CB protocol and the two full-block RB protocols considered below. In each case a producer uploads transactions while a block is being formed, but waits for delivery of that block before starting the next one. We isolate this calculation in the following lemma to avoid repeating the same argument for each protocol. Let r denote the transaction arrival rate at one producer, and let v be the rate of transaction parcels for which that producer can upload all addressed copies. So, $r = D$ and $v = S/n$ in the preceding proof. For a block with B transaction parcels, let $\theta(B)$ be the time from the producer's last transaction upload until all processors deliver the block. We call this the *remaining communication time*. It includes the transmission of metadata and later messages, their propagation, and any waiting caused by other messages.

Lemma 1 (Streamed blocks). *Suppose that in continuous service, transaction parcels arrive at a constant rate $0 < r < v$ and consecutive blocks contain exactly $B > 0$ transaction parcels. Assume that the producer uploads available transactions at rate v, waits for delivery of the present block before starting the next, and has the same remaining communication time $\theta(B) \geq 0$ for every block. Buffers are initially empty. For $j \geq 1$, let t_j be the time at which the producer finishes uploading the transactions of block j, including their addressed copies. Under these assumptions,*

$$t_1 = B/r, \qquad t_j = \max\{jB/r,\ t_{j-1} + \theta(B) + B/v\} \quad (j > 1). \tag{5}$$

Moreover, writing $\ell_{\mathrm{tx}}(B)$ for the worst transaction latency, this latency is bounded if and only if $B/r \geq B/v + \theta(B)$, and then $\ell_{\mathrm{tx}}(B) = B/r + \theta(B)$. If, in addition, $\theta(B) = \theta_0 + \theta_1 B$ for constants $\theta_0 > 0$ and $\theta_1 \geq 0$, the smallest feasible fixed size also minimises latency and is

$$B^* = \frac{r\theta_0}{1 - r(1/v + \theta_1)}, \qquad \ell_{\mathrm{tx}}(B^*) = \frac{B^*}{r} + \theta_0 + \theta_1 B^*, \tag{6}$$

provided $r(1/v + \theta_1) < 1$.

The proof is in Appendix D.1. For each protocol below, we calculate its remaining communication time $\theta(B)$ and apply the lemma to determine the permitted block sizes and resulting latency.

For the CB and full-block RB protocols analysed in this section, $v = S/n$. When a producer is permitted to start its next block, after all processors have delivered the preceding one, the number of transactions accumulated in the meantime is $r\theta(B^*) = B^*(1 - r/v) < B^*$. Thus the threshold procedure **Produce block** agrees with the fixed-size calculation at B^* in continuous service: it does not immediately form a block larger than B^*.

Table 1. Remaining communication times for the CB and full-block RB protocols. All rows use $v = S/n$ and (6). The conditions retain the n-copy convention before the leading-order simplification of RB.

Protocol	r	θ_0	θ_1	Rate condition
CB, single	D	$C(M+\lambda) + 2\delta$	0	$S > nD$
CB, multi	D/n	$C(M) + C(n\lambda) + 2\delta$	0	$S > D$
RB, single	D	$C(2M+\lambda) + 3\delta$	n/S	$S > 2nD$
RB, multi	D/n	$C(M) + C(n(M+\lambda)) + 3\delta$	n^2/S	$S > (n+1)D$

4.1.2 Consistent Broadcast by all processors. The protocol is specified in Figure D.1 of Appendix D.2. Each producer has input rate $r = D/n$. After the initial metadata finishes, all processors receive the blocks together, broadcast their n votes, and deliver together. The remaining time to delivery is therefore $C(M) + C(n\lambda) + 2\delta$, the second row of Table 1. Appendix D.1 verifies the simultaneous sending and download conditions once for all four rows.

Claim 6. *Suppose all processors are synchronised and correct. If $S > D$ and B is set to minimise latency, then latency for the protocol of Figure D.1 is:*

$$\left(\frac{2Mn + 2\lambda n^2}{S} + 4\delta\right)\left(1 + \frac{1}{2((S/D) - 1)}\right)$$

This is (6) with the second row of Table 1; the corresponding B^* is recorded in (35). The first factor approaches the latency at low transaction rates. The second is close to 1 when D is small relative to S, and diverges as D approaches the bottleneck S.

In the protocols of Figures 7 and D.1, a producer waits for delivery before starting its next block. Beginning the next block before that delivery would be a different protocol, requiring its own analysis, especially in a layered DAG where blocks depend on earlier layers (Appendix B). We compare the two protocols as specified, rather than claiming optimality among all implementations of CB.

4.1.3 Comparing latency for Consistent Broadcast protocols. The difference between the first two rows of Table 1 is informative. Distributing transaction input among all processors changes the throughput bottleneck from S/n to S, while each processor's votes for all producers contribute $n^2\lambda/S$ to the remaining communication time. So higher throughput comes with a larger fixed cost as n grows.

Table 2 fixes the parameters shared by the CB, RB and erasure-coded RB comparison plots. Figure 8 first fixes the incoming rate at 10000 transactions per second and varies n; its second panel fixes $n = 200$ and measures transaction rate in thousands per second. As elsewhere, an input in transactions per second is converted to parcels per time unit using the stated transaction size.

Table 2. Shared parameters for Figures 8, 9 and 10. These are analytical examples, rather than experimental results.

Parameter	Value
Upload and download capacity S	10 Gbps per processor
Transaction size	2500 bits
Block metadata M	1000 bits (single); $500 + 500n$ bits (multi)
Vote size λ; propagation δ	500 bits; 0.2 seconds

4.2 Reliable Broadcast

We consider versions of Bracha's Reliable Broadcast [12] in which recipients echo the full block, then broadcast votes after $n - f$ matching echoes. A processor that has not voted may also vote after $f + 1$ matching votes. Delivery requires the block and $n - f$ votes. An echo has size $B + M$ and a vote has size λ. The extra echo stage, rather than a different batching argument, distinguishes the final two rows of Table 1. The single- and multi-sender protocols are specified in Figures D.2 and D.3 of Appendix D.3.

4.2.1 Reliable Broadcast by a single sender. After the initial transactions finish uploading, the sender must still broadcast metadata; every processor then echoes the block to all and finally broadcasts its vote. Consequently $\theta(B) = C(M) + C(B + M) + C(\lambda) + 3\delta$. The third row of Table 1 and Lemma 1 give the following result for Figure D.2.

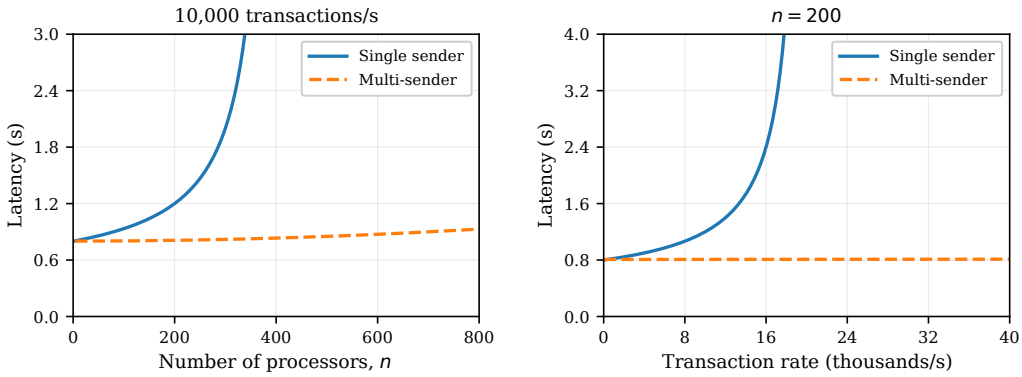

Fig. 8. Latency for Consistent Broadcast protocols: parameters are explained in Section 4.1.3

Claim 7. *Suppose all processors are correct. If $S/(Dn) > 2$ and the block size B is set to minimise latency, then the latency of the protocol of Figure D.2 is:*

$$\left(\frac{(4M+2\lambda)n}{S}+6\delta\right)\left(1+\frac{3/2}{(S/(Dn))-2}\right).$$

The first factor is the low-throughput latency approached when Dn is small compared with S. The second factor grows without bound as Dn approaches $S/2$, reflecting the two consecutive data-transmission stages in this protocol.

4.2.2 Reliable Broadcast by all processors. In Figure D.3, each processor echoes all n blocks and broadcasts n votes, using the interleaving operation $\oplus$ defined in Appendix D.2. So $\theta(B) = C(M) + C(n(B+M)) + C(n\lambda) + 3\delta$. The final row of Table 1 gives the unsimplified result

$$\ell^*_{\text{tx}} = \left(\frac{Mn+(M+\lambda)n^2}{S}+3\delta\right)\frac{2S-D}{S-(n+1)D}, \qquad S > (n+1)D. \tag{7}$$

This retains the finite-n counts, but still uses continuous service. For a simpler comparison, the leading-order approximation gives:

Claim 8. *Suppose all processors are synchronised and correct. If $S > (n+1)D$ and the block size B is set to minimise latency, then, after the approximation described in Section 2, latency for the protocol of Figure D.3 is:*

$$\left(\frac{2(M+\lambda)n^2}{S}+6\delta\right)\left(1+\frac{1}{(S/(Dn))-1}\right).$$

Corollary 2. *Set $M+\lambda = n\lambda'$. If $S > (n+1)D$ and the block size B is set to minimise latency, then, after the approximation described in Section 2, latency for the protocol of Figure D.3 is:*

$$\left(\frac{2\lambda' n^3}{S}+6\delta\right)\left(1+\frac{1}{(S/(Dn))-1}\right).$$

The three remaining block-size expressions are collected in Appendix D.1; no new clearing induction is needed for any of these rows. The specifications in Figures D.1, D.2 and D.3 retain their respective evidence rules.

4.2.3 Comparing latencies for Reliable Broadcast. Unlike Best-effort and Consistent Broadcast, the RB protocols of Figures D.2 and D.3 have leading-order bottlenecks differing only by a factor of two: $2Dn = S$ for a single sender

and $Dn = S$ for simultaneous senders. The reason for the factor of two is that a single sender pays two consecutive expensive stages, its initial broadcast and the all-to-all echo. By contrast, in the multi-sender protocol, the single stage in which every processor echoes the producers' full blocks dominates. When $M + \lambda = n\lambda'$, the fixed term is cubic in n, as Corollary 2 shows, compared with a linear term for constant single-sender metadata. This helps explain the use of CB rather than full-block RB in practical DAG dissemination.

Pipelining can overlap a single sender's next initial broadcast with others' echoes, removing the factor-two separation between those data stages. It does not remove the multi-sender requirement to echo all blocks. Section 5 examines how coding and early forwarding change these costs.

Figure 9 plots the latencies of the single- and multi-sender protocols in Figures D.2 and D.3, using the parameters of Table 2. Its first two panels fix incoming rates of 10000 and 1000 transactions per second and vary n; the lower panel fixes $n = 200$ and measures rate in thousands of transactions per second. The plotted expressions use the leading-order simplification; the more precise finite-n expression is (7).

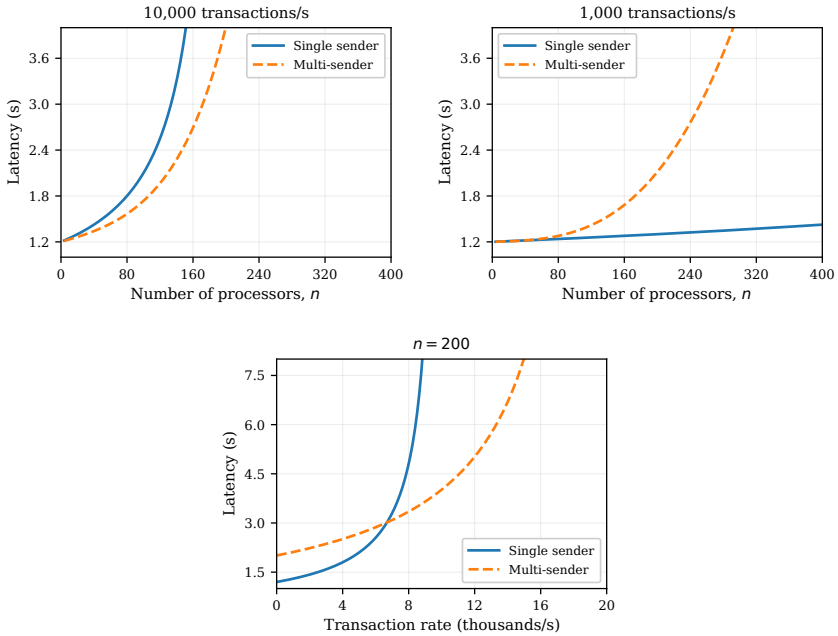

Fig. 9. Latency for RB: parameters are explained in Section 4.2.3

5 RELIABLE BROADCAST WITH ERASURE CODING AND PIPELINING

5.1 Reliable Broadcast by a single sender with erasure coding

The use of erasure coding considered here changes two features of the preceding calculation: a processor forwards a fragment rather than a complete block, and transmission begins only after the block and its authentication information have been computed. We use the Merkle-tree-based authenticated dispersal of Cachin and Tessaro [17] (AVID-H), based on information dispersal [62].

We use the following functions. As before, a block is a finite sequence of parcels, and $|m|$ denotes the number of parcels in a block m.

- An injective function F maps m to $(f_1(m), \dots, f_n(m))$. Each component has $|m|/(f+1)$ parcels, with padding when necessary, and any $f+1$ components with distinct indices determine m.
- For each $i \in [1, n]$, the function G_i maps m to the authenticated fragment $(i, f_i(m), x, y)$. Here x is the root of the Merkle tree with leaves $f_1(m), \dots, f_n(m)$, and y is a *fingerprint* witnessing that $f_i(m)$ is its ith leaf.
- The verification function V has $V(i, z, x, y) = \text{true}$ exactly when y is a valid witness that z is the ith leaf of a Merkle tree with root x. Write V_x for the set of tuples (i, z, x, y) passing this test.
- The reconstruction function R takes a finite set X of fragment tuples. It first checks that the tuples belong to one V_x, have distinct indices, and number at least $f+1$. It reconstructs a candidate block from $f+1$ of them and encodes that candidate again. It returns the block only if the resulting Merkle root is x and every tuple in X agrees with the encoding; otherwise it returns false. Thus a valid Merkle witness alone does not establish that the fragments encode a block. When R is applied to a set of protocol messages below, it acts on their enclosed fragment tuples, ignoring the message tags, depths and producer identifiers.

The protocols are specified in Figures D.4 and D.5 of Appendix D.4. We first consider the non-pipelined single-sender protocol: the sender waits for delivery before sending its next block, and each processor waits for a complete fragment before echoing it. So dispersal and echoing require separate transmission times.

For the latency calculations we assume $n = 3f + 1$ and write $\rho = n/(f+1)$ for the exact coding expansion. Under the stipulated fingerprint-size model, one fragment has size $\rho(B+M)/n + \lambda \log n$, and the total initial fragment-upload work is

$$P(B) = \rho(B + M) + n\lambda \log n.$$

The simplified claims below set $\rho = 3$, under the approximation of Section 2.[19]

For both non-pipelined coded protocols we need the same second timing calculation. Unlike the streaming rule of Lemma 1, a producer starts transmitting a block only after its contents have arrived. The following lemma therefore concerns block delivery times; isolating the argument avoids repeating it for the single- and multi-sender protocols.

Lemma 2 (Blocks encoded after formation). *Suppose that in continuous service, transaction parcels arrive at constant rate $r > 0$ and are partitioned into consecutive blocks of exactly $B > 0$ parcels. Assume that the producer starts communicating each block as soon as its contents are fixed and the preceding block has been delivered, and that the time from this start until delivery by all processors is the same value $T_c(B) \geq 0$ for every block. Buffers are initially empty. For $j \geq 1$, let t_j be the delivery time of block j. Under these assumptions,*

$$t_1 = B/r + T_c(B), \qquad t_j = \max\{jB/r, t_{j-1}\} + T_c(B) \quad (j > 1).$$

Consequently, transaction latency is bounded if and only if $B/r \geq T_c(B)$, and then its worst value is $B/r + T_c(B)$. If, in addition, $T_c(B) = T_0 + \kappa B$, where $T_0 > 0$, $\kappa \geq 0$ and $r\kappa < 1$, the smallest feasible fixed size is

$$B^* = \frac{rT_0}{1 - r\kappa} = rT_c(B^*).$$

This size minimises latency, whose value is $2T_c(B^*)$.

[19] We use the natural logarithm in the modelled fingerprint size, including in the plots. For a binary Merkle tree padded to the next power of two, the literal path contains $\lceil \log_2 n \rceil$ hashes; our $\lambda \log n$ term is therefore a simplified size convention, rather than an exact count for a fixed hash length. Appendix E.3 retains integral fragment lengths in the reconstruction lower bound.

The complete communication times for the two protocols are given in Table 3; their derivations and download checks appear in Appendix D.4.

Table 3. Complete communication times for non-pipelined coded RB. Here $P(B) = \rho(B+M) + n\lambda \log n$ and $C(X) = nX/S$; the table precedes the factor-3 and lower-order simplifications.

Protocol	r	$T_c(B)$	Rate condition
Single sender	D	$2P(B)/S + C(\lambda) + 3\delta$	$S > 2\rho D$
All processors	D/n	$(n+1)P(B)/S + C(n\lambda) + 3\delta$	$S > \rho(n+1)D/n$

At the calculated size B^*, the communication time of one block equals B^*/r, the time taken for the next B^* transactions to arrive. Thus the protocol can repeat the same block size without a growing backlog in the continuous-service calculation. Appendix D.1 proves the lemma; Appendix D.4 gives the substitutions and the terms omitted from the simplified formulas.

Claim 9. *Suppose all processors are correct and $M = c\lambda$, where c is fixed. If $S > 6D$ and the block size B is set to minimise latency, then, after the stated lower-order approximation,[20] the protocol of Figure D.4 has latency:*

$$\left(\frac{4n\log(n)\lambda}{S} + 6\delta\right)\left(1 + \frac{1}{(S/(6D)) - 1}\right).$$

5.2 Reliable Broadcast by all processors with erasure coding

The multi-sender version is specified in Figure D.5 of Appendix D.4. Each processor receives input at rate D/n and sends its own designated fragments. It then echoes all received fragments, interleaving messages from distinct producers, before broadcasting votes. The second row of Table 3 gives:

Claim 10. *Suppose all processors are synchronised and correct. If $S > 3D$,[21] then, setting blocksize B to minimise latency and after removing lower order terms, the protocol of Figure D.5 has latency:*

$$\left(\frac{6nM + 2n^2\log(n)\lambda}{S} + 6\delta\right)\left(1 + \frac{1}{(S/(3D)) - 1}\right).$$

Independent metadata. For the DAG application, independent metadata M^{in} can be sent uncoded before the fragments, while only the transactions and dependent metadata M^{de} are encoded (the commitment must then bind both parts). Missing independent metadata must remain retrievable from a correct holder; Appendix D.4 states this requirement. In Table 3, the multi-sender cycle changes by the constant $[n - \rho(n+1)]M^{\text{in}}/S$, with $M = M^{\text{in}} + M^{\text{de}}$. Its coefficient of B is unchanged. The resulting simplified latency is:

Claim 11. *Suppose all processors are synchronised and correct. If $S > 3D$, then, setting blocksize B to minimise latency and making the same lower-order approximations as above, the independent-metadata variant of multi-sender Reliable Broadcast with erasure coding has latency*

$$\left(\frac{2nM^{in} + 6nM^{de} + 2n^2\log(n)\lambda}{S} + 6\delta\right)\left(1 + \frac{1}{(S/(3D)) - 1}\right).$$

[20] The displayed formula omits $6c\lambda + n\lambda$ relative to $2n\lambda \log n$ in the fixed communication term. Equation (38) retains these terms.

[21] Before the two approximations used here, the rate condition is $S > (n+1)D/(f+1)$. For $n = 3f+1$, this is $S > 3(n+1)D/(n+2)$. Replacing $n/(f+1)$ by 3 first gives the intermediate condition $S > 3(n+1)D/n$, and then suppressing the lower order distinction between n and $n+1$ gives the displayed condition $S > 3D$.

5.3 Reliable Broadcast by a single sender with erasure coding and pipelining

The pipelined single-sender protocol is specified in Figure D.6 of Appendix D.5. Pipelining removes the two waits responsible for consecutive fragment-transmission costs. The leader constructs a new block after queuing the preceding block's initial fragments, without echoing a fragment itself. Nonleaders forward their designated fragment parcels as they arrive, and echo only after validating the complete fragment. The leader broadcasts its own echo share and participates in the vote and certificate messages, although it does not forward a received fragment. This is the protocol used for the stable-leader DispersedSimplex comparison in Section 6.3.

For the latency bound we use continuous service, with the block size specified in Appendix D.5. Since successive blocks overlap, a processor will often have several messages to send at once. It sends shares first, then fragment data, and then certificates. The appendix takes the interval between successive blocks to be slightly longer than the upload time that each processor requires for one block, and shows that each stage of the protocol is then delayed by at most the time needed to upload two shares. We do not attempt to optimise the lower order terms in the resulting bound. The appendix also specifies a rule which ensures that shares and fragment data are eventually sent in faulty executions, and which plays no role when all processors are correct.

Claim 12. *Suppose $n = 3f + 1$ with $f \geq 1$, $\lambda > 0$, and all processors are synchronised and correct. In continuous service, and with the priority rule specified in Appendix D.5, if $S > 3D$ and the block size is chosen as in (42), then the protocol of Figure D.6 has latency at most*

$$\left(\frac{6M + 2n\lambda(\log(n) + 8)}{S}\right)\left(1 + \frac{1}{(S/(3D)) - 1}\right) + 3\delta.$$

5.4 Latency lower bounds for protocols using AVID-style erasure coding

The preceding calculations concern particular ways of distributing encoded blocks. We now establish two lower bounds. The first follows from block formation and the data needed for reconstruction; the second also uses the propagation delays incurred when fragments have to be forwarded.

Encoding and reconstruction. We consider the single-sender setting and use the following assumptions for AVID-style erasure coding, following the Merkle-tree-based construction of Cachin and Tessaro [17]. The leader encodes each finite nonempty block separately into $n \geq 2$ fragments authenticated under a common Merkle root. The block's contents and the root are fixed before any of its fragment data or authentication information is uploaded. The root depends on all the encoded fragments; here we consider the usual arrangement in which the data and its authentication information are sent only after this commitment is available.

Let $k_{\mathrm{rec}} \in \{1, \ldots, n\}$ be the number of distinct complete authenticated fragments required for reconstruction. A nonleader delivers a transaction only after receiving at least k_{rec} such fragments and reconstructing a block containing that transaction. Fragments may arrive directly from the leader or through other processors. Different blocks may contain the same transaction, and different processors may deliver it through different blocks. Each fragment's data and authentication information are received in full for that block; parcels already received for another block do not replace these transmissions. No particular voting or certificate rule is required, and we impose no bound on unfinished transmissions.

For a block with B transaction parcels, suppose that the total size $P(B)$ of its n equal authenticated fragments is

$$P(B) = \rho B + A, \qquad \rho, A > 0, \tag{8}$$

where ρ and A are constants independent of B. Here A is the fixed contribution to the total fragment size, beyond the part proportional to B. Each fragment therefore has size $P(B)/n$, including authentication information and framing (the information identifying its block and fragment and marking its boundaries). An encoding with these sizes and reconstruction threshold is assumed available. In particular, a nonleader receives at least $k_{\mathrm{rec}}P(B)/n$ parcels for a block it reconstructs. The following statement uses continuous service, as in the preceding calculations.

Fragment forwarding. For the second bound below, we additionally take $k_{\mathrm{rec}} \geq 2$ and consider the designated-fragment arrangement used in the preceding protocols. For each block, a nonleader receives data and authentication information for at most one fragment directly from the leader: its designated fragment. Every parcel of any other fragment it uses for reconstruction, including the authentication information counted in that fragment, reaches it by forwarding through another nonleader. A processor forwards a parcel only after receiving it, but may begin forwarding before the complete fragment arrives. Thus at least $k_{\mathrm{rec}} - 1$ complete authenticated fragments must each traverse at least two communication links after the block is fixed. This assumption concerns fragment data and authentication information, not the small echo shares defined in Appendix D.5.

Proposition 1 (Latency lower bounds). *Under the encoding and reconstruction assumptions above, finite transaction latency requires $nS > k_{\mathrm{rec}}\rho D$, and*

$$\ell_{\mathrm{tx}} \geq \frac{2k_{\mathrm{rec}}A}{nS - k_{\mathrm{rec}}\rho D} + \delta. \tag{9}$$

Under the additional fragment-forwarding assumption, with $k_{\mathrm{rec}} \geq 2$, this strengthens to

$$\ell_{\mathrm{tx}} \geq \max\left\{\frac{2k_{\mathrm{rec}}A}{nS - k_{\mathrm{rec}}\rho D} + \delta,\ \frac{(2k_{\mathrm{rec}} - 1)A}{nS - k_{\mathrm{rec}}\rho D} + 2\delta\right\}. \tag{10}$$

The proof, given in Appendix E, counts the fragments actually received by one nonleader. Small blocks incur fixed overhead too frequently, while large blocks delay the arrival of all the transactions they contain and take longer to reconstruct. To allow repeated transactions, we assign each transaction only to the first block through which that processor reconstructs it (this is an assignment for counting, not a change to the blocks).

For a code with $\rho = n/k_{\mathrm{rec}}$, block metadata of size M, and authentication and framing of size m_F per fragment, we have $A = nM/k_{\mathrm{rec}} + nm_F$. Provided $M + k_{\mathrm{rec}}m_F > 0$, the reconstruction bound simplifies to

$$S > D, \qquad \ell_{\mathrm{tx}} \geq \frac{2(M + k_{\mathrm{rec}}m_F)}{S - D} + \delta. \tag{11}$$

With designated-fragment forwarding and $k_{\mathrm{rec}} \geq 2$, it strengthens to

$$\ell_{\mathrm{tx}} \geq \max\left\{\frac{2(M + k_{\mathrm{rec}}m_F)}{S - D} + \delta,\ \frac{(2 - 1/k_{\mathrm{rec}})(M + k_{\mathrm{rec}}m_F)}{S - D} + 2\delta\right\}. \tag{12}$$

The usual choice $k_{\mathrm{rec}} = f + 1$ gives the exact expansion $\rho = n/(f + 1)$ used earlier. Both assertions hold independently of the agreement mechanism, and the second uses the stated forwarding arrangement rather than an additional Byzantine-correctness argument. Neither requires the protocol to upload every fragment before delivery. We do not assert that these bounds are tight for Reliable Broadcast.

5.5 Comparing latency for Reliable Broadcast protocols with erasure coding

The non-pipelined single-sender calculation pays two fragment-transmission stages, whereas early forwarding and overlapping blocks pay one in the leading bandwidth constraint. The corresponding simplified bottlenecks are $S/6$ and

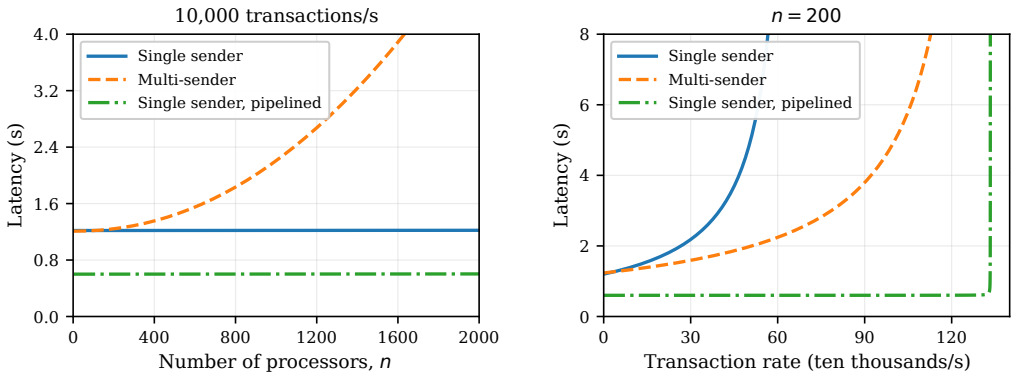

Fig. 10. Latency for RB with erasure coding: parameters are explained in Section 5.5

$S/3$. Multi-sender coding also has a leading bottleneck $S/3$, but each processor authenticates and forwards fragments for all producers, giving larger metadata terms as n grows. [22]

Figure 10 plots the calculated latencies of the single-sender, multi-sender and pipelined single-sender protocols of Figures D.4, D.5 and D.6, respectively. It does not plot the lower bounds of Proposition 1. It uses Table 2; the left panel fixes 10000 transactions per second and varies n, and the right fixes $n = 200$ and measures rate in units of 10000 transactions per second.

6 LATENCY FOR SMR PROTOCOLS

6.1 The difficulty of making apples-to-apples comparisons

For a number of reasons, making a direct comparison between protocols for the single-sender and multi-sender settings is complicated. To start with (as noted in Section 3.5), the two settings make inherently different assumptions. In some contexts, one may consider the assumption that transactions are divided evenly between processors without repetition in the multi-sender setting as generous: it is not immediately clear how such a division is to be achieved in a decentralised fashion, and whether doing so will induce substantial extra latency [30]. In other contexts, such a partition of transactions (or an approximation to it) is likely to arise naturally, while relaying all transactions to a leader may induce extra latency [33]. Our analysis so far also fails to compare how protocols will fare in the context of Byzantine actors.

Different protocols also give different guarantees in terms of data availability. In comparing latency for versions of Sailfish, the version using Consistent Broadcast is seen to have lower latency than others, but Consistent Broadcast gives weaker guarantees than Reliable Broadcast in terms of ensuring that correct processors receive the necessary blocks. These weaker guarantees may be abused by Byzantine actors and, in some cases, though not those considered here, may even lead to increased latency in the case that all processors are correct. Further complications arise from the fact that, in the single-sender setting, the use of erasure coding is the only method we have seen which allows for a throughput bottleneck at $D = \Theta(S)$ (rather than $Dn = \Theta(S)$). In reality, the use of erasure coding may be computationally expensive. Gossip networks are an alternative approach, but we leave an analysis of this method to future work. As noted in

[22] These bottlenecks refer to the protocols of Figures D.4, D.5 and D.6. More recent dissemination methods can reduce expansion towards 1.5 or below in favourable executions [50, 65]; $S/3$ is not a universal single-sender limit.

previous sections, there are also various reasons why the use of pipelining makes more sense in the single-sender setting than in the multi-sender setting.[23]

Since it is difficult to make an entirely apples-to-apples comparison, we focus on comparing representative low-latency protocols for the single-sender and multi-sender settings, while bearing in mind that there are trade-offs that remain hidden if one just considers the corresponding latency figures.

6.2 Latency for DAG-based SMR protocols

In this section, we consider latency for standard 'certified' DAG-based SMR protocols in the multi-sender setting and assume the reader is familiar with such protocols. Each processor now *delivers* a transaction upon adding a block containing that transaction to its finalised ledger. However, it is also useful while reasoning about such protocols to consider weaker notions of delivery, such as Reliable Broadcast delivery (RB-delivery) or Consistent Broadcast delivery (CB-delivery), since DAG-based protocols often use Reliable Broadcast or Consistent Broadcast as primitives.

In fact, analysing latency for such protocols is generally rather straightforward, given the analysis we have already carried out in previous sections. This is because calculating latency now amounts to counting the number of rounds of Consistent Broadcast/Reliable Broadcast (or possibly Reliable Broadcast with erasure coding, whichever is employed by the protocol in question for the task of block propagation) that are required to finalise each block.

The example of Sailfish. To make things concrete, we focus on the example of Sailfish [69], and assume familiarity with that protocol in what follows. However, the analysis we carry out here is easily adapted to deal with other DAG-based protocols.

We take $n = 3f + 1$. Each block is a vertex of the DAG, and its pointers to earlier blocks are directed edges. The *causal history* of a vertex consists of the earlier vertices reachable from it by following these edges. A Sailfish vertex in layer d has *strong edges* to at least $2f + 1 = n - f$ vertices from layer $d - 1$. It may also contain *weak edges* to older vertices that are not already in its causal history. The latter are important for the latency of non-leader vertices. A *strong path* is a directed path consisting entirely of strong edges. [24] Sailfish's weak-edge rule then adds an edge to any omitted older vertex that is not already in the new vertex's causal history. Consequently every correct layer-d vertex is in the causal history of every correct layer-$(d + 2)$ vertex in the executions we analyse.

Sailfish directly commits leader vertices; non-leader vertices are ordered when they lie in the causal history of a committed leader. In the good case analysed here, a correct layer-d leader is directly committed once it is available locally and the processor has received the first messages of the broadcasts for $2f + 1$ layer-$(d + 1)$ vertices having strong paths to that leader. One does not wait for those $2f + 1$ broadcasts to complete. In this good case, the next-layer rule also requires the preceding leader vertex to be available and includes it among the strong edges; the timeout rules are not invoked. Since every correct layer-$(d + 1)$ processor has the preceding correct leader available before constructing its vertex, these first messages provide the required strong paths.

[23] On the other hand, a very recent paper [61] describes a new DAG-based protocol called Starfish, which does employ a form of pipelining in the context of erasure codes.

[24] Our synchronisation assumption means that correct processors begin constructing a layer together. It does not require a processor whose next-layer rule is enabled after receiving $n - f$ preceding-layer vertices to wait until all n preceding-layer vertices have been processed. In particular, under the simultaneous-event convention of Section 2, even if all n layer-d vertices become available in the same displayed time slot, the layer-$(d + 1)$ leader may process $n - f$ of them first and construct its vertex before processing a particular remaining non-leader vertex b. So b may legitimately be omitted from that leader's strong edges. Equivalently, this is the limiting behaviour obtained when b arrives an arbitrarily small amount after the $n - f$ vertices that trigger construction: the arbitrarily small timing difference disappears from the latency calculation, while the discrete consequence of omitting b does not. In the all-correct synchronised execution considered below, the additional layer is enough for any such omitted correct layer-d broadcast to complete before a layer-$(d + 2)$ vertex is constructed.

Now consider a correct non-leader block b in layer d. If b is already in the causal history of the layer-$(d+1)$ leader, then its transactions are finalised when that leader is committed: after the leader block is delivered and the processor receives the requisite first messages from $2f+1$ processors of layer $d+2$. If b is omitted from that leader's causal history, the weak-edge argument above places b in the causal history of the layer-$(d+2)$ leader. In the worst case, the transactions in b are therefore finalised when the layer-$(d+2)$ leader is committed, after receipt of the requisite first messages from $2f+1$ processors of layer $d+3$.

Choice of broadcast primitive. As with other certified DAG protocols, Sailfish uses a broadcast primitive to disseminate each proposed block. The original version uses Reliable Broadcast. Our preceding analysis shows that transmitting full-block echoes in this multi-sender setting creates a much lower throughput bottleneck than Consistent Broadcast. We therefore also consider a version using Consistent Broadcast for block propagation. We apply the analogous good-case rule using CB-delivery and the initial proposal messages of the next-layer broadcasts.

We also note that there are optimisations which can sometimes be employed to reduce latency in the optimistic case: for example, one may be able to RB-deliver after two rounds of communication in certain optimistic cases [5, 70]. For the sake of simplicity, we do not consider such optimisations in what follows.

Sailfish with Consistent Broadcast as the underlying primitive. As in previous sections that considered the multi-sender setting, we simplify the analysis by assuming that all processors are correct and are 'in-sync', i.e., that all processors begin construction of each layer of the DAG simultaneously. To calculate the latency, we consider transactions included in a block b produced by p_i in layer d of the DAG. As already calculated in Section 4.1.2, the maximum delay between receipt of a transaction in b and the point at which all processors CB-deliver b is:

$$\left(\frac{2Mn+2\lambda n^2}{S}+4\delta\right)\left(1+\frac{1}{2((S/D)-1)}\right)$$

We then have to consider the latency induced by the time to CB-deliver blocks in layers $d+1$ and $d+2$ (recall, b will lie in the causal history of a leader block in at least one of these layers). For each of these layers, however, we *do not* have to consider any equivalent of the initial delay between receipt of a transaction in b and the time at which p_i starts sending b. For each of these layers the contribution to latency is:

$$\left(\frac{Mn+\lambda n^2}{S}+2\delta\right)\left(1+\frac{1}{(S/D)-1}\right)$$

.

In the worst case, b is first included in the causal history of the layer-$d+2$ leader. Finalising that leader requires the initial proposal messages from $2f+1$ processors of layer $d+3$; the corresponding broadcasts need not themselves complete. In the CB protocol analysed here, each such first message contains the full proposed block. These proposals are sent concurrently, and their aggregate incoming size at any processor is at most $n(B+M)$ under our n-rather-than-$n-1$ convention, so the final contribution is $C(B+M)+\delta$. This gives total latency:

$$\left(\frac{4(Mn+\lambda n^2)}{S}+8\delta\right)\left(1+\frac{3}{4((S/D)-1)}\right)+C(B+M)+\delta$$

Substituting in the value for B given by (35), this is:

$$\left(\frac{4(Mn+\lambda n^2)}{S}+8\delta\right)\left(1+\frac{1}{(S/D)-1}\right)+\frac{Mn}{S}+\delta. \tag{13}$$

If $M = n\lambda$, then this is:

$$\left(\frac{9\lambda n^2}{S} + 9\delta\right)\left(1 + \frac{8}{9((S/D) - 1)}\right). \tag{14}$$

We note that the expressions in (13) and (14) correspond to the worst case in which b is not in the causal history of the layer-$d + 1$ leader and is therefore finalised through the layer-$d + 2$ leader. If b is already in the causal history of the layer-$d + 1$ leader and $M = n\lambda$, then one full layer is saved and the latency is:

$$\left(\frac{7\lambda n^2}{S} + 7\delta\right)\left(1 + \frac{6}{7((S/D) - 1)}\right). \tag{15}$$

Equations (14) and (15) therefore describe two distinct inclusion cases. The 7δ expression applies when b is included in the causal history of the layer-$(d + 1)$ leader. The 9δ expression is the worst case permitted by the protocol rule: the layer-$(d + 1)$ leader proceeds after $n - f$ preceding-layer vertices without b, and b first enters the relevant leader's causal history through layer $d + 2$. The latter remains the worst case even when the ordering difference that causes the omission is arbitrarily small, because the omission incurs one complete additional DAG layer. We therefore use the 9δ case for the worst-case Sailfish comparisons below. In both cases the final term corresponds to the full initial proposal messages required by the Sailfish commit rule; we do not assume a separate independent-metadata optimisation.

Sailfish with Reliable Broadcast as the underlying primitive. The calculations here are entirely analogous to those for Consistent Broadcast, but use instead the analysis of Section 4.2.2. As in that section, we use the leading-order expression obtained after identifying $n - 1$, n, and $n + 1$ and removing lower order terms; the corresponding unsimplified finite-n expression should be used close to the bottleneck. The latency in this approximation is:

$$\left(\frac{4(M + \lambda)n^2}{S} + 12\delta\right)\left(1 + \frac{1}{(S/(Dn)) - 1}\right) + C(B + M) + \delta$$

Substituting in the value for B in (37) and making the same leading-order approximation gives:

$$\left(\frac{4(M + \lambda)n^2}{S} + 12\delta\right)\left(1 + \frac{1 + 1/(4n)}{(S/(Dn)) - 1}\right) + \frac{Mn}{S} + \delta.$$

Here the remaining omitted contribution is the lower-order term $M/[S((S/(Dn))-1)]$, consistent with the approximation already made in Section 4.2.2.

Sailfish using Reliable Broadcast with erasure coding as the underlying primitive. For this version of Sailfish we use the independent-metadata variant of multi-sender Reliable Broadcast with erasure coding described at the end of Section 5.2. Write M^{in} and M^{de} for the sizes of the independent and dependent metadata. In particular, the DAG-edge information needed to determine the relevant paths is independent of the transactions. It is therefore sent uncoded in the first message, while the transaction payload and dependent metadata are erasure coded.

As in the preceding Sailfish calculations, in the worst case we must account for the Reliable-Broadcast latency of the target layer and two following layers. Using Claim 11, these three layers contribute

$$\left(\frac{4nM^{\text{in}} + 12nM^{\text{de}} + 4n^2\log(n)\lambda}{S} + 12\delta\right)\left(1 + \frac{1}{(S/(3D)) - 1}\right).$$

The final Sailfish step is different from waiting for another Reliable Broadcast to complete. To commit the relevant leader, a processor needs the requisite first messages from $2f + 1$ processors in the next layer and must be able to determine their strong paths. For the variant just described, that information is contained in the uncoded independent

metadata; there is no need to reconstruct the transaction payloads of those next-layer vertices. The messages are sent concurrently, and the aggregate incoming independent metadata is at most nM^{in} under our n-rather-than-$n-1$ convention. Thus the final contribution is

$$C(M^{\text{in}}) + \delta = \frac{nM^{\text{in}}}{S} + \delta.$$

The resulting worst-case latency is therefore

$$\left(\frac{4nM^{\text{in}} + 12nM^{\text{de}} + 4n^2\log(n)\lambda}{S} + 12\delta\right)\left(1 + \frac{1}{(S/(3D)) - 1}\right) + \frac{nM^{\text{in}}}{S} + \delta.$$

The exact finite-n rate condition and the approximation of the expansion factor by 3 are the same as in Section 5.2. In particular, the former $C(B + M) + \delta$ term is not used here: the finalisation test needs only the independently available DAG metadata, not a full uncoded block and not a reconstructed next-layer payload.

6.3 Latency for SMR protocols using erasure coding and pipelining in the single-sender setting

As noted in Section 6.1, there are a number of reasons why making apples-to-apples comparisons between single-sender and multi-sender SMR protocol is complicated. For this reason, we compare representative low-latency protocols for the single-sender and multi-sender settings. In this section, we therefore consider latency for SMR protocols using erasure coding and pipelining in the single-sender setting. In particular, we consider the version of Simplex called *DispersedSimplex* described by Shoup [65].

Latency analysis for DispersedSimplex in the case of a stable leader. We assume familiarity with DispersedSimplex in what follows and consider latency in the context of a stable leader, assuming all processors are correct.

In fact, analysing DispersedSimplex in this setting is very simple given the work we already carried out in Section 5.3. In that section, we described a multi-shot protocol for Reliable Broadcast using erasure coding and pipelining. In doing so, we took care to specify the protocol so that it is essentially identical to DispersedSimplex in the good case that there is a correct stable leader and message delivery is reliable. Of course, the instructions for DispersedSimplex must also allow for the sending of messages, such as *complaint certificates*, in the case that the leader is faulty or message delivery is not reliable. However, these instructions do not impact latency in the good case. The latency analysis of Section 5.3 therefore carries over directly, giving the following latency bound if votes are of size λ and metadata is of size M:

$$\left(\frac{6M + 2n\lambda(\log(n) + 8)}{S}\right) \cdot \left(1 + \frac{1}{(S/(3D)) - 1}\right) + 3\delta.$$

As throughout our single-sender setting, we assume that incoming transactions are received at the client processor of the designated leader. The expression above uses the additional pipelining available when the same correct leader remains in place. Appendix C derives the corresponding rotating-leader calculation and the non-pipelined variant used in the experiments of Section 8.

6.4 Latency analysis for Tendermint

While our principal focus in Section 6 is to compare low-latency single-sender and multi-sender protocols, it is also useful to compare them with standard single-sender protocols that do not use erasure coding.

Standard terminology. A *view* is a protocol iteration with a designated leader. A *quorum certificate* (QC) records matching votes from $n - f$ distinct processors for the same block and view. A stage-k QC, also called a k-QC, uses votes

from voting stage k. A processor's *lock* records a certificate that restricts which proposals it may vote for in subsequent views. Its *highest-QC* is a known QC with greatest view number, using the highest voting stage to break ties.

Waiting in Tendermint. Tendermint-style protocols use locking to preserve safety across views: a new leader must avoid proposing a value that conflicts with a value on which correct processors may already be locked. One fixed-wait schedule is to wait 2Δ before proposing. The purpose of this wait is to allow the leader to learn relevant lock information even when a correct processor enters the view up to Δ later and its next-view message then takes up to another Δ to reach the leader. However, if the leader already has a stage-1 or stage-2 QC from the immediately preceding view, that certificate identifies a value that is safe to extend, so this additional wait is unnecessary on the successful path. Tendermint-style formulations and implementations differ in their timeout behaviour, so rather than select one schedule as canonical we parameterise the additional fixed wait. Let $w \geq 0$ denote this wait: $w = 2\Delta$ represents the fixed-wait schedule, while $w = 0$ represents the successful path in which a previous-view QC allows the leader to propose without the additional wait.

The Tendermint-style message-sending schedule is given in Figure D.7 of Appendix D. We suppose that stage-1 votes, stage-2 votes, QCs, and the next-view message are all of size λ. A next-view message carries the sender's current lock/highest-QC information, so there is no separate transmission of the lock. This version of the protocol does not use pipelining.

To make the transaction-input model unambiguous, we analyse a fixed correct leader p_ℓ and suppose that incoming transactions are received at its client processor. The same in-view message calculation applies once a rotating leader has its transaction block, but any additional client-to-leader routing is outside the present model. As elsewhere, $C(X) = nX/S$.

Suppose p_ℓ enters view v at time t. The protocol of Figure D.7 allows the leader to propose no earlier than $C(\lambda) + w$ after entering the view. This pre-proposal interval represents the specified transmission allowance for next-view messages, each of size λ, together with the additional wait w. Once the proposal starts, broadcasting a block containing B transaction parcels and M metadata parcels, broadcasting the stage-1 votes, and broadcasting the stage-2 votes take

$$C(B + M) + 2C(\lambda) + 3\delta.$$

So the proposal-to-proposal spacing is

$$T_{\mathrm{T}}(w) = C(B + M) + 3C(\lambda) + w + 3\delta. \tag{16}$$

Here $r = D$ and (16) has coefficient $\kappa = n/S$. The block-size calculation in Section 3.6 therefore gives $Dn < S$ and

$$B_{\mathrm{T}}(w) = \frac{D\big((3\lambda + M)n + (w + 3\delta)S\big)}{S - Dn}. \tag{17}$$

With this choice, $B_{\mathrm{T}}(w)/D = T_{\mathrm{T}}(w) > C(\lambda) + w$, so an initially empty leader simply continues collecting transactions after the proposal gate opens and proposes once the first $B_{\mathrm{T}}(w)$ transactions are available.

A transaction that just misses the preceding block waits at most one proposal-to-proposal interval and then the time from its own proposal to the 2-QC. Thus

$$L_{\mathrm{T}}(w) = 2C(B + M) + 5C(\lambda) + w + 6\delta.$$

Since this latency increases with B, the smallest permitted block size (17) minimises it. Substituting gives

$$L_{\mathrm{T}}(w) = \frac{(6\lambda n + 2Mn)/S + 2w + 6\delta}{(S/(Dn)) - 1} + \frac{(5\lambda + 2M)n}{S} + w + 6\delta. \tag{18}$$

The choices $w = 2\Delta$ and $w = 0$ recover the two successful-path schedules compared below.

6.5 Latency analysis for HotStuff

HotStuff [54, 78] obtains linear communication complexity within a view by relaying the voting phases through the leader. This reduces total communication but adds leader-relayed transmissions and, in the non-pipelined three-vote variant below, extra message-delay phases. Following the convention above, we do not attempt to select a unique canonical HotStuff implementation. The message-sending schedule for the concrete, non-pipelined three-vote variant that we analyse is given in Figure D.8 of Appendix D. The references provide context for the HotStuff family.

To make the comparison with Tendermint use the same transaction-input convention, we again analyse a fixed correct leader p_ℓ and suppose incoming transactions are received at its client processor. We suppose NEWVIEW messages, votes, and QCs are all of size λ, and analyse the all-correct synchronised execution. Although the protocol is allowed to act after receiving $n - f$ messages, the timing calculation below uses $C(\lambda) = n\lambda/S$, i.e., the time by which all n synchronised messages of the relevant phase have arrived. This conservative convention removes any residual quorum tail from the next phase and is the same n-rather-than-$n - f$ convention used elsewhere in the paper.

If p_ℓ enters view v at time t, collecting the NEWVIEW messages at the leader takes at most $C(\lambda) + \delta$. The leader then broadcasts a proposal containing B transaction parcels and metadata of size M. Each of the three voting stages requires votes to reach the leader, followed by a broadcast of the resulting QC. From the start of the proposal through receipt of the 3-QC, these messages therefore take $C(B + M) + 6C(\lambda) + 7\delta$: one transmission of the proposal, three transmissions of votes, and three transmissions of certificates. Thus a complete successful view has duration

$$T_{\mathrm{H}} = C(B + M) + 7C(\lambda) + 8\delta. \tag{19}$$

Again $r = D$ and (19) has coefficient $\kappa = n/S$. The block-size calculation in Section 3.6 gives $Dn < S$ and

$$B_{\mathrm{H}} = \frac{D((7\lambda + M)n + 8\delta S)}{S - Dn}. \tag{20}$$

As for Tendermint, this rule also handles the initial view. Since $B_{\mathrm{H}}/D = T_{\mathrm{H}} > C(\lambda) + \delta$, an initially empty leader simply keeps collecting transactions after the NEWVIEW messages have arrived and proposes once the first B_{H} transactions are available.

A transaction that just misses the preceding block waits one proposal-to-proposal interval and then the time from its own proposal through receipt of the 3-QC. This gives

$$L_{\mathrm{H}} = 2C(B + M) + 13C(\lambda) + 15\delta.$$

Since this latency increases with B, the smallest permitted block size (20) minimises it. Substitution yields

$$L_{\mathrm{H}} = \frac{(14\lambda n + 2Mn)/S + 16\delta}{(S/(Dn)) - 1} + \frac{(13\lambda + 2M)n}{S} + 15\delta. \tag{21}$$

Section 7.3 compares this expression with the Tendermint-style family parameterised by w, separating HotStuff's extra leader-relayed traffic and message-delay phases from any fixed proposal wait paid by that schedule.

7 COMPARING LATENCIES

The preceding sections derive latency expressions for the concrete broadcast and SMR protocols that we analyse. The primitive-level comparisons are given alongside those derivations; this section focuses on the resulting SMR comparisons for common parameter choices. Section 6.1 explains why the single-sender and multi-sender settings are not entirely apples-to-apples; the numerical comparisons below should be read with those qualifications in mind. Unless stated otherwise, the plotted comparisons in this section set the actual network delay equal to the known delay bound, $\delta = \Delta$.

7.1 Sailfish and erasure-coded single-sender protocols

We first compare Sailfish with Consistent Broadcast against the stable-leader and rotating-leader DispersedSimplex schedules analysed in Section 6.3 and Appendix C. The introduction explains why we use these protocols as concrete representatives of the multi-sender DAG and single-sender erasure-coded approaches. The comparison does not claim that either protocol is uniquely canonical or optimal within its wider design family.

We suppose that processors can upload/download at a rate of 10 Gbps. Transactions are 2500 bits (about 300 bytes, similar to typical Bitcoin transactions). In the single-sender setting, we suppose M is 1000 bits, while in the multi-sender setting, we suppose $M = 500 + 500n$ bits. We set $\lambda = 500$ bits and $\delta = \Delta = 0.2$ seconds. The first graph in Figure 11 supposes the incoming transaction rate D is 10^6 transactions per second and shows the resulting latency as a function of n. The second graph fixes $n = 400$ and shows latency as a function of the number of incoming transactions per second divided by 10^5.

The stable-leader DispersedSimplex variant has the lowest latency over much of the displayed range, while the rotating-leader curve is appreciably higher and can exceed Sailfish before the common DispersedSimplex bottleneck is reached. For the baseline DispersedSimplex dissemination scheme analysed by the displayed formulas, erasure coding expands the transaction data by a factor of approximately 3, so both DispersedSimplex curves have their leading bottleneck at $D = S/3$, while the Sailfish+CB curve has its leading bottleneck at $D = S$.

The factor 3 is not an inherent limitation of the single-sender approach. The more recent version of DispersedSimplex includes an optimisation that reduces the data expansion to approximately 1.5 in favourable executions with a stable leader [65]. The protocols of *The Carnot Bound* [50] go further: they can optimistically use data expansion rates below 1.5 when fewer processors are faulty and fall back to 1.5 when necessary. The curves in Figure 11 deliberately continue to show the factor-3 protocol that we analyse in detail; obtaining exact curves for the newer coding schemes requires their own message and timing calculations. Qualitatively, if the effective data expansion is ρ, the bandwidth bottleneck moves from the baseline $S/3$ towards S/ρ.

7.2 Comparison with Tendermint and HotStuff

Most single-sender protocols do not use erasure coding, so it is also useful to compare Sailfish with Tendermint and HotStuff. Section 6.4 parameterises a concrete two-vote Tendermint-style schedule by its fixed per-cycle wait w, and Section 6.5 specifies the concrete HotStuff variant used in the comparison.

For Figure 12, processors can upload/download at 1 Gbps, transactions are 2500 bits, $M = 1000$ bits in the single-sender setting, and $\lambda = 500$ bits. We set $\delta = \Delta = 0.2$ seconds. The first graph fixes D at 2000 transactions per second and varies n; the second fixes $n = 30$ and varies the incoming transaction rate. The two Tendermint-style curves show the reference cases $w = 2\Delta$ and $w = 0$. For reference, the figure also shows Sailfish with Reliable Broadcast.

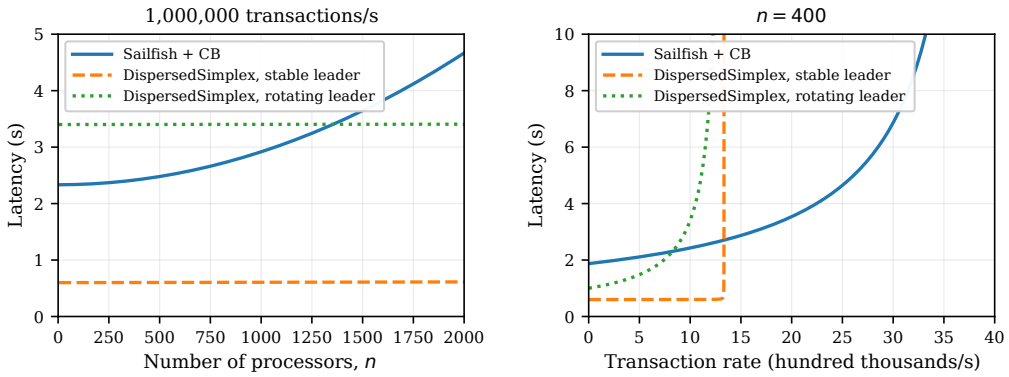

Fig. 11. Latency for Sailfish and stable- and rotating-leader DispersedSimplex. Parameters are explained in Section 7.1. The DispersedSimplex curves correspond to the factor-3 dissemination scheme analysed in this paper. The Sailfish curve uses the worst-inclusion 9δ case; Equation (15) gives the 7δ case in which the next-layer leader already includes the target block.

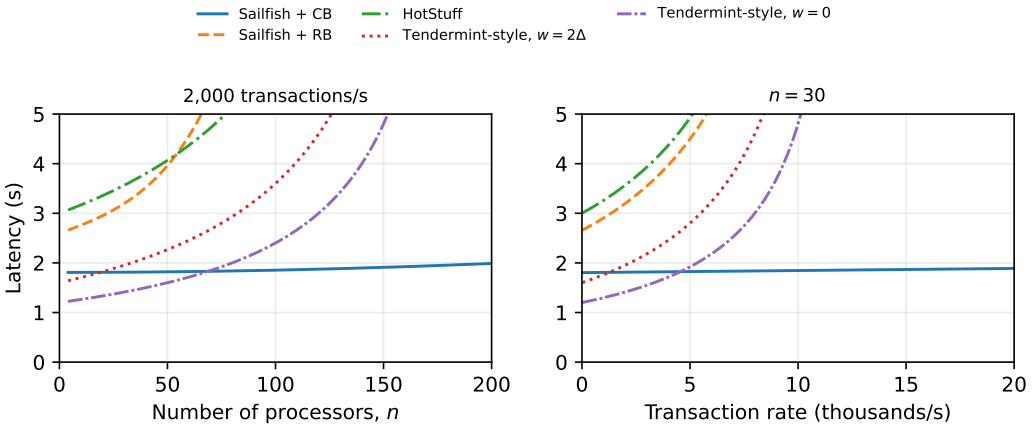

Fig. 12. Comparing latency for DAG-based and standard SMR protocols. Parameters are explained in Section 7.2. The two Tendermint-style curves correspond to $w = 2\Delta$ and $w = 0$.

7.3 Effect of the fixed wait

The Tendermint-style analysis separates the fixed per-cycle wait w from the actual network delay δ. A delay-bound-based schedule may take $w = 2\Delta$, while $w = 0$ represents the same successful-path message pattern with no additional fixed wait. Let

$$\alpha := \frac{Dn}{S} < 1.$$

Subtracting Equation (18) from Equation (21) gives

$$L_{\mathrm{H}} - L_{\mathrm{T}}(w) = \frac{8n\lambda/S + (9+\alpha)\delta - (1+\alpha)w}{1-\alpha}. \tag{22}$$

Thus HotStuff has lower latency exactly when

$$(1+\alpha)w > 8n\lambda/S + (9+\alpha)\delta.$$

For $w = 0$ the two-vote Tendermint-style schedule is faster throughout the admissible region. For $w = 2\Delta$, HotStuff can instead win when the known bound Δ is sufficiently larger than the actual delay. These are latency comparisons for the specific successful-path schedules analysed here, not claims that they have identical recovery or adversarial guarantees.

8 EVALUATION

We implement and evaluate variants of the Sailfish and DispersedSimplex SMR protocols and compare their empirical performance against our theoretical model. The two implementations represent the multi-sender DAG and single-sender erasure-coded approaches considered in the paper. Both are written in Rust. The source code for Sailfish is available at [67] and for DispersedSimplex at [66].

Experimental setup. We conduct our experiments on the Google Cloud Platform (GCP), deploying all nodes within a single region to ensure consistent baseline network conditions. Each node runs on an e2-standard-4 instance [41], equipped with 4 vCPUs, 16 GB of memory, and up to 8 Gbps network bandwidth, running Ubuntu 22.04. The native ping latency between nodes is below 1 ms. To emulate realistic network conditions in a controlled manner, we use the Linux traffic control (tc) utility [53], which supports fine-grained configuration of delay, bandwidth, and packet loss. We configure tc to enforce a uniform 100 ms one-way delay between all nodes and cap each node's upload and download bandwidth separately at 100 Mbps. This setup provides reproducible experiments under realistic network constraints.

We use TCP connections for data transfer between nodes. TCP employs congestion control, which initially limits the number of in-flight packets (typically to around 10) before additional packets can be sent. This results in higher latency at the start of a connection. As more data is transmitted, the congestion window automatically expands, enabling lower latency and higher throughput. Similarly, TCP send and receive buffers start with small default values to reduce memory usage and are gradually increased as the connection progresses, which can also introduce initial latency. To avoid these startup effects, we ensure that all TCP connections are properly warmed up (i.e., the congestion window and buffer sizes have scaled to stable values) before running our benchmarks.

After warmup, transactions are injected for 120 seconds. The experimental latency statistic is computed per block rather than per transaction. For block b_k, let t_{k-1} be the time at which the preceding block b_{k-1} is formed, and let t'_k be the time at which b_k is finalised by the last processor. We record the block-cycle latency $Y_k := t'_k - t_{k-1}$ and report the median of the Y_k values over the 120-second interval. Under the steady batching used in these experiments, transactions for b_k begin accumulating after b_{k-1} is formed. Hence Y_k upper-bounds the latency of every transaction in b_k and closely approximates the latency of the first—and therefore highest-latency—transaction accumulated for that block. In the idealised stable Pipes execution, successive block cycles are symmetric, so the median across blocks estimates the steady-state worst-within-block transaction latency predicted by the model while being less sensitive to an atypically slow block. As throughout the paper, D denotes the total transaction-data arrival rate. Transactions in the SMR experiments are 512 bytes.

Implementation details. We simplify the implementations of both Sailfish and DispersedSimplex. In our Sailfish implementation, we employ the CB primitive to disseminate proposed blocks and wait for all n vertices in each layer, rather than $n - f$ as specified in the original protocol. Furthermore, incoming transactions are first accumulated

in a buffer and proposed collectively in the next block proposal. This differs from the design in Figure D.1, where incoming transactions can be included in the block currently being disseminated: the theoretical analysis of this modified implementation is adapted accordingly.

For DispersedSimplex, we implement a non-pipelined variant in which the leader waits to obtain a quorum certificate for the previous block before issuing a new proposal. Similar to Sailfish, the leader accumulates incoming transactions in a buffer and proposes them collectively in the next block proposal. Furthermore, we omit actual erasure coding and instead transmit data of equivalent size to an erasure-coded chunk to avoid the computational overheads of encoding, decoding, and chunk verification. Additionally, both SMR implementations exclude digital signatures to eliminate cryptographic computation overhead, as this cost is not accounted for in our theoretical model.

For Sailfish, the buffering change relative to Figure D.1 can be incorporated explicitly. Put

$$A := \frac{Mn + \lambda n^2}{S}, \qquad x := \frac{S}{D}.$$

The Consistent-Broadcast calculation of Section 4.1.2 still gives the latency-minimising block size

$$B = \frac{M + \lambda n + 2\delta S/n}{x - 1}. \tag{23}$$

Because the experimental implementation waits for all n vertices of the preceding layer, a target block is in the causal history of the next-layer leader. We therefore use the shorter, one-next-leader case discussed in Section 6.2. If transactions can be streamed into the block currently being disseminated, that case has latency

$$L_{\text{stream}} = 3A + \frac{Mn}{S} + 7\delta + \frac{3(A + 2\delta)}{x - 1}.$$

In the experimental implementation, however, a producer first accumulates a full block and then queues the block as a unit. This adds exactly the extra block-upload term

$$\frac{nB}{S} = \frac{A + 2\delta}{x - 1},$$

and hence the adjusted Sailfish prediction is

$$L_{\text{Sail,exp}} = 3A + \frac{Mn}{S} + 7\delta + \frac{4(A + 2\delta)}{x - 1}. \tag{24}$$

For the experimental prediction we use the same stripped-down communication accounting as the implementation: fixed metadata, signatures, and control-message sizes are suppressed. Equation (24) then reduces to

$$L_{\text{Sail,pred}} = 7\delta + \frac{8\delta}{(S/D) - 1}.$$

Transactions in the SMR experiments are 512 bytes, so an offered rate of r transactions per second corresponds to $D = 4096r$ bits per second. With $S = 100$ Mbps and $\delta = 0.1$ seconds, the Predicted curve in Figure 13 is therefore obtained directly from

$$L_{\text{Sail,pred}}(r) = 0.7 + \frac{0.8}{10^8/(4096r) - 1} \quad \text{seconds.}$$

so the batching modification is material: relative to the corresponding streaming calculation, it adds $(A + 2\delta)/(x - 1)$ before the fixed terms are suppressed.

Appendix C derives the worst-case Pipes latency for this non-pipelined DispersedSimplex schedule. In the notation used there it is

$$L_{\mathrm{np}} = \left(\frac{12M + 4n\lambda\log(n)}{S} + 4\delta\right)\left(1 + \frac{1}{(S/(6D)) - 1}\right) + \delta,$$

with latency-minimising block size given by Equation (32) with $h = 2$. For the stripped-down experimental implementation, suppressing the fixed metadata and control-message terms gives

$$L_{\mathrm{exp}} = \frac{4\delta}{1 - 6D/S} + \delta, \qquad B_{\mathrm{exp}} = \frac{2\delta SD}{S - 6D}.$$

So this implementation has a Pipes bottleneck at $D = S/6$. With 512-byte transactions, an offered rate of r transactions per second again corresponds to $D = 4096r$ bits per second. The Predicted curve in Figure 14 is obtained directly from

$$L_{\mathrm{DS,pred}}(r) = 0.1 + \frac{0.4}{1 - 6(4096r)/10^8} \quad \text{seconds.}$$

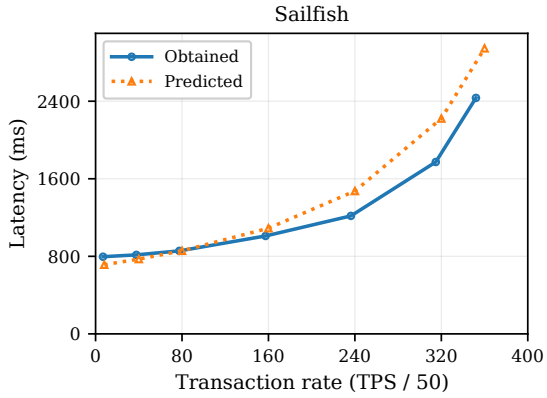

Fig. 13. Throughput vs. latency for Sailfish at $n = 40$.

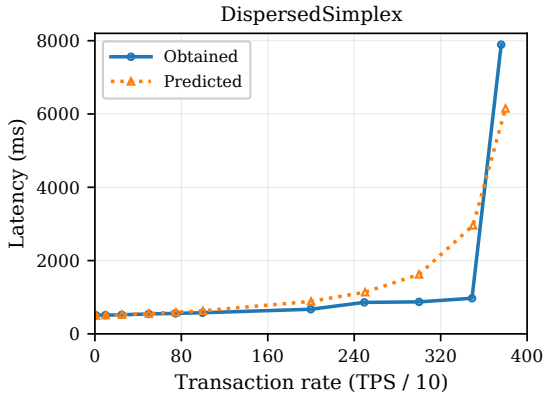

Fig. 14. Throughput vs. latency for DispersedSimplex at $n = 40$.

Performance discussion. We vary D to obtain varying throughput and latency for these protocols. The performance comparison for Sailfish is shown in Figure 13. The measured median block-cycle latency in our implementation is generally lower than the Predicted curve. One plausible explanation is that blocks are proposed in bursts, followed by idle periods during which only minimal traffic (e.g., vote messages) is exchanged. The Linux tc subsystem regulates outgoing bandwidth using token-bucket–based schedulers, which can permit short transmission bursts above the configured steady-state rate when sufficient tokens accumulate during idle periods [34]. We did not separately measure the burst contribution, so we treat this as an explanation of the discrepancy rather than as an experimental conclusion.

The performance comparison for DispersedSimplex is shown in Figure 14. The measured median block-cycle latency in our implementation is again generally lower than the Predicted curve over the middle part of the measured range. As for Sailfish, temporary bursting permitted by the token-bucket scheduler is a plausible explanation, but we did not separately measure that effect. As D approaches the predicted bottleneck $S/6$, the available bandwidth margin disappears. Near this regime the observed latency rises sharply and can exceed the Pipes prediction because storage, system-level delays, and other implementation overheads are not included in the model. The finite experiment is

consistent with the predicted saturation behaviour; it does not by itself establish divergence at the mathematical bottleneck.

Scope of the experimental comparison. The experiments in this section test the simplified implementations just described. In particular, DispersedSimplex is non-pipelined, Sailfish waits for all n vertices in each layer, and both implementations omit cryptographic operations whose computational cost is outside the Pipes model. The experiments therefore do not directly test the security or performance of the full protocols, nor do they test the stable-leader pipelined DispersedSimplex expression of Section 6.3. They also use a single configured network delay, so they do not test the separate comparison in Section 7.3 as the ratio δ/Δ varies. The Obtained points are medians over the block-cycle quantities Y_k, not the maximum latency observed over the entire 120-second run and not medians over individual transaction latencies. Their comparison with the Pipes prediction uses the regular steady-state batching assumption described above, under which Y_k tracks the highest transaction latency within a typical block. Finally, the token-bucket explanation above describes one way in which the experimental network can depart from the model's constant per-time-slot bandwidth assumption; the figures should therefore be read as a comparison of scaling and numerical predictions for these implementations, not as evidence that the Pipes expression is always a lower bound for a practical deployment.

9 RELATED WORK

Latency metrics in consensus. Consensus latency is commonly expressed in communication rounds or in multiples of a network-delay bound. Under asynchrony one instead studies round latency, worst-case or expected, because message delay itself is unbounded; Byzantine broadcast, for example, has classical worst-case round lower bounds, while randomized leader selection can yield constant expected round latency [2, 31]. In partial synchrony and synchrony, responsiveness asks whether latency depends on the actual delay δ rather than only on the pessimistic bound Δ [5, 60]. These measures normally abstract from message size. Bagaria et al. and subsequent work instead introduce an explicit network-capacity parameter for gossip-style protocols [9, 44, 59, 63]; related analyses also discretise executions into bandwidth-sensitive rounds [36]. Their capacity is essentially network-wide, whereas Pipes models the upload and download queues of each processor and keeps the dependence on n, which is needed for the single-sender/multi-sender comparisons in this paper.

Bandwidth-sensitive models and queueing frameworks. LogP combines communication latency, overhead, gap and processor count; LogGP also models long messages [6, 22]. Deterministic network calculus describes arrival and service constraints and derives backlog, delay and composition bounds [21, 47]. Pipes is closely related to this viewpoint: constant input and fixed-rate buffers form a deterministic queueing system whose traffic and dependencies are generated by a consensus protocol. Reservation and packetized service guarantees are established techniques [80], as are the event recurrences used to analyse synchronization and flow control [81]. Symbolic performance analysis has also been developed for dataflow graphs [82].

Our analysis begins with a protocol's sending and delivery rules. The shared streaming calculation separates batching from communication-stage costs, while the lower bounds in Section 5.4 combine the downloads required for reconstruction with the propagation delays incurred by fragment forwarding. We do not claim the recurrence algebra as new. Shoup's DispersedSimplex estimates likewise account for bandwidth, and additionally for computational costs that Pipes abstracts away [65].

Communication complexity, extension protocols, and data availability. A large literature instead measures the total number of messages or bits. Classical Byzantine-agreement lower bounds and successive protocol designs range from very high communication costs to quadratic and sub-quadratic constructions [1, 14, 16, 18, 19, 26, 27, 38, 45, 46, 48, 54, 78]. Extension protocols make the input length explicit. Writing ℓ for the input length in bits and κ for the security parameter in this bound, these protocols use erasure coding to obtain communication $O(n\ell + \text{poly}(n)\kappa)$ [10, 35, 37, 57]. Closely related coding ideas appear in data-availability systems and dispersal protocols [20, 28, 29, 58, 71, 74, 77], while pipelining is widely used to overlap dissemination and agreement [3, 16, 68, 78]. Communication complexity is an important input to performance, but it does not by itself determine which processor becomes the bottleneck or how queues grow as offered load approaches capacity.

Erasure-coded single-sender and DAG protocols. The factor-3 expansion in our baseline DispersedSimplex calculation is a property of that dissemination scheme, not a lower bound for leader-based protocols. The fuller DispersedSimplex construction replaces the $(n, n-2f)$ code by an $(n, n-f)$ code in favourable executions, reducing the expansion towards 1.5 without increasing optimistic proposal-commit or consecutive-proposal latency [65]. The Carnot constructions use a recovery round to permit still more aggressive optimistic coding, approaching expansion 1 when the execution is favourable and falling back to safer rates when necessary [50].

DAG-based protocols pursue the complementary strategy of distributing dissemination across processors. Narwhal demonstrated the system-level throughput benefits of a DAG dissemination layer and multiple workers [25]; many subsequent protocols also use the DAG itself in consensus [7, 8, 23, 24, 40, 42, 43, 69, 72, 73]. Their aggregate communication can be high even when throughput is high, which is one reason that per-processor bandwidth rather than aggregate communication is central to our comparisons.

Other protocol families. Our calculations focus on quorum-based SMR, but the same methodology can in principle be applied to Nakamoto-style protocols [56]. Existing work on such protocols already studies the interaction between confirmation latency, communication, coding, and finite network capacity [9, 52, 76, 79].

10 CONCLUSION AND FUTURE WORK

The Pipes model combines network delay, per-processor upload/download capacity, transaction arrival rate, and protocol-specific delivery or finalisation rules in a single latency calculation. This lets us study not only low-load latency but also the point at which continuing transaction input causes queues, and hence latency, to become unbounded. The model is deliberately simple, but it provides a common analytical language for comparing protocol designs whose round complexity and aggregate communication complexity alone do not reveal their practical latency/throughput trade-off.

The Best-effort Broadcast results also show that block formation matters for latency independently of coding or gossip. When the same blocks are sent to everyone and outstanding upload work is bounded, the protocols described in the main text are optimal. Requiring only direct receipt of a full block permits a smaller tight bound, attained by using staggered block boundaries for different recipients. The latter protocol delivers every transaction to everyone, but does not give them a common sequence of blocks.

For AVID-style erasure coding, the lower bounds combine the time needed to form a block with the downloads required to reconstruct it. When a nonleader receives only its designated fragment directly from the leader, the other reconstruction fragments must be forwarded through another processor. Their two propagation delays strengthen the lower bound even when forwarding begins before a complete fragment arrives. The proofs allow overlapping block

contents and arbitrary interleaving under the stated transmission rules, without imposing voting rules or bounds on unfinished transmissions. Whether these bounds are tight for Reliable Broadcast remains a question for future work.

Our main comparison between leader-based and DAG-based designs illustrates this point. A DAG can distribute dissemination across processors and thereby sustain a high offered load, while a stable leader can also avoid becoming the bandwidth bottleneck by using erasure coding and pipelining. For the specific baseline variants analysed here, Sailfish reaches the higher bottleneck, whereas stable-leader DispersedSimplex has lower latency over much of the admissible range. The difference in their bottlenecks comes principally from the factor-3 data expansion in the baseline DispersedSimplex dissemination scheme. For these baseline schemes, transmitting all initial fragments with expansion ρ gives the leading single-sender bandwidth constraint $D < S/\rho$; newer schemes can reduce ρ substantially. Hybrid designs such as Autobahn and BBCA-Chain suggest further comparisons between these architectural choices [39, 55].

The Tendermint/HotStuff comparison exposes a different trade-off. Writing w for the additional fixed wait in the two-vote Tendermint-style schedule, the three-phase HotStuff variant analysed here wins only when that wait is large enough to compensate for HotStuff's additional leader-relayed traffic and message-delay phases. Parallel-leader uses of HotStuff, such as VABA, may behave differently because the communication load is spread across leaders [4].

Several limitations are natural directions for future work. We analyse optimistic executions with correct processors and synchronous delivery; faults, view changes, heterogeneous and time-varying delays, and different quorum-completion patterns can all change the queueing picture. The network model also abstracts from transport protocols and congestion control, peer-to-peer topology, minimum packet sizes, and computational costs such as erasure coding and signature generation. Extending Pipes to incorporate these effects, and applying it to further quorum-based and Nakamoto-style protocols, would test how much of the simple bandwidth/latency structure survives in more detailed models. The effect of limiting the number of simultaneously undelivered blocks is another direction for future work.

Use of generative AI. The original version of this paper, available as IACR ePrint Report 2025/1116, and the version accepted for publication at the IEEE Symposium on Security and Privacy were prepared without the use of generative AI. Generative AI tools, including ChatGPT (OpenAI) and Claude (Anthropic), were used in preparing the present journal version, principally to collate, reorganise, and edit material from those earlier versions and to assist with subsequent revisions. The authors reviewed the resulting manuscript and take full responsibility for its contents.

Acknowledgement. This work is partially funded by an academic gift grant by Stellar Foundation and Sui Foundation.

REFERENCES

[1] Ittai Abraham, TH Hubert Chan, Danny Dolev, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. 2019. Communication complexity of byzantine agreement, revisited. In *ACM PODC'19*. 317–326.

[2] Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, and Ling Ren. 2019. Synchronous Byzantine Agreement with Expected O (1) Rounds, Expected Communication, and Optimal Resilience. In *FC'19*. Springer, 320–334.

[3] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. 2020. Sync hotstuff: Simple and practical synchronous state machine replication. In *IEEE S&P'20*. IEEE, 106–118.

[4] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. 2019. Asymptotically optimal validated asynchronous byzantine agreement. In *ACM PODC'19*. 337–346.

[5] Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. 2021. Good-case latency of byzantine broadcast: A complete categorization. In *ACM PODC'21*. 331–341.

[6] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. 1995. LogGP: Incorporating Long Messages into the LogP Model—One Step Closer Towards a Realistic Model for Parallel Computation. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '95)*. 95–105. https://doi.org/10.1145/215399.215427

[7] Balaji Arun, Zekun Li, Florian Suri-Payer, Sourav Das, and Alexander Spiegelman. 2025. Shoal++: High Throughput {DAG}{BFT} Can Be Fast and Robust!. In *USENIX NSDI '25*. 813–826.

[8] Kushal Babel, Andrey Chursin, George Danezis, Lefteris Kokoris-Kogias, and Alberto Sonnino. 2023. Mysticeti: Low-Latency DAG Consensus with Fast Commit Path. *arXiv preprint arXiv:2310.14821* (2023).

[9] Vivek Bagaria, Sreeram Kannan, David Tse, Giulia Fanti, and Pramod Viswanath. 2019. Prism: Deconstructing the blockchain to approach physical limits. In *ACM CCS'19*. 585–602.

[10] Amey Bhangale, Chen-Da Liu-Zhang, Julian Loss, and Kartik Nayak. 2022. Efficient adaptively-secure byzantine agreement for long messages. In *ASIACRYPT'22*. Springer, 504–525.

[11] Dan Boneh, Ben Lynn, and Hovav Shacham. 2001. Short signatures from the Weil pairing. In *International conference on the theory and application of cryptology and information security*. Springer, 514–532.

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

[13] Ethan Buchman. 2016. *Tendermint: Byzantine fault tolerance in the age of blockchains.* Ph. D. Dissertation. University of Guelph.

[14] Ethan Buchman, Jae Kwon, and Zarko Milosevic. 2018. The latest gossip on BFT consensus. *arXiv preprint arXiv:1807.04938* (2018).

[15] Aaron Buchwald, Stephen Buttolph, Andrew Lewis-Pye, Patrick O'Grady, and Kevin Sekniqi. 2024. Frosty: Bringing strong liveness guarantees to the snow family of consensus protocols. *arXiv preprint arXiv:2404.14250* (2024).

[16] Vitalik Buterin and Virgil Griffith. 2017. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437* (2017).

[17] Christian Cachin and Stefano Tessaro. 2005. Asynchronous verifiable information dispersal. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*. IEEE, 191–201.

[18] Miguel Castro, Barbara Liskov, et al. 1999. Practical byzantine fault tolerance. In *OSDI (99, 1999)*. 173–186.

[19] Pierre Civit, Muhammad Ayaz Dzulfikar, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, Jovan Komatovic, and Manuel José Ribeiro Vidigueira. 2022. Byzantine Consensus is Θ (nˆ 2): The Dolev-Reischuk Bound is Tight even in Partial Synchrony! *DISC'22* 11 (2022), 1–11.

[20] Shir Cohen, Guy Goren, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. 2023. Proof of availability and retrieval in a modular blockchain architecture. In *FC'23*. Springer, 36–53.

[21] Rene L. Cruz. 1991. A Calculus for Network Delay. I. Network Elements in Isolation. *IEEE Transactions on Information Theory* 37, 1 (1991), 114–131. https://doi.org/10.1109/18.61109

[22] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. 1993. LogP: Towards a Realistic Model of Parallel Computation. *ACM SIGPLAN Notices* 28, 7 (1993), 1–12. https://doi.org/10.1145/173284.155333

[23] Xiaohai Dai, Guanxiong Wang, Jiang Xiao, Zhengxuan Guo, Rui Hao, Xia Xie, and Hai Jin. 2024. LightDAG: A Low-latency DAG-based BFT Consensus through Lightweight Broadcast. *Cryptology ePrint Archive* (2024).

[24] George Danezis and David Hrycyszyn. 2018. Blockmania: from block dags to consensus. *arXiv preprint arXiv:1809.01620* (2018).

[25] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. 2022. Narwhal and tusk: a dag-based mempool and efficient bft consensus. In *Eurosys'22*. 34–50.

[26] Danny Dolev and Rüdiger Reischuk. 1985. Bounds on information exchange for Byzantine agreement. *Journal of the ACM (JACM)* 32, 1 (1985), 191–204.

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

[28] EigenLabs. 2023. Intro to EigenDA: Hyperscale Data Availability for Rollups. https://www.blog.eigenlayer.xyz/intro-to-eigenda-hyperscale-data-availability-for-rollups/ Accessed on March 20, 2024.

[29] Ethereum. 2024. Data availability | ethereum.org. https://ethereum.org/en/developers/docs/data-availability/ Accessed on March 20, 2024.

[30] Ethereum. n.d.. Proposer Builder Separation. https://ethereum.org/en/roadmap/pbs/

[31] Michael J Fischer and Nancy A Lynch. 1982. A lower bound for the time to assure interactive consistency. *Information processing letters* 14, 4 (1982), 183–186.

[32] 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.

[33] Arbitrum Foundation. n.d.. Decentralized Timeboost Specification. https://research.arbitrum.io/t/decentralized-timeboost-specification/9676

[34] Linux Foundation. n.d.. TC-TBF Linux Manual Page. https://man7.org/linux/man-pages/man8/tc-tbf.8.html

[35] Chaya Ganesh and Arpita Patra. 2021. Optimal extension protocols for byzantine broadcast and agreement. *Distrib. Comput.* 34, 1 (Feb. 2021), 59–77. https://doi.org/10.1007/s00446-020-00384-1

[36] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. 2024. The bitcoin backbone protocol: Analysis and applications. *J. ACM* 71, 4 (2024), 1–49.

[37] Juan A. Garay, Rosario Gennaro, Charanjit Jutla, and Tal Rabin. 2000. Secure distributed storage and retrieval. *Theor. Comput. Sci.* 243, 1–2 (jul 2000), 363–389. https://doi.org/10.1016/S0304-3975(98)00263-1

[38] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling byzantine agreements for cryptocurrencies. In *ACM SOSP'17*. 51–68.

[39] Neil Giridharan, Florian Suri-Payer, Ittai Abraham, Lorenzo Alvisi, and Natacha Crooks. 2024. Autobahn: Seamless high speed BFT. In *ACM SOSP'24*. 1–23.

[40] Adam Gkagol, Damian Lesniak, Damian Straszak, and Michał Swietek. 2019. Aleph: Efficient atomic broadcast in asynchronous networks with byzantine nodes. In *AFT'19*. 214–228.

[41] Google Cloud Platform n.d.. Google Cloud Platform - General-purpose machine family for Compute Engine. https://cloud.google.com/compute/docs/general-purpose-machines#e2_machine_types Online; accessed 06-April-2025.

[42] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. 2021. All you need is dag. In *ACM PODC'21*. 165–175.

[43] Idit Keidar, Oded Naor, Ouri Poupko, and Ehud Shapiro. 2023. Cordial Miners: Fast and Efficient Consensus for Every Eventuality. In *DISC'23*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik.

[44] Lucianna Kiffer, Joachim Neu, Srivatsan Sridhar, Aviv Zohar, and David Tse. 2024. Nakamoto Consensus under Bounded Processing Capacity. In *ACM CCS'24*. 363–377.

[45] Valerie King, Jared Saia, Vishal Sanwalani, and Erik Vee. 2006. Scalable leader election. In *SODA'06*. 990–999.

[46] Leslie Lamport, Robert Shostak, and Marshall Pease. 2019. The Byzantine generals problem. In *Concurrency: the works of leslie lamport*. 203–226.

[47] Jean-Yves Le Boudec and Patrick Thiran. 2001. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Lecture Notes in Computer Science, Vol. 2050. Springer. https://doi.org/10.1007/3-540-45318-0

[48] Andrew Lewis-Pye. 2022. Quadratic worst-case message complexity for State Machine Replication in the partial synchrony model. *arXiv preprint arXiv:2201.01107* (2022).

[49] Andrew Lewis-Pye, Kartik Nayak, and Nibesh Shrestha. 2026. The Pipes Model for Latency and Throughput Analysis. In *Proceedings of the 2026 IEEE Symposium on Security and Privacy*. IEEE, 1970–1988. https://doi.org/10.1109/SP63933.2026.00255

[50] Andrew Lewis-Pye and Patrick O'Grady. 2026. The Carnot Bound: Limits and Possibilities for Bandwidth-Efficient Consensus. https://doi.org/10.48550/arXiv.2603.11797 arXiv:2603.11797 [cs.DC]

[51] Andrew Lewis-Pye and Ehud Shapiro. 2025. Morpheus Consensus: Excelling on trails and autobahns. https://doi.org/10.48550/arXiv.2502.08465 arXiv:2502.08465 arXiv:2502.08465, version 2.

[52] Chenxin Li, Peilun Li, Dong Zhou, Zhe Yang, Ming Wu, Guang Yang, Wei Xu, Fan Long, and Andrew Chi-Chih Yao. 2020. A decentralized blockchain with high throughput and fast confirmation. In *USENIX ATC'20*. 515–528.

[53] Linux traffic control n.d.. Linux traffic control. https://linux.die.net/man/8/tc Online; accessed 06-Nov-2025.

[54] Dahlia Malkhi and Kartik Nayak. 2023. Hotstuff-2: Optimal two-phase responsive bft. *Cryptology ePrint Archive* (2023).

[55] Dahlia Malkhi, Chrysoula Stathakopoulou, and Maofan Yin. 2024. BBCA-CHAIN: One-Message, Low Latency BFT Consensus on a DAG. In *International Conference on Financial Cryptography and Data Security*.

[56] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008).

[57] Kartik Nayak, Ling Ren, Elaine Shi, Nitin H Vaidya, and Zhuolun Xiang. 2020. Improved Extension Protocols for Byzantine Broadcast and Agreement. In *DISC'20*.

[58] Kamilla Nazirkhanova, Joachim Neu, and David Tse. 2022. Information dispersal with provable retrievability for rollups. In *AFT'22*. 180–197.

[59] Joachim Neu, Srivatsan Sridhar, Lei Yang, David Tse, and Mohammad Alizadeh. 2022. Longest chain consensus under bandwidth constraint. In *ACM AFT'22*. 126–147.

[60] Rafael Pass and Elaine Shi. 2018. Thunderella: Blockchains with optimistic instant confirmation. In *EUROCRYPT'18*. Springer, 3–33.

[61] Nikita Polyanskii, Sebastian Mueller, and Ilya Vorobyev. 2025. Starfish: A high throughput BFT protocol on uncertified DAG with linear amortized communication complexity. *Cryptology ePrint Archive* (2025).

[62] Michael O Rabin. 1989. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM (JACM)* 36, 2 (1989), 335–348.

[63] Coretti-Drayton Sandro, Matthias Fitzi, Aggelos Kiayias, Giorgos Panagiotakos, and Alexander Russell. 2024. High-Throughput Blockchain Consensus under Realistic Network Assumptions. In *https://iohk.io/en/research/library/papers/high-throughput-blockchain-consensus-under-realistic-network-assumptions/*.

[64] 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.

[65] Victor Shoup. 2024. Sing a Song of Simplex. In *38th International Symposium on Distributed Computing (DISC 2024) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 319)*. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 37:1–37:22. https://doi.org/10.4230/LIPIcs.DISC.2024.37 Full version: Cryptology ePrint Archive, Report 2023/1916.

[66] Nibesh Shrestha. n.d.. DispersedSimplex Implementation. https://github.com/nibeshrestha/hydrangea/tree/dispersed-simplex

[67] Nibesh Shrestha. n.d.. Sailfish Implementation. https://github.com/nibeshrestha/pipes/tree/sailfish

[68] Nibesh Shrestha, Ittai Abraham, Ling Ren, and Kartik Nayak. 2020. On the optimality of optimistic responsiveness. In *ACM CCS'20*. 839–857.

[69] Nibesh Shrestha, Rohan Shrothrium, Aniket Kate, and Kartik Nayak. 2025. Sailfish: Towards improving the latency of dag-based bft. In *2025 IEEE S&P*. IEEE, 1928–1946.

[70] Nibesh Shrestha, Qianyu Yu, Aniket Kate, Giuliano Losa, Kartik Nayak, and Xuechao Wang. 2025. Optimistic, Signature-Free Reliable Broadcast and Its Applications. In *ACM CCS'25*.

[71] Alex Skidanov and Illia Polosukhin. 2019. Nightshade: Near protocol sharding design. *URL: https://nearprotocol. com/downloads/Nightshade. pdf* 39 (2019).

[72] Alexander Spiegelman, Balaji Aurn, Rati Gelashvili, and Zekun Li. 2024. Shoal: Improving DAG-BFT Latency And Robustness. In *FC'24*.

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

[74] Espresso Systems. 2023. Designing the Espresso Sequencer: Combining HotShot Consensus with Tiramisu DA. https://hackmd.io/@EspressoSystems/HotShot-and-Tiramisu Accessed on March 20, 2024.
[75] Preston Vander Vos, Alberto Sonnino, Giorgos Tsimos, Philipp Jovanovic, and Lefteris Kokoris-Kogias. 2025. BlueBottle: Fast and Robust Blockchains through Subsystem Specialization. https://doi.org/10.48550/arXiv.2511.15361 arXiv:2511.15361 [cs.DC]
[76] Lei Yang, Yossi Gilad, and Mohammad Alizadeh. 2022. Coded Transaction Broadcasting for High-throughput Blockchains. arXiv:2205.01797 [cs.NI] https://arxiv.org/abs/2205.01797
[77] Lei Yang, Seo Jin Park, Mohammad Alizadeh, Sreeram Kannan, and David Tse. 2022. {DispersedLedger}:{High-Throughput} byzantine consensus on variable bandwidth networks. In *USENIX NSDI '22*. 493–512.
[78] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT consensus with linearity and responsiveness. In *ACM PODC'19*. 347–356.
[79] Haifeng Yu, Ivica Nikolić, Ruomu Hou, and Prateek Saxena. 2020. OHIE: Blockchain Scaling Made Simple. In *IEEE S&P'20*. 90–105. https://doi.org/10.1109/SP40000.2020.00008
[80] Abhay K. Parekh and Robert G. Gallager. 1993. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking* 1, 3 (1993), 344–357. https://doi.org/10.1109/90.234856
[81] François Baccelli, Guy Cohen, Geert Jan Olsder, and Jean-Pierre Quadrat. 1992. *Synchronization and Linearity: An Algebra for Discrete Event Systems*. Wiley.
[82] Adnan Bouakaz, Pascal Fradet, and Alain Girault. 2017. Symbolic Analyses of Dataflow Graphs. *ACM Transactions on Design Automation of Electronic Systems* 22, 2, Article 39 (2017), 1–25. https://doi.org/10.1145/3007898

A BEST-EFFORT BROADCAST: LOWER BOUNDS

We first prove Claim 2, assuming all three conditions in the definition of a simple-broadcast protocol. We then prove Claim 3, which assumes only direct receipt of a full block from the transaction's source, and give a protocol that attains the latter bound. Finally, we extend both results to the multi-sender setting. Throughout, the input rate is D, the leader's upload capacity is S, and sending to all includes the sender. Each recipient's copy of a block contains its full metadata. The metadata size is $M = M^{\text{in}} + M^{\text{de}} > 0$, apart from the possible omission of independent metadata from the first block. This first-block exception changes the counts below by a constant only.

As elsewhere in the paper, the displayed bounds suppress time-slot rounding. We give the integral-slot allowances at the end of the proofs. The proofs fix an arbitrary execution satisfying the hypotheses and bound its worst transaction latency. They therefore also bound the protocol's latency, which takes the worst case over executions.

A.1 The tight bound for simple-broadcast protocols

Proof of Claim 2. The basic idea is to bound how frequently the leader can form blocks. Upload capacity gives an upper bound on this frequency, because every block requires metadata. Bounded transaction latency gives a lower bound, because a transaction arriving just after one block is formed must wait for a later block. Combining the two bounds gives the result.

Suppose that transaction latency in the execution is bounded by L; otherwise the result is immediate. For each block that is eventually broadcast, consider the time at which the producer forms the block's dependent metadata. We say that the transaction contents of the block are *fixed* at this time. Since dependent metadata, such as a signature on the block, depends on the transactions included in the block, it cannot be formed before all of those transactions have arrived. So, every transaction in a block has arrived by the time its contents are fixed. If $M^{\text{de}} = 0$, so that there is no dependent metadata to form, we instead say that the transaction contents are fixed at the time at which the last of the block's transactions arrives at the producer's client processor. The producer may already have uploaded some transaction parcels by the time the contents are fixed, but the dependent metadata cannot be uploaded before then. Let $N(t)$ count the nonempty broadcast blocks whose transaction contents are fixed by time t; blocks that are never broadcast are not counted. Independent metadata may be uploaded earlier, and we allow that in the count.

Let $Q(t)$ be the number of addressed metadata parcels still to be uploaded for the $N(t)$ blocks whose contents were fixed by time t, including copies not yet placed in the upload buffer. The third condition in the definition of a simple-broadcast protocol gives a constant K such that $Q(t) \leq K$ at every time (only this consequence of the third condition is needed below).

Counting the blocks. Consider a timeslot $t > L$. Since the execution starts at time 0 with no transactions, and transactions arrive at constant rate D, the number of transaction parcels arriving during the interval from 0 to $t - L$ is $D(t - L)$. If transaction latency is at most L, every recipient has delivered all of these transactions by time t. By condition (1), each recipient must receive every transaction parcel directly from the leader as part of a full block. The leader must therefore have uploaded at least $nD(t - L)$ addressed transaction parcels by time t.

Now consider the $N(t)$ blocks whose contents have been fixed by time t. By condition (2), the leader must send each such block to all n recipients. Their metadata therefore requires $nMN(t)$ addressed parcels, apart from the possible omission of at most nM^{in} parcels from the first block. By the definition of $Q(t)$, at most $Q(t)$ of these metadata parcels remain to be uploaded. Hence at least $nMN(t) - Q(t) - nM^{\text{in}}$ addressed metadata parcels have already been uploaded by time t.

The transaction parcels and metadata parcels counted above are distinct, and the leader can upload at most St addressed parcels by time t. Adding the two counts therefore gives

$$St \geq nD(t - L) + nMN(t) - Q(t) - nM^{\text{in}}. \tag{25}$$

Rearranging this inequality, we have

$$\frac{N(t)}{t} \leq \frac{S - nD}{nM} + \frac{nDL + Q(t) + nM^{\text{in}}}{nMt}.$$

Since $Q(t) \leq K$ and K is independent of t, the second term on the right tends to zero as t tends to infinity. It follows that

$$\limsup_{t \to \infty} \frac{N(t)}{t} \leq \frac{S - nD}{nM}. \tag{26}$$

Additional copies of transactions and blocks can only increase the upload work; the count does not discard their cost or assume that their first deliveries occur in the same order at different recipients.

Gaps between block-formation times. Since transactions continue to arrive at a positive rate, bounded transaction latency requires infinitely many nonempty broadcast blocks. Moreover, inequality (25), together with $Q(t) \leq K$ and $M > 0$, implies that $N(t)$ is finite for every finite t. Several blocks may have their transaction contents fixed at the same time. We therefore consider the times at which at least one of the blocks counted by $N(t)$ has its transaction contents fixed. There are infinitely many such times, and only finitely many of them occur before any given time. So, we can list them in increasing order as $t_1 < t_2 < \cdots$, where this sequence tends to infinity.

Consider a transaction arriving immediately after t_i. It cannot belong to any block whose contents have already been fixed. Consequently, every block through which a recipient delivers that transaction has its contents fixed at time t_{i+1} or later. Each of the n recipients must then receive the dependent metadata for one such block. These need not be copies of the same block, but in each case the dependent metadata cannot be uploaded before that block's contents are fixed. So, if $M^{\text{de}} > 0$, at least nM^{de} addressed metadata parcels must be uploaded at or after t_{i+1}. The leader can upload at most S addressed parcels per time unit, and the last required parcel takes a further δ time to reach its recipient. If $M^{\text{de}} = 0$, the bound below follows instead from the fact that a recipient cannot have received a full block before

its contents have been fixed and its last parcel has reached that recipient. Taking the arrival time of the transaction arbitrarily close to t_i from above therefore gives

$$L \geq t_{i+1} - t_i + \frac{nM^{\mathrm{de}}}{S} + \delta.$$

This bound does not assume metadata is sent immediately when it becomes available.

Put $h = L - \delta - nM^{\mathrm{de}}/S$. Since $t_{i+1} > t_i$, the inequality above requires $h > 0$ and shows that every gap $t_{i+1} - t_i$ is at most h. In particular, $t_j \leq t_1 + (j-1)h$ for every $j \geq 1$. Therefore, by any time $t \geq t_1$, at least $\lfloor (t - t_1)/h \rfloor + 1$ distinct block-formation times have occurred. Since $N(t)$ counts at least one block at each of these times, we obtain

$$\liminf_{t \to \infty} \frac{N(t)}{t} \geq \frac{1}{h}. \tag{27}$$

Combining (26) and (27) gives $h \geq nM/(S - nD)$ and hence

$$L \geq \frac{nM}{S - nD} + \frac{nM^{\mathrm{de}}}{S} + \delta = \frac{(M + (1-\alpha)M^{\mathrm{de}})n}{(1-\alpha)S} + \delta, \qquad \alpha = Dn/S. \tag{28}$$

□

Attaining the bound. In the protocol of Figure 3, the block size is fixed for each fixed choice of parameters below the bottleneck. The proof of Claim 1 shows that the preceding metadata and the intervening transactions are cleared by the next block completion. In particular, the outstanding broadcast work is bounded independently of elapsed time. The protocol attains (28) under the displayed-formula convention, establishing optimality in the class specified in Claim 2.

Time-slot rounding. In an integral-slot execution, take the transaction in slot $t_i + 1$. If $M^{\mathrm{de}} > 0$, the last of the required nM^{de} metadata parcels cannot leave the upload buffer before $t_{i+1} + nM^{\mathrm{de}}/S - 1$. For $M^{\mathrm{de}} = 0$, it suffices that a full block copy cannot be uploaded before its contents have been fixed. So the gap bound uses $h = L - \delta - nM^{\mathrm{de}}/S + 2$ instead. Endpoint terms in (25) are bounded constants and disappear after division by t. The literal integral-slot lower bound is therefore the right-hand side of (28) minus at most two time slots.

A.2 The lower bound assuming only direct receipt

Proof of the lower bound in Claim 3. Suppose that transaction latency is bounded by L; otherwise the result is immediate. We count completed block copies separately at each recipient, which avoids both the requirement that all recipients receive the same blocks and any bound on unfinished upload work.

For each recipient, assign each transaction to the block through which that recipient first delivers it; break simultaneous ties by a fixed ordering of blocks. (This is only a way of counting first deliveries. Transactions are not removed from blocks, and their assignments may differ between recipients.)

Fix a recipient p, and consider a block b to which at least one transaction is assigned for p. Let t_0 be the infimum of the arrival times of the transactions assigned to b for p, let t_1 be the time at which the transaction contents of b are fixed, and let t_2 be the time at which p receives the full block b. Each transaction assigned to b for p is first delivered by p at time t_2. Since latency is at most L, each of these transactions arrives no earlier than $t_2 - L$, and so $t_2 \leq t_0 + L$. The dependent metadata in the copy of b sent to p must be uploaded after t_1, requiring at least M^{de}/S time, and propagation takes a further δ. If $M^{\mathrm{de}} = 0$, it suffices instead that a full copy of the block cannot be uploaded before its contents have been fixed. Consequently,

$$t_1 \leq t_2 - \delta - M^{\mathrm{de}}/S.$$

The transactions assigned to b for p all belong to b, and so have arrived by time t_1. By the choice of t_0, they therefore all arrive in $[t_0, t_1]$. Put $h = L - \delta - M^{\mathrm{de}}/S$. The two inequalities above give $t_1 - t_0 \leq h$. Since transactions arrive at constant rate D, at most Dh transactions arrive in an interval of length h. So, at most Dh transactions are assigned to b for p. Other transactions in b may have arrived before t_0, but p first delivers those through other blocks, to which they are assigned.

Now consider a time $t > L$. By time t, the recipient p has delivered all $D(t-L)$ transactions arriving by $t - L$. Each of these is assigned for p to a block that p has received in full by time t, and each such block accounts for at most Dh of them. Only finitely many blocks can be received in full by time t: in continuous service a non-empty block may contain an arbitrarily small quantity of transaction data, but all blocks other than possibly the first contain $M > 0$ metadata parcels. Since $D(t-L) > 0$, it follows that $h > 0$, and that p must have received at least $(t-L)/h$ such blocks in full by time t. The same holds for each of the n recipients. Counting the metadata in these block copies and the minimum required transaction uploads gives

$$St \geq nD(t-L) + \frac{nM(t-L)}{h} - nM^{\mathrm{in}}. \tag{29}$$

The last term allows for omission of independent metadata from the first block received at each recipient. Dividing by t and taking the limit yields $S \geq nD + nM/h$, and hence

$$L \geq \frac{nM}{S - nD} + \frac{M^{\mathrm{de}}}{S} + \delta.$$

□

Time-slot rounding. In integral slots, the interval $[t_0, t_1]$ contains at most $L - \delta - M^{\mathrm{de}}/S + 2$ arrival slots. Substituting this value of h in (29) gives the displayed lower bound minus at most two slots, as in Appendix A.1. The attainment argument below concerns continuous service; it does not assert that the same unrounded expression is the exact optimum for every choice of integral parameters.

A.3 A protocol attaining the broader bound

The idea is to use a separate sequence of blocks for each recipient. Each sequence contains every transaction exactly once, but the block boundaries for different recipients are staggered. This lets the leader finish one recipient's block at a time, rather than having to transmit dependent metadata to everyone after a common block boundary.

We work in continuous service and put

$$T = \frac{nM}{S - nD}, \qquad t_k = \frac{kT}{n} \quad (k \geq 0), \qquad j(k) = 1 + ((k-1) \bmod n) \quad (k \geq 1). \tag{30}$$

At time t_k, for $k \geq 1$, the leader fixes the contents of a block b_k intended only for $p_{j(k)}$. Its transaction parcels are those arriving in

$$[\max\{0, t_k - T\}, t_k).$$

Each block also has independent and dependent metadata of sizes M^{in} and M^{de}, respectively. We include the full independent metadata even in the initial blocks. For each recipient, these transaction intervals partition the incoming stream: its first block contains the available prefix, and subsequent blocks cover consecutive intervals of length T.

As in the main-text protocol, transactions may be sent before the block's contents are fixed. The leader maintains a separate, initially empty local FIFO queue for each recipient, containing one copy of every transaction not yet uploaded

to that recipient. These local queues are distinct from the physical upload buffer. Parcels are added to the physical buffer only at the rates specified next, so that the buffer serves them in the prescribed order without rearranging its contents.

The upload schedule. During each interval $[t_k, t_{k+1})$, for $k \geq 0$, the leader performs the following operations in order:

(1) If $k \geq 1$, upload the dependent metadata of b_k to $p_{j(k)}$ at rate S, taking M^{de}/S time. When $k = 0$, leave this amount of time idle instead.[25]
(2) Upload the independent metadata of b_{k+1} to $p_{j(k+1)}$ at rate S, taking M^{in}/S time.
(3) For the remainder of the interval, upload transactions from the local queues at rate S/n to each recipient, preserving their order of arrival.

The independent metadata in the second operation is computable at t_k; it may, for example, refer to the preceding block in the same recipient's sequence, whose contents are already fixed, using a fixed initial value for the first block. The first two operations occupy M/S time, strictly less than $T/n = M/(S - nD)$. During the third operation, the combined upload rate is S. Thus every operation respects the leader's upload capacity.

The schedule determines the order and sizes of the transaction and metadata portions sent to each recipient. A recipient can therefore parse its stream by counting parcels, without extra messages for block boundaries. Independent metadata need not precede all the transaction parcels of its block. The recipient delivers a block's transactions when all its parcels, including both parts of its metadata, have arrived.

Proof of attainment in Claim 3. We first show by induction that every local transaction queue is empty at each time t_k. This is true at $t_0 = 0$. Suppose it holds at t_k. During the first M/S time of the interval, each queue accumulates DM/S transaction parcels. In the remaining time, each queue is served at rate S/n while new transactions continue to arrive at rate D. The net clearing rate is therefore $S/n - D > 0$. The identity

$$\left(\frac{S}{n} - D\right)\left(\frac{T}{n} - \frac{M}{S}\right) = \frac{DM}{S}$$

shows that the queue empties exactly at t_{k+1}. Until that time it has enough pending transactions to sustain the specified service. This also proves the inductive step in the initial interval, where the first operation is idle rather than an upload.

It follows that all transaction parcels of b_k have been uploaded to its recipient by t_k. Its independent metadata was uploaded in $[t_{k-1}, t_k)$, and its dependent metadata is uploaded by $t_k + M^{\mathrm{de}}/S$. Since the leader is the only sender and never uploads at a combined rate exceeding S, no recipient receives parcels at a rate exceeding its download capacity S. Uniform propagation therefore adds δ and no download waiting. The full block reaches its recipient by

$$t_k + \frac{M^{\mathrm{de}}}{S} + \delta.$$

At each recipient, a transaction waits at most T for the boundary of its block, including the shorter initial blocks. Its latency is consequently at most

$$T + \frac{M^{\mathrm{de}}}{S} + \delta = \frac{nM}{S - nD} + \frac{M^{\mathrm{de}}}{S} + \delta,$$

which matches the lower bound. □

[25] There is no block b_0, so the leader has no dependent metadata to send at the start of the first interval. Leaving this time idle means that every interval follows the same schedule, which simplifies the proof below. It does not affect optimality: the leader uploads at full capacity throughout every later interval, and latency is the worst case over all transactions. Using the idle time to send transactions would reduce latency only for some of the transactions arriving before t_1.

At the time a block's contents are fixed, only its dependent metadata remains to be uploaded. This is cleared before the next block is fixed, so the outstanding work for formed blocks is at most M^{de}. So the construction also satisfies condition (3); the improvement comes from allowing different blocks for different recipients, not from accumulating unfinished transmissions. It delivers the same transactions to everyone, but does not provide a common sequence of blocks for use by a subsequent consensus protocol.

A.4 The multi-sender setting

In the multi-sender setting, every processor produces blocks. When considering a processor in this role, we refer to it as a *source*: each source receives transactions at rate D/n at its client processor and has upload capacity S. We consider the two lower bounds in turn, and then describe a protocol attaining the second.

The lower bound for simple-broadcast protocols. Suppose that all three simple-broadcast conditions hold. The proof of Claim 2 applies to each source separately, with input rate D/n in place of D. That proof considers only the parcels that a source must upload, and so remains valid when other processors are also sending: contention at the download buffers can only postpone receipt, and so cannot reduce latency. This gives

$$\ell_{\mathrm{tx}} \geq \frac{nM}{S-D} + \frac{nM^{\mathrm{de}}}{S} + \delta.$$

The protocol in Figure 5 attains this expression and has bounded outstanding broadcast work, as established in Claim 4.

The lower bound assuming only direct receipt. Now suppose only that condition (1) holds. Applying the proof of the lower bound in Claim 3 to each source, in the same way, gives

$$\ell_{\mathrm{tx}} \geq \frac{nM}{S-D} + \frac{M^{\mathrm{de}}}{S} + \delta,$$

which is the bound (2).

A protocol attaining the latter bound. Each source uses the construction of Appendix A.3 at input rate D/n, now with $T = nM/(S-D)$ and $t_k = kT/n$. All sources start together. At time t_k, source p_i fixes a block intended for

$$p_{j_i(k)}, \qquad j_i(k) = 1 + ((i+k-2) \bmod n).$$

The transaction interval and the three upload operations are otherwise unchanged.

The single-source induction shows that each source's transaction queues empty at every t_k. We must also check the download capacities. During a dependent-metadata operation, the recipients $p_{j_i(k)}$ are distinct as i varies, so each recipient receives at rate at most S from one source. The same is true during an independent-metadata operation, with $k+1$ in place of k. During transaction uploads, each source sends at rate S/n to each recipient, giving combined rate S at each recipient. Uniform propagation preserves these bounds. Hence download buffers add no waiting, and the construction attains (2). The two-slot allowance for each lower bound remains valid in integral slots; as before, the equality for the construction is stated in continuous service.

B BEST-EFFORT BROADCAST WITH A LAYERED DAG

Section 3.3 lets a processor determine all metadata for its next block without waiting for the current layer from the other processors. In a layered DAG, by contrast, a block in layer $d+1$ must contain pointers to blocks in layer d. We

isolate that additional dependency while retaining the simplifying assumption used in the main comparison: all correct processors begin each layer simultaneously. Write $\alpha = D/S \in (0,1)$, split the metadata as $M = M^{\text{in}} + M^{\text{de}}$, and retain the notation $C(X) = nX/S$ and $EC(X) = C(X)/(1-\alpha)$ from Section 3.3.

To see how the three regimes below arise, recall that in the protocol of Figure 5 a processor appends the independent metadata for its next block to sendbuffer at the moment it completes the present block. In a layered DAG this is not possible: the independent metadata for a layer-$(d+1)$ block consists of pointers to the layer-d blocks, and so is not available until those blocks have arrived. The only issue is whether the previous layer arrives before sendbuffer has cleared.

- In Regime A, the layer-d blocks arrive while the processor is still clearing the dependent metadata of its own layer-d block, together with the transactions that have arrived in the meantime. The pointers can then be appended before sendbuffer empties, so waiting for the previous layer costs nothing and latency is as in Claim 4.
- In Regime B, the layer-d blocks arrive too late for this, and nothing is gained by sending the independent metadata early. All metadata for a block is then sent together, once the block's transactions have been sent. The time between successive blocks is still determined by the time needed to clear sendbuffer.
- In Regime C, the incoming transaction rate is low enough that sendbuffer clears before the previous layer arrives. The time between successive blocks is then determined by the time taken to send a block's metadata and for it to arrive, and processors have spare upload capacity.

The conditions for the three regimes, and the resulting latencies, are as follows:

Regime	Condition	Latency
A	$EC(M^{\text{de}}) \geq C(M^{\text{de}}) + \delta$	$EC(M) + C(M^{\text{de}}) + \delta$
B	not A, and $EC(M) \geq C(M) + \delta$	$EC(M) + C(M) + \delta$
C	$EC(M) < C(M) + \delta$	$2(C(M) + \delta)$

Equivalently, these are

$$\frac{(M + (1-\alpha)M^{\text{de}})n}{(1-\alpha)S} + \delta, \qquad \frac{(2-\alpha)Mn}{(1-\alpha)S} + \delta, \qquad \frac{2Mn}{S} + 2\delta,$$

respectively.

To see the calculation for Regime A, suppose all processors finish producing their layer-d blocks at time t, with sendbuffer empty, and append the dependent metadata for those blocks. They continue collecting and sending transactions for layer $d+1$. All layer-d blocks arrive by $t + C(M^{\text{de}}) + \delta$, at which point each processor appends the independent metadata for its layer-$(d+1)$ block. The regime condition ensures that this happens before the queue created by the preceding dependent metadata and continuing transaction arrivals has cleared. So sendbuffer remains busy until $t' = t + EC(M)$, just as in Section 3.3. At t' each processor completes its layer-$(d+1)$ block by appending its dependent metadata, again with sendbuffer empty immediately beforehand. This restores the same state for the next layer. A transaction arriving just after t is delivered by $t' + C(M^{\text{de}}) + \delta$, giving the stated latency and block size $B = (D/n)EC(M)$.

In Regimes B and C, continue streaming transactions but append all metadata for a block together, once its transactions have cleared sendbuffer and the preceding layer has arrived. Suppose this is done for layer d at time t. Its metadata and the transactions arriving behind it clear sendbuffer by $t + EC(M)$, while all layer-d blocks arrive by $t + C(M) + \delta$. The next block is therefore completed at $t' = t + \max\{EC(M), C(M) + \delta\}$, with the same state restored. Its metadata reaches all processors by $t' + C(M) + \delta$. Considering a transaction arriving just after t gives the resulting latency bound. The corresponding block size is $B = (D/n)(t' - t)$. Regime B is governed by $EC(M)$ and Regime C by $C(M) + \delta$, giving the

two remaining expressions in the table. The first layer can use the same block size and has no preceding-layer wait, so it does not exceed these bounds. This is the only aspect of the layered-DAG calculation used in Figure 6.

C ADDITIONAL DISPERSEDSIMPLEX CALCULATIONS

The stable-leader calculation in Section 6.3 uses early fragment forwarding and overlaps successive dispersals. Here we record the corresponding rotating-leader schedules and the stripped-down schedule used in the experiments. For these calculations, we omit the upload times of echo shares, votes and certificates, while retaining their network delays and the stated fragment sizes. We also use the same factor-3 approximation and identification of $n-1$ with n as in Section 5.3. As in that section, the current leader sends the initial fragments but does not forward its own fragment. This prevents initial dispersal and forwarding from competing for the leader's upload capacity. We assume that the designated leader has the transaction data for its proposal available at the prescribed rate D; any additional routing of transactions between successive leaders is outside this calculation. Let

$$P(B) := 3(B+M) + n\lambda\log(n)$$

be the total size of the initial fragments sent by the leader for one block. Under the stated approximation, each recipient also sends $P(B)$ parcels when it forwards its own fragment to all processors. Put $h = 2$ if a recipient waits for its complete fragment before forwarding it, and $h = 1$ if fragment parcels are forwarded as they arrive. The next leader proposes when the preceding dispersal and echo step have completed, concurrently with the final vote for the preceding block. Under the approximation above, the proposal-to-proposal interval is therefore

$$T_h(B) = \frac{hP(B)}{S} + 2\delta = \frac{3h(B+M) + hn\lambda\log(n)}{S} + 2\delta. \tag{31}$$

For $h = 2$ the leader-to-recipient and recipient-to-all serialisation costs are paid consecutively; for $h = 1$ they form a pipeline and the serialisation cost is paid once. In both cases the two network delays remain.

The block-size calculation (3), with $r = D$ and $\kappa = 3h/S$, gives

$$B_h = \frac{D\big(3hM + hn\lambda\log(n) + 2\delta S\big)}{S - 3hD}, \qquad S > 3hD. \tag{32}$$

A transaction arriving just after a proposal at time t is included in the next proposal, at $t + T_h(B)$. The dispersal and echo step for its own block then finish at $t + 2T_h(B)$, when the final vote is sent. This vote arrives after one more network delay δ. Thus the latency is $2T_h(B) + \delta$, which is increasing in B, and substituting the block size above gives

$$L_h = \left(\frac{6hM + 2hn\lambda\log(n)}{S} + 4\delta\right)\left(1 + \frac{1}{(S/(3hD)) - 1}\right) + \delta. \tag{33}$$

So the no-early-forwarding schedule ($h = 2$) has bottleneck $D = S/6$, while early forwarding ($h = 1$) has bottleneck $D = S/3$, the same leading bottleneck as the stable-leader calculation but with the additional network-delay term caused by rotating the leader. The rotating-leader curve in Section 7.1 uses $h = 1$; the experimental variant below uses $h = 2$.

The experimental DispersedSimplex variant. Section 8 implements the $h = 2$ schedule: a new proposal is not issued until the quorum certificate for the preceding block has been obtained, and the implementation transmits data of the same size as an erasure-coded fragment rather than performing the coding operation itself. Suppressing the fixed metadata and control-message terms in (33) gives

$$L_{\text{exp}} = \frac{4\delta}{1 - 6D/S} + \delta, \qquad D < S/6, \tag{34}$$

with block size $B = 2\delta DS/(S - 6D)$. This is the Pipes prediction compared with the implementation in Section 8.

D PROTOCOL SPECIFICATIONS AND SUPPORTING CALCULATIONS

This appendix proves the shared timing statements, records their protocol-specific substitutions, and gives the complete specifications for Sections 4 and 5. The results and comparisons remain in the main text.

D.1 Shared block-timing arguments

The lemmas here concern the continuous-service interpretation used for the displayed ordinary broadcast calculations. They do not replace the protocol-specific communication counts or assert exact timings for indivisible parcels.

Proof of Lemma 1. Let $\tau = B/r$ be the time between the arrivals of the last transactions of successive blocks, and put $T(B) = B/v + \theta(B)$. We first establish the recurrence. Before the first block is complete, the producer uploads transactions as they arrive, since $v > r$. It therefore finishes uploading them at $t_1 = \tau$.

For $j > 1$, let $t'_j = t_{j-1} + \theta(B)$ be the delivery time of block $j - 1$, when the producer may start uploading block j. Since the transactions of block $j - 1$ must already have arrived, $t'_j \geq (j - 1)\tau$. If all B transactions of block j are available at t'_j, uploading them takes B/v time. Otherwise the producer first works on the transactions already available and then on later arrivals. Because $v > r$, it either catches up with those arrivals and finishes at $j\tau$, or continues at full rate and finishes at $t'_j + B/v$. So:

$$t_j = \max\{j\tau, t'_j + B/v\},$$

which is (5).

Induction in this recurrence gives

$$t_j = \tau + (j - 1)\max\{\tau, T(B)\}.$$

If $T(B) > \tau$, even the last transaction in block j, arriving at $j\tau$, waits at least $(j - 1)(T(B) - \tau)$ before its initial upload is complete. Latency is therefore unbounded. If $T(B) \leq \tau$, then $t_j = j\tau$ and all processors deliver block j at $j\tau + \theta(B)$. The first transactions in that block arrive immediately after $(j - 1)\tau$, so the supremum of their latencies is $\tau + \theta(B)$. This proves the feasibility condition and latency expression.

Finally, if $\theta(B) = \theta_0 + \theta_1 B$, feasibility is

$$B \geq r\big(\theta_0 + (1/v + \theta_1)B\big).$$

Equation (3), with $T_0 = \theta_0$ and $\kappa = 1/v + \theta_1$, gives the value of B^* in (6). Since $B/r + \theta_0 + \theta_1 B$ increases with B, this value also minimises latency. □

Derivation of the entries in Table 1. Lemma 1 requires two quantities for each protocol: the rate v at which a producer can upload all addressed copies of its transaction parcels, and the remaining communication time $\theta(B)$. We first derive these for the four protocols in the table, and then check that the lemma applies.

A producer must upload n addressed copies of each transaction parcel and has upload capacity S, so $v = S/n$. After the last transaction upload, its metadata takes $C(M)$ to upload and δ to arrive. CB then has one stage in which votes are broadcast, which takes $C(\lambda) + \delta$ when there is a single producer and $C(n\lambda) + \delta$ when all n processors are producers. RB inserts an echo stage before those votes, of duration $C(B + M) + \delta$ or $C(n(B + M)) + \delta$, respectively. Adding these durations gives the four values of $\theta(B)$ in the table.

These durations suppose that every processor uploads the messages of each stage at its full rate S, beginning as soon as the preceding stage ends, and that the messages are received δ after they are uploaded. Three points need to be checked.

All processors begin and end each stage together. In the multi-sender protocols, the producers have equal input rates and equal block sizes, and begin with empty buffers at the same time. In each stage, every processor therefore has the same quantity of data to send as every other, and it interleaves the messages that it sends on behalf of different producers (interleaving is defined in Appendix D.2). So, each stage begins at the same time at every processor, and ends at the same time. In particular, a processor that acts upon receiving $n - f$ messages, rather than n, does not act any earlier, because the messages from different senders are completed at the same time. The convention for simultaneous events in Section 2 can affect which messages a processor uses to meet such a threshold, but not the time at which it does so.

Download buffers add no waiting. Each processor divides its uploads equally among the n recipients. Since it uploads at rate at most S, it sends to each recipient at rate at most S/n. Each processor therefore receives at a total rate of at most S, which is its download capacity. This holds for every stage, including the stage in the single-sender protocols in which only the producer is sending. Since the delay is the same between all pairs of processors, parcels arrive at the same rates at which they were uploaded, and are received δ after they are uploaded.

Blocks contain exactly B^* transactions. Lemma 1 concerns blocks of a fixed size, whereas **Produce block** forms a block as soon as it has collected at least B transactions. We must therefore check that, when $B = B^*$, a producer never has more than B^* transactions waiting when it is permitted to start a new block. While a producer waits for delivery of its present block, $r\theta(B^*) = B^*(1 - r/v) < B^*$ transactions arrive. So, when it resumes, it does not yet have a full block. The remaining $B^* r/v$ transactions arrive in time B^*/v, which is exactly the time needed to upload the addressed copies of all B^* transactions. The producer therefore completes a block of exactly B^* transactions at a time when it has no transactions left to upload, just as for the first block, and the same cycle repeats. By induction, Lemma 1 applies to every block. □

For the second, third and fourth rows respectively, the block-size expressions are

$$B^*_{\text{CB,multi}} = \frac{M + \lambda n + 2\delta S/n}{(S/D) - 1}, \tag{35}$$

$$B^*_{\text{RB,single}} = \frac{D(2nM + n\lambda + 3\delta S)}{S - 2nD}, \tag{36}$$

$$B^*_{\text{RB,multi}} = \frac{D(M + (M + \lambda)n + 3\delta S/n)}{S - D - Dn}. \tag{37}$$

Substitution into (6) proves Claims 6 and 7, and gives (7) for the last row. Replacing the stated lower-order terms there gives Claim 8 and Corollary 2. In particular, $S > (n + 1)D$, not $S > nD$, is the unsimplified rate condition for the last row under the n-copy convention. Omitting self-sends would change several stages and is not a change to only one denominator.

Proof of Lemma 2. Put $\tau = B/r$. The transactions of block j have all arrived at $j\tau$. The first block therefore starts at τ and is delivered at $t_1 = \tau + T_c(B)$. For $j > 1$, the producer must wait both for time $j\tau$ and for delivery of block $j - 1$ at t_{j-1}. It then takes $T_c(B)$ time to deliver block j, giving the stated recurrence.

Induction gives

$$t_j = \tau + T_c(B) + (j - 1)\max\{\tau, T_c(B)\}.$$

If $T_c(B) > \tau$, the waiting time from the last transaction's arrival at $j\tau$ to delivery grows without bound. Otherwise block j is delivered at $j\tau + T_c(B)$, while its first transactions arrive immediately after $(j-1)\tau$. The worst transaction latency is therefore $\tau + T_c(B)$.

When $T_c(B) = T_0 + \kappa B$, the feasibility condition is $B \geq r(T_0 + \kappa B)$. The block-size calculation (3) gives $B^* = rT_0/(1 - r\kappa)$, and $B^*/r = T_c(B^*)$. The latency increases with B, and at B^* it equals $2T_c(B^*)$, as claimed. The proof uses the requirement that communication of successive blocks does not overlap; the priority pipeline requires its own scheduling argument. □

D.2 Consistent Broadcast

The single-sender specification and the first worked calculation are in Section 4.1.1. For the multi-sender version, simultaneous votes are interleaved, as we now define.

Interleaving. For $I = \{i_1 < \cdots < i_k\}$ and sequences $m_i = (m_{i,1}, \ldots, m_{i,x_i})$, write $\oplus_I m_i$ for $m_{i_1,1}, \ldots, m_{i_k,1}, m_{i_1,2}, \ldots$, omitting undefined terms. If m_i is intended for p_i, $\oplus^*_I m_i$ applies that order to addressed parcels. Thus simultaneous messages advance evenly, rather than one complete message preceding the next. The multi-sender CB/RB specifications use $\oplus$; coded dispersal uses $\oplus^*$. The addressed message information identifies the constituent parcels.

Fig. D.1. Consistent Broadcast by all processors

The following instructions are for p_i.
at time slot 0 **do**: Initialise; set send = true and, for all j, $\texttt{vote}_j$ = true, $d_j = 1$
at every time slot t **do**:
  Produce block ▷ As specified in Section 3.6
  Let $I = \{j : \texttt{vote}_j = \text{true}$ and p_i has received a first valid block b_j of depth d_j from $p_j\}$
  Set sendbuffer = sendbuffer$||\oplus_I$ (VOTE, $j, d_j, H(b_j)$); for all $j \in I$, set $\texttt{vote}_j$ = false
  For each j for which p_i has received $n - f$ matching votes for b'_j of depth d_j and has also received b'_j **do**:
    Deliver b'_j; set $\texttt{vote}_j$ = true, $d_j = d_j + 1$; if $j = i$, set send = true
  Transfer to upload ▷ As specified in Figure 4

D.3 Reliable Broadcast

Here we give the sending instructions and block-size calculations for the single- and multi-sender variants analysed in Section 4.2.

Fig. D.2. Reliable Broadcast by a designated sender

There is a designated processor p_ℓ. The following instructions are for p_i.
at time slot 0 **do**: Initialise; set $d = 1$ and send = echo = vote = true
at every time slot t **do**:
  Produce block ▷ As specified in Section 3.6
  If echo = true and p_i has received a first valid block b of depth d from p_ℓ **do**:
    Set sendbuffer = sendbuffer$||$(ECHO, d, b) and echo = false
  If vote = true and p_i has received $n - f$ matching ECHOs for b' **do**:
    Set sendbuffer = sendbuffer$||$(VOTE, $d, H(b')$) and vote = false
  If vote = true and p_i has received $f + 1$ matching votes for some hash x **do**:
    Set sendbuffer = sendbuffer$||$(VOTE, d, x) and vote = false
  If p_i has received $n - f$ matching votes for b' and has also received b' **do**:
    Deliver b'; set send = echo = vote = true and $d = d + 1$

Transfer to upload ▷ As specified in Figure 4

Fig. D.3. Reliable Broadcast by all processors

The following instructions are for p_i.
at time slot 0 **do**: Initialise; set send = true and, for all j, $\mathsf{echo}_j = \mathsf{vote}_j = \text{true}$, $d_j = 1$
at every time slot t **do**:
  Produce block ▷ As specified in Section 3.6
  Let $I = \{j : \mathsf{echo}_j = \text{true}$ and p_i has received a first valid block b_j of depth d_j from $p_j\}$
  Append $\oplus_I(\text{ECHO}, j, d_j, b_j)$ to sendbuffer; for $j \in I$, set echo_j = false
  Let I contain those j with $\mathsf{vote}_j = \text{true}$ for which p_i has received $n - f$ matching ECHOs for b'_j
  Append $\oplus_I(\text{VOTE}, j, d_j, H(b'_j))$ to sendbuffer; for $j \in I$, set vote_j = false
  Let I contain those j with $\mathsf{vote}_j = \text{true}$ for which p_i has received $f + 1$ matching votes for some hash x_j
  Append $\oplus_I(\text{VOTE}, j, d_j, x_j)$ to sendbuffer; for $j \in I$, set vote_j = false
  For each j for which p_i has received $n - f$ matching votes for b'_j and has also received b'_j **do**:
    Deliver b'_j; set $\mathsf{echo}_j = \mathsf{vote}_j = \text{true}$, $d_j = d_j + 1$; if $j = i$, set send = true
  Transfer to upload ▷ As specified in Figure 4

D.4 Reliable Broadcast with erasure coding

The functions F, G_i, V and R used below are defined in Section 5.1; the following protocols use the packet-size and lower-order conventions of that section. The resulting protocol is a modification of Figure D.2 and is shown in Figure D.4. It uses the addressed interleaving $\oplus^*$ defined at the start of Appendix D.2. For a set X of ECHO messages, define $R_{\text{val}}(X, d, x, k) = \text{true}$ exactly when: (i) $|X| \geq k$; (ii) the messages in X come from distinct processors and each has the form $(\text{ECHO}, d, j, z', x, y')$ with $V(j, z', x, y') = \text{true}$; and (iii) $R(X) \neq \text{false}$. In the multi-sender protocol below, $R^*_{\text{val}}(X, j', d, x, k)$ is the same predicate with the producer identifier j' inserted in every ECHO message, so those messages have the form $(\text{ECHO}, j', d, j, z', x, y')$.

In the blockwise encoding and transmission arrangement considered here, the entire block is known before the leader starts sending any fragment. The protocol nevertheless remains deliberately non-pipelined: the leader waits for one block to be delivered before beginning the next, and nonleaders wait for a full fragment before echoing it. Section 5.3 removes these restrictions.

Fig. D.4. Reliable Broadcast using erasure coding with a designated sender

There is a designated processor p_ℓ. The following instructions are for p_i.
at time slot 0 **do**:
  Initialise
  Set $d = 1$, send = true, echo = true, vote = true
at time slot t **if** send = true **and** $p_i = p_\ell$:
  Party p_i collects *txns*, which is all data parcels received by the client processor but not yet added to block-txns at any previous time slot.
  block-txns = block-txns||*txns*
  If |block-txns| $\geq B$ **do**: ▷ B as specified in Section 5.1
    Set metadata = metadata for the current block
    Set b = block-txns||metadata
    For each $j \in [1, n]$, set $m_j = (d, j, z, x, y)$, where $(j, z, x, y) = G_j(b)$
    Add $\oplus^*_{j\in[1,n]} m_j$ to upload buffer
    block-txns = {}

Set send = false
at every time slot t **do**:
If echo = true **and** p_i has received a first message (d, i, z, x, y) from p_ℓ such that
$V(i, z, x, y)$ = true **do**:
Set sendbuffer = sendbuffer||(ECHO, d, i, z, x, y)
Set echo = false
If vote = true **do**:
If there exists x such that p_i has received a set of messages X with
$R_{\text{val}}(X, d, x, n - f)$ = true **do**:
Set sendbuffer = sendbuffer||(VOTE, d, x)
Set vote = false
If vote = true **do**:
If there exists x such that p_i has received (VOTE, d, x) from $f + 1$ distinct processors **do**:
Set sendbuffer = sendbuffer||(VOTE, d, x)
Set vote = false
If there exists x such that:
(i) p_i has received (VOTE, d, x) from $n - f$ distinct processors, and;
(ii) p_i has received a set of messages X with $R_{\text{val}}(X, d, x, f + 1)$ = true; then **do**:
Deliver $R(X)$
Set send = true, echo = true, vote = true, $d = d + 1$
Transfer to upload

Substitution for Claim 9. The first row of Table 3 gives two complete fragment stages followed by votes. In the execution considered, each stage begins with empty upload buffers, and the messages are interleaved evenly among recipients. The corresponding messages therefore finish together, and the aggregate incoming rate at each download buffer is at most S, by the count in Appendix D.1. With $\rho = 3$ and $M = c\lambda$, the communication time is

$$T_{\text{c}}(B) = \frac{6(B + c\lambda) + n\lambda(2\log n + 1)}{S} + 3\delta.$$

Lemma 2 gives $S > 6D$ and the block size

$$B^* = \frac{D(6c\lambda + n\lambda(2\log n + 1) + 3\delta S)}{S - 6D}. \tag{38}$$

At this optimum the next block fills exactly during the preceding block's communication interval, so the threshold rule implements the fixed-size schedule. The latency is $2T_{\text{c}}(B^*)$. Omitting $6c\lambda + n\lambda$ from the fixed part of this expression gives Claim 9; retaining it gives the unsimplified expression. No new clearing induction is needed. □

Fig. D.5. Reliable Broadcast by all processors using erasure coding

The following instructions are for p_i.
at time slot 0 **do**:
Initialise
Set send = true
$\forall j$, echo$_j$ = true, vote$_j$ = true, $d_j = 1$
at time slot t **if** send = true **do**:
Party p_i collects *txns*, which is all data parcels received by the client processor but not
yet added to block-txns at any previous time slot.
block-txns = block-txns||*txns*
If |block-txns| $\geq B$ **do**: ▷ B as specified in Section 5.2
Set metadata = metadata for the current block

Set b = block-txns||metadata
For each $j \in [1,n]$, set $m_j = (d_i, j, z, x, y)$, where $(j, z, x, y) = G_j(b)$
Add $\oplus^*_{j\in[1,n]} m_j$ to upload buffer
block-txns = {}
Set send = false

at every time slot t **do**:
Let $I = \{j :$ echo_j = true **and** p_i has received a first message of the form (d_j, i, z_j, x_j, y_j) (for some z_j, x_j, y_j) from p_j with $V(i, z_j, x_j, y_j)$ = true$\}$
Set sendbuffer = sendbuffer|| $\oplus_I$ (ECHO, j, d_j, i, z_j, x_j, y_j)
$\forall j \in I$ set echo_j = false
Let $I = \{j :$ vote_j = true **and** there exists x_j s.t. p_i has received a set of messages X_j with $R^*_{\text{val}}(X_j, j, d_j, x_j, n - f)$ = true$\}$
Set sendbuffer = sendbuffer|| $\oplus_I$ (VOTE, j, d_j, x_j)
$\forall j \in I$ set vote_j = false
Let $I = \{j :$ vote_j = true **and** there exists x_j s.t. p_i has received (VOTE, j, d_j, x_j) from $f + 1$ distinct processors$\}$
Set sendbuffer = sendbuffer|| $\oplus_I$ (VOTE, j, d_j, x_j)
$\forall j \in I$ set vote_j = false
$\forall j$ s.t. there exists x_j for which p_i has received (VOTE, j, d_j, x_j) from $n - f$ distinct processors **and** s.t. p_i has also received a set of messages X_j with $R^*_{\text{val}}(X_j, j, d_j, x_j, f + 1)$ = true
Deliver $R(X_j)$
Set echo_j = true, vote_j = true, $d_j = d_j + 1$
If $j = i$, set send = true
Transfer to upload

Substitution for Claim 10. Each producer has transaction arrival rate $r = D/n$. After the initial dispersal, each processor echoes one fragment for each of the n producers and then broadcasts one vote for each producer. Thus the second row of Table 3 applies under its balanced simultaneous-service assumptions. Setting $\rho = 3$ gives

$$T_{\text{c}}(B) = \frac{3(n+1)(B+M) + n(n+1)\lambda \log n + n^2\lambda}{S} + 3\delta.$$

Lemma 2 gives $S > 3(n+1)D/n$ and

$$B^* = \frac{D(3(n+1)M + n(n+1)\lambda \log n + n^2\lambda + 3\delta S)}{Sn - 3(n+1)D}. \tag{39}$$

At this value of B, the next block's transactions arrive during exactly one communication interval, so the threshold rule produces blocks of this size. Substituting into $2T_{\text{c}}(B^*)$, then replacing $n + 1$ by n and suppressing $n^2\lambda$ relative to the fingerprint term, gives Claim 10. The intermediate denominator above, not just $S - 3D$, matters near the finite-n threshold. □

The independent-metadata variant. Claim 11 concerns the variant of the protocol of Figure D.5 described at the end of Section 5.2: each producer sends the independent metadata of its block to all processors uncoded, before the fragments, and encodes only the transactions and the dependent metadata. The commitment used by the Reliable Broadcast instance must then bind the independent metadata as well as the Merkle root of the coded part, so that a faulty sender cannot associate different independent metadata with the same coded part. This adds only a constant to the message sizes, which we absorb into the existing terms of size λ. If the sender is faulty, a correct processor that reconstructs the coded part without having received the independent metadata must also be able to obtain it from a correct processor that has. No such retrieval occurs in the all-correct executions analysed here.

We now give the timing calculation. In the protocol of Figure D.5, the independent metadata is encoded with the rest of the block, and so contributes $\rho(n+1)M^{\text{in}}/S$ to the communication time $T_{\text{c}}(B)$ in Table 3. In the variant, this is replaced by the time nM^{in}/S needed to send the independent metadata uncoded to all processors. With $\rho = 3$, Lemma 2 therefore gives the same rate condition and

$$B^* = \frac{D\big(nM^{\text{in}} + 3(n+1)M^{\text{de}} + n(n+1)\lambda \log n + n^2\lambda + 3\delta S\big)}{Sn - 3(n+1)D}. \tag{40}$$

The latency is $2T_{\text{c}}(B^*)$. Substituting this block size, and making the same simplifications as above, proves Claim 11.

D.5 Reliable Broadcast with erasure coding and pipelining

The protocol of Section 5.3 modifies that of Figure D.4 in two ways, which together constitute the "pipelining". First, the leader does not wait for a block to be delivered before producing the next: it produces a new block whenever a further B transactions have arrived, so that several blocks are generally in progress at once. Second, nonleaders forward the parcels of their fragments as they arrive, so that dispersal and forwarding overlap rather than taking place one after the other. The protocol is shown in Figure D.6. We explain below what is new in the specification, and then prove Claim 12.

Fig. D.6. Reliable Broadcast using erasure coding and pipelining with a designated sender

There is a designated processor p_ℓ. The following instructions are for p_i.
at time slot 0 **do**:
  Initialise; set $d = 1$, $C = \{\}$, $Q_S = Q_F = Q_C = \{\}$
  For all d', set $\texttt{echo}(d') = \texttt{true}$, $\texttt{vote}(d') = \texttt{true}$
at time slot t **if** $p_i = p_\ell$:
  Party p_i collects *txns*, all data parcels received by the client processor but not yet added to block-txns
  block-txns = block-txns||*txns*
  If |block-txns| $\geq B$ **do**: ▷ B as specified in Section 5.3
    Set metadata = metadata for the current block and b = block-txns||metadata
    For each $j \in [1, n]$, set $m_j = (d, j, z, x, y)$, where $(j, z, x, y) = G_j(b)$
    Queue the addressed sequence $\oplus^*_{j\in[1,n]} m_j$ in Q_F
    Queue one addressed copy of $(\text{ECHO}, d, x)_i$ for each recipient in Q_S
    Set $\texttt{echo}(d) = \texttt{false}$, block-txns = $\{\}$, $d := d + 1$
at every time slot t **do**:
  Update certificates ▷ As specified below
  For each depth d', if $p_i \neq p_\ell$, $\texttt{echo}(d') = \texttt{true}$, and p_i has received the full designated-fragment message selected for depth d'
  (d', i, z, x, y) from p_ℓ with $V(i, z, x, y) = \texttt{true}$ **do**:
    Queue one addressed copy of $(\text{ECHO}, d', x)_i$ for each recipient in Q_S
    Set $\texttt{echo}(d') = \texttt{false}$
  For each depth d' being processed, if $\texttt{vote}(d') = \texttt{true}$ **do**:
    If $\exists x$ such that C contains an echo-certificate for x of depth d' and
    $R^\dagger_{\text{block}}(d', x) \neq \text{false}$ **do**:
      Queue one addressed copy of $(\text{VOTE}, d', x)_i$ for each recipient in Q_S
      Set $\texttt{vote}(d') = \texttt{false}$
  For each depth d' being processed, if $\texttt{vote}(d') = \texttt{true}$ **do**:
    If $\exists x$ such that p_i has received $f + 1$ vote-certificate shares for x of depth d' **do**:
      Queue one addressed copy of $(\text{VOTE}, d', x)_i$ for each recipient in Q_S
      Set $\texttt{vote}(d') = \texttt{false}$
  For each vote-certificate in C, with root x and depth d', if
  $R^\dagger_{\text{block}}(d', x) \neq \text{false}$ **do**:
    Deliver $R^\dagger_{\text{block}}(d', x)$ (if not already delivered)
  Queue certificates ▷ As specified below

If $p_i \neq p_\ell$: Queue forwarded fragments	▷ As specified below
Check the finite-wait rule and move parcels to the upload buffer	▷ As specified below

Changes to the specification. We say that the dth block produced by the leader has *depth* d. The leader's counter d specifies the next block it will produce, and all processors handle receiving, voting and delivery independently at every depth d'; the flags $\texttt{echo}(d')$ and $\texttt{vote}(d')$ are initially true and record whether a processor may still send the corresponding share. The remaining changes are as follows.

- *Forwarding and echo shares.* A nonleader selects the first message (d', i, z, x, y) that it receives from the leader at each depth, which contains its *designated fragment*, and forwards each parcel of that message once, as it arrives. An ECHO message no longer contains the fragment. Instead, once the selected message is complete and valid, the nonleader sends an *echo share* $(\text{ECHO}, d', x)_i$ to all processors: this is its signature share on (ECHO, d', x), where x is the root of the Merkle tree. We refer to x as the *commitment* to the block. The leader sends its echo share when it produces the block, and does not forward a fragment.
- *Vote shares and certificates.* Similarly, a *vote share* $(\text{VOTE}, d', x)_i$ is a signature share on (VOTE, d', x). Matching echo shares from $n - f$ distinct processors can be combined to form an *echo-certificate*, and matching vote shares to form a *vote-certificate*. Each share and each certificate has size λ, using constant-size threshold signatures [11, 64]. A processor votes once it has an echo-certificate and the block, and we call this an *ordinary vote*. As before, it also votes upon receiving $f + 1$ matching vote shares, and we call this the *relay rule*. It delivers once it has the block and a vote-certificate.
- *The certificate pool.* To ensure that the protocol remains as similar as possible to that later considered in Section 6.3, we suppose that every processor maintains a *certificate pool* C. Whenever a processor receives at least $n - f$ shares of some echo/vote-certificate, or receives a certificate, and C does not already contain the corresponding certificate, it adds the certificate to C ("Update certificates") and sends it to all processors ("Queue certificates").
- *Reconstruction.* $R^{\dagger}_{\text{val}}(X, d, x)$ is the fragment-validity predicate corresponding to R_{val} with threshold $f + 1$: the messages in X must come from distinct processors, have the form (d, j, z', x, y') with $V(j, z', x, y') = \text{true}$, and satisfy $R(X) \neq \text{false}$. The function $R^{\dagger}_{\text{block}}(d, x)$ returns the leader's locally generated block at depth d if that block has commitment x; otherwise it returns $R(X)$ for a locally available set X satisfying $R^{\dagger}_{\text{val}}(X, d, x) = \text{true}$, and returns false if neither is available.

Scheduling. Since several blocks are in progress at once, a processor will often have more than one message to send. Shares are small, while the fragment data for a block is large. So, a share that had to wait for fragment data could be delayed for a long time, while interrupting fragment data to send a share delays that data only slightly. Processors therefore send shares first, then fragment data, and finally certificates, which are not needed when all processors are correct. Formally, each processor maintains FIFO queues Q_S for shares, Q_F for fragment data and Q_C for certificates, and places addressed parcels in these queues once they are ready to be sent ("Queue forwarded fragments" does this for each newly received parcel that a nonleader is to forward). At each time slot, it moves up to S parcels to its upload buffer, taking parcels from Q_S first, then from Q_F, and then from Q_C. The copies of a message for different recipients are interleaved, as are the n fragments that the leader sends for a block, so a processor always sends to all recipients at the same rate.

In a faulty execution, shares could delay fragment data indefinitely. To prevent this, we add a *finite-wait rule*: if any parcel has been waiting in Q_S or Q_F for more than L, where L is the interval between successive blocks specified in (41), then the processor switches permanently to moving parcels from the three queues to the upload buffer in the order in which they were placed there. We show below that this never occurs when all processors are correct.

Correctness. The rules above affect only the order in which a processor sends its messages, and not the conditions under which it echoes, votes or delivers. Agreement therefore holds as for the protocol of Figure D.4: a correct processor sends at most one echo share and at most one vote share at each depth, and any two sets of $n - f$ processors have a correct processor in common, so two certificates of the same kind at the same depth cannot be for different commitments. Forwarding parcels before a fragment has been checked is also harmless, since each recipient checks every fragment against the commitment before using it. Validity and totality require, in addition, that every share and fragment parcel which a correct processor is to send is eventually uploaded. The finite-wait rule ensures this: once it applies, each waiting parcel has only finitely many parcels ahead of it. Certificates may be delayed indefinitely in a faulty execution, but they are not needed. If a correct processor delivers, then at least $f + 1$ correct processors have sent vote shares. So, every correct processor eventually votes, by the relay rule if not before, and can form a vote-certificate from the $n - f$ correct vote shares. Moreover, the first correct vote is an ordinary vote, so an echo-certificate exists. If the leader is faulty, at least $f + 1$ correct nonleaders therefore have valid designated fragments, which they forward, and every correct processor can reconstruct the block.

The block interval. We now prove Claim 12. We suppose that all processors are correct and, as elsewhere, we work in continuous service. Write

$$P = 3(B + M) + n\lambda \log n, \qquad T_F = P/S, \qquad a = n\lambda/S.$$

Here P is the total size of the n fragments that the leader sends for each block, so T_F is the time that the leader needs to upload them at full capacity. Each nonleader receives a designated fragment of size P/n and forwards it to n processors, so it also needs time T_F to upload the fragment data for each block. Similarly, a is the time needed to send one share or one certificate to all processors. For each block, each processor sends an echo share, a vote share and two certificates. So, the upload time that each processor requires for each block is $T_F + 4a$. We take the interval between successive blocks to be slightly longer than this:

$$L = T_F + 8a = \frac{3(B + M) + n\lambda \log n + 8n\lambda}{S}. \tag{41}$$

The leader produces a block each time a further B transactions have arrived, which takes time B/D. The block-size calculation (3), applied to $B/D \geq L$, gives $S > 3D$ and

$$B^* = \frac{D(3M + n\lambda(\log n + 8))}{S - 3D}. \tag{42}$$

We take $B = B^*$, so $B/D = L$. Starting with no transactions, the leader therefore forms block b_i at $t_i = iL$, for $i \geq 1$.

Download buffers. As specified above, each processor sends to all recipients at the same rate. Since it uploads at rate at most S, it sends to each recipient at rate at most S/n. Each processor therefore receives at a total rate of at most S, and download buffers add no waiting: every parcel is received exactly δ after it is uploaded. Note also that all nonleaders send and receive their corresponding messages at the same times: the contents of the messages differ, but their sizes and the times at which they are ready to be sent agree.

Bounds for each stage. For block b_i, let $t_{i,1}$ be the time at which the leader finishes uploading the fragments, let $t_{i,2} = t_{i,1} + \delta$ be the time at which each nonleader has received its complete designated fragment, and let $t_{i,3}$ be the time at which each nonleader finishes forwarding that fragment. We say that a share becomes *ready* when it is placed in Q_S. We establish the following bounds, for every block b_i:

(a) Each share is uploaded within $2a$ of becoming ready.
(b) $t_i + T_F + a \leq t_{i,1} \leq t_i + T_F + 2a$.
(c) $t_{i,3} \leq t_{i,2} + 2a$.
(d) At every processor, the vote share for b_i becomes ready at some time in the interval $[t_{i,2} + a + \delta,\ t_{i,2} + 2a + \delta]$.

Certificates are sent only when a processor has no shares or fragment data waiting, and so play no role in these bounds. We prove the bounds together, by induction on the order in which the events concerned occur. So, in proving a bound for one event, we may assume that all of the bounds hold for events occurring earlier.

We first note a consequence of the bounds, for those events to which they have already been shown to apply. A nonleader's echo share for b_i becomes ready at $t_{i,2}$, when it has received its complete designated fragment, while the leader's becomes ready at t_i. Since $t_{i+1} = t_i + L$, it follows from (b) that a processor's echo shares for successive blocks become ready at least $L - a$ apart. Similarly, it follows from (b) and (d) that a processor's vote shares for successive blocks become ready at least $L - 2a$ apart. Since $L - 2a = T_F + 6a$, we conclude:

($*$) In any time interval of length at most $T_F + 5a$, at most one echo share and at most one vote share become ready at any given processor.

Proof of (a). Suppose that a share becomes ready at time s. By the induction hypothesis, any share that is still waiting at time s became ready after $s - 2a$. By ($*$), these shares, together with the new one, include at most one echo share and at most one vote share. Since shares are sent before anything else, and in the order in which they become ready, the new share waits for at most one other share. It is therefore uploaded by time $s + 2a$.

It follows from (a) and ($*$) that a processor spends at most $2a$ uploading shares during any time interval of length at most $T_F + 3a$: the shares uploaded during such an interval became ready during an interval of length at most $T_F + 5a$, and so consist of at most one echo share and at most one vote share.

Proof of (b). At time t_i, the leader places its echo share for b_i in Q_S and the fragments of b_i in Q_F. Since shares are sent first, the echo share is uploaded before the fragments, which gives the lower bound. The fragments of b_{i-1} have already been uploaded by time t_i, since $t_{i-1,1} \leq t_{i-1} + T_F + 2a < t_i$. Since the leader spends at most $2a$ of the interval $[t_i, t_i + T_F + 2a]$ uploading shares, the remaining time suffices to upload the fragments of b_i.

Proof of (c). The leader does not begin uploading the fragments of b_i before $t_i + a$. So, by (b), a nonleader receives the parcels of its designated fragment during $[t_i + a + \delta, t_{i,2}]$, an interval of length at most $T_F + a$. By (b) and (c) for b_{i-1}, it has finished forwarding its fragment of b_{i-1} before this interval begins. The parcels arrive at rate at most S/n and each is forwarded to n processors, so fragment data arrives no faster than the nonleader can upload it, and is left waiting only as a result of time that the nonleader spends uploading shares. Since the nonleader spends at most $2a$ of the interval $[t_i + a + \delta, t_{i,2} + 2a]$ uploading shares, and no parcels arrive after $t_{i,2}$, it finishes forwarding by $t_{i,2} + 2a$.

Proof of (d). Each nonleader's echo share for b_i becomes ready at $t_{i,2}$. It takes time a to upload and so, by (a), it is received by all processors at some time in $[t_{i,2} + a + \delta, t_{i,2} + 2a + \delta]$. An echo-certificate requires shares from $n - f \geq 3$ processors. So, no processor can form one before it receives the echo share of a nonleader other than itself, and no echo-certificate exists before $t_{i,2} + a + \delta$. A processor cannot send an ordinary vote without an echo-certificate, and the relay rule cannot be applied before some processor has sent an ordinary vote. So, no vote share for b_i becomes ready before $t_{i,2} + a + \delta$.

On the other hand, by $t_{i,2} + 2a + \delta$ every processor has received the echo shares of all $n - 1 \geq n - f$ nonleaders. By (c), it has also received the fragments forwarded by all nonleaders, from which it can reconstruct the block. So, the vote share for b_i has become ready at every processor by this time.

Delivery and transaction latency. By (d) and (a), every vote share for b_i has been uploaded by $t_{i,2} + 4a + \delta$ and received by $t_{i,2} + 4a + 2\delta$. Every processor then has vote shares from at least $n - f$ processors, and has the block. Using (b), all processors therefore deliver b_i by

$$t_{i,2} + 4a + 2\delta \leq t_i + T_F + 6a + 3\delta = t_i + L - 2a + 3\delta.$$

The transactions in b_i arrive during $[t_i - L, t_i]$, giving

$$\ell_{\text{tx}} \leq 2L - 2a + 3\delta \leq 2L + 3\delta.$$

Substituting (42) into (41) gives

$$L = \frac{3M + n\lambda(\log n + 8)}{S}\left(1 + \frac{1}{(S/(3D)) - 1}\right),$$

and hence the expression in Claim 12. We do not claim that this choice of block interval, or this order of sending, minimises latency. The calculation uses continuous service, and is not an exact timing assertion for integral slots.

The finite-wait rule is not used when all processors are correct. It remains to show that, in the execution analysed above, no parcel waits in Q_S or Q_F for more than L. The finite-wait rule then plays no role, and Claim 12 holds for the protocol as specified. By (a), shares wait for at most $2a$. By (b), the leader uploads the fragments of each block within $T_F + 2a < L$ of placing them in Q_F, and the proof of (c) shows that a nonleader forwards each fragment parcel within $2a$ of receiving it.

Certificates are uploaded when a processor has no shares or fragment data waiting. By (b) and (c), a processor has fragment data waiting for at most $T_F + 3a$ in each period of length L. By (a) and $(*)$, it uploads at most two shares during the remaining $5a$, which leaves at least $3a$ in each such period for the two certificates that it sends for each block.

D.6 Protocol specifications for SMR

The following are the Tendermint-style and HotStuff schedules referred to in Section 6.

Here v is the view number, $\texttt{lead}(v)$ is its designated leader, and $\texttt{starttime}(v)$ is the time at which the processor entered that view. Block metadata includes the view number, so votes on $H(b)$ identify both the block and its view. A processor can form a k-QC after receiving matching stage-k votes from $n - f$ distinct processors; receiving such a certificate also includes forming it from the votes received. A complaint-QC consists of complaints for the same view from $n - f$ distinct processors. The initial lock is a fixed certificate for the genesis block in view 0. The parameter TIMEOUT is chosen large enough that neither the initial view nor any subsequent view times out in the successful executions analysed in Section 6. A valid proposal is one that satisfies the block-validation and locking rules of the protocol under consideration; the figures specify the message-sending schedules used in the latency calculations.

The instruction **Collect locally** appends to block-txns all transactions received by the client processor that have not previously been collected. It does not append them to sendbuffer: the leader queues the whole block only when the proposal condition is met. Transactions received while send = false remain at the client processor and are collected after delivery re-enables send. The instructions **Initialise** and **Transfer to upload** retain their meanings from Figure 4.

Fig. D.7. The Tendermint-style schedule analysed here, with fixed per-cycle wait w.

The following instructions are for p_i.
at time slot 0 **do**:
  Initialise; set $v = 1$, $\texttt{starttime}(1) = 0$, send = true, vote1 = true, vote2 = true
  Set complained = false and initialise the lock to the genesis certificate
  Execute the "upon entering view" instruction for view 1
upon entering view v **do**:
  Form a λ-sized next-view message c_v carrying the current lock/highest-QC information
  Add c_v to the upload buffer, with all parcels addressed to $\texttt{lead}(v)$
at every time slot t **do**:
  If send = true and $p_i = \texttt{lead}(v)$ **do**: Collect locally
  If send = true, $p_i = \texttt{lead}(v)$, $t - \texttt{starttime}(v) \geq C(\lambda) + w$, and |block-txns| $\geq B$ **do**:
    Set metadata = metadata for the current block
    Set sendbuffer = sendbuffer||block-txns||metadata, block-txns = $\{\}$, send = false
  If vote1 = true and p_i has received a valid block proposal b for view v **do**:
    Set sendbuffer = sendbuffer||$(1\text{-VOTE}, H(b))_i$, vote1 = false
  If vote2 = true and p_i has received a 1-QC for a view-v block b **do**:
    Set sendbuffer = sendbuffer||$(2\text{-VOTE}, H(b))_i$, vote2 = false
    Set the local lock to the 1-QC
  If p_i has received a 2-QC for a view-v block b **do**:
    Deliver b and all ancestor blocks
    Set $v := v + 1$, $\texttt{starttime}(v) = t$, send = vote1 = vote2 = true
    Set complained = false and execute the "upon entering view" instruction
  If complained = false and $t - \texttt{starttime}(v) \geq$ TIMEOUT **do**:
    Set sendbuffer = sendbuffer||$(\text{COMPLAIN}, v)_i$, complained = true
  If p_i has received a complaint-QC for view v **do**:
    Set $v := v + 1$, $\texttt{starttime}(v) = t$, send = vote1 = vote2 = true
    Set complained = false and execute the "upon entering view" instruction
  Transfer to upload ▷ As specified in Figure 4

Fig. D.8. HotStuff

The following instructions are for p_i.
at time slot 0 do:
  Initialise
  Set $v = 1$, $\texttt{starttime}(1) = 0$, send = true, vote1 = true, vote2 = true, vote3 = true
  Set send1QC = true, send2QC = true, send3QC = true, complained = false
  Initialise the lock to the genesis certificate
at every time slot t **do**:
  Add a NEWVIEW message for view v carrying the current lock/highest-QC information
  to the upload buffer, with all parcels addressed to $\texttt{lead}(v)$,
  if not already done
  If $p_i = \texttt{lead}(v)$ **and** send = true **do**: Collect locally
  If $p_i = \texttt{lead}(v)$ and send = true, and p_i has received
    $n - f$ NEWVIEW messages for view v, and |block-txns| $\geq B$ **do**: ▷ B as in Equation (20)
      Set metadata = metadata for the current block
      Set sendbuffer = sendbuffer||block-txns||metadata
      Set block-txns = $\{\}$, send = false
  If vote1 = true **and** p_i has received a valid block proposal b for view v from $\texttt{lead}(v)$ **do**:
    Set vote1 = false
    Add $(1\text{-VOTE}, H(b))_i$ to upload buffer, with all parcels addressed to $\texttt{lead}(v)$
    ▷ Stage 1 vote on hash of b signed by p_i
  If $p_i = \texttt{lead}(v)$ **and** send1QC = true **and** p_i has received $n - f$ stage 1 votes for b corresponding to view v **do**:
    Set send1QC = false
    Form Q_1, a threshold 1-QC for b

Set sendbuffer = sendbuffer$||Q_1$
If vote2 = true **and** p_i has received a 1-QC for a view v block b from lead(v) **do**:
Set vote2 = false
Add (2-VOTE, $H(b)$)$_i$ to upload buffer, with all parcels addressed to lead(v)
▷ Stage 2 vote on hash of b signed by p_i
If p_i = lead(v) **and** send2QC = true **and** p_i has received $n - f$ stage 2 votes for b corresponding to view v **do**:
Set send2QC = false
Form Q_2, a threshold 2-QC for b
Set sendbuffer = sendbuffer$||Q_2$
If vote3 = true **and** p_i has received Q_2, a 2-QC for a view v block b, from lead(v) **do**:
Set vote3 = false
Set lock = Q_2
Add (3-VOTE, $H(b)$)$_i$ to upload buffer, with all parcels addressed to lead(v)
▷ Stage 3 vote on hash of b signed by p_i
If p_i = lead(v) **and** send3QC = true **and** p_i has received $n - f$ stage 3 votes for b corresponding to view v **do**:
Set send3QC = false
Form Q_3, a threshold 3-QC for b
Set sendbuffer = sendbuffer$||Q_3$
If p_i has received a 3-QC for a view v block b **do**:
Deliver b and all ancestor blocks
Set send = vote1 = vote2 = vote3 = true, starttime($v+1$) = t, $v = v+1$
Set send1QC = send2QC = send3QC = true, complained = false
If complained = false and t − starttime(v) ≥ TIMEOUT **do**:
Set sendbuffer = sendbuffer$||$(COMPLAIN, v)$_i$, complained = true ▷ Signed complaint for view v
If p_i has received a complaint-QC for view v **do**:
Form threshold certificate c for view $v+1$, set sendbuffer = sendbuffer$||c$
Set send = vote1 = vote2 = vote3 = true, starttime($v+1$) = t, $v = v+1$
Set send1QC = send2QC = send3QC = true, complained = false
Transfer to upload ▷ As specified in Figure 4

E LATENCY LOWER BOUNDS FOR PROTOCOLS USING AVID-STYLE ERASURE CODING

This appendix proves both assertions of Proposition 1. We first bound the number of previously unreconstructed transactions that one block can provide at a fixed nonleader, and count the actual downloads needed to provide all incoming transactions. We then strengthen the timing inequality when all but at most one of the required fragments reach that processor by forwarding through another nonleader. Both arguments allow block contents to overlap, without counting unfinished transmissions or requiring every processor to deliver a transaction through the same block.

E.1 The reconstruction bound

Proof. Suppose transaction latency is bounded by $L < \infty$; otherwise the latency inequality is immediate. Fix a nonleader p. For each block, consider the first time at which p has both received the required k_{rec} complete authenticated fragments and reconstructed the block. Assign each transaction to a block for which this time is earliest among blocks containing that transaction, breaking simultaneous ties by a fixed ordering of blocks. Every transaction has such a block because both conditions are required for delivery. (This assignment is only for counting: no transactions are removed from blocks and no change to the protocol is made.)

Index the blocks with at least one assigned transaction as $b_1, b_2, \ldots$. For each i, let B_i be the total number of transaction parcels in b_i, and let $B'_i > 0$ be the number assigned to it, so that $B'_i \le B_i$. Let $t_{i,0}$ be the infimum of the arrival times of its assigned transactions, $t_{i,1}$ the time its contents are fixed, and $t_{i,2}$ the first time p has both the required fragments and a successful reconstruction of b_i. (We use the infimum because, in continuous service, there need not be a first such transaction.) Processor p does not deliver any transaction assigned to b_i before $t_{i,2}$, since both conditions are

required for delivery. Since latency is at most L, each of these transactions therefore arrives no earlier than $t_{i,2} - L$, and so $t_{i,2} \leq t_{i,0} + L$.

The number of new transactions supplied by a block. All B'_i assigned transactions arrive between $t_{i,0}$ and $t_{i,1}$. Constant input rate D therefore gives

$$t_{i,1} - t_{i,0} \geq B'_i/D.$$

By $t_{i,2}$, processor p has received at least k_{rec} distinct complete authenticated fragments of that block. Recall from (8) that the n fragments of a block with B transaction parcels have total size $\rho B + A$, where ρ and A are positive constants, so that each fragment of b_i has size $(\rho B_i + A)/n$. The fragments received by p therefore contain at least $k_{\mathrm{rec}}(\rho B_i + A)/n$ parcels. None can arrive before $t_{i,1} + \delta$, because transmission starts only after the block is fixed. Processor p has total download capacity S, so

$$t_{i,2} - t_{i,1} \geq \frac{k_{\mathrm{rec}}(\rho B_i + A)}{nS} + \delta.$$

Forwarding, other downloads and additional delivery conditions can only postpone reconstruction or delivery. Combining these inequalities, and using $B_i \geq B'_i$, yields

$$L \geq \left(\frac{1}{D} + \frac{k_{\mathrm{rec}}\rho}{nS}\right) B'_i + \frac{k_{\mathrm{rec}}A}{nS} + \delta. \tag{43}$$

Consequently $B'_{\max} := \sup_i B'_i$ is finite and positive. (This is a bound on the assigned transactions, not an assumption that the protocol's blocks are disjoint.)

The required downloads. For $t \geq L$, let $\mathcal{I}_t = \{i : t_{i,2} \leq t\}$ be the indices of counted blocks which p has reconstructed by time t, having received the required fragments. Their assigned transactions include all arrivals by $t - L$, and the assignments are disjoint. Hence

$$\sum_{i \in \mathcal{I}_t} B'_i \geq D(t - L), \qquad |\mathcal{I}_t| \geq \frac{D(t-L)}{B'_{\max}}.$$

For each block in $\mathcal{I}_t$, the required fragments have actually passed through p's download buffer. Recall our assumption that each fragment's data and authentication information are received in full for each block separately: parcels received for one block do not count towards the fragments of another. The parcels received for different blocks in $\mathcal{I}_t$ are therefore distinct, and their numbers can be added. Since at most St parcels have been received by time t,

$$\begin{aligned} St &\geq \frac{k_{\mathrm{rec}}}{n} \sum_{i \in \mathcal{I}_t} (\rho B_i + A) \\ &\geq \frac{k_{\mathrm{rec}}}{n} D(t-L) \left(\rho + \frac{A}{B'_{\max}}\right). \end{aligned}$$

In particular, only finitely many counted blocks can have been reconstructed in finite time, since each requires at least $k_{\mathrm{rec}}A/n > 0$ received parcels. Dividing by t and letting $t \to \infty$ gives

$$S \geq \frac{k_{\mathrm{rec}}\rho D}{n} + \frac{k_{\mathrm{rec}}AD}{nB'_{\max}}. \tag{44}$$

All quantities in the second term are positive. So $nS > k_{\mathrm{rec}}\rho D$ and

$$B'_{\max} \geq \frac{k_{\mathrm{rec}}AD}{nS - k_{\mathrm{rec}}\rho D}.$$

Taking a sequence of assigned sizes tending to $B'_{\max}$ in (43), and substituting this lower bound, we obtain

$$\begin{aligned} L &\geq \left(\frac{1}{D} + \frac{k_{\mathrm{rec}}\rho}{nS}\right)\frac{k_{\mathrm{rec}}AD}{nS - k_{\mathrm{rec}}\rho D} + \frac{k_{\mathrm{rec}}A}{nS} + \delta \\ &= \frac{2k_{\mathrm{rec}}A}{nS - k_{\mathrm{rec}}\rho D} + \delta. \end{aligned}$$

The argument applies to every finite latency bound L, proving the first assertion of the proposition. It counts only completed downloads needed for reconstruction; other fragments and control messages need not have been transmitted. □

E.2 The additional constraint from fragment forwarding

Proof of the forwarding assertion in Proposition 1. Consider the fixed nonleader p, the assignment of transactions and the notation from the preceding proof. Suppose $k_{\mathrm{rec}} \geq 2$ and the additional fragment-forwarding assumption holds: for each block, p receives at most one fragment directly from the leader, and every parcel of any other fragment that it uses for reconstruction reaches it by forwarding through another nonleader. We strengthen the timing inequality (43); the count of downloads leading to (44) remains unchanged.

For a counted block b_i, processor p receives at most one distinct fragment directly from the leader. At least $k_{\mathrm{rec}} - 1$ of its complete reconstruction fragments are therefore forwarded through another nonleader. Every parcel counted in these fragments traverses at least two communication links after $t_{i,1}$, when the block is fixed. No such parcel can arrive at p before $t_{i,1} + 2\delta$, even if each intermediary forwards it immediately upon receipt. These fragments contain at least $(k_{\mathrm{rec}} - 1)(\rho B_i + A)/n$ parcels. The download capacity of p gives

$$t_{i,2} - t_{i,1} \geq \frac{(k_{\mathrm{rec}} - 1)(\rho B_i + A)}{nS} + 2\delta.$$

The assigned transactions require $t_{i,1} - t_{i,0} \geq B'_i/D$, as before. Since $B_i \geq B'_i$ and $t_{i,2} - t_{i,0} \leq L$, it follows that

$$L \geq \left(\frac{1}{D} + \frac{(k_{\mathrm{rec}} - 1)\rho}{nS}\right)B'_i + \frac{(k_{\mathrm{rec}} - 1)A}{nS} + 2\delta. \tag{45}$$

The inequality (44) still counts all k_{rec} received fragments, including the designated fragment. So

$$B'_{\max} \geq \frac{k_{\mathrm{rec}}AD}{nS - k_{\mathrm{rec}}\rho D}.$$

Taking assigned sizes tending to $B'_{\max}$ in (45) and substituting gives

$$\begin{aligned} L &\geq 2\delta + \frac{(k_{\mathrm{rec}} - 1)A}{nS} + \left(\frac{1}{D} + \frac{(k_{\mathrm{rec}} - 1)\rho}{nS}\right)\frac{k_{\mathrm{rec}}AD}{nS - k_{\mathrm{rec}}\rho D} \\ &= \frac{(2k_{\mathrm{rec}} - 1)A}{nS - k_{\mathrm{rec}}\rho D} + 2\delta. \end{aligned}$$

Together with (9), this proves (10). □

E.3 Integral slots

For discrete timeslots, suppose each authenticated fragment of a block with B transaction parcels contains at least $(\rho B + A)/n$ parcels, with padding retained in its actual length. Input and receipt occur at the beginning of slots, and an upload in slot t reaches its destination's download buffer in slot $t + \delta$. For the times in the proof above, inclusive

counting gives

$$t_{i,1} - t_{i,0} \geq \lceil B_i'/D \rceil - 1, \qquad t_{i,2} - t_{i,1} \geq \left\lceil \frac{k_{\mathrm{rec}}(\rho B_i + A)}{nS} \right\rceil + \delta - 1.$$

So (43) holds with $L + 2$ in place of L. In the count of downloads leading to (44), the download capacity through slot t is $S(t + 1)$ and the input count changes by bounded endpoint terms; these disappear after division by t. The same proof therefore gives

$$nS > k_{\mathrm{rec}}\rho D, \qquad \ell_{\mathrm{tx}} \geq \frac{2k_{\mathrm{rec}}A}{nS - k_{\mathrm{rec}}\rho D} + \delta - 2. \tag{46}$$

For example, authenticated fragment length $\lceil (B + M)/k_{\mathrm{rec}} \rceil + m_F$ satisfies the required size inequality with $\rho = n/k_{\mathrm{rec}}$ and $A = \rho M + nm_F > 0$.

Under the additional fragment-forwarding assumption, the earliest forwarded parcel for b_i can be received at p in slot $t_{i,1} + 2\delta$. By slot $t_{i,2}$, at least $(k_{\mathrm{rec}} - 1)(\rho B_i + A)/n$ such parcels have been received, so

$$t_{i,2} - t_{i,1} \geq \left\lceil \frac{(k_{\mathrm{rec}} - 1)(\rho B_i + A)}{nS} \right\rceil + 2\delta - 1.$$

Combining this with the same inclusive formation count makes (45) hold with $L+2$ in place of L. The count of downloads still uses all k_{rec} fragments. Consequently, for $k_{\mathrm{rec}} \geq 2$,

$$\ell_{\mathrm{tx}} \geq \max\left\{ \frac{2k_{\mathrm{rec}}A}{nS - k_{\mathrm{rec}}\rho D} + \delta - 2,\ \frac{(2k_{\mathrm{rec}} - 1)A}{nS - k_{\mathrm{rec}}\rho D} + 2\delta - 2 \right\}. \tag{47}$$

In addition, the required forwarded fragments cannot arrive before two propagation delays after block formation, so $\ell_{\mathrm{tx}} \geq 2\delta$ also holds in integral slots.