The Pipes Model for Latency and Throughput Analysis

ANDREW LEWIS-PYE, London School of Economics, UK
KARTIK NAYAK, Duke University, USA
NIBESH SHRESTHA, Supra Research, USA

Protocols for State-Machine-Replication (sometimes called 'blockchain' protocols) generally make use of rotating *leaders* to drive consensus. In typical protocols [17] (henceforth called 'single-sender' protocols), the leader is a single processor responsible for making and disseminating proposals to others. Since the leader acts as a bottleneck, apparently limiting throughput, a recent line of research has investigated the use of 'multi-sender' protocols in which many processors distribute proposals in parallel. Examples include DAG-based protocols such as DAG-Rider [37], Bullshark [63], Sailfish [59], Cordial Miners [38], Mysticeti [7], and variants such as Autobahn [36]. However, existing models do not allow for a formal analysis to determine whether these protocols can actually handle higher throughputs than single-sender protocols such as PBFT [17], Tendermint [12], and HotStuff [67].

In this paper, we describe a very simple model that allows for such an analysis. For any given protocol, the model allows one to calculate latency as a function of network bandwidth, network delays, the number of processors n, and the incoming transaction rate. Each protocol has a *latency bottleneck*: 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 aim of building to an analysis for state-of-the-art State-Machine-Replication (SMR) protocols, we begin by considering protocols for simpler primitives, such as Best-effort Broadcast and Reliable Broadcast. For Best-effort Broadcast, we establish a tight lower bound on latency for single-sender and multi-sender protocols when blocks are distributed without the use of techniques such as erasure coding. Perhaps unsurprisingly, a key difference between the single-sender and multi-sender approaches in this case is a factor n in the point at which the latency bottleneck appears. However, for other primitives such as Reliable Broadcast, our results may be more surprising: the factor n difference now disappears, and maximum throughput for the two approaches differs by a constant factor, while multi-sender approaches will generally have latency that grows more quickly with n. For state-of-the-art SMR protocols, the picture that emerges is one with seemingly inherent trade-offs. If one compares single-sender protocols that use pipelining and erasure coding, such as DispersedSimplex [56], with DAG-based protocols such as Sailfish or Bullshark, the former are seen to have lower latency for a wide range of throughputs, while the benefit of the latter protocols is that they have a latency bottleneck which is higher by a constant factor.

[Andy: Once experiments are added, obviously we'll add a bit about them here.]

1 INTRODUCTION

Background. The common timing assumptions considered in Distributed Computing are the synchronous, asynchronous, and partially synchronous settings. In the synchronous setting, message delivery is reliable, in the sense that there is a known upper bound Δ on message delays. In the most standard form of the asynchronous setting [29], messages are always delivered, but there is no upper bound on message delays. In the partially synchronous setting [24], there is an unknown global-stablization-time (GST) after which messages are always delivered within the known bound Δ. When considering protocols for State-Machine-Replication (SMR) [54] (i.e., 'blockchain' protocols), the latter setting formalizes the idea that a protocol must be *consistent* in asynchrony and *live* given sufficiently long periods of synchrony. These settings have been extensively studied over almost half a century. When combined with a variety of adversarial models (e.g., Byzantine/omission/crash faults) and various possible setup assumptions (e.g., PKI or otherwise), one sees a rich tapestry of results in which even small changes in the model considered can lead to very different feasibility outcomes.

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 models give no meaningful way to analyze some other important properties of a blockchain protocol, such as the maximum throughput it can handle.

These models also give limited ability to analyze (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 *finalized* 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 *communcation complexity*, e.g., the number of bits of data that must be sent to achieve finalization 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 communcation 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 [17] (henceforth called 'single-sender' protocols), have a single designated (potentially rotating) 'leader' who is responsible for making and disseminating proposals 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 [37], Bullshark [63], Shoal [6], Sailfish [59], Cordial Miners [38], Mysticeti [7], and variants such as Autobahn [36]. Some of these protocols serve as the consensus mechanism for major blockchains with market caps in the billions of dollars.[1] However, the models presently used in distributed computing have no way of formally analyzing the claim that 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 such as PBFT, Tendermint [12], and HotStuff [67].

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 *latency bottleneck*: this is an

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

[2]We note that some existing papers (e.g., [8, 39, 49]) do consider models that place high level limits on the 'capacity' of a network to deliver messages. However, these models are highly tailored to the analysis of longest-chain protocols and do

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 latency 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:

> *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, [14]) 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 optimized 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: it establishes a lower bound on the latency that a protocol can achieve 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. [Andy: Later, if experiments back it up, it would be nice to add a sentence here saying something along the lines that our experiments do suggest the model is useful.]

not allow for a granular comparison of single-sender and multi-sender protocols: 'capacity' is modeled as a network-wide parameter and those models do not allow for an analysis of the extent to which specific communication channels between pairs of processors may act as a bottleneck. See Section 8 for a more detailed comparison.

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. This is particularly important in the context of distributed computing, where:

(i) Running experiments over a large number of processors is expensive;
(ii) Comparing and interpreting the results of experiments is often difficult due to differences in implementation details (a model is useful in the specific sense that it factors out such differences).

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: we consider Best-effort Broadcast, Consistent Broadcast, Reliable Broadcast, and Reliable Broadcast with erasure coding. 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.

Best-effort Broadcast. For Best-effort Broadcast, where processors deliver a transaction upon receiving a full block containing that transaction, we establish a tight lower bound on latency for single-sender and multi-sender protocols when blocks are distributed without the use of techniques such as erasure coding. To establish this bound, it is necessary to distinguish between two types of metadata, since they have different impacts on latency:

- *Dependent metadata*, such as the leader'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, the leader may be able to determine an appropriate hash pointer to a previous block before the transactions are decided.

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 $\alpha \in (0,1)$, we establish a formal version of the statement that any single-sender Best-effort Broadcast protocol that "does not use techniques like erasure coding" must have latency at least:

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

The same result holds for multi-sender protocols, but with $\alpha = Dn/S$ replaced by $\alpha = D/S$. We also describe single-sender and multi-sender protocols showing that these bounds are tight. The latency bottleneck therefore appears at $D = S/n$ in the single-sender setting, and appears at $D = S$ in the multi-sender setting.

Consistent Broadcast, Reliable Broadcast and Reliable Broadcast with erasure coding. For other primitives, our results are more nuanced. In each case, we consider standard protocols and establish an expression for latency in terms of S, D, n, Δ, and other parameters (such as the size of metadata), when block sizes are set to minimize latency. We refer the reader to Sections 4–6 for a precise description of the results. A key take-away is that, for Reliable Broadcast or Reliable Broadcast with erasure coding, the factor n difference in the latency bottleneck now disappears, and maximum throughput for the two approaches differs by a constant factor, while multi-sender approaches will generally have latency that grows more quickly with n. The constant factor difference in the latency bottleneck depends on whether or not protocols use *pipelining*,[3] and the precise form of erasure coding used. For example, if $S/Dn > 2$, 'votes' are of size λ, and the block size (number of

[3]An example of pipelining is given in Section 5.3.

transactions) is set to minimize latency, then latency for a standard form of Reliable Broadcast in the single-sender setting (specified in Figure 10, and which does not use erasure coding or pipelining) is:[4]

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

Note that, if $S/Dn > 2$, the second term in parentheses above is greater than 1 and tends to infinity as $2Dn$ approaches S. If $2Dn$ is small compared to S, then the latency is roughly equal to the first term in parentheses. So, one can think of the first term as giving the approximate latency when $2Dn$ is small compared to S, while latency tends to infinity as $2Dn$ approaches S. The corresponding latency for the multi-sender case if block metadata is of size $n\lambda$ is:

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

However, the factor 2 difference in the latency bottleneck disappears if one uses pipelining in the single-sender setting (the use of pipelining in the multi-sender setting does not have the same impact since the all-to-all 'echo' round is the most expensive in terms of resulting latency).

SMR protocols. With respect to DAG-based protocols, we focus on the case of 'certified' protocols such as DAG-Rider, Bullshark, Shoal, and Sailfish, which use some form of Consistent/Reliable Broadcast as the underlying method of block propagation: the term 'certified' [51] 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 simply to limit the length of the paper: we leave it to future work to analyse protocols, such as Cordial Miners and Mysticeti, in which block producers just send blocks directly to all others (Best-effort Broadcast).[5] In fact, analyzing latency for certified DAG-based protocols is uncomplicated once we have analysed the simpler primitives discussed above. This is because the calculation 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) that are required to finalize each block.

To make things concrete, we focus on the example of Sailfish [59]: the analysis we carry out is also easily adapted to deal with other well-known DAG-based protocols. As far as we are aware, Sailfish has latency that is at least on par with other existing DAG-based protocols in the 'good case', with the following caveat:[6] while the Sailfish paper assumes Reliable Broadcast is used as the underlying primitive for block propagation, doing so produces a latency bottleneck which is $O(S/n)$, thereby defeating the apparent aim of using a multi-sender protocol. The latency bottleneck is $O(S)$ if Consistent Broadcast is used for block propagation, and we therefore focus on this version of the protocol. In this case, if block metadata is of size $n\lambda$, latency is:

[4]The figure of 6Δ in this expression may initially surprise the reader: the figure is higher than one might expect due to the way in which we calculate latency from the first time at which a correct processor receives a transaction, rather than the first time at which a correct processor propagates a block containing the transaction.

[5]For some protocols using Best-effort Broadcast, the latency analysis will be complicated (even when processors are correct) by an extra requirement to pass on blocks proposed by *other* processors when necessary. For the sake of simplicity, we avoid such considerations here.

[6]Of course, specific implementations can also introduce their own inefficiencies. For example, the code given in the original Sailfish paper implements a variant of the protocol built on Narwhal [22], which introduces some inefficiencies. A very recent paper [51], which appeared as we were completing the present draft, also introduces a protocol called Starfish, which may have latency lower than Sailfish in some regimes. We leave a detailed analysis of Starfish to future work, but note that on a *qualitative* level the analysis will be similar to that for Sailfish when Reliable Broadcast with erasure coding is used for block propagation.

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

The 9Δ figure appearing in the left term in parentheses may be higher than some readers expect. This is because we calculate latency in a way which is more stringent than in some other papers. See Section 7.2 for details.

For reasons that we expand on in Section 7.1 it is difficult to make an apples-to-apples comparison between protocols for the single-sender and multi-sender settings. For example, the expression giving latency for Sailfish above assumes the use of Consistent Broadcast. Among protocols we are aware of in the single-sender setting, that with the most competitive latency is DispersedSimplex [56]. The latter protocol uses a form of Reliable Broadcast with erasure coding, and Reliable Broadcast gives stronger guarantees than Consistent Broadcast. On the other hand, latency for DispersedSimplex is minimized in the context of a stable leader, enabling a form of pipelining. It is not obvious that the case in which there is a stable leader has a precise (or similarly useful) analogue in the case of DAG-based protocols.

Since it is difficult to make an entirely apples-to-apples comparison, we take the approach of comparing the lowest latency protocols for each of 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. Latency for DispersedSimplex in the case of a stable leader is:

$$\left(\frac{6M + 2n\lambda(\log(n) + 4)}{S}\right) \cdot \left(1 + \frac{1}{(S/3D) - 1}\right) + 2n\lambda/S + 3\Delta.$$

To make this concrete, suppose processors can upload/download at a rate of 1 Gbps. Suppose transactions are 2500 bits (about 300 bytes, similar to typical Bitcoin transactions). In the single-sender setting, suppose M is 1000 bits. Set $\lambda = 500$ bits and $\Delta = 0.2$ seconds. The first graph in Figure 1 supposes the incoming transaction rate D is 10^5 transactions per second and shows the resulting latency (in seconds) as a function of n. The second graph (on the right) in Figure 1 fixes $n = 400$ and shows the resulting latency as a function of the number of incoming transactions per second divided by 10^4. One can see that DispersedSimplex has significantly lower latency until one reaches its latency bottleneck, but that the latency bottleneck for DispersedSimplex is a third of that for Sailfish (when Sailfish uses Consistent Broadcast as the underlying primitive for block propagation).

Unlike DispersedSimplex, most single-sender (or multi-sender) protocols do not use erasure coding. It is therefore also interesting to compare latencies for standard protocols like Tendermint and Sailfish. To this end, we consider a standard version of Tendermint (see Figure 25) in Section 7.4. Latency is calculated to be:

$$\frac{(6\lambda n + 2Mn)/S + 10\Delta}{(S/Dn) - 1} + \frac{(5\lambda + 2M)n}{S} + 8\Delta.$$

To compare with latency for Sailfish, suppose again that processors can upload/download at a rate of 1 Gbps and that transactions are 2500 bits. In the single-sender setting, suppose M is 1000 bits. Set $\lambda = 500$ bits and $\Delta = 0.2$ seconds. The first graph in Figure 2 supposes the incoming transaction rate D is 2000 transactions per second and shows the resulting latency (in seconds) as a function of n. The second graph (on the right) in Figure 2 fixes $n = 30$ and shows the resulting latency as a function of the number of incoming transactions per second divided by 10^3.

HotStuff. HotStuff was introduced as a modification of Tendermint that is 'optimistically responsive' and requires only linear communication complexity within each 'view'. Of course, the hope is

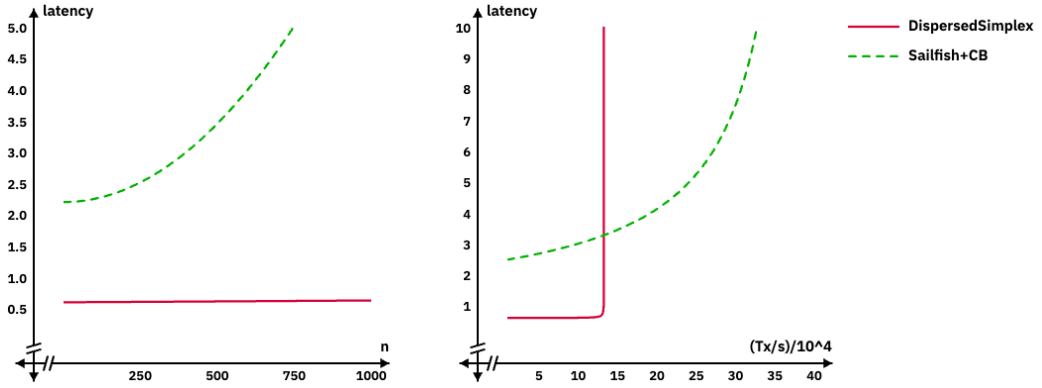

Fig. 1. Latency for Sailfish and DispersedSimplex: parameters are explained in Section 1

that reducing communication complexity will allow for better scalability. However, the reduced communication complexity is achieved by relaying all messages via the leader and the objection has been raised that the leader becomes a 'bottleneck' with this approach: this is not reflected in the communication complexity but should significantly impact 'real-world' latency. In Section 7.5, we analyse latency for HotStuff. In the calculations resulting from our model, latency for HotStuff is strictly greater than latency for Tendermint for all parameter values.

Summary. For state-of-the-art SMR protocols, the picture that emerges is one with trade-offs. If one compares single-sender protocols that use pipelining and erasure coding, such as DispersedSimplex, with DAG-based protocols such as Sailfish or Bullshark, the former are seen to have lower latency for a wide range of throughputs, while the benefit of the latter protocols is that they have a latency bottleneck which is higher by a constant factor. The comparison between DAG-based protocols and standard protocols such as Tendermint and HotStuff is somewhat simpler, because the former have a latency bottleneck which is greater by roughly a factor of n.

[Andy: Later, add comments on experiments.]

Paper structure. Section 2 defines the model. Section 3 considers Best-effort Broadcast. Sections 4–6 consider Reliable Broadcast, Reliable Broadcast with erasure coding, and Consistent Broadcast, respectively. Section 7 considers SMR protocols. Section 8 describes related work and Section 9 has some final comments.

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.*[7] 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.

[7]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.

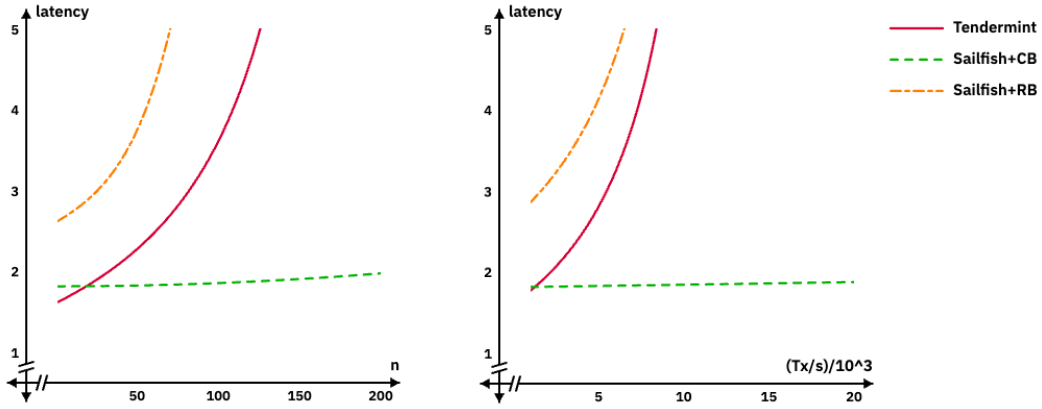

Fig. 2. Latency for Sailfish and Tendermint: parameters are explained in Section 1

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. For simplicity, we suppose time is divided into discrete slots[8] and that the message delay between any two processors is always Δ time slots.[9] We note that a protocol executed by the system of processors may function in partial synchrony or asynchrony [24], where the assumptions above may not always hold. However, our goal is to analyze throughput and latency under optimistic conditions — thus, 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 formalize 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 $t + \Delta$. 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 .

[8]Generally, we will think of each time slot as being sufficiently short that a 'rounding error' of plus or minus 1 when counting a number of time slots is not overly significant. This will simplify the presentation of a number of proofs.
[9]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.

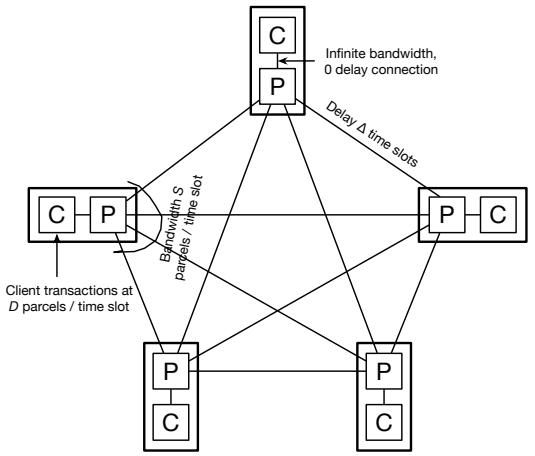

Fig. 3. Depicting a 5-processor system. Each processor involves a client processor (denoted C) and consensus processor (denoted P).

Modeling connections to clients. In a state machine replication protocol, the n processors receive transactions from external clients. Thus, the processors utilize their bandwidth to also receive transactions from these clients. To 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.[10] 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.

[10]Whether or not the download buffer of *client* processors is rate limited will not meaningfully impact our analysis.

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 finalized log. The relevant notion of delivery will be made precise in each context. A protocol has latency r if r 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 $t + r$. 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).

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 analyzing 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 4 describes the protocol.[11] The goal is for the leader to produce a block of transactions and send it to all processors along with some metadata, while minimizing latency. As mentioned in the introduction, 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.

[11]For simplicity of presentation, 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.

There is a designated processor p_ℓ. The following instructions are for p_i.
at time slot 0 **do**
    block-txns = {} ▷ {} denotes the empty sequence
    sendbuffer = {} ▷ sendbuffer is a FIFO queue distinct from the upload buffer
    recv-block = {}
at each time slot t **do**
    if $p_i = p_\ell$ **then**
        p_i collects *txns*, which is all data parcels received by the client processor but not yet added to sendbuffer
        block-txns = block-txns $\cup$ *txns*
        sendbuffer = sendbuffer||*txns*
        Dequeue the first S/n parcels in sendbuffer and add each to the upload buffer n times, once for each recipient ▷ Send some parcels
        if |block-txns| $\geq \frac{\alpha}{1-\alpha}M$ **then** ▷ M denotes the total size of metadata ▷ α is as specified in Section 3.1
            Set metadata$^{\text{de}}$ = *dependent* metadata for the current block
            Set metadata$^{\text{in}}$ = *independent* metadata for the next block ▷ dependent and independent metadata as specified in Section 3.1
            sendbuffer = sendbuffer||metadata$^{\text{de}}$||metadata$^{\text{in}}$
            block-txns = {}
    upon receiving parcels m from p_ℓ **do**
        recv-block = recv-block||m
        if |recv-block| $\geq \frac{M}{1-\alpha}$ **then** ▷ recv-block includes metadata
            Deliver recv-block as (metadata$^{\text{in}}$, *txns*, metadata$^{\text{de}}$) received from p_ℓ
            recv-block = {}

Fig. 4. Best-effort Broadcast by a single designated processor

- *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.[12]

We let M denote the total size of the metadata for each block, while the size of dependent and independent metadata are denoted M^{de} and M^{in} respectively (we assume these sizes are fixed and do not vary between blocks). As we will see, it is important to distinguish these different forms of metadata because they have different impacts on latency.[13]

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.[14] 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[15] of some of the parcels in sendbuffer and adds these addressed parcels to its upload buffer (adding parcels to the upload

[12]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.
[13]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.
[14]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.
[15]For simplicity, we assume that p_ℓ sends data to all processors, including itself.

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 parameterization. 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 $\frac{\alpha}{1-\alpha}M$ allows the leader to send metadata along with the transactions, while also minimizing latency.

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

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 4 is:*

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

Proof. 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 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). Recall that the leader receives $D = \alpha S/n$ transactions at its client processor every time slot. This means a set of $B := \frac{\alpha}{1-\alpha}M$ transactions is received in time:

$$\frac{B}{D} = \frac{Bn}{\alpha S} = \frac{Mn}{(1-\alpha)S}.$$

Note that the metadata added to sendbuffer at t is of size M and that $B + M = \frac{M}{1-\alpha}$. From each of these block data parcels added to sendbuffer, the leader needs to form n addressed data parcels to add to the upload buffer. S parcels are removed from the upload buffer of the leader at each timeslot, corresponding to S/n parcels from sendbuffer. To remove the addressed versions of the $B+M = \frac{M}{1-\alpha}$ block parcels from the upload buffer therefore takes $\frac{Mn}{(1-\alpha)S}$ time, which is precisely the amount of time it takes for the B transactions to arrive at the client processor of the leader. This means that,

when the leader completes the next block, b say, it will once again hold that sendbuffer does not contain any transactions and that the upload buffer does not contain any addressed transactions. Let t' be the time at which the leader completes b, i.e., adds the dependent metadata for b (together with the independent metadata for the next block) to sendbuffer.

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 metadata$^{\mathrm{de}}$ for b from the upload buffer requires $\frac{M^{\mathrm{de}}n}{S}$ time slots. As claimed, this gives a latency of:

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

□

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 $B = \frac{\alpha}{1-\alpha}M$ minimizes the latency. As α tends to 1, block size and latency tends 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 metadata$^{\mathrm{in}}$ first, followed by the transactions in the block. Then send metadata$^{\mathrm{de}}$ corresponding to the block. This ensures that the full bandwidth for data transmission is always utilized. 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 4 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 4 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 4 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 4 is:

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

Our next aim is to show that *any* leader-based protocol in which the leader sends all data to all other processors has at least this latency.

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.

Intuitively, it may seem clear that the protocol of Figure 4 is the best one can possibly do if one wishes to minimize latency for a single-sender protocol, and if one is restricted to using a protocol in which the leader must send *all* data to *all* processors without using methods like erasure coding or a gossip network. However, the question remains: is there a formal sense in which one can establish that the latency of the protocol is optimal? To answer this question we will need some new techniques. First, though, we need to formalise what we mean by 'protocols that don't use techniques like erasure coding or a gossip network'.

Simple-broadcast protocols. We say that a protocol is a *simple-broadcast* protocol if both:

(1) 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, and;
(2) 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 ,[16] i.e., p_i must receive every block data parcel m of b from p_j before transactions in the block are delivered.[17]

The intention of this definition is to exclude the use of techniques such as erasure coding or a gossip network for improving latency. The definition does not rule out standard approaches to Best-effort Broadcast, such as the leader simply waiting until a full block is formed and then sending it to

[16] 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.
[17] 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.

all processors. However, the protocol of Figure 4 is rather specific in the way it separately treats independent and dependent metadata, and has lower latency than such standard approaches.

The following claim establishes that the protocol of Figure 4 is exactly optimal amongst simple-broadcast protocols for the single-sender setting.

Claim 2. *Consider the single-sender setting, suppose all processors are correct, and let M and M^{de} be as defined previously. If $D = \alpha S/n$ for $\alpha \in (0,1)$, then latency for any simple-broadcast protocol is at least:*

$$f(M, M^{de}, D, S, n, \Delta) := \frac{(M + (1-\alpha)M^{de})n}{(1-\alpha)S} + \Delta.$$

Proof. By the definition of a simple-broadcast protocol, each block sent by p_ℓ must be sent to all processors. So, for each block b sent by p_ℓ, we can consider the first timeslot, $t_1(b)$ say, by which it holds, for every data parcel m of b and every processor p_i, that p_ℓ has removed a copy of m addressed to p_i from its upload buffer (meaning that b will be received by all processors by $t_1(b) + \Delta$).

We can assume that each transaction received by the client processor of p_ℓ is included in precisely one block sent by the leader. To see this, note that, if some transaction is not included in any block, then latency is unbounded. If a transaction `tr` is included in two blocks, b_1 and b_2 say, then, without loss of generality, suppose that $t_1(b_2) \geq t_1(b_1)$. In this case, one can remove `tr` from b_2 without increasing latency. We can also assume that each block sent by p_ℓ includes at least one transaction, since one can eliminate the sending of empty blocks without increasing latency.

The basic idea. For each block b sent by p_ℓ, let $t_0(b)$ be the first time slot at which p_ℓ receives a transaction at its client processor that is included in b. We define the *block latency for b* to be $\ell(b) := t_1(b) + \Delta - t_0(b)$. Note that $\ell(b)$ lower bounds latency, since at least one transaction in b is received at client processor of p_ℓ at t_0, and $t_1(b) + \Delta$ is the first time slot at which transactions in b are delivered by all correct processors. The basic idea behind the remainder of the proof is to use an averaging argument to show that *mean* block latency is at least $f(M, M^{\mathrm{de}}, D, S, n, \Delta)$, so that the latter value must lower bound $\ell(b)$ for at least one b. (In fact, mean block latency might not be defined, i.e., might not come to a limit as the number of blocks considered tends to infinity, so we have to be careful in formalising this idea, but the above indicates the spirit of the argument that follows.)

Accountancy for time slots in $[t_0(b), t_1(b)]$. To lower bound $t_1(b) - t_0(b)$, we consider which data parcels are removed from the upload buffer of p_ℓ at each time slot in the interval $[t_0(b), t_1(b)]$. Let $M_-^{\mathrm{in}}(b)$ be the number of addressed independent metadata parcels for b which are removed from the upload buffer at time slots strictly *before* $t_0(b)$, and let $M_+^{\mathrm{in}}(b)$ be the number of addressed independent metadata parcels for b which are removed from the upload buffer at time slots at or *after* $t_0(b)$. If $B(b)$ transaction parcels are included in b then:

(i) Removing addressed transactions in b from the upload buffer accounts for at least $B(b)n/S$ timeslots in $[t_0(b), t_1(b)]$;
(ii) Removing addressed dependent metadata parcels for b accounts for at least $M^{\mathrm{de}}n/S$ timeslots in $[t_0(b), t_1(b)]$, and;
(iii) Removing addressed independent metadata parcels for b accounts for a further $M_+^{\mathrm{in}}(b)/S$ timeslots in $[t_0(b), t_1(b)]$.

However, we must also consider that there may be time slots in the interval $[t_0(b), t_1(b)]$ at which the leader removes data parcels corresponding to blocks *other than b* from the upload buffer: suppose

that removing data parcels corresponding to blocks other than b from the upload buffer accounts for $x(b)$ time slots in $[t_0(b), t_1(b)]$.

Let the blocks sent by the leader be $b_1, b_2, \ldots$, ordered by $t_1(b)$ and with ties broken by least hash. For each $r \in \mathbb{N}_{\geq 1}$, set $\mathbb{E}_r[\ell(b)] = \sum_{i=1}^{r} \ell(b_r)/r$. We also extend this notation in the obvious way to other terms, such as $\mathbb{E}_r[x(b)]$. By linearity of expectation and the observations above it follows that:

$$\mathbb{E}_r[\ell(b)] \geq n\,\mathbb{E}_r[B(b)]/S + nM^{\mathrm{de}}/S + \mathbb{E}_r[M_+^{\mathrm{in}}(b)]/S + \mathbb{E}_r[x(b)] + \Delta. \tag{1}$$

The remainder of the proof consists of two steps:

(1) First, we lower bound $\mathbb{E}_r[x(b)]$ for all sufficiently large r.
(2) Then we lower bound $\mathbb{E}_r[B(b)]$ for all sufficiently large r.

Combining these bounds with (1) suffices to establish the claim.

Step 1. We aim to bound $\mathbb{E}_r[x(b)]$ for sufficiently large r. Since the dependent metadata for any block b cannot be determined until after all the transactions for the block have been received, the time slots at which the leader removes the dependent metadata for b from its upload buffer contribute at least nM^{de}/S to the $x(b')$ values of other blocks.[18] Similarly, the time slots prior to $t_0(b)$ at which the leader removes the $M_-^{\mathrm{in}}(b)$ addressed independent metadata parcels for b from the upload buffer contribute $M_-^{\mathrm{in}}(b)/S$ to the $x(b')$ values of other blocks. Let us write $b_i \rightharpoonup x(b_j)$ if metadata for b_i is removed from the upload buffer of p_ℓ in the interval $[t_0(b_j), t_1(b_j)]$, i.e., the removal of metadata for b_i from the upload buffer contributes positively to $x(b_j)$. Note that:

$(*)$ If latency is bounded, there must exist r^* such that it is never the case $b_i \rightharpoonup x(b_{i+r})$ for $r \geq r^*$.

To argue that $(*)$ holds, note that at most S blocks b' can share the same value $t_1(b')$. So, if $b_i \rightharpoonup x(b_{i+r})$, then $\ell(b_{i+r}) \geq r/S$.

Now consider the blocks $b_1, \ldots, b_r$. The idea is now to show that, if r is large, then almost all the blocks b_j in this sequence (all but at most the last r^*) contribute at least $nM^{\mathrm{de}}/S + M_-^{\mathrm{in}}(b_j)/S$ to $\sum_{i=1}^{r} x(b_i)$. This allows us to lower bound $\mathbb{E}_r[x(b)]$. For each $j < r - r^*$, the time slots at which the leader removes the dependent metadata for b_j from its upload buffer must contribute at least nM^{de}/S to $\sum_{i=1}^{r} x(b_i)$. Similarly, for $j < r - r^*$, the time slots prior to $t_0(b_j)$ at which the leader removes the $M_-^{\mathrm{in}}(b_j)$ addressed independent metadata parcels for b_j from the upload buffer contribute $M_-^{\mathrm{in}}(b_j)/S$ to $\sum_{i=1}^{r} x(b_i)$. It follows that, for each $\epsilon > 0$, the following holds for all sufficiently large r:

$$\mathbb{E}_r[x(b)] \geq (1-\epsilon)\left(nM^{\mathrm{de}}/S + \mathbb{E}_r[M_-^{\mathrm{in}}(b)]/S\right). \tag{2}$$

Putting equations (1) and (2) together, and since $\mathbb{E}[M_-^{\mathrm{in}}] + \mathbb{E}[M_+^{\mathrm{in}}] = \mathbb{E}[M_-^{\mathrm{in}} + M_+^{\mathrm{in}}] = nM^{\mathrm{in}}$, we conclude that for each $\epsilon > 0$, it holds for all sufficiently large r that:

$$\mathbb{E}_r[\ell(b)] \geq n\,\mathbb{E}_r[B(b)]/S + nM^{\mathrm{de}}/S + (1-\epsilon)Mn/S + \Delta.$$

If latency is bounded, so that the lim inf values below are defined, this means that:

$$\liminf_r \mathbb{E}_r[\ell(b)] \geq n \liminf_r \mathbb{E}_r[B(b)]/S + nM^{\mathrm{de}}/S + Mn/S + \Delta. \tag{3}$$

[18]We allow a rounding error of one time slot here (which also applies in the proof of Claim 1). In the very first time slot at which p_ℓ removes dependent metadata for b from its upload buffer, it may be that no transactions arrive that are included in blocks other than b. However, for all subsequent time slots at which p_ℓ removes dependent metadata for b from its upload buffer, transactions will arrive at the client processor of p_ℓ and these transactions must be included in blocks other than b, because the dependent metadata for b has already been determined.

Step 2. To lower bound $\liminf_r \mathbb{E}_r[B(b)]$, note that it takes time at least $t(r) := n\,\mathbb{E}_r(B)r/S + Mnr/S$ to remove blocks $b_1, \ldots, b_r$ from the upload buffer. Since $D \cdot t(r)$ many transactions arrive at the client processor of the leader in this time, for latency to be bounded we require that, for each $\epsilon > 0$, it holds for all sufficiently large r that:

$$r\,\mathbb{E}_r[B(b)] \geq (1-\epsilon)\,(D(n\,\mathbb{E}_r(B)r/S + Mnr/S))\,. \tag{4}$$

To see that (4) must hold, note that there must exist an upper bound, B^* say, on the number of transactions included in a block, if latency is to be bounded. For each r, let $y(r)$ be the set of transactions that have been received at the client processor of p_ℓ by $t(r)$, but which are not included in $b_1, \ldots, b_r$. The failure of (4) means that $|y(r)|$ is unbounded, i.e., can be arbitrarily large for large r. The transactions in $y(r)$ must be divided between at least $|y(r)|/B^*$ many blocks. Since at most S blocks b' can share the same value $t_1(b')$, some transaction in $y(r)$ has latency at least $|y(r)|/SB^*$.

From (4), it follows that:

$$\liminf_r \mathbb{E}_r[B(b)] \geq \frac{DMn}{S - nD}. \tag{5}$$

Putting steps 1 and 2 together. Putting (5) together with (3), it follows that:

$$\liminf_r \mathbb{E}_r[\ell(b)] \geq \frac{(M + (1-\alpha)M^{\mathrm{de}})n}{(1-\alpha)S} + \Delta,$$

so that latency is also bounded by the r.h.s. of this inequality, as claimed. □

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 parallelized, potentially leading to a more even use of the available bandwidth.

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 latency 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.

Claim 3. *Suppose $D = \alpha S$ for $\alpha \in (0,1)$ and that all processors are correct. Then latency for the protocol in Figure 6 is optimal for 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. The proof is essentially the same as the proofs of Claims 1 and 2. The only difference here is that every processor is 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. □

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. [37]). As described in Section 3.4, 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

Initialize
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 $\cup$ *txns*
sendbuffer = sendbuffer||*txns*

Transfer to upload
Dequeue the first S/n parcels in 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:
recv-block$_j$ = recv-block$_j$||m
If fullblock(recv-block$_j$) = true:
Deliver block received from p_j
recv-block$_j$ = {}

Fig. 5. Some procedures used in Figure 6 (and in later sections)

The following instructions are for p_i.
at time slot 0 **do**
Initialize ▷ As specified in Figure 5
at each time slot t **do**
Collect txns ▷ As specified in Figure 5
Transfer to upload ▷ As specified in Figure 5

if |block-txns| = $\frac{\alpha}{1-\alpha}M$ **then** ▷ M denotes the total size of metadata
▷ $\alpha = D/S$
Set metadata$^{\text{de}}$ = *dependent* metadata for the current block
Set metadata$^{\text{in}}$ = *independent* metadata for the next block
sendbuffer = sendbuffer||metadata$^{\text{de}}$||metadata$^{\text{in}}$
block-txns = {}
Receive blocks ▷ As specified in Figure 5

Fig. 6. Best-effort Broadcast by each processor

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 6 is:*

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

Analysis. Consider the setting above, where $M^{\mathrm{de}} = \lambda$, $M^{\mathrm{in}} = n\lambda$, 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., [22]) consider high bandwidth networks (large S) and relatively small-to-moderate n.

3.4 Best-effort Broadcast with a layered DAG

In this section, we consider how the analysis of Section 3.3 changes in the case that one is required to build a structured DAG. *The reader may skip to Section 3.5 without impacting their ability to understand later sections.*

The analysis of Section 3.3 applies to a scenario in which each processor can choose n previous blocks to point to at the beginning of the process of forming a new block. However, many DAG-based protocols build a more structured DAG consisting of 'layers': each block in layer $d+1$ must reference (via hash pointers in the metadata) $\Theta(n)$ blocks from layer d. Analyzing latency for protocols of this form introduces certain complexities:

(a) Correct processors may not be perfectly 'in sync'. In particular, processors may not begin building their blocks for a given layer at the same time as each other.
(b) Even if each correct processor begins construction for layer $d+1$ at exactly the same time, processors must wait to receive blocks in layer d before they can determine the hash pointers to be included in the metadata for their block in layer $d+1$.

In this section, we subdue complexity (a) above, i.e. for the sake of simplicity, we consider a scenario in which correct processors begin building their blocks for each layer simultaneously. In this simplified setting, we consider the impact of complexity (b) on latency. Setting $\alpha = D/S$, this leads to three regimes of interest:

(i) **Regime A**: $EC(M^{\mathrm{de}}) \geq C(M^{\mathrm{de}}) + \Delta$.
(ii) **Regime B**: Regime A does not hold, but $EC(M) \geq C(M) + \Delta$.
(iii) **Regime C**: Neither of the above.

We consider these three regimes separately in what follows.

Regime A. In this case, suppose that all processors (perfectly synchronized) finish production of layer d at t by adding M^{de} to their buffer, but have not yet received full blocks for layer d from other processors. Each processor will then receive the layer d blocks of other processors by $t + C(M^{\mathrm{de}}) + \Delta$, fully using their bandwidth all the while, and so can add M^{in} (the pointers to blocks from the previous layer) at this time. Figure 7 shows the resulting protocol. Since the time to clear the block metadata and all incoming transactions from sendbuffer is unchanged depending on whether we add M^{in} to sendbuffer at t or $t + C(M^{\mathrm{de}}) + \Delta$, the latency analysis is then essentially the same as for Section 3.3. As before, the latency is:

$$EC(M) + C(M^{\mathrm{de}}) + \Delta = \frac{(M + (1-\alpha)M^{\mathrm{de}})n}{(1-\alpha)S} + \Delta.$$

The following instructions are for p_i.
at time slot 0 **do**
  Initialize ▷ As specified in Figure 5
  Set $d = 1$
at each time slot t **do**:
  Collect txns ▷ As specified in Figure 5
  Transfer to upload ▷ As specified in Figure 5
  If $d > 1$ and received layer $d - 1$ = true
    Set metadata$^{\text{in}}$ = independent metadata for the current block
    Set sendbuffer = sendbuffer||metadata$^{\text{in}}$
  If |block-txns| $\geq \frac{\alpha}{1-\alpha}M$ ▷ $\alpha = D/S$
    Set metadata$^{\text{de}}$ = dependent metadata for the current block
    sendbuffer = sendbuffer||metadata$^{\text{de}}$
    block-txns = {}
    $d := d + 1$
  Receive blocks ▷ As specified in Figure 5

Fig. 7. Best-effort Broadcast for a layered DAG in Regime A

Regime B. In this case, we can ensure that processors use their entire bandwidth at each time slot by adding all metadata at the end of the block. The resulting protocol is shown in Figure 8. This gives latency of:

$$EC(M) + C(M) + \Delta = \frac{(2-\alpha)Mn}{(1-\alpha)S} + \Delta.$$

The following instructions are for p_i.
at time slot 0 **do**
  Initialize
  Set $d = 1$
at each time slot t **do**:
  Collect txns
  Transfer to upload
  If sendbuffer is free of transactions **and** ($d = 1$ **or** received layer $d - 1$ = true):
    Set metadata$^{\text{in}}$ = independent metadata for the current block
    Set metadata$^{\text{de}}$ = dependent metadata for the current block
    sendbuffer = sendbuffer||metadata$^{\text{in}}$||metadata$^{\text{de}}$
    $d := d + 1$
  Receive blocks

Fig. 8. Best-effort Broadcast for a layered DAG in Regimes B or C

Regime C. In this case, the protocol of Figure 8 still applies, but now the latency analysis is different, because the delay Δ dominates the wait to receive blocks from the previous layer. For the same protocol, latency is now:

$$2(C(M) + \Delta) = \frac{2Mn}{S} + 2\Delta.$$

This can be explained as follows. Let us say t is the time at which metadata for layer $d-1$ was added to the sendbuffer by p_i. So, any transaction arriving at p_i after time t will be sent as a part of layer d. The metadata sent by p_i at time t will be received by all parties at time $t + C(M) + \Delta$; similarly, party p_i will receive layer $d-1$ metadata from all parties at time $t + C(M) + \Delta$. Party p_i keeps sending transactions for layer d starting at time t. Once it has received metadata for layer $d-1$ (at time $t + C(M) + \Delta$), it will start adding metadata for layer d in the sendbuffer. This will arrive at all parties at $t + 2C(M) + 2\Delta$.

One can view the first $C(M) + \Delta$ term from layer $d-1$ as being the queueing delay for layer d. The latter $C(M) + \Delta$ constitutes the time for completing the layer d block. Observe that, the transactions corresponding to layer d are pipelined; they are sent in the intervening time before layer $d-1$ blocks are entirely received. Moreover, the value of $C(M) + \Delta$ determines the block size.

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 latency 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.

To illustrate these 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. 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 9. The left-hand graph in Figure 9 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 emphasize (as a point of academic interest) that there are some regions in which the single-sender protocol of Figure 4 does have (slightly) lower latency than the multisender protocol of Figure 6.

It should be emphasized 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.

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

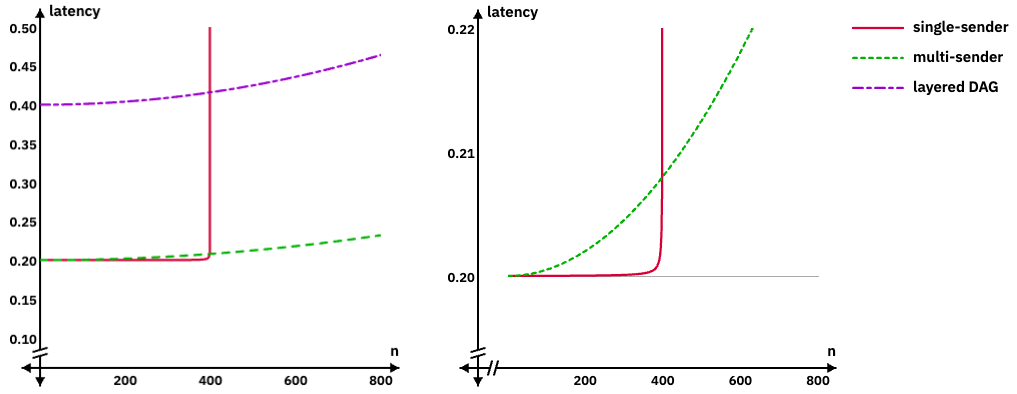

Fig. 9. Latency for BEB: parameters are explained in Section 3.5. The graph to the right is a zoomed-in version of the one to the left to display the difference between single-sender and multi-sender approaches.

all metadata is determined at the end of the block production process. Thus, we will no longer distinguish between independent and depedent 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) With Claims 2 and 3, we established tight lower bounds on latency for simple-broadcast protocols in the single-sender and multi-sender settings. In future sections, we will not aim to establish tight lower bounds, but will instead analyse latency for certain well-known protocols. In each case, we will determine the optimal *block size* for the given protocol.

4 RELIABLE BROADCAST

4.1 Reliable Broadcast by a single sender

We start by considering a standard reliable broadcast protocol in the single-sender setting. A designated leader receives D transaction parcels per time slot and sends blocks to all others using a version of Bracha's broadcast [11], under the assumption that the number of faulty processors is bounded by $f < n/3$ (while the protocol assumes $f < n/3$, our *analysis* assumes all processors are correct, so that we do not have to consider download bandwidth consumed by the sending of extra messages by faulty processors). The protocol is shown in Figure 10. We assume the reader is familiar with Bracha's broadcast in what follows and analyse only the resulting latency, rather than re-verifying correctness.

In analysing latency for the protocol of Figure 10, we make the following simplification: if a block b contains B transaction parcels and metadata of size M, then we also suppose that the message (ECHO, d, b) is of size $B + M$. We suppose that *votes*, i.e. messages of the form $(\text{VOTE}, d, H(b))$, are of a fixed size λ.

There is a designated processor p_ℓ. The following instructions are for p_i.
at time slot 0 do:
  Initialize ▷ As specified in Figure 5
  Set $d = 1$, send = true, echo = true, vote = true ▷ d is block depth

at time slot t **if** send = true **and** $p_i = p_\ell$:
  Collect txns ▷ As specified in Figure 5
  If |block-txns| = B **do**: ▷ B as specified in Section 4.1
    Set metadata = metadata for the current block
    Set sendbuffer = sendbuffer||metadata ▷ d included in metadata
    Set send = false

at every time slot t **do**:
  If echo = true **and** p_i has received a first block b of depth d from p_ℓ **do**:
    Set sendbuffer = sendbuffer||(ECHO, d, b)
    Set echo = false
  If vote = true **do**:
    If p_i has received (ECHO, d, b') from $n - f$ distinct processors (for some fixed b') **do**:
      Set sendbuffer = sendbuffer||(VOTE, $d, H(b')$) ▷ H a hash function
      Set vote = false
  If vote = true **do**:
    If p_i has received (VOTE, $d, H(b')$) from $f + 1$ distinct processors (for some fixed b') **do**:
      Set sendbuffer = sendbuffer||(VOTE, $d, H(b')$)
      Set vote = false
  If p_i has received (VOTE, $d, H(b')$) from $n - f$ distinct processors (for some fixed b')
  and if p_i has also received b' **do**:
    Deliver b'
    Set send = true, echo = true, vote = true, $d = d + 1$
  Transfer to upload ▷ As specified in Figure 5

Fig. 10. Reliable Broadcast by a designated sender

Claim 4. *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 10 is:*

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

Note that, if $S/Dn > 2$, the second terms in parentheses above is ≥ 1 and tends to infinity as $2Dn$ approaches S. If $2Dn$ is small compared to S, then the latency is roughly equal to the first term in parentheses. So, one can think of the first term as giving the approximate latency when $2Dn$ is small compared to S, while latency tends to infinity as $2Dn$ approaches S. Figure 13 gives a plot of the resulting latency for parameter values that are explained in Section 4.3.

Proof. Suppose B transactions are included in each block and that, at some time slot t, the leader delivers a block of depth $d - 1$ and begins sending transactions for the next block b of depth d. All processors will then deliver b by time $t' := t + 2(B + M)n/S + n\lambda/S + 3\Delta$ at the earliest. Since $D(2(B + M)n/S + n\lambda/S + 3\Delta)$ transactions will arrive at the client processor of the leader between

t and t', if latency is to be bounded we require:

$$B \geq \frac{D(2(B+M)n + n\lambda + 3\Delta S)}{S}.$$

From this it follows that:

$$B \geq \frac{D(2nM + n\lambda + 3\Delta S)}{S - 2nD}.$$

Set:

$$B := \frac{D(2nM + n\lambda + 3\Delta S)}{S - 2nD}. \tag{6}$$

Consider next 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 $S/2n$ transactions arrive at the client processor of p_ℓ (since $S/Dn > 2$), and are immediately added to sendbuffer. Each 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., 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 the next block at time:

$$t' := t + C(M) + C(B+M) + C(\lambda) + 3\Delta = t + B(S - 2nD)/(DS) + Bn/S.$$

To see that the second equality above holds, feed the expression for B in (6) into the expression on the right. In the interval $(t, t']$, the leader p_ℓ will therefore receive $B(S-2nD)/S + BnD/S$ many transaction parcels, which will be immediately added to sendbuffer at t'. To complete the block, the leader requires a further

$$B - B(S - 2nD - nD)/S = nBD/S$$

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:

$$\frac{(3B + 4M + 2\lambda)n}{S} + 6\Delta.$$

Substituting in the value for B in (6) gives the latency as claimed. Note also that increasing B increases latency according to the calculations above, so that B as specified in (6) minimises latency. □

A comment on optimisations. The protocol of Figure 10 is in no way 'optimal': the aim at this point is just to analyse latency for standard protocols. There are many ways in which the protocol could be adjusted to reduce latency, and we will consider some of these in the sections that follow.

4.2 Reliable Broadcast by all processors

We consider next a related primitive for the multi-sender setting: every processor receives D/n transaction parcels at its client processor at each time slot and simultaneously reliably broadcasts to all processors. The protocol, which references the $\oplus$ function defined in Figure 11, is shown in Figure 12. The $\oplus$ function is used to evenly spread the sending of data parcels, and we presume that addressed data parcels contain information that allows processors to recover each m_i from $\oplus_I m_i$.

Suppose $I \subseteq \{1, \dots, n\}$ and that, for each $i \in I$, m_i is a sequence of data parcels $m_{i,1}, \dots, m_{i,x_i}$ (where $m_{i,x}$ is undefined for $x > x_i$). Then $\oplus_I m_i$ is the sequence of data parcels formed as follows:

Let $I = \{i_1, \dots i_k\}$, where each $i_{k'} < i_{k'+1}$ for $k' \in [1, k)$.
Set $x^* = \max\{x_i : i \in I\}$
Set m to be the empty sequence.
For $x = 1$ to x^* do:
  For $j = 1$ to k do:
    If $m_{i_j,x}$ is defined, set $m := m||m_{i_j,x}$
Output m

Fig. 11. Defining $\oplus$

In analysing latency for the protocol of Figure 12, we make the following simplification: if a block b contains B transaction parcels and metadata of size M, then we also suppose that the message (ECHO, d, b) is of size $B + M$. We suppose that *votes*, i.e., messages of the form (VOTE, $d, H(b)$), are of a fixed size λ and analyse latency in the case that all processors are synchronized (i.e., all begin the protocol at precisely the same time) and are correct.

Claim 5. *Suppose all processors are synchronized and correct. If $S/Dn > 1$ and the block size B is set to minimise latency, then, once lower order terms are removed, latency for the protocol of Figure 12 is:*

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

Note that, if $S/Dn > 1$, the second terms in parentheses above is ≥ 1 and tends to infinity as Dn approaches S. If Dn is small compared to S, then the latency is roughly equal to the first term in parentheses.

Proof. With a calculation exactly analogous to that in the proof of Claim 4, we may conclude that, for latency to be bounded, we require B to be greater than or equal to:

$$\frac{D(M + (M+\lambda)n + 3\Delta S/n)}{S - D - Dn}. \tag{7}$$

So, set B equal to this value.

Consider next what happens between timeslot 0 and the first timeslot, t_1 say, at which any processor p_i adds metadata to sendbuffer. At each timeslot in this interval, less than S/n transactions arrive at the client processor of p_i (since $S/Dn > 1$), and are immediately added to sendbuffer. Each is then added to p_i'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, each processor's sendbuffer does not contain any transactions and that each processor's upload buffer does not contain any addressed transactions.

The following instructions are for p_i.
at time slot 0 **do**:
  Initialize
  Set send = true
  $\forall j$, echo_j = true, vote_j = true, $d_j = 1$

at time slot t **if** send = true:
  Collect txns
  If |block-txns| = B **do**: ▷ B as specified in Section 4.2
    Set metadata = metadata for the current block
    Set sendbuffer = sendbuffer||metadata ▷ d_i included in metadata
    Set send = false

at every time slot t **do**:
  Let $I = \{j :$ echo_j = true **and** p_i has received a first block b_j of depth d_j from $p_j\}$
  Set sendbuffer = sendbuffer|| $\oplus_I$ (ECHO, j, d_j, b_j)
  $\forall j \in I$ set echo_j = false
  Let $I = \{j :$ vote_j = true **and** p_i has received (ECHO, j, d_j, b'_j) from $n - f$
  distinct processors (for some fixed b'_j)$\}$
  Set sendbuffer = sendbuffer|| $\oplus_I$ (VOTE, $j, d_j, H(b'_j)$)
  $\forall j \in I$ set vote_j = false
  Let $I = \{j :$ vote_j = true **and** p_i has received (VOTE, j, d_j, b'_j) from $f + 1$
  distinct processors (for some fixed b'_j)$\}$
  Set sendbuffer = sendbuffer|| $\oplus_I$ (VOTE, $j, d_j, H(b'_j)$)
  $\forall j \in I$ set vote_j = false
  $\forall j$ s.t. p_i has received (VOTE, $j, d_j, H(b'_j)$) from $n - f$ distinct processors (for some fixed b'_j)
  and s.t. p_i has also received b'_j **do**:
    Deliver b'_j
    Set echo_j = true, vote_j = true, $d_j = d_j + 1$
    If $j = i$, set send = true
  Transfer to upload

Fig. 12. Reliable Broadcast by all processors

Now suppose that, at some timeslot t, some processor p_i 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 the block at time:

$$t' := t + C(M) + C((B + M)n) + C(\lambda n) + 3\Delta = t + Bn(S - D - Dn)/(DS) + Bn^2/S.$$

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

$$B - B(S - D - Dn)/S - BnD/S = BD/S$$

transaction parcels, which will arrive in time nB/S, i.e. $C(B)$, which is exactly the time that p_i requires to clear sendbuffer of all transactions. It therefore holds that when the next layer of blocks

is delivered at $t'' := t' + nB/S$, each processor's 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 layer of blocks (which processors start sending at t') is delivered. This is:

$$\frac{Bn + 2Bn^2 + 2Mn + 2(M+\lambda)n^2}{S} + 6\Delta.$$

Substituting in the value for B in (7), equating $S - D(n+1)$ with $S - Dn$, and removing lower order terms gives the latency below as claimed:

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

As in the proof of Claim 4, we may also observe that increasing B increases latency according to the calculations above, so that B as specified in (7) minimises latency. □

Of course, DAG-based protocols generally have metadata which is $\Theta(n)$:

Corollary 2. *Set $M + \lambda = n\lambda'$. If $S/Dn > 1$ and the block size B is set to minimise latency, then, once lower order terms are removed, latency for the protocol of Figure 12 is:*

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

4.3 Comparing latencies for Reliable Broadcast

There are at least two significant differences between the results of Sections 3 and 4. First, Section 3 demonstrated a factor n difference in the point at which the latency bottleneck appears for the single-sender and multi-sender approaches: for the single-sender approach the bottleneck was at $Dn = S$, while for the multi-sender approach the bottleneck was at $D = S$. For Reliable Broadcast, however, the corresponding bottlenecks are $2Dn = S$ and $Dn = S$, and so differ only by a factor of 2. The factor of 2 arises because, in the single-sender case, there are *two* rounds which are equally expensive in terms of latency: first the leader sends their block to all, and then all others echo it to all. In the multi-sender case, the single round which dominates in terms of latency cost is that in which each processor has to echo-to-all *all* blocks produced by other processors. As discussed later, this factor of 2 can be eliminated by using *pipelining* techniques in the single sender case: in the single sender case, one can remove the need for the leader to echo their own blocks, so that the leader may immediately progress to broadcasting their next block while others echo the current one. While we do not formally analyse pipelining in this Section, we will do so in Section 5.3. Such techniques would not seem to apply (or, at least, not so effectively) to the multi-sender case, since the expensive round in that case is anyway the all-to-all echoing of all blocks.

A second observation is that the first term in parentheses in the statement of Corollary 2 is already *cubic* in n in the case that metadata is $\Theta(n)$, while if metadata in the single-sender case is of constant size then the corresponding term in the single-sender case is $O(n)$. Perhaps to avoid this issue, practical instantiations of DAG-based protocols tend to use a form of *Consistent Broadcast* rather than Reliable Broadcast. We will analyze Consistent Broadcast later in Section 6.1.

To illustrate these trade-offs, we suppose again that 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. We set $\lambda = 500$ bits and $\Delta = 0.2$ seconds. The first graph in Figure 13 supposes the incoming transaction rate D is 10000 transactions per second and shows

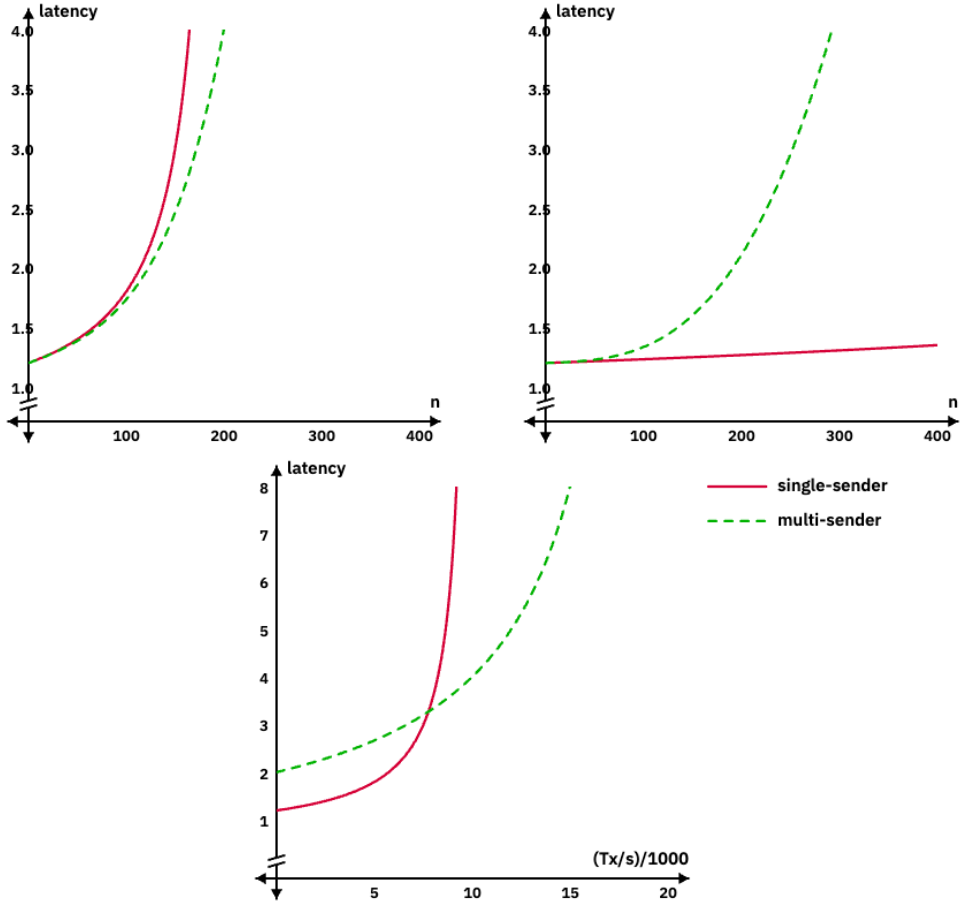

Fig. 13. Latency for RB: parameters are explained in Section 4.3

the resulting latency (in seconds) as a function of n. The second graph (on the right) in Figure 9 supposes the incoming transaction rate D is 1000 transactions per second and shows the resulting latency as a function of n. The lower graph fixes $n = 200$ and shows the resulting latency as a function of the number of incoming transactions per second divided by 1000: one can see that the latency bottleneck in the single-sender case appears at 10000 transactions per second.

5 RELIABLE BROADCAST WITH ERASURE CODING

5.1 Reliable Broadcast by a single sender with erasure coding

We next consider an erasure coding version of Reliable Broadcast in the single-sender setting. The protocol is based on that described by Cachin and Tessaro [16], which uses a scheme for erasure coding originally described by Rabin [52]. We assume familiarity with those papers in what follows. At a high level, the protocol uses certain basic functionalities:

- An injective function F, which converts a sequence of parcels m into a tuple $F(m) = (f_1(m), \dots, f_n(m))$, where each $f_i(m)$ consists of $3|m|/n$ parcels.[19] F satisfies the property that m can be recovered from any $f+1$ of the values $f_1(m), \dots, f_n(m)$.
- For each $i \in [1, n]$, a function G_i, which converts a sequence of parcels m into a 'fragment' $(i, f_i(m), x, y)$, where:
  - $f_i(m)$ is the i^{th} component of $F(m)$;
  - x is the Merkle root of the Merkle tree with leaves $f_1(m), \dots, f_n(m)$ (we also refer to x as the 'Merkle root of m'), and;
  - y is a *fingerprint* of $\Theta(\log(n))$ data parcels, which acts as a witness that $f_i(m)$ is the i^{th} leaf of a Merkle tree with root x.
- A *verification* function V such that $V(i, z, x, y) = \text{true}$ iff y is a witness that x is the Merkle root of a Merkle tree with i^{th} leaf z. For a given x, we let V_x be the set of tuples of the form (i, z, x, y) such that $V(i, z, x, y) = \text{true}$.
- A *reconstruction* function R such that, for any x and any set X of $f+1$ distinct tuples in V_x, X={ $(i_1, z_1, x, y_1), \dots, (i_{f+1}, z_{f+1}, x, y_{f+1})$} with $i_j \neq i_{j'}$ for $j \neq j'$, $R(X)$ outputs the unique m such that, for each $j \in [1, f+1]$, $G_{i_j}(m) = (i_j, z_j, x, y_j)$ (so that $z_j = f_{i_j}(m)$) if there exists such, and outputs false otherwise.

The resulting protocol is a modification of the protocol for Reliable Broadcast in Figure 10 and is shown in Figure 16. We note that the instructions must now be altered to incorporate the requirement of erasure coding that the the entire block be known before the leader starts sending any part of the block to others. The protocol uses a function $\oplus^*$, which is defined in Figure 14 below, and also uses the function defined in Figure 15. While the protocol decreases latency compared to that in Figure 10 through the use of erasure coding, there remain various ways in which the protocol is non-optimal. For example, the leader waits until one block is delivered before beginning to send data for the next (i.e., there is no use of 'pipelining'). Another inefficiency is that other processors wait until receiving their full 'fragment' $G_i(b)$ of a block b before beginning to echo that fragment to others. We will address these issues in Section 5.3.

Suppose that, for each $i \in [1, n]$, m_i is a sequence of data parcels $m_{i,1}, \dots, m_{i,x_i}$ (where $m_{i,x}$ is undefined for $x > x_i$). For each i and x, let $m^i_{i,x}$ be the addressed data parcel which is $m_{i,x}$ intended for recipient p_i. Then $\oplus^*_{i\in[1,n]} m_i$ is the sequence of addressed data parcels formed recursively as follows:

Set $x^* = \max\{x_i : i \in [1, n]\}$
Set m to be the empty sequence.
For $x = 1$ to x^* do:
  For $i = 1$ to n do:
    If $m_{i,x}$ is defined, set $m := m||m^i_{i,x}$
Output m

Fig. 14. Defining $\oplus^*$

Since the verification of correctness is essentially the same as in [16], we focus on analyzing latency. To analyze the latency, we make the following simplifications. We suppose that *votes*, i.e. messages of the form (VOTE, d, x), are of a fixed size λ (essentially, the size of a hash), and that the block metadata (perhaps a hash and a signature) is of size $c\lambda$ for some small constant c. If a block b

[19] As in previous sections, we ignore issues of integer rounding for the sake of simplicity.

$R_{\text{val}}(X, d, x, k)$
If $X < k$, **output** false
If it does not hold that every message in X is from a different processor p_j and is of the form $(\text{ECHO}, d, j, z', x, y')$ such that $V(j, z', x, y') = \text{true}$;
output false
If it holds for any (equivalently all, assuming hashes are unique) $X' \subseteq X$ of size $f + 1$ that $R(X') = \text{false}$;
output false
Otherwise output true

Fig. 15. The function R_{val}

contains B transaction parcels and metadata of size $c\lambda$, then we suppose that a message $(d, G_i(b))$ is of size $3(B + c\lambda)/n + \lambda \log(n)$ and a message $(\text{ECHO}, d, G_i(b))$ is also of size $3(B + c\lambda)/n + \lambda \log(n)$. We analyse latency in the case that all processors are correct.

Claim 6. *Suppose all processors are correct. If $S > 6D$, then setting blocksize B to minimise latency (and after removing some small terms), the protocol of Figure 16 has latency:*

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

Figure 21 gives a plot of the resulting latency for parameter values explained in Section 5.4.

Proof. Note that, at the first time slot at which the leader completes a block, i.e., transfers metadata to its upload buffer, its upload buffer will be empty prior to this transfer. Set t to be any time slot at which the leader completes a block b and suppose (inductively) that its upload buffer was empty prior to this transfer of data. All correct processes will then send an ECHO message corresponding to b by time $t + 3(B + c\lambda)/S + \lambda n \log(n)/S + \Delta$, by which time the leader's upload buffer will also be empty. All correct processes will then send a VOTE message for b by time:

$$t + 6(B + c\lambda)/S + 2n\log(n)\lambda/S + 2\Delta,$$

by which time the upload buffers of all correct processors will be empty. All correct processes will then deliver the block by time:

$$t' = t + 6(B + c\lambda)/S + n\lambda(2\log(n) + 1)/S + 3\Delta, \tag{8}$$

by which time the upload buffers of all correct processors will be empty. Since D transactions arrive at the client processor of the leader at every time slot, for latency to be bounded, we require:

$$B \geq \frac{D(6(B + c\lambda) + \lambda n(2\log(n) + 1) + 3\Delta S)}{S}.$$

This is equivalent to B being greater than:

$$\frac{D(6c\lambda + \lambda n(2\log(n) + 1) + 3\Delta S}{S - 6D} \tag{9}$$

So, set B equal to the expression in (9) above. For this value of B, the leader will then able to complete the next block b' immediately upon all correct processors delivering b at t'. The latency is then $2(t' - t)$, i.e., $(t' - t)$ to deliver the current block and $(t' - t)$ time to deliver the previous block,

There is a designated processor p_ℓ. The following instructions are for p_i.
at time slot 0 **do**:
  Initialize
  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 $\cup$ {*txns*}
  If |block-txns| $\geq B$ **do**: ▷ B as specified in Section 5.1
    Set metadata = metadata for the current block
    Set b = *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

Fig. 16. Reliable Broadcast using erasure coding with a designated sender

during which time transactions for the current block are queued. Approximating $6c + n(2\log(n) + 1)$ as $2n\log(n)$ (which will be reasonable so long as n is not small), this gives a latency of

$$\frac{12B + 4n\log(n)\lambda}{S} + 6\Delta.$$

Substituting in the value for B, and again approximating $6c + n(2\log(n) + 1)$ as $2n\log(n)$, gives the latency as stated in the claim. Note also that increasing B increases latency in the calculations above, so that our choice of B minimises latency. □

5.2 Reliable Broadcast by all processors with erasure coding

We next consider an erasure coding version of Reliable Broadcast in the multi-sender setting. The protocol uses a modified form of R_{val}, shown in Figure 17, and appears in Figure 18.

$R^*_{\text{val}}(X, j', d, x, k)$
If $X < k$, **output** false
If it does not hold that every message in X is from a different processor p_j and is of the form (ECHO, j', d, j, z', x, y') such that $V(j, z', x, y') =$ true;
output false
If it holds for any (equivalently all, assuming hashes are unique) $X' \subseteq X$ of size $f + 1$ that $R(X') =$ false;
output false
Otherwise output true

Fig. 17. The function R^*_{val}

To analyse the latency, we make similar simplifications to those used in Section 5.1. We suppose that *votes*, i.e., messages of the form (VOTE, j, d, x), are of a fixed size λ, and that block metadata is of size M. If a block b contains B transaction parcels and metadata of size M, then we suppose that a message $(d, G_j(b))$ is of size $3(B + M)/n + \lambda\log(n)$ and a message (ECHO, $i, d, G_j(b)$) is also of size $3(B + M)/n + \lambda\log(n)$. We analyse the latency in the case that all processors are synchronized (i.e., all begin the protocol at precisely the same time) and correct.

Claim 7. *Suppose all processors are synchronized and correct. If $S > 3D$, then, setting blocksize B to minimise latency and after removing lower order terms, the protocol of Figure 16 has latency:*

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

Proof. Note that, at the first time slot at which any processor completes a block, i.e. transfers metadata to its upload buffer, its upload buffer will be empty prior to this transfer. Now let t be any time slot at which a processor p_i completes a block b and suppose (inductively) that its upload buffer was empty prior to this transfer of data. All correct processes will then send an ECHO message corresponding to b by time $t + 3(B + M)/S + \lambda n\log(n)/S + \Delta$, by which time the leader's upload buffer will also be empty. All correct processes will then send a VOTE message for b by time:

$$t + 3(B + M)/S + \lambda n\log(n)/S + 3n(B + M)/S + n^2\lambda\log(n)/S + 2\Delta,$$

by which time the upload buffers of all correct processors will be empty. All correct processes will then deliver the block by time:

$$t' = t + 3(B + M)/S + \lambda n\log(n)/S + 3n(B + M)/S + n^2\lambda\log(n)/S + n^2\lambda/S + 3\Delta,$$

by which time the upload buffers of all correct processors will be empty. Recall that each process receives D/n transactions at its client processor at each time slot. For latency to be bounded we therefore require:

$$B \geq \frac{D(3(B + M) + \lambda n\log(n) + 3n(B + M) + n^2\lambda\log(n) + n^2\lambda + 3\Delta S)}{Sn},$$

The following instructions are for p_i.
at time slot 0 **do**:
Initialize
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 $\cup$ {*txns*}
If |block-txns| $\geq B$ **do**: ▷ B as specified in Section 5.2
Set metadata = metadata for the current block
Set b = *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

Fig. 18. Reliable Broadcast by all processors using erasure coding

which means B must be greater than or equal to:

$$\frac{D(3(n+1)M + n(n+1)(\log(n)+1)\lambda + 3\Delta S)}{Sn - 3(n+1)D}. \tag{10}$$

So, set B equal to the expression in (10) above. Setting B to this value, the leader will be able to complete the next block b' immediately upon all correct processors delivering b at t'. The latency

is then $2(t' - t)$. Approximating $n + 1$ as n and $\log(n) + 1$ as $\log(n)$, this gives a latency of

$$\frac{6(B + M)n + 2n^2 \log(n)\lambda}{S} + 6\Delta.$$

Substituting in the value for B, and again approximating $n + 1$ as n and $\log(n) + 1$ as $\log(n)$, gives the latency as stated in the claim. Note that increasing B beyond the value specified in (10) only increases latency. □

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

Recall that our motivation for studying the latency of (multi-shot versions of) primitives such as Best-effort Broadcast, Consistent Broadcast, and Reliable Broadcast, is (at least partly) so that we can later extend this analysis to determine latency for state-of-the-art SMR protocols. For example, in Section 7.3, we will analyse good-case latency for a leader-based protocol in the single-sender setting that uses erasure coding as well as various other optimisations, such as pipelining. As preparation for that analysis, we next consider a modification of the protocol in Figure 16 that also uses such optimisations. This modified protocol is designed to be as close as possible to the SMR protocol considered in Section 7.3 (DispersedSimplex), so that our analysis here will later carry over with minimal effort. At a high level, the principal modifications are as follows:

- At the cost of reducing fault-tolerance by one, we remove the requirement that the leader echoes block fragments and sends votes. This means that after sending the fragments for one block b, the leader can immediately begin sending fragments for the next block.
- Rather than having other processors wait until receiving a full fragment before they begin the process of echoing that fragment, we have them pass on those data parcels as they arrive. Since they cannot verify that the fragment is well-formed until the entire message is received, they must send a separate message upon receipt of the full fragment, which indicates that they have received the fragment and it is well-formed.
- This separate message is formed using a threshold signature scheme [10, 55] and is a *share* of a *certificate* for the message (ECHO, d, x), where x is the corresponding Merkle root and d is the block depth. We let m_i denote p_i's share of a certificate for m, and suppose that a certificate for m can be constructed from any set of $n - f$ distinct shares. A certificate for m of the form $m = (\text{ECHO}, d, x)$ is called an echo-certificate for x (of depth d).
- To make the protocol as close as possible to that considered in Section 7.3, we also use a threshold signature scheme for votes. A certificate for m of the form $m = (\text{VOTE}, d, x)$ is called a vote-certificate for x (of depth d).

Some further details. To implement the above, while ensuring that the protocol remains as similar as possible to that later considered in Section 7.3, we suppose that each processor maintains a *certificate pool* C. Whenever a processor receives at least $n - f$ shares of some echo/vote-certificate (and if C does not already contain the corresponding certificate), it generates a certificate, adds it to C, and sends the certificate to all processors. Similarly, whenever a processor receives a certificate that does not already belong to C, it enumerates it into C and sends that certificate to all processors.

The protocol, which uses the procedures defined in Figure 19, is shown in Figure 20.

Latency. To analyse the latency, we make similar simplifications to those used in Section 5.1. We suppose that *votes*, i.e., messages of the form $(\text{VOTE}, d, x)_i$, are of a fixed size λ. Similarly, echo-messages, i.e., messages of the form $(\text{ECHO}, d, x)_i$, are of a fixed size λ, and vote/echo-certificates

are of size λ.[20] If a block b contains B transaction parcels and metadata of size M, then we suppose that a message $(d, G_j(b))$ is of size $3(B+M)/n + \lambda \log(n)$. We analyse latency in the case that all processors are correct.

$R^{\dagger}_{\text{val}}(X, d, x)$
If $X \neq f+1$, **output** false
If it does not hold that every message in X is from a different processor p_j and is of the form (d, j, z', x, y') such that $V(j, z', x, y') = \text{true}$;
output false
If it holds that
$R(X) = \text{false}$;
output false
Otherwise output true

Update certificates
For any x and d' such that p_i has received at least $n - f$ shares of an echo/vote-certificate for x of depth d', construct the corresponding certificate and enumerate it into C (if not already in C).
For and x and d' such that such that p_i has received an echo/vote-certificate for x of depth d', enumerate that certificate into C (if not already in C).

Send certificates
For each certificate $c \in C$ that has not previously been added to sendbuffer:
Set sendbuffer := sendbuffer$||c$

Gossip from leader
For each data parcel m received from p_ℓ and not yet added to sendbuffer:
sendbuffer := sendbuffer$||m$

Fig. 19. $R^{\dagger}_{\text{val}}$, 'Update certificates', 'Send certificates' and 'Gossip from leader'

To calculate latency when B is well chosen, we first consider how the protocol is intended to function. For some time interval L, and for $t_i := iL$, the idea is that the leader will disseminate fragments for some first block b_1 from t_1 until t_2, and will disseminate fragments for a second block b_2 from t_2 until t_3, and so on. Other processors will begin to receive fragments of b_i at $t_i + \Delta$, will immediately start gossiping those fragments on to other processors, and will receive the full fragments by $t_{i+1} + \Delta$. The hope is that, at this time, they will be ready to start gossiping fragments for b_{i+1}, i.e. doing so will not lead to an increasing backlog at their sendbuffer. However, we must take into account that, during the interval between $t_i + \Delta$ and $t_{i+1} + \Delta$ (and in the steady state), processors other than p_ℓ will also have to send to all processors: (i) an echo-message for some block, (ii) an echo-certificate for some block, (iii) a vote for some block, and (iv) a vote-certificate for some block. If B transactions are included in each block, then clearing these messages from sendbuffer,

[20]We assume that threshold signatures are used to create these certificates. In practice, there is a trade-off between the amount of computation and communication. Threshold signatures such as BLS will incur high computation per signature verification but requires only one signature verification for an n-sized certificate. The signature size is smaller (λ-sized) as well. On the other hand, Ed25519 will incur lower computation per signature verification but an n-sized certificate consists of n signatures, each of which is λ-sized. [Kartik: Nibesh, can we add a citation for this?]

There is a designated processor p_ℓ. The following instructions are for p_i.
at time slot 0 **do**:
Initialize, set $d = 1$, $C = \{\}$, for all d' set $\mathtt{echo}(d') = \mathtt{true}$, $\mathtt{vote}(d') = \mathtt{true}$

at time slot t **if** $p_i = p_\ell$:
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 $\cup$ {*txns*}
If |block-txns| $\geq B$ **do**: ▷ B as specified in Section 5.3
Set metadata = metadata for the current block
Set b = *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 = {}, $d := d + 1$

at every time slot t **if** $p_i \neq p_\ell$ **do**:
Update certificates ▷ As specified in Figure 19
If $\mathtt{echo}(d) = \mathtt{true}$ **and** p_i has received a first (full) message (d, i, z, x, y) from p_ℓ such that $V(i, z, x, y) = \mathtt{true}$ **do**:
Set sendbuffer = sendbuffer||$(\mathrm{ECHO}, d, x)_i$ ▷ Send share of echo-certificate
Set $\mathtt{echo}(d) = \mathtt{false}$, $d := d + 1$
For all $d' \leq d$, **if** $\mathtt{vote}(d') = \mathtt{true}$ **do**:
If $\exists x$ s.t. C contains an echo certificate for x of depth d' **and** p_i has received a set of messages X with $R^\dagger_{\mathrm{val}}(X, d', x) = \mathtt{true}$ **do**: ▷ $R^\dagger_{\mathrm{val}}$ as specified in Figure 19
Set sendbuffer = sendbuffer||$(\mathrm{VOTE}, d', x)_i$
Set $\mathtt{vote}(d') = \mathtt{false}$
For all $d' \leq d$, **if** $\mathtt{vote}(d') = \mathtt{true}$ **do**:
If $\exists x$ s.t. p_i has received $f + 1$ signature shares of a vote-certificate for x of depth d' **do**:
Set sendbuffer = sendbuffer||$(\mathrm{VOTE}, d', x)_i$
Set $\mathtt{vote}(d') = \mathtt{false}$
For all x and $d' \leq d$ such that:
(i) C contains a vote-certificate for x of depth d', and;
(ii) p_i has received a set of messages X with $R^\dagger_{\mathrm{val}}(X, d', x) = \mathrm{true}$, **do**:
Deliver $R(X)$ (if not already delivered)
Send certificates ▷ As specified in Figure 19
Gossip from leader ▷ As specified in Figure 19
Transfer to upload ▷ As specified in Figure 5

Fig. 20. Reliable Broadcast using erasure coding and pipelining with a designated sender

together with all the fragments of b_i that they must gossip, takes time at least:

$$L := \frac{3(B + M) + n\lambda \log(n) + 4n\lambda}{S}. \tag{11}$$

This means we are sending data at a rate of at most:

$$\frac{BS}{3(B + M) + n\lambda \log(n) + 4n\lambda}.$$

For latency to be bounded, we require this rate to be at least D, meaning B is at least:

$$\frac{D(3M + n\lambda(\log(n) + 4))}{S - 3D}. \tag{12}$$

Set B equal to this value. In this case, it takes precisely L time slots for the transactions in each block to arrive at the client processor of p_ℓ, so that p_ℓ can indeed start sending b_i at t_i (for $i \geq 1$). It takes time $(3(B + M) + n\lambda \log(n))/S$ for the leader to clear the fragments for each block from its upload buffer, meaning that each of the other processors is able to clear from their upload buffers the corresponding fragment of each block, as well as the required messages of types (i)-(iv) above, in time at most $(3(B + M) + n\lambda \log(n))/S + 4n\lambda/S = L$.

Finally, to calculate latency, note that the first transactions in b_i to arrive at the client processor of p_ℓ do so at t_{i-1}. Correct processors add echo-messages for b_i to their sendbuffer at $t_{i+1} + \Delta$, which are cleared from their upload buffer[21] by $t_{i+1} + \Delta + n\lambda/S$ and are received by all other processors by $t_{i+1} + n\lambda/S + 2\Delta$. Correct processors add votes for b_i to their sendbuffer at this time, which are cleared from their sendbuffer and received by all correct processors by time $t_{i+1} + 2n\lambda/S + 3\Delta$, whereupon they deliver the block. This gives a total latency of $2L + 2n\lambda/S + 3\Delta$, which is:

$$\left(\frac{6M + 2n\lambda(\log(n) + 4)}{S}\right) \cdot \left(1 + \frac{1}{(S/3D) - 1}\right) + 2n\lambda/S + 3\Delta.$$

This justifies the following claim:

Claim 8. *Suppose all processors are correct. If $S > 3D$, then setting blocksize B to minimise latency, the protocol of Figure 20 has latency:*

$$\left(\frac{6M + 2n\lambda(\log(n) + 4)}{S}\right) \cdot \left(1 + \frac{1}{(S/3D) - 1}\right) + 2n\lambda/S + 3\Delta.$$

5.4 Comparing latency for Reliable Broadcast protocols with erasure coding

To illustrate the trade-offs between the different settings, we suppose again that 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. We set $\lambda = 500$ bits and $\Delta = 0.2$ seconds. The first graph in Figure 21 supposes the incoming transaction rate D is 10000 transactions per second and shows the resulting latency (in seconds) as a function of n. The second graph (on the right) in Figure 21 fixes $n = 200$ and shows the resulting latency as a function of the number of incoming transactions per second divided by 10000.

6 CONSISTENT BROADCAST

In this section, we consider Consistent Broadcast protocols, in which processors send a block to all others, who then send a vote for the block to all others. Correct processors deliver the block upon receiving $n - f$ votes for the block. This suffices to rule out equivocation, but gives weaker guarantees than Reliable Broadcast: when the sender is faulty there is no guarantee that all correct processors receive the block just because some deliver it.

In Section 7.2, we will consider latency for DAG-based protocols that use Consistent Broadcast as the underlying mechanism for block-propagation. In this context, the idea is that the *certificate*

[21] Here we make the mild assumption that the timeslots at which processors clear echo-messages, votes and certificates from their sendbuffer are disjoint. For some input parameters this may not be true, giving an increase in latency bounded by $6n\lambda/S$ (and where this bound could be reduced to $n\lambda/S$ by prioritising the sending of echo-messages first, then votes, then certificates).

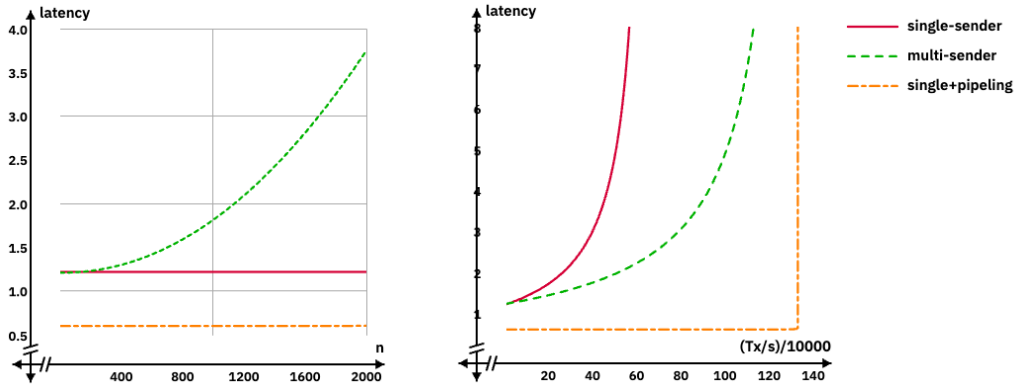

Fig. 21. Latency for RB with erasure coding: parameters are explained in Section 5.4

for a block produced by a successful instance of Consistent Broadcast suffices to ensure both non-equivocation and data-availability, and that the certificate (normally produced using a threshold signature scheme) can be used as a pointer to the block during the process of forming the DAG. This means that any processor receiving a correctly formed block can regard it as valid without having received and verified the blocks that it points to.

6.1 Consistent Broadcast by a single sender

There is a designated processor p_ℓ. The following instructions are for p_i.
at time slot 0 do:
  Initialize
  Set $d = 1$, send = true, vote = true

at time slot t **if** send = true **and** $p_i = p_\ell$:
  Collect txns
  If |block-txns| = B **do**: ▷ B as specified in Section 6.1
    Set metadata = metadata for the current block
    Set sendbuffer = sendbuffer||metadata ▷ d included in metadata
    Set send = false

at every time slot t **do**:
  If vote = true **and** p_i has received a first block b of depth d from p_ℓ **do**:
    Set sendbuffer = sendbuffer||(VOTE, d, $H(b)$)
    Set vote = false
  If p_i has received (VOTE, d, $H(b')$) from $n - f$ distinct processors (for some fixed b')
  and if p_i has also received b' **do**:
    Deliver b'
    Set send = true, vote = true, $d = d + 1$
  Transfer to upload

Fig. 22. Consistent Broadcast by a designated sender

We first consider the single-sender setting: the corresponding protocol is shown in Figure 22. In analysing the latency, we make the following simplifications. We suppose that *votes*, i.e. messages of the form (VOTE, $d, H(b)$), are of a fixed size λ, that metadata is of size M, and we analyse latency in the case that all processors are correct.

Claim 9. *Suppose all processors are correct. If $S > Dn$ and B is set to minimise latency, then latency for the protocol of Figure 22 is:*

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

Proof. Since the time between each block delivery is:

$$C(B+M) + C(\lambda) + 2\Delta,$$

and since D transactions arrive at the client processor of p_ℓ at each timeslot, for latency to be bounded we require:

$$B \geq \frac{D((B+M+\lambda)n + 2\Delta S)}{S}.$$

This means B is greater than or equal to:

$$\frac{D((M+\lambda)n + 2\Delta S)}{S - Dn}. \tag{13}$$

So, set B equal to this value.

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 the instruction "Collect *txns*"). 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 the next 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:

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

Substituting in the value for B in (13) gives the latency as claimed. □

6.2 Consistent Broadcast by all processors

The following instructions are for p_i.
at time slot 0 **do**:
  Initialize
  Set send = true
  $\forall j$, $\texttt{vote}_j$ = true, $d_j = 1$

at time slot t **if** send = true:
  Collect txns
  If |block-txns| = B **do**: ▷ B as specified in Section 6.2
    Set metadata = metadata for the current block
    Set sendbuffer = sendbuffer||metadata ▷ d_i included in metadata
    Set send = false

at every time slot t **do**:
  Let $I = \{j : \texttt{vote}_j = \text{true}$ **and** p_i has received a first block b_j of depth d_j from $p_j\}$
  Set sendbuffer = sendbuffer|| $\oplus_I$ (VOTE, $j, d_j, H(b_j)$)
  $\forall j \in I$ set echo_j = false
  $\forall j$ s.t. p_i has received (VOTE, $j, d_j, H(b'_j)$) from $n - f$ distinct processors (for some fixed b'_j)
  and s.t. p_i 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

Fig. 23. Consistent Broadcast by all processors

Next, we consider the mult-sender setting and the protocol of Figure 23. In analysing latency for this protocol, we make the following simplifications. We suppose that *votes*, i.e. messages of the form (VOTE, $j, d_j, H(b_j)$), are of a fixed size λ, that metadata is of size M, and we analyse latency in the case that all processors are synchronized and correct.

Claim 10. *Suppose all processors are synchronized and correct. If $S > D$ and B is set to minimise latency, then latency for the protocol of Figure 23 is:*

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

Proof. The proof is analagous to that of Claim 9. The only differences are that the second phase, in which votes are sent to all, now takes time $\lambda n^2/S$ (since processors must now send votes for n blocks simultaneously), and that each processor receives D/n transactions at their client processor per time slot (rather than the leader receiving D per time slot). The value of B that minimizes latency is now:

$$\frac{M + \lambda n + 2\Delta S/n}{(S/D) - 1}. \tag{14}$$

□

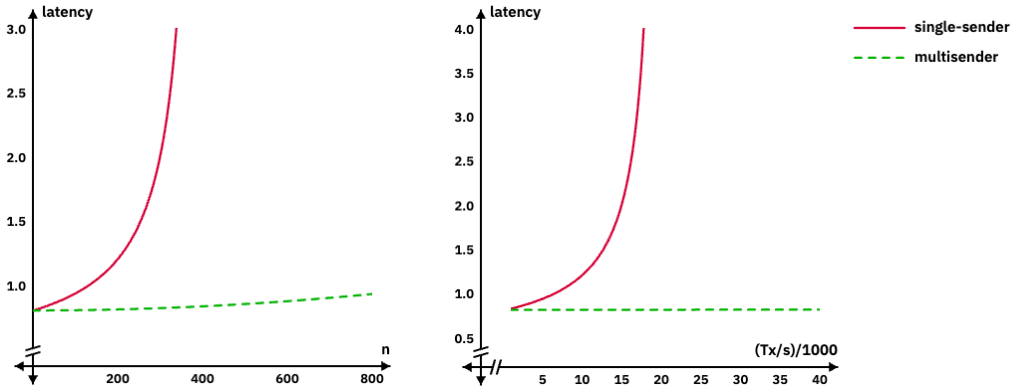

Fig. 24. Latency for Consistent Broadcast protocols: parameters are explained in Section 6.3

A comment on pipelining. We do not consider a pipelined version of the protocol for Figure 23 in this paper. In part, this is because SMR protocols for the multi-sender setting generally (with exceptions) construct a DAG which is built in layers, so that blocks of layer $d + 1$ cannot be constructed prior to the point at which blocks for layer d are received. However, even in this case, there are contexts in which processors may begin to distribute some of the data for a block before the required metadata is known (see Section 3.4). We leave an analysis of such considerations for future work. [Kartik: Just for my understanding: So a protocol like Mysticeti would be one that applies/can apply such a pipelining optimization, correct? the primary concern is that, in case of equivocation, different parties may point to different equivocating blocks.]

6.3 Comparing latency for Consistent Broadcast protocols

To illustrate the trade-offs between the different settings, we suppose again that 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. We set $\lambda = 500$ bits and $\Delta = 0.2$ seconds. The first graph in Figure 24 supposes the incoming transaction rate D is 10000 transactions per second and shows the resulting latency (in seconds) as a function of n. The second graph (on the right) in Figure 24 fixes $n = 200$ and shows the resulting latency as a function of the number of incoming transactions per second divided by 1000.

7 LATENCY FOR SMR PROTOCOLS

7.1 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-senders settings is complicated. To start with (as noted in Section 3.5), the single-sender and multi-sender 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 [26]. 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 [30]. 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 latency 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 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.[22]

Since it is difficult to make an entirely apples-to-apples comparison, we focus on comparing the best (i.e. lowest latency) protocols for each of 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.

7.2 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 (RBC-delivery) or Consistent Broadcast delivery (CBC-delivery), since DAG-based protocols often use Reliable Broadcast or Consistent Broadcast as primitives.

In fact, analyzing 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 finalize each block.

The example of Sailfish. To make things concrete, we focus on the example of Sailfish [59], 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.

As with many other DAG-based protocols, one complexity is that the latency incurred by a given transaction may depend on the position of the corresponding block in the DAG. If a block b from layer d is 'pointed to' by a leader block b' for layer $d + 1$, then (assuming all leaders are correct and that message delays are always Δ), all transactions in b will be finalised by all correct processors after they RBC-deliver b' and receive the first messages from $2f + 1$ processors of layer $d + 2$.[23] On the other hand, latency for transactions that are included in blocks not pointed to by b' may be greater. To deal with this complexity, we can suppose that, while each block points to at least $n - f$ blocks from the previous layer, blocks also have pointers to the most recently observed block by

[22]On the other hand, a very recent paper [51] describes a new DAG-based protocol called Starfish, which does emply a form of pipelining in the context of erasure codes.

[23]While Sailfish explicitly uses Reliable Broadcast (without erasure coding) for each block, we may also consider a version using Consistent Broadcast. In this case, all transactions in b will be finalised by all correct processors when they CBC-deliver b' and then receive the first messages from the $2f + 1$ processors of the next layer.

each processor (meaning n pointers in all). Under our assumption of synchrony and correct leaders, this means that each block produced by a correct processor for layer d will either be pointed to by a leader block for layer $d+1$ or by a leader block for layer $d+2$ (or both).

We also note that there are optimizations which can sometimes be employed to reduce latency in the optimistic case: for example, one may be able to RBC-deliver after two rounds of communication in certain optimistic cases [5, 60]. For the sake of simplicity, we do not consider such optimizations 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 6.2, the maximum delay between receipt of a transaction in b and the point at which all processors CBC-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 CBC-deliver blocks in layers $d+1$ and $d+2$ (recall, we assume that b will be pointed to by 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:

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

.

Since all transactions in b will be finalized by all correct processors when they CBC-deliver a leader block for layer $d+2$ (or, perhaps, $d+1$) and then receive a first message from the leader of the next layer, this gives total latency:[Kartik: Nibesh, the additive term here should be $C(M)$ only, correct? we can interpret a block as receiving vote first and then block content?][Nibesh: we send the entire vertex. Though, the independent metadata can be sent first which is sufficient for committing.]

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

Substituting in the value for B given by (14), this is:

$$\left(\frac{4(Mn+\lambda n^2)}{S}+8\Delta\right)\left(1+\frac{1}{(S/D)-1}\right)+\frac{Mn}{S}+\Delta. \tag{15}$$

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

$$\left(\frac{9\lambda n^2}{S}+9\Delta\right)\left(1+\frac{8}{9((S/D)-1)}\right). \tag{16}$$

We note that the expressions in (15) and (16) correspond to the worst case that b is not pointed to by the leader block for layer $d+1$. If b is pointed to by the leader block for layer $d+1$ and $M=n\lambda$ then the latency is:

$$\left(\frac{7\lambda n^2}{S}+7\Delta\right)\left(1+\frac{6}{7((S/D)-1)}\right). \tag{17}$$

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. The latency in this case 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 (7) and removing small terms, this is:

$$=\left(\frac{4(M+\lambda)n^2}{S}+12\Delta\right)\left(1+\frac{5}{4((S/Dn)-1)}\right)+Mn/S+\Delta.$$

Sailfish using Reliable Broadcast with erasure coding as the underlying primitive. Once again, the calculations here are entirely analogous to those for Consistent Broadcast, but now we use instead the analysis of Section 5.2. The latency in this case is:

$$\left(\frac{12nM+4n^2\log(n)\lambda}{S}+12\Delta\right)\left(1+\frac{1}{(S/3D)-1}\right)+C(B+M)+\Delta.$$

Substituting in the value of B from (10), this is:

$$\left(\frac{12nM+4n^2\log(n)\lambda}{S}+12\Delta\right)\left(1+\frac{13}{12((S/3D)-1)}\right)+Mn/S+\Delta.$$

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

As noted in Section 7.1, 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 take the approach of comparing the best (i.e. lowest latency) protocols for each of 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 [56].

[Nibesh: For leader-based protocols, the external client has to send the transactions to all the nodes as they do not know who the leader is. In DAG, they can only to one or some nodes to ensure their transactions are included in the block.] [Andy: I've added some comments along these lines to Section 7.1. Let me know if you think we need more.] [Kartik: I interpreted Andy's text in section 7.1 a little differently from answering the question Nibesh raised, and his text makes sense to me in that context. Re: Nibesh's comment: protocols need to deal with worst case, so we would have to send these transactions to at least $t+1$ parties anyway? so we are only left with a factor of 3?][Nibesh: In the common case, we only need to send one party in DAG-based protocol. For leader-based, we need to send it to everyone as they do not know the leader.]

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, analyzing 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 if votes are of size λ and metadata is of size M:

$$\left(\frac{6M + 2n\lambda(\log(n) + 4)}{S}\right) \cdot \left(1 + \frac{1}{(S/3D) - 1}\right) + 2n\lambda/S + 3\Delta.$$

7.4 Latency analysis for Tendermint

While our principal focus in Section 7 is to compare the lowest latency protocols for each of the single-sender and multi-sender settings, it is also interesting to compare these with 'standard' single-sender protocols, which do not use techniques such as erasure coding. In this section, we analyze latency for Tendermint.

Since there are multiple versions of the protocol, and for concreteness, we consider the version of Tendermint described in Figure 25. It is not our aim here to *explain* the Tendermint protocol, and so it would be a distraction to make all of the instructions entirely explicit (e.g., how a processor determines whether a block proposal received from the leader is valid, or what it means for a processor to 'set their lock'): the figure is intended only to specify when messages are sent, and our analysis below will also specify message sizes. The version of the protocol described in Figure 25 does not make use of any optimizations such as pipelining.

To calculate latency in the single-sender setting, we consider the case that there is a single processor p_ℓ which is the leader of every view, i.e., $p_\ell = \mathtt{lead}(v)$ for all v. Since the protocol does not make any optimizations that take advantage of a stable leader this calculation will also reflect latency in the case of rotating leaders. We suppose 'votes' (stage 1 or 2) are of size λ. For the sake of simplicity, we also suppose that 'locks' are of size λ (the use of a threshold signature scheme allowing for locks of approximately the size of a signature).

If p_ℓ enters view v at t, then it will complete a block for view v (i.e., place the corresponding metadata on sendbuffer) by $t + C(\lambda) + 2\Delta$ [Kartik: shouldn't this be $t + C(M) + 2\Delta$] [Andy: NEWVIEW message is of size λ.] at the earliest. If the block includes B transaction parcels and metadata of M parcels, all processors will then send stage 1 votes by $t + C(\lambda) + C(B + M) + 3\Delta$ and stage 2 votes by $t + 2C(\lambda) + C(B + M) + 4\Delta$ at the earliest. Processors therefore receive a 2-QC for the block and enter view $v + 1$ by $t + 3C(\lambda) + C(B + M) + 5\Delta$ at the earliest. This means data is sent at a rate of at most:

$$\frac{B}{3C(\lambda) + C(B + M) + 5\Delta}.$$

For latency to be bounded, this must be greater than or equal to D, so:

$$B \geq D\left(\frac{(3\lambda + B + M)n}{S} + 5\Delta\right).$$

This means B is at least:

$$\frac{D((3\lambda + M)n + 5\Delta S)}{S - Dn}. \tag{18}$$

So, set B equal to this value: as in previous sections, one can then easily verify that the bound $3C(\lambda) + C(B + M) + 5\Delta$ on the length of a view calculated above is tight (we suppose TIMEOUT is set to some greater value), except for the first view. Latency must be measured from the time at which a transaction first arrives at the client processor of p_ℓ. For a transaction included in the block for view v, this could be any time after the leader 'completes' the block for view $v - 1$ by adding the corresponding metadata to sendbuffer (the first view is a special case, but accords the bounds

The following instructions are for p_i.
at time slot 0 do:
  Initialize
  Set $v = 1$, $\texttt{starttime}(1) = 0$, send = true, vote1 = true, vote2 = true

at time slot t **if** send = true **and** $p_i = \texttt{lead}(v)$ **and** $t - \texttt{starttime}(v) = C(\lambda) + 2\Delta$ **do**:
  ▷ Leader waits for lock info; λ as specified in Section 7.4
  Collect txns ▷ As specified in Figure 5
  If |block-txns| $\geq B$ **do**: ▷ B as specified in Section 7.4
    Set metadata = metadata for the current block
    Set sendbuffer = sendbuffer||metadata
    Set send = false

at every time slot t **do**:
  If vote1 = true **and** p_i has received a valid block proposal for view v from $\texttt{lead}(v)$ **do**:
    Set sendbuffer = sendbuffer||$(1\text{-VOTE}, H(b))_i$ ▷ Stage 1 vote on hash of b signed by p_i
    Set vote1 = false
  If vote2 = true **and** p_i has received a 1-QC for a view v block b **do**:
    Set sendbuffer = sendbuffer||$(2\text{-VOTE}, H(b))_i$ ▷ Stage 2 vote on hash of b signed by p_i
    Set vote2 = false, Set lock
    Set sendbuffer = sendbuffer||lock
  If p_i has received a 2-QC for a view v block b **do**:
    Deliver b and all ancestor blocks
    Form threshold certificate c for view $v + 1$, set sendbuffer = sendbuffer||c
    Set send = true, vote1 = true, vote2 = true, $\texttt{starttime}(v + 1) = t$, $v = v + 1$
  If $t - \texttt{starttime}(v) \geq$ TIMEOUT **do**:
    Set sendbuffer = sendbuffer||$(\text{COMPLAIN}, v)_i$ ▷ 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 = true, vote1 = true, vote2 = true, $\texttt{starttime}(v + 1) = t$, $v = v + 1$
  Transfer to upload ▷ As specified in Figure 5

Fig. 25. Tendermint

below). Latency is therefore:

$$\frac{(5\lambda + 2B + 2M)n}{S} + 8\Delta.$$

Subtituting in the value for B in (18), we concluded that latency is:

$$\frac{(6\lambda n + 2Mn)/S + 10\Delta}{(S/Dn) - 1} + \frac{(5\lambda + 2M)n}{S} + 8\Delta. \tag{19}$$

7.5 Latency analysis for HotStuff

The principal aim of HotStuff [44, 67] is to obtain a protocol with linear communication complexity within each view. While HotStuff incurs low communication complexity, all the messages are relayed through the leader and hence it becomes the bottleneck. This has the possibility of impacting the 'real-world' latency. In this section, we use our model to analyze latency for HotStuff.

For concreteness, we consider the version of HotStuff described in Figure 26. As for Tendermint, it is not our aim here to explain the protocol: the figure is intended only to specify when messages are sent, and our analysis below will also specify message sizes. The version of the protocol described in Figure 26 is intended to reflect that in the original paper [67], and does not make use of any optimizations such as pipelining.

As for Tendermint, to calculate latency in the single-sender setting, we consider the case that there is a single processor p_ℓ which is the leader of every view, i.e., $p_\ell = \mathtt{lead}(v)$ for all v. Since the protocol does not make any optimizations that take advantage of a stable leader, this calculation will also reflect latency in the case of rotating leaders. We suppose 'NEWVIEW' messages, 'votes' and QCs (stage 1, 2, or 3) are of size λ. We analyze latency in the case that all processors are correct and are synchronized.

If p_ℓ enters view v at t, then it will complete a block for view v (i.e., place the corresponding metadata on sendbuffer) by $t + C(\lambda) + \Delta$ [Kartik: shouldn't this be $t + C(M) + \Delta$] [Andy: NEWVIEW message is of size λ.] at the earliest. If the block includes B transaction parcels and metadata of M parcels, all processors will then send stage 1 votes by $t + C(\lambda) + C(B + M) + 2\Delta$, stage 2 votes by $t + 3C(\lambda) + C(B + M) + 4\Delta$, and stage 3 votes by $t + 5C(\lambda) + C(B + M) + 6\Delta$ at the earliest. [Kartik: We note that for each vote message, only the leader incurs $C(\lambda)$ time to receive and send votes; other processors incur strictly lesser time since they are sending/receiving λ-sized messages from only one processor (leader).] All processors then receive a stage 3 QC (and enter view $v + 1$) by $t + 7C(\lambda) + C(B + M) + 8\Delta$ at the earliest. This means data is sent at a rate of at most:

$$\frac{B}{7C(\lambda) + C(B + M) + 8\Delta}.$$

For latency to be bounded, this must be greater than or equal to D, so:

$$B \geq D\left(\frac{(7\lambda + B + M)n}{S} + 8\Delta\right).$$

This means B is at least:

$$\frac{D((7\lambda + M)n + 8\Delta S)}{S - Dn}. \tag{20}$$

So, set B equal to this value: as in previous sections, one can then easily verify that the bound $7C(\lambda) + C(B + M) + 8\Delta$ on the length of a view calculated above is tight (we suppose TIMEOUT is set to some greater value), except for the first view. Latency must be measured from the time at which a transaction first arrives at the client processor of p_ℓ. For a transaction included in the block for view v, this could be any time after the leader 'completes' the block for view $v - 1$ by adding the corresponding metadata to sendbuffer (the first view is a special case, but accords the bounds below). Latency is therefore:

$$\frac{(13\lambda + 2B + 2M)n}{S} + 15\Delta.$$

Subtituting in the value for B in (20), we concluded that latency is:

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

Comparing with equation (19), we see that latency for HotStuff is strictly greater than latency for Tendermint, for all parameter values.

The following instructions are for p_i.
at time slot 0 do:
Initialize
Set $v = 1$, $\texttt{starttime}(1) = 0$, $\texttt{send} = \texttt{true}$, $\texttt{vote1} = \texttt{true}$, $\texttt{vote2} = \texttt{true}$, $\texttt{vote3} = \texttt{true}$
Set $\texttt{send1QC} = \texttt{true}$, $\texttt{send2QC} = \texttt{true}$, $\texttt{send3QC} = \texttt{true}$

at every time slot t **do**:

Add a NEWVIEW message for view v to upload buffer, with all parcels addressed to $\texttt{lead}(v)$, if not already done

If $p_i = \texttt{lead}(v)$ **and** $\texttt{send} = \texttt{true}$ **and** p_i has received $n - f$ NEWVIEW messages for view v **do**:
Collect txns ▷ As specified in Figure 5
If |block-txns| $\geq B$ **do**: ▷ B as specified in Section 7.5
Set metadata = metadata for the current block
Set sendbuffer = sendbuffer||metadata
Set send = false

If $\texttt{vote1} = \texttt{true}$ **and** p_i has received a valid block proposal b for view v from $\texttt{lead}(v)$ **do**:
Set $\texttt{vote1} = \texttt{false}$
Add $(\text{1-VOTE}, H(b))_i$ 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** $\texttt{send1QC} = \texttt{true}$ **and** p_i has received $n - f$ stage 1 votes for b corresponding to view v **do**:
Set $\texttt{send1QC} = \texttt{false}$
Form Q_1, a threshold 1-QC for b
Set sendbuffer = sendbuffer$||Q_1$

If $\texttt{vote2} = \texttt{true}$ **and** p_i has received a 1-QC for a view v block b from $\texttt{lead}(v)$ **do**:
Set $\texttt{vote2} = \texttt{false}$
Add $(\text{2-VOTE}, H(b))_i$ to upload buffer, with all parcels addressed to $\texttt{lead}(v)$
▷ Stage 2 vote on hash of b signed by p_i

If $p_i = \texttt{lead}(v)$ **and** $\texttt{send2QC} = \texttt{true}$ **and** p_i has received $n - f$ stage 2 votes for b corresponding to view v **do**:
Set $\texttt{send2QC} = \texttt{false}$
Form Q_2, a threshold 2-QC for b
Set sendbuffer = sendbuffer$||Q_2$

If $\texttt{vote3} = \texttt{true}$ **and** p_i has received Q_2, a 2-QC for a view v block b, from $\texttt{lead}(v)$ **do**:
Set $\texttt{vote3} = \texttt{false}$
Set $\texttt{lock} = Q_2$
Add $(\text{3-VOTE}, H(b))_i$ to upload buffer, with all parcels addressed to $\texttt{lead}(v)$
▷ Stage 3 vote on hash of b signed by p_i

If $p_i = \texttt{lead}(v)$ **and** $\texttt{send3QC} = \texttt{true}$ **and** p_i has received $n - f$ stage 3 votes for b corresponding to view v **do**:
Set $\texttt{send3QC} = \texttt{false}$
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 $\texttt{send} = \texttt{true}$, $\texttt{vote1} = \texttt{true}$, $\texttt{vote2} = \texttt{true}$, $\texttt{starttime}(v+1) = t$, $v = v + 1$
Set $\texttt{send1QC} = \texttt{true}$, $\texttt{send2QC} = \texttt{true}$, $\texttt{send3QC} = \texttt{true}$

If $t - \texttt{starttime}(v) \geq$ TIMEOUT **do**:
Set sendbuffer = sendbuffer$||(\text{COMPLAIN}, v)_i$ ▷ 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 $\texttt{send} = \texttt{true}$, $\texttt{vote1} = \texttt{true}$, $\texttt{vote2} = \texttt{true}$, $\texttt{starttime}(v+1) = t$, $v = v + 1$
Set $\texttt{send1QC} = \texttt{true}$, $\texttt{send2QC} = \texttt{true}$, $\texttt{send3QC} = \texttt{true}$

Transfer to upload ▷ As specified in Figure 5

Fig. 26. HotStuff

7.6 Comparing latency for SMR protocols

For the reasons explained in Section 7.1 (since it is difficult to make an entirely apples-to-apples comparison), we first take the approach of focussing on the best (i.e. lowest latency) protocols for each of the single-sender and multi-sender settings. We consider Sailfish with Consistent Broadcast as the underlying primitive for block propagation as a representative of DAG-based protocols: as noted in Section 7.3 our analysis is easily adapted to consider other DAG-based protocols. As a representative of an SMR protocol in the single-sender setting, we consider DispersedSimplex with a stable leader. Later, we will also compare latency with Tendermint and HotStuff.

We suppose again that 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. We set $\lambda = 500$ bits and $\Delta = 0.2$ seconds. The first graph in Figure 27 supposes the incoming transaction rate D is 10^6 transactions per second and shows the resulting latency (in seconds) as a function of n. The second graph (on the right) in Figure 27 fixes $n = 400$ and shows the resulting latency as a function of the number of incoming transactions per second divided by 10^5. One can see that DispersedSimplex has significantly lower latency until one reaches its latency bottleneck, but that the factor of 3 caused by the use of erasure coding means that the latency bottleneck for DispersedSimplex is a third of that for Sailfish (when Sailfish uses Consistent Broadcast as the underlying primitive for block propagation). We note that Locher and Shoup [57] have shown that the factor of 3 can be reduced to 1.5, but doing so requires further rounds of communication, which may increase latency and reduce the efficacy of pipelining. We leave it to future work to determine whether this approach can be used to further lower latency in the single-sender setting.

[Andy: Add Sailfish + erasure coding to the figure. Also explain why Dispersed Simplex is quicker for lower throughputs. Explain why our figures for Sailfish are higher than you might think, and that they will be lower for other cases. Add a comparison between Sailfish, DispersedSimplex and Tendermint (with another figure).]

Comparison with Tendermint and HotStuff. Since most single-sender (or multi-sender) protocols do not use erasure coding, it is also interesting to compare latencies for standard protocols like Tendermint, HotStuff, and Sailfish. As noted in Section 7.5, latency for HotStuff is strictly greater than that for Tendermint (over all parameter values).

To give a concrete example, suppose that processors can upload/download at a rate of 1 Gbps and that transactions are 2500 bits. In the single-sender setting, suppose M is 1000 bits. Set $\lambda = 500$ bits and $\Delta = 0.2$ seconds. The first graph in Figure 28 supposes the incoming transaction rate D is 2000 transactions per second and shows the resulting latency (in seconds) as a function of n. The second graph (on the right) in Figure 28 fixes $n = 30$ and shows the resulting latency as a function of the number of incoming transactions per second divided by 10^3.

8 RELATED WORK

Latency of consensus protocols. The 'latency' of a consensus protocol depends on several factors such as the underlying networking model assumed, the behavior of Byzantine parties, the amount of data transmitted, and the actual network delay. Due to this, the literature considers several latency metrics that are expressed in different ways under different assumptions.

In the existing literature, latency is often measured in terms of the number of 'rounds' of communication required for asynchronous and partially synchronous protocols (and sometimes

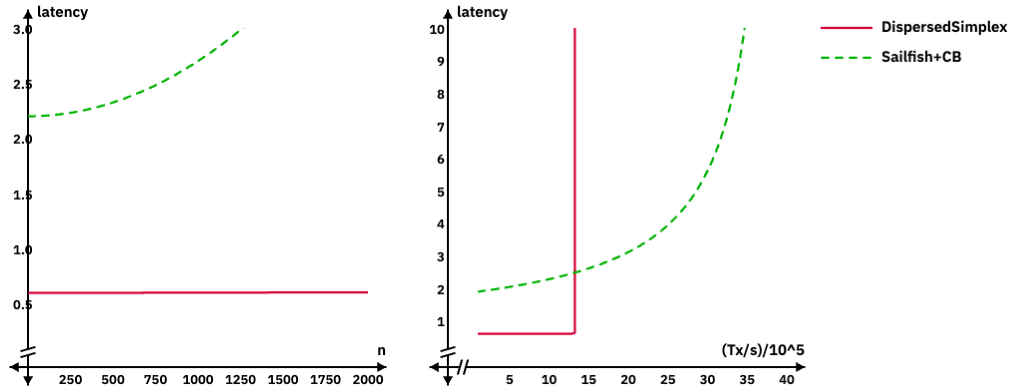

Fig. 27. Latency for SMR protocols: parameters are explained in Section 7.6

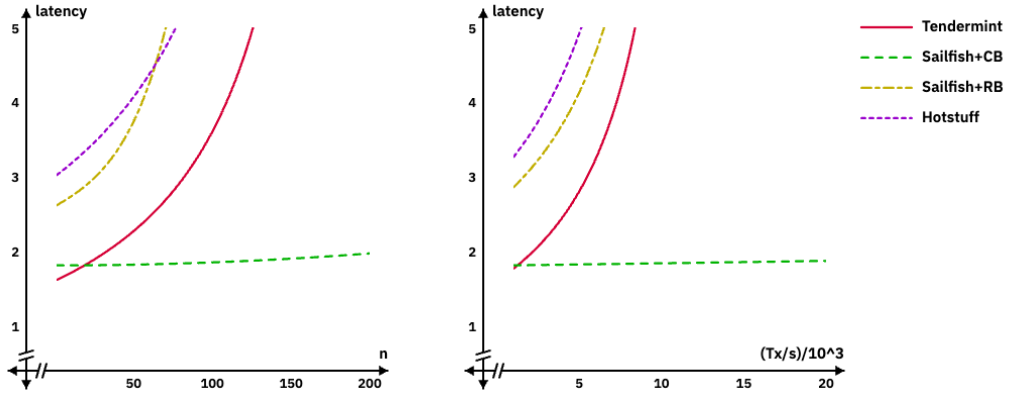

Fig. 28. Comparing latency for DAG-based and standard SMR protocols: parameters are explained in Section 7.6

synchronous protocols too), and in terms of the pessimistic network delay Δ for synchronous protocols [5]. This is because, under asynchrony, the network delay can be arbitrary, and so one cannot provide a bound on latency in terms of Δ.

Many theoretical studies have focused on the worst-case latency (given arbitrary behaviour by a bounded adversary). For Byzantine broadcast, the worst-case number of rounds required is $f + 1$ to tolerate up to f Byzantine faults [28]. Intuitively, in many protocols, this is because "leaders" may behave maliciously, e.g., not send any messages, and so need to be changed until one has an honest leader. An alternative latency measure is the 'expected (round) latency'. At a high level, protocols may select leaders uniformly at random, and so, if the correct parties are at least a constant fraction of the total number of parties, the expected latency is $O(1)$ rounds [2].

In addition, many prior works do not take into consideration the amount of data sent when describing the latency of the protocol. For instance, we say Byzantine reliable broadcast and partial synchrony broadcast have a latency of 2 rounds and 3 rounds respectively [5]; for larger data sizes, the rounds may be longer in duration but the metric does not account for it. Even when expressed in Δ, they assume that the same Δ parameter is sufficient to send any amount of data. This is

reasonable in scenarios where small fixed size inputs are agreed upon. An exception to this is [8] in which Bagaria et al. describe a model (further developed in [39, 49, 53]) that is highly tailored to the analysis of longest-chain protocols. A basic similarity with our approach is that they consider a notion of 'network capacity' C and suppose that sending a block containing B transactions should take time $\Delta := B/C+D$ for some fixed delay D. However, their model then divides the execution into discrete rounds of length Δ (somewhat akin to the analysis in [33]). Any block sent during round r arrives at the beginning of round $r+1$. To limit the number of blocks that can be sent in each round, they consider an 'environment'. All messages are sent to and delivered by the environment, and the environment is allowed to process at most $C\Delta$ transactions in each round. The "capacity" therefore acts as a network-wide limit on the ability to pass messages/transactions from each round to the next and the model does not allow for an analysis of the extent to which specific communication channels between pairs of processors may act as a bottleneck. Another crucial distinction is that, to simplify calculations, they carry out a sort of 'mean-field' approximation in the limit of large n (and assume that the network is somehow able to deal with delivering transactions to infinitely many processors in finite time). By contrast, the impact of n on latency is a focus of our analysis. The approach described by our model is therefore much more granular, as is required (for example) for any detailed analysis comparing latencies for singe-sender and multi-sender protocols.

A related aspect is the responsiveness of the protocol, where the latency can be expressed as a function of the actual network delay δ instead of the pessimistic bound Δ [50].

In our analysis, while the underlying protocols tolerate Byzantine faults and are secure under asynchrony, we only consider optimistic scenarios where there are no faulty parties and the network is entirely synchronous. Moreover, our work expresses the latency as a function of the amount of data that needs to be propagated. However, even if the protocol is responsive, the analysis assumes that the actual network delay δ is the same as the pessimistic bound Δ. In that sense, our latency measure is an upper bound on the actual latency.

Communication complexity. Communication complexity refers to the number of bits transmitted by all correct parties. The Dolev-Reischuk bound states that the number of messages sent by correct parties (and so also the communication complexity) is $\Omega(f^2)$ in the worst case for Byzantine Broadcast/Agreement [23] (where f is the number of Byzantine parties). Intuitively, this is because every Byzantine party can request some message from every correct party. Communication complexity has been thought to be directly related to the throughput of the protocol, since sending more protocol metadata implies sending fewer transaction data parcels for the same total amount of data sent. Thus, in BFT protocols, we have seen works that focus on improving the communication complexity from exponential communication [41] to $O(n^4)$ [24] to $O(n^3)$ [13, 15, 17] to $O(n^2)$ [18, 42, 44, 67] to sub-quadratic [1, 35, 40].

Extension protocols, data availability oracles, pipelining. The line of work on extension protocols explicitly considers consensus protocols for ℓ-bit long messages where ℓ is a parameter. For a large ℓ, they obtain a communication complexity of $O(n\ell + \mathrm{poly}(n)\kappa)$ where κ is a security parameter [9, 32, 34, 47]. These protocols rely on erasure coding techniques to improve communication complexity, and thus improve throughput. Data availability oracles such as EigenDA and Tiramisu also use related ideas to separate data dissemination from consensus to improve the throughput of the protocol [19, 25, 27, 48, 61, 64, 66]. Finally, several state machine replication protocols have considered "pipelining", i.e., transmitting a block before committing the previous block it extends, as a means to improve throughput and latency [3, 15, 58, 67].

DAG-based protocols. It has often been observed that, in practice, low communication complexity does not necessarily imply high throughput. Narwhal and Tusk [22] showed that building on a

DAG-based dissemination layer, where each party is responsible for transmitting transactions, allows for a much higher throughput. In fact, Narwhal is a system-level optimization where every party can utilize several *worker nodes* to transmit data and scale the throughput (albeit at the cost of higher underlying network bandwidth). The use of several worker machines implies using several CPUs and larger bandwidth. While larger bandwidth can be incorporated in our model, the use of several CPUs is not captured by our model. [Kartik: Andy check.] Due to the transmission of data and protocol messages by several parties at once, these works typically incur higher communication complexity. In addition to being a dissemination layer, several works rely on the underlying DAG structure to reach consensus [6, 7, 20, 21, 31, 37, 38, 59, 62, 63]. Early DAG-based protocols had high round-latency and so improving the round-latency to match that of non-DAG-based counterparts has been an active area of research.

Quorum-based vs. Nakamoto-style protocols. All of our analysis in this work has been designed with a focus on "quorum-based" protocols. However, there exist several "Nakamoto-style" works that have focused on improving latency and communication complexity [8, 43, 65, 68]. The methods developed in this work can also be applied to the analysis of Nakamoto-style protocols.

9 CONCLUSION AND FUTURE WORK

Designing SMR protocols with low latency and high throughput has been a highly active area of research. This has been particularly true over the last decade, since the launch of Bitcoin [46] initiated worldwide interest in 'blockchains'. However, research in this area has been bottlenecked by existing analytical tools, which measure latency via round-complexity, relying on communication complexity to take some account of message sizes. The stark distinction between theory and practice became obvious to the community due to the release and adoption of DAG-based protocols. For the first time, these protocols challenged the narrative that low communication complexity implies the ability to deal with high throughputs. DAG-based protocols were seen to produce low latency in practice, even if theoretical metrics do not reflect this fact. In achieving these goals, the community introduced optimizations such as Narwhal [22].

In this paper, we take a first-principles approach to understanding and analyzing network-level performance. The key contribution of the paper is a new approach to analyzing latency and throughput — our latency measure considers the network delay, the need to send data parcels, and the complex interaction between different processors to actually deliver these messages. This approach combines communication complexity and traditional round-latency in a single metric, while providing bounds on what throughput can be obtained by a protocol.

One key result of our analysis is a comparison between latency for leader-based and DAG-based protocols. While today it is colloquially understood that DAG-based protocols outperform leader-based protocols in latency and throughput, our analysis suggests that there are trade-offs between the two approaches. At a high-level, the key to obtaining low latency and high throughputs is (as far as possible) to use the full bandwidth of each processor at all times, so that no single party is a bottleneck. With leader-based protocols, one can use erasure codes to achieve this. With this approach, leader-based protocols, that rely on a fixed leader who pipelines data dissemination, have the ability to achieve low latency until a 'latency bottleneck' is reached for sufficiently high throughputs. The advantage of DAG-based protocols is that the latency bottleneck occurs at throughput which is higher by a constant factor. This is because erasure coding incurs sending a constant factor more data. There are other works, such as Autobahn [36] and BBCA-Chain [45], that use ideas from both of these approaches. Analyzing and comparing these works using our approach is an avenue for future work.

Another key result is a latency comparison between HotStuff and Tendermint. HotStuff reduced communication complexity to linear communication complexity per-view so that the protocol can be scaled to a large number of parties. This was achieved by sending data through the leader at the expense of increasing the number of rounds. On the other hand, Tendermint incurred a quadratic communication complexity. Our latency analysis shows that HotStuff does not outperform Tendermint in terms of latency or the latency bottleneck for any value of n. The leader indeed acts as a bottleneck, and the approach does not improve either latency or the maximum throughput obtained. We do, however, note that protocols may still potentially use HotStuff as a sub-protocol to achieve improved latency and throughput. For instance, the VABA protocol [4] uses n parallel instances of HotStuff, where every party acts as a leader for different instances of HotStuff and thus performs the same amount of work. It may be the case that the use of Hotstuff in this context allows for a higher latency bottleneck (high throughput).

While our analysis is a significant step forward, it makes several simplifying assumptions, leaving a number of avenues for future work. One large simplification is that our analysis only deals with the best case scenario: latency analysis is for the case that everyone is correct and the network is synchronous. Of course, processor or network failure can significantly impact latency, but the precise impact may not be obvious. For instance, if a leader fails in a single-sender protocol, it is clear that latency increases, but if we also expect the next leader to transmit double the amount of transaction data, this will have further knock-on effects impacting latency. In a DAG-based protocol, in contrast, data is still disseminated by honest processors when the leader is faulty. Similarly, while Δ may be an upper bound on network delays, shorter message delays will generally also be possible. For instance, the geographic distribution of processors will not generally be uniform and, even if it is, in practice there may be variance in message delays over time. As an example, compare two protocols requiring two different quorum sizes. In the presence of stragglers, a protocol that requires a 51% quorum may be faster than one that requires a 95% quorum. Also, the metric we compare is the worst case latency under the best case scenario. Considering other metrics such as average case or expected case or best case is an avenue for future work.

As described earlier, our analysis also considers a simplified network model:

(i) We ignore the underlying complications of protocols like TCP: ensuring guaranteed message delivery using retries, the underlying buffer sizes, the underlying congestion control algorithms used;
(ii) We assume that every party is directly connected to every other party — extending this to peer-to-peer networks is an interesting avenue for future work;
(iii) We assume that processors are capable of sending arbitrarily small-sized parcels, although there may be limitations stemming from the underlying network layer on sending at least a few tens or hundreds of bytes.

Significantly, our analysis also assumes that computation is free – in practice, computing operations for erasure codes and digital signatures may incur non-trivial costs, affecting the system's throughput and latency. The use of CPU cycles to perform these computations may also have downstream effects in the larger blockchain system, which may need the CPU for other tasks, e.g., execution of transactions.

Acknowledgment. This work is partially funded by an academic gift grant by Stellar Foundation.

REFERENCES

[1] I. Abraham, T. H. Chan, D. Dolev, K. Nayak, R. Pass, L. Ren, and E. Shi. Communication complexity of byzantine agreement, revisited. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 317–326,

2019.

[2] I. Abraham, S. Devadas, D. Dolev, K. Nayak, and L. Ren. Synchronous byzantine agreement with expected o (1) rounds, expected communication, and optimal resilience. In *International Conference on Financial Cryptography and Data Security*, pages 320–334. Springer, 2019.

[3] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and M. Yin. Sync hotstuff: Simple and practical synchronous state machine replication. In *Proceedings - 2020 IEEE Symposium on Security and Privacy, SP 2020*, Proceedings - IEEE Symposium on Security and Privacy, pages 106–118, United States, May 2020. Institute of Electrical and Electronics Engineers Inc. Publisher Copyright: © 2020 IEEE.; 41st IEEE Symposium on Security and Privacy, SP 2020 ; Conference date: 18-05-2020 Through 21-05-2020.

[4] I. Abraham, D. Malkhi, and A. Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 337–346, 2019.

[5] I. Abraham, K. Nayak, L. Ren, and Z. Xiang. Good-case latency of byzantine broadcast: A complete categorization. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 331–341, 2021.

[6] B. Aurn, Z. Li, F. Suri-Payer, D. Sourva, and A. Spiegelman. Shoal++: High throughput dag bft can be fast! *arXiv preprint 2405.20488*, 2024.

[7] K. Babel, A. Chursin, G. Danezis, L. Kokoris-Kogias, and A. Sonnino. Mysticeti: Low-latency dag consensus with fast commit path. *arXiv preprint arXiv:2310.14821*, 2023.

[8] V. Bagaria, S. Kannan, D. Tse, G. Fanti, and P. Viswanath. Prism: Deconstructing the blockchain to approach physical limits. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 585–602, 2019.

[9] A. Bhangale, C.-D. Liu-Zhang, J. Loss, and K. Nayak. Efficient adaptively-secure byzantine agreement for long messages. In *Advances in Cryptology – ASIACRYPT 2022: 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5–9, 2022, Proceedings, Part I*, page 504–525, Berlin, Heidelberg, 2022. Springer-Verlag.

[10] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the weil pairing. In *International conference on the theory and application of cryptology and information security*, pages 514–532. Springer, 2001.

[11] G. Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

[12] E. Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, University of Guelph, 2016.

[13] E. Buchman, J. Kwon, and Z. Milosevic. The latest gossip on bft consensus. *arXiv preprint arXiv:1807.04938*, 2018.

[14] A. Buchwald, S. Buttolph, A. Lewis-Pye, P. O'Grady, and K. Sekniqi. Frosty: Bringing strong liveness guarantees to the snow family of consensus protocols. *arXiv preprint arXiv:2404.14250*, 2024.

[15] V. Buterin and V. Griffith. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437*, 2017.

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

[17] M. Castro, B. Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, number 1999 in 99, pages 173–186, 1999.

[18] P. Civit, M. A. Dzulfikar, S. Gilbert, V. Gramoli, R. Guerraoui, J. Komatovic, and M. J. Ribeiro Vidigueira. Byzantine consensus is θ (nˆ 2): The dolev-reischuk bound is tight even in partial synchrony! *LIPIcs–Leibniz International Proceedings in Informatic*, (11):1–11, 2022.

[19] S. Cohen, G. Goren, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman. Proof of availability and retrieval in a modular blockchain architecture. In *International Conference on Financial Cryptography and Data Security*, pages 36–53. Springer, 2023.

[20] X. Dai, G. Wang, J. Xiao, Z. Guo, R. Hao, X. Xie, and H. Jin. Lightdag: A low-latency dag-based bft consensus through lightweight broadcast. *Cryptology ePrint Archive*, 2024.

[21] G. Danezis and D. Hrycyszyn. Blockmania: from block dags to consensus. *arXiv preprint arXiv:1809.01620*, 2018.

[22] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman. Narwhal and tusk: a dag-based mempool and efficient bft consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 34–50, 2022.

[23] D. Dolev and R. Reischuk. Bounds on information exchange for byzantine agreement. *Journal of the ACM (JACM)*, 32(1):191–204, 1985.

[24] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[25] EigenLabs. Intro to eigenda: Hyperscale data availability for rollups, 2023. Accessed on March 20, 2024.

[26] Ethereum. Proposer builder separation. https://ethereum.org/en/roadmap/pbs/.

[27] Ethereum. Data availability | ethereum.org, 2024. Accessed on March 20, 2024.

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

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

[30] A. Foundation. Decentralized timeboost specification. https://research.arbitrum.io/t/decentralized-timeboost-specification/9676.

[31] A. Gągol, D. Leśniak, D. Straszak, and M. Świętek. Aleph: Efficient atomic broadcast in asynchronous networks with byzantine nodes. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pages 214–228, 2019.

[32] C. Ganesh and A. Patra. Optimal extension protocols for byzantine broadcast and agreement. *Distrib. Comput.*, 34(1):59–77, Feb. 2021.

[33] J. Garay, A. Kiayias, and N. Leonardos. The bitcoin backbone protocol: Analysis and applications. *Journal of the ACM*, 71(4):1–49, 2024.

[34] J. A. Garay, R. Gennaro, C. Jutla, and T. Rabin. Secure distributed storage and retrieval. *Theor. Comput. Sci.*, 243(1–2):363–389, jul 2000.

[35] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th symposium on operating systems principles*, pages 51–68, 2017.

[36] N. Giridharan, F. Suri-Payer, I. Abraham, L. Alvisi, and N. Crooks. Autobahn: Seamless high speed bft. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 1–23, 2024.

[37] I. Keidar, E. Kokoris-Kogias, O. Naor, and A. Spiegelman. All you need is dag. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 165–175, 2021.

[38] I. Keidar, O. Naor, O. Poupko, and E. Shapiro. Cordial miners: Fast and efficient consensus for every eventuality. In *37th International Symposium on Distributed Computing (DISC 2023)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2023.

[39] L. Kiffer, J. Neu, S. Sridhar, A. Zohar, and D. Tse. Nakamoto consensus under bounded processing capacity. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 363–377, 2024.

[40] V. King, J. Saia, V. Sanwalani, and E. Vee. Scalable leader election. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm*, SODA '06, page 990–999, USA, 2006. Society for Industrial and Applied Mathematics.

[41] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. In *Concurrency: the works of leslie lamport*, pages 203–226. 2019.

[42] A. Lewis-Pye. Quadratic worst-case message complexity for state machine replication in the partial synchrony model. *arXiv preprint arXiv:2201.01107*, 2022.

[43] C. Li, P. Li, D. Zhou, Z. Yang, M. Wu, G. Yang, W. Xu, F. Long, and A. C.-C. Yao. A decentralized blockchain with high throughput and fast confirmation. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, pages 515–528, 2020.

[44] D. Malkhi and K. Nayak. Hotstuff-2: Optimal two-phase responsive bft. *Cryptology ePrint Archive*, 2023.

[45] D. Malkhi, C. Stathakopoulou, and M. Yin. Bbca-chain: One-message, low latency bft consensus on a dag. In *International Conference on Financial Cryptography and Data Security*, 2024.

[46] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[47] K. Nayak, L. Ren, E. Shi, N. H. Vaidya, and Z. Xiang. Improved Extension Protocols for Byzantine Broadcast and Agreement. In H. Attiya, editor, *34th International Symposium on Distributed Computing (DISC 2020)*, volume 179 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 28:1–28:17, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[48] K. Nazirkhanova, J. Neu, and D. Tse. Information dispersal with provable retrievability for rollups. In *Proceedings of the 4th ACM Conference on Advances in Financial Technologies*, pages 180–197, 2022.

[49] J. Neu, S. Sridhar, L. Yang, D. Tse, and M. Alizadeh. Longest chain consensus under bandwidth constraint. In *Proceedings of the 4th ACM Conference on Advances in Financial Technologies*, pages 126–147, 2022.

[50] R. Pass and E. Shi. Thunderella: Blockchains with optimistic instant confirmation. In *Advances in Cryptology–EUROCRYPT 2018: 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29-May 3, 2018 Proceedings, Part II 37*, pages 3–33. Springer, 2018.

[51] N. Polyanskii, S. Mueller, and I. Vorobyev. Starfish: A high throughput bft protocol on uncertified dag with linear amortized communication complexity. *Cryptology ePrint Archive*, 2025.

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

[53] C.-D. Sandro, M. Fitzi, A. Kiayias, G. Panagiotakos, and A. Russell. High-throughput blockchain consensus under realistic network assumptions. In *https://iohk.io/en/research/library/papers/high-throughput-blockchain-consensus-under-realistic-network-assumptions/*, 2024.

[54] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[55] V. Shoup. 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*, pages 207–220. Springer, 2000.

[56] V. Shoup. Sing a song of simplex. *Cryptology ePrint Archive*, 2023.
[57] V. Shoup and T. Locher. Minicast: Minimizing the communication complexity of reliable broadcast. Springer-Verlag, 2025.
[58] N. Shrestha, I. Abraham, L. Ren, and K. Nayak. On the optimality of optimistic responsiveness. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 839–857, 2020.
[59] N. Shrestha, R. Shrothrium, A. Kate, and K. Nayak. Sailfish: Towards improving the latency of dag-based bft. *Cryptology ePrint Archive, Paper 2024/472*, 2024.
[60] N. Shrestha, Q. Yu, A. Kate, G. Losa, K. Nayak, and X. Wang. Optimistic, signature-free reliable broadcast and its applications. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, 2025.
[61] A. Skidanov and I. Polosukhin. Nightshade: Near protocol sharding design. *URL: https://nearprotocol. com/downloads/Nightshade. pdf*, 39, 2019.
[62] A. Spiegelman, B. Aurn, R. Gelashvili, and Z. Li. Shoal: Improving dag-bft latency and robustness. In *International Conference on Financial Cryptography and Data Security*, 2024.
[63] A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias. Bullshark: The partially synchronous version. *arXiv preprint arXiv:2209.05633*, 2022.
[64] E. Systems. Designing the espresso sequencer: Combining hotshot consensus with tiramisu da, 2023. Accessed on March 20, 2024.
[65] L. Yang, Y. Gilad, and M. Alizadeh. Coded transaction broadcasting for high-throughput blockchains, 2022.
[66] L. Yang, S. J. Park, M. Alizadeh, S. Kannan, and D. Tse. {DispersedLedger}:{High-Throughput} byzantine consensus on variable bandwidth networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 493–512, 2022.
[67] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.
[68] H. Yu, I. Nikolić, R. Hou, and P. Saxena. Ohie: Blockchain scaling made simple. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 90–105, 2020.

A NOTES ON DISPERSED SIMPLEX

In Section 5.3, we analyzed the DispersedSimplex protocol assuming a fixed leader with pipelining. In this section, we will consider the original setting where the leader changes after each block. This protocol is similar to the one in Section 5.1 except that sending the next block happens concurrently with the vote step (called the commit step in the original paper).

To analyze the latency, we make the following simplifications. We suppose that *votes* are of a fixed size λ, and that the block metadata (perhaps a hash and a signature) is of size $c\lambda$ for some small constant c. If a block b contains B' transaction packets and metadata of size $c\lambda$, then we suppose that an erasure coded message $(d, G_i(b))$ is of size $3(B' + c\lambda)/n + \lambda \log(n)$ and an echo message is also of size $3(B' + c\lambda)/n + \lambda \log(n)$.

(1) Suppose t is the time at which the block at layer $d-1$ is added to the buffer by the leader of layer $d-1$. This means any transaction that arrives at the leader after time t will be included in a layer d block (or a later block).
(2) First, we assume that processors do not begin echoing fragments until receiving a full fragment (we will also consider early forwarding below). In this case, fragments of the layer $d-1$ block will arrive at all respective parties by time $t + C(3(B' + c\lambda)/n + \lambda \log(n)) + \Delta$. All parties will send their echo message and forward their fragment at this time.
(3) Assuming parties send their echo + forward after receiving the full fragment (and ignoring the size of the echo message), this will arrive at all parties at time $t' = t + C(\lambda \log(n) + \Delta + C(2(B' + c\lambda)/n + \lambda \log(n)) + \Delta$. This is equal to:

$$t + \frac{6(B' + c\lambda)}{S} + \frac{2\lambda n \log(n)}{S} + 2\Delta$$

(4) In the next step, the leader of layer d proposes a block, and all parties also send their vote share for layer $d-1$. Ignoring the size of the vote share, this will arrive at all parties at

$$t + \frac{9(B' + c\lambda)}{S} + \frac{3\lambda n \log(n)}{S} + 3\Delta$$

(5) Parties will then forward fragments of the layer d block and send their echo share. This will arrive at all parties at time:

$$t + \frac{12(B' + c\lambda)}{S} + \frac{4\lambda n \log(n)}{S} + 4\Delta$$

(6) All parties will receive the vote share at (ignoring its size):

$$t + \frac{12(B' + c\lambda)}{S} + \frac{4\lambda n \log(n)}{S} + 5\Delta$$

For a small c, the $\lambda n \log(n)$ term dominates $c\lambda$. Thus, the latency is given by

$$\frac{12B'}{S} + \frac{4\lambda n \log(n)}{S} + 5\Delta$$

Next, observe that, in steps 3 and 5, parties can actually start forwarding packets of data as soon as they receive them from the leader. If such pipelining is used, the latency for each of these steps will reduce by $\frac{3(B'+c\lambda)}{S} + \frac{\lambda n \log(n)}{S}$. Thus, the latency in the presence of pipelining is given by:

$$\frac{6B'}{S} + \frac{2\lambda n \log(n)}{S} + 5\Delta$$

Extending to a multi-sender setting. We now extend the analysis above to a setting where a leader must aggregate transactions from each of the n parties before making a proposal. These transactions can be piggy-backed together with the echo message + block forwarding. Now, assume that each party sends a block of size B, and thus, $B' = Bn$. The analysis above then changes as follows:

(1) In the earlier analysis, any transaction that arrives at the leader after time t will be part of a layer d block (or a later block). When the leader aggregates from other parties, the time t is for any transaction that arrives at any party after they have sent their layer $d-1$ block to the leader of layer $d-1$. This adds a latency of: $(\frac{Bn}{S} + \Delta)$.
Again, if pipelining is employed so that transactions are transmitted to the leader as they arrive on the client processor, then the additional latency can be reduced to Δ. (Note that this is possible to do since the parties are sending these blocks directly to the leader without erasure coding.)

(2) The echo message in step 5 involves sending the block for layer $d+1$ to the corresponding leader. This requires $\frac{Bn}{S}$ time.
Again, if pipelining is employed, then there is no additional latency.

Thus, the latency incurred without pipelining is:

$$\frac{14Bn}{S} + \frac{4\lambda n \log(n)}{S} + 6\Delta$$

Thus, the latency incurred with pipelining is:

$$\frac{6Bn}{S} + \frac{2\lambda n \log(n)}{S} + 6\Delta$$

Latency of Sailfish with consistent broadcast expressed with the B term. In comparison, the latency of Sailfish where the leader of layer d points to all blocks in layer $d-1$ is given by:

$$(C(M) + C(n\lambda) + 2\Delta) + 2(C(B+M) + C(n\lambda) + 2\Delta) + (C(M) + \Delta)$$

The first term $C(M) + C(n\lambda) + 2\Delta$ is the queueing delay, $2(C(B+M) + C(n\lambda) + 2\Delta)$ is the time to complete consistent broadcasts for 2 layers, and $C(M) + \Delta$ is the time for receiving sufficiently many "first votes" to commit the leader block. Thus, the latency is given by:

$$\frac{2Bn}{S} + \frac{7\lambda n^2}{S} + 7\Delta$$

Local variables:

struct vertex v: ▷ The struct of a vertex in the DAG

$v.round$ - the round of v in the DAG

$v.source$ - the party that broadcast v

$v.block$ - a block of transactions

$v.edges$ - a set of vertices in $v.round - 1$ that represent strong edges (aka. independent metadata)

$v.dependentMeta$ - dependent metadata

Upon receiving CBC-deliver all n vertices from previous round, construct a new round r vertex v with transactions from a buffer. Send vertex v via a CBC primitive.

Fig. 29. Basic Sailfish