HAL open science

A Comparison of Message Exchange Patterns in BFT Protocols

Fábio Silva, Ana Alonso, José Pereira, Rui Oliveira

To cite this version:

Fábio Silva, Ana Alonso, José Pereira, Rui Oliveira. A Comparison of Message Exchange Patterns in BFT Protocols. 20th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS), Jun 2020, Valletta, Malta. pp.104-120, 10.1007/978-3-030-50323-9_7 . hal-03223258

HAL Id: hal-03223258

https://inria.hal.science/hal-03223258

Submitted on 10 May 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

A Comparison of Message Exchange Patterns in BFT Protocols
(Experience Report)

Fábio Silva, Ana Alonso, José Pereira, and Rui Oliveira

INESC TEC and U. Minho
{fabio.l.silva,ana.n.alonso}@inesctec.pt, {jop,rco}@di.uminho.pt

Abstract. The performance and scalability of byzantine fault-tolerant (BFT) protocols for state machine replication (SMR) have recently come under scrutiny due to their application in the consensus mechanism of blockchain implementations. This led to a proliferation of proposals that provide different trade-offs that are not easily compared as, even if these are all based on message passing, multiple design and implementation factors besides the message exchange pattern differ between each of them. In this paper we focus on the impact of different combinations of cryptographic primitives and the message exchange pattern used to collect and disseminate votes, a key aspect for performance and scalability. By measuring this aspect in isolation and in a common framework, we characterise the design space and point out research directions for adaptive protocols that provide the best trade-off for each environment and workload combination.

1 Introduction

The popularization of cryptocurrencies backed by blockchain implementations such as Bitcoin has led to a renewed interest in consensus protocols, particularly in protocols that can tolerate Byzantine faults to prevent malicious participants from taking fraudulent economic advantage from the system. Instead of using established BFT protocols such as PBFT [7] to totally order transactions, permissionless blockchains such as Bitcoin's [14] and Ethereum [6] currently use protocols based on Proof-of-Work [14], as scalability in the number of processes is known to be an issue for classic BFT consensus protocols. This, however, represents a trade-off: the ability to scale to large numbers of processes with a possibly very dynamic membership comes at the cost of increased transaction latency and probabilistic transaction finality.

An alternative path is taken by permissioned blockchains such as Hyperledger Fabric [1], which use classical consensus protocols to totally order transactions, motivating the need for higher scalability in BFT protocols. The result has been that a variety of BFT protocols have been proposed, which, having identified the number and/or size of the messages to be exchanged as the bottleneck for scalability, take advantage of different message exchange patterns combined with different cryptographic primitives [9,17,4,2,12,8].

The proposed protocols might be generally compared through decision latency and throughput measurements in a common experimental setup, and relative scalability evaluated by varying only the number of processes. There are however a number of implementation factors that can affect performance and hide the impact of the abstract protocol, including the programming language, concurrency control strategy, and networking and cryptographic libraries.

We argue that a more interesting result for the proposal and implementation of future protocols can come from assessing the impact of the selected message exchange patterns and cryptographic primitives, in isolation of other implementation factors, as these are key aspects for protocol performance and strongly impact scalability to a large number of processes.

This work makes the following contributions:

- We propose an experimental harness for reliably reproducing the performance and scalability characteristics of the vote dissemination and collection phases in a BFT protocol, including the message exchange pattern and the cryptographic primitive.
- We run experiments for four message exchange patterns and three cryptographic primitives, thus characterizing the design space for these protocols in terms of resource usage (CPU and network) and potential parallelism. We use these results to draw lessons for future research and development.

The rest of this paper is structured as follows. Section 2 briefly describes existing protocols in terms of the message exchange patterns and cryptographic primitives used. Section 3 proposes a model for reproducing and measuring the impact of these protocol features, Section 4 presents the results obtained, which are discussed in Section 5. Finally, Section 6 presents the main lessons learned and outlines future work.

2 Background

A practical BFT protocol for state machine replication was initially proposed by Castro and Liskov [7]. It allows clients to submit requests to a set of processes that order and execute them. The challenge lies in ensuring that all correct processes execute the same sequence of requests, regardless of crash faults, where processes forget what has not yet been saved to a persistent log and byzantine faults, where malicious processes behave arbitrarily, possibly generating messages that do not comply with the protocol. The number of tolerated faults is bound to the number of processes in such a way that it requires $3t + 1$ processes to provide safety and liveness in the presence of t faults.

The challenge is addressed with a three phase message exchange protocol. In a first stage, *pre-prepare* the current leader proposes a sequence number for a pending request. In a second stage, *prepare*, all processes that recognize the sender of the message as the leader acknowledge it to all others. This is however not enough for agreement, as malicious processes might be sending different messages to different destinations. Therefore, in the third phase, *commit*, processes that got $2t$ acknowledgments will then confirm the outcome to all others.

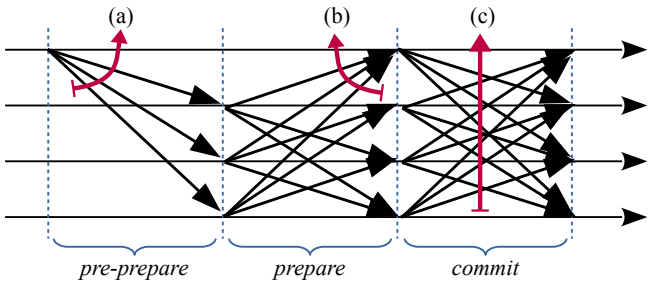

Fig. 1. Message exchange pattern in BFT agreement in three phases. It includes (a) sending a message to all others; (b) receiving a message from all others; and (c) quadratic number of messages being transmitted in the network.

Upon reception of $2t + 1$ confirmations, the request can be executed. Note that messages need to be authenticated to prevent malicious processes from forging messages by impersonating other processes.

The resulting message exchange pattern (*broadcast*) is depicted in Figure 1. Even if these messages do not convey the requests themselves and are restricted to protocol information, it is problematic for performance in various ways. First, (a) it requires each process to send a message to all others, which without a true broadcast medium consumes CPU time in transport and network layers. Second, each process has to receive and handle messages from all others (b). In fact, even if only $2t+1$ replies are needed for progress, in the normal case where no process is faulty, $3t+1$ messages will have to be delivered and decoded, thus consuming CPU time. Finally, as all-to-all messages are exchanged, the network bandwidth used grows quadratically with the number of processes. These issues are bound to become scalability bottlenecks.

The typical answer to these issues is to design a message exchange pattern that trades latency for bandwidth and exploits parallelism. For instance, instead of directly sending a message to all destinations, it is first sent to a selected subset that then relays it to some other subset. This avoids any of the processes having to deal directly with all destinations and enables message exchange to be done in parallel by the various intermediate processes. A similar strategy can be used when collecting acknowledgments.

An option used in agreement protocols in the crash-stop model [11], is to employ a *centralized* pattern, i.e., to rely on the central coordinator to collect messages from all processes and then re-broadcast to all, making the number of messages grow linearly with the number of processes at the expense of an additional communication step. Another option is to organize processes in a logical ring and have each of them add to and forward the message to the next (*ring* pattern). This is the option taken in Ring Paxos [13] and in the Chain configuration of the Aliph protocol [2]. Gossiping is a well known efficient distributed information dissemination and aggregation strategy, hence it has also been proposed in this context [12]. In this case, each bit of information is routed to a

Table 1. Representative protocols for different message exchange patterns (rows) and cryptographic primitive combinations (columns).

	Simple/Set	**Threshold**	**Aggregate**
Broadcast	PBFT [7]	n/a	n/a
Centralized		SBFT [9], HotStuff [17]	LibraBFT [4]
Ring	Chain [2]		
Gossip			Multi-level [12]

small random subset of destinations, where it can be combined and forwarded (*gossip* pattern).

Unfortunately, the assumption of byzantine faults makes this harder to achieve than in the variants of Paxos for the crash-stop fault model, as a process cannot trust others to correctly forward the information contained in messages from others unless the original (*simple*) cryptographic signature is included verbatim. This works when disseminating information but is less useful when collecting information from other processes, as the agreement protocol needs to do in *prepare* and *commit* phases, as multiple signatures need to be included (*set*), making message size grow with the number of processes. It is nonetheless viable and is used in the Chain configuration of Aliph [2].

Some protocols employ cryptographic techniques that enable signatures to be combined to mitigate this increase in message size. Designating specific processes to act as collectors, which combine a pre-defined number of signatures into a single one (*threshold signatures*), can be used in protocol phases that require the collection of a minimum number of replies/confirmations [9,17]. Alternatively, other protocols leverage techniques that allow signatures to be aggregated at each step (*aggregate signatures*), thus eschewing the need to define specific processes to carry out this operation but, in turn, verification requires knowing exactly which signatures have been aggregated [12,4].

Table 1 lists representative combinations of message exchange patterns and cryptographic primitives for creating digital signatures used in BFT protocols. Notice that for the broadcast message pattern used in the original PBFT protocol [7] there is no need to use a cryptographic primitive to combine message signatures, as these are sent directly. The other options might lead to useful combinations, as the computational effort required by different cryptographic primitives needs to be weighed against savings in the amount of data that is transmitted.

3 Model

To assess the impact of each combination of message exchange pattern and cryptographic primitive we built a cycle-based simulation of the core phases of a byzantine fault tolerant protocol. This allows us to highlight the impact of these two factors without the experimental noise that would result from implementation details such as language, concurrency, networking, serialization and

cryptographic libraries. This also allows us to exhaustively experiment with all combinations, including those that haven't been tried before.

The protocol model is as follows. It reproduces only the common path of the replication protocol, namely, the *pre-prepare*, *prepare*, and *commit* phases of PBFT [7] as shown in Figure 1. A designated process (the coordinator) starts a protocol instance by disseminating a proposal. Each process, upon receiving that proposal, disseminates a first phase vote. Upon collecting first phase votes from two thirds of processes (a first phase certificate), a process disseminates a second phase vote. Agreement is reached when one third of processes collect a second phase certificate (second phase votes from two thirds of processes). The model thus omits request execution, interaction with clients, and the view change protocol, needed to deal with failure of the coordinator.

The key to achieving different message exchange patterns is to allow each process to forward information. In this case, the relevant information consists of the votes for each phase of the protocol: instead of a process having to send a vote directly to all others, as in the original PBFT protocol, it is possible for the vote to be forwarded by intermediate processes, thus avoiding the need for direct communication. We do this in a simple fashion: each process is able to send all votes collected so far in each phase instead of just sending out its own. The decision for when these votes are sent and to whom depends on a strategy parameter, which leads to different message exchange patterns. Based on the protocols described in Section 2, the considered message exchange patterns are:

Broadcast: The coordinator broadcasts the proposal and each process broadcasts its own votes.

Centralized: The coordinator broadcasts the proposal and each process sends its own votes only to the coordinator. Upon collecting a certificate, the coordinator forwards it to the remaining processes.

Ring: Processes are disposed in a logical ring. The coordinator sends the proposal to its successor. A process forwards the proposal and collected votes to its successor until it forwards a second phase certificate.

Gossip: The coordinator sends the proposal to *fanout* processes. A process forwards the proposal and collected votes to *fanout* processes every time it receives a set that contains messages (either proposal or votes) it does not know about, until it forwards a second phase certificate. The destinations are picked from a random permutation of all possible destinations, in a cyclic order, to ensure deterministic termination [15].

All patterns except Broadcast require processes to forward collected votes. Votes must be authenticated and same phase votes from distinct processes, if correct, differ only in their signature. These signatures can be sent individually, as a set, or make use of cryptographic techniques to reduce the size of messages as follows:

Set: A simple approach is to forward a set containing known signatures. However, this entails that message size will be proportional to the number of signatures.

Threshold: Threshold signatures allow any process to convert a set of signatures into a single signature. However, this can only happen when the set contains a pre-defined number of signatures – the threshold value. Up until that point the whole set must be forwarded. In this context, the threshold value should be two thirds of the number of processes (the size of certificates).

Aggregation: With signature aggregation, processes can aggregate any number of signatures into a single signature at any moment, but forwards must include information about which processes' signatures have been aggregated. Additionally, for the gossip pattern, forwarded information must also include how many times each signature has been aggregated, as these may, in turn, be further aggregated.

Regarding the simulator, in each cycle, each active process runs to completion, sequentially processing all pending messages. In detail, a process is active if it is the coordinator at the start of the protocol or if there is an incoming message, ready to be received. Each process can thus receive and send multiple messages per cycle. Messages sent in a cycle are made available at the destination in the next cycle. This allows us to obtain several interesting metrics:

Number of cycles to reach a decision: The number of cycles required to reach a decision is the primary metric, as it provides a measurement of how many communication steps are required.

Number of messages sent and received: The number of messages sent and received provide a measurement of network bandwidth used. By recording these metrics individually for each process, we are also able to point out the cases where the load is asymmetrically distributed.

Message size in bytes: The overhead that the message exchange pattern combined with the cryptographic primitive entails in bytes. The space taken by view, sequence number, requests, among others, are not regarded. This metric is calculated from the content of the messages exchanged and is key to assess the impact of collecting multiple votes in each forwarded message.

Number of active processes: The number of active processes is a measure of parallelism, pointing out how many processes are able to make progress in parallel and how evenly computational load is distributed.

Actual CPU time: Since the implementation used to process each message is complete, i.e., includes de-serialization of the input signatures, protocol state changes, cryptographic operations, and serialization of output signatures, and would be usable in a real implementation, we measure the used CPU time using hardware counters, and consider this as a measure of computational effort.

The protocol model and cycle-based simulators have been implemented in C++ and executed in a Linux server with dual *AMD Opteron 6172* processors (2100 MHz and 24 cores/hardware threads) with 128GB RAM. All cryptography is provided by the *Chia-Network BLS signatures library.*[1]

[1] `https://github.com/Chia-Network/bls-signatures`

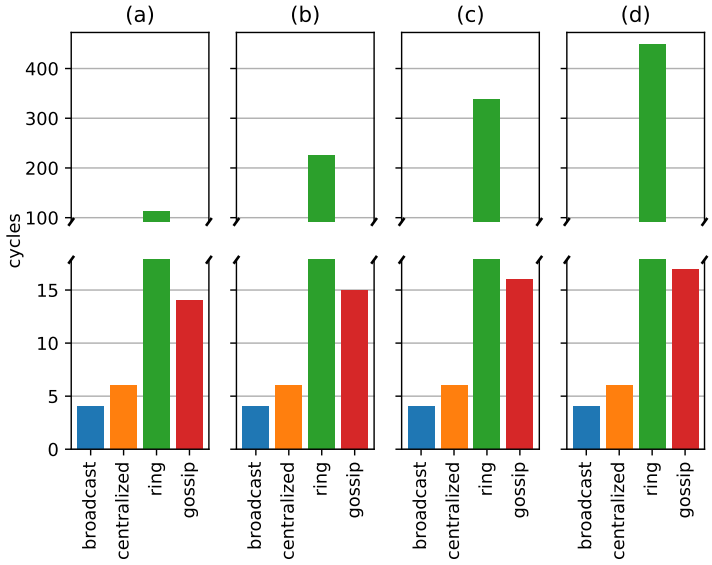

Fig. 2. Number of cycles needed to reach a decision in an agreement instance, by each process, per message exchange pattern for: (a) 49 processes; (b) 97 processes; (c) 145 processes; and (d) 193 processes.

4 Results

The cycle-based simulator and protocol model are now used to obtain results for each relevant message exchange pattern and cryptographic primitive. It should be pointed out that the broadcast pattern uses only simple message signatures, as each process only sends its own vote and sends it directly to every other process. On the other hand, in centralized, ring and gossip patterns, processes forward collected votes and thus are evaluated with all cryptographic primitive options. In addition, the number of destinations in the gossip pattern is set as $fanout = 2$, the lowest value that can still define the pattern. These experiments don't account for either network or process faults, so there is no need for retransmissions, no messages are lost and messages are always correct. All experiments are repeated with 49, 97, 145 and 193 processes, the number of processes required to tolerate 16, 32, 48 and 64 malicious processes, respectively.

Because the gossip message pattern includes an element of randomness, the results of several runs were analysed. In order to consolidate the results of those runs, we first ranked the measurements for each metric per run. Then we calculated the average value per rank. An alternative could be to use the identity of each process to calculate average measurements. However, the identity of processes in different runs is ultimately unrelated. Thus, the ranking method provides better predictive ability, allowing us to provide, for example, an esti-

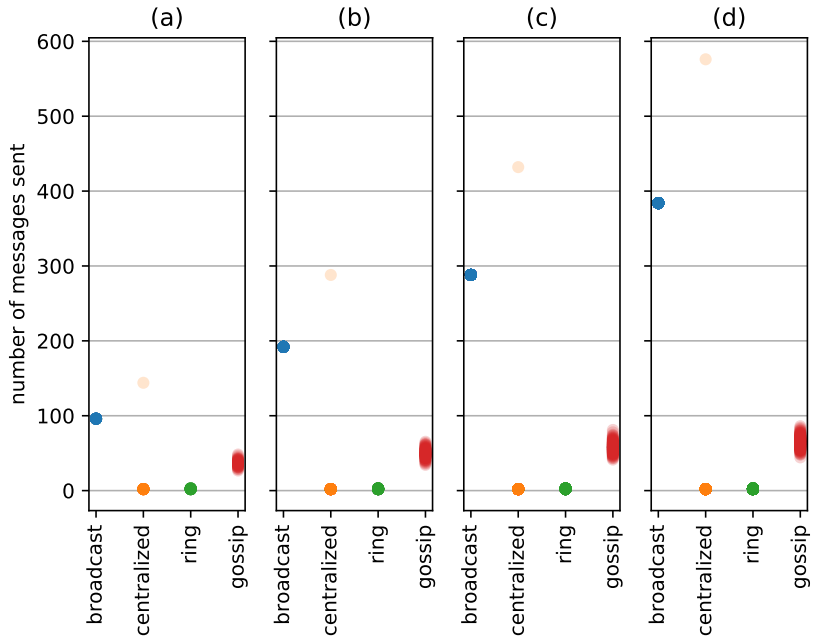

Fig. 3. Number of messages sent in an agreement instance, by each process, per message exchange pattern for: (a) 49 processes; (b) 97 processes; (c) 145 processes; and (d) 193 processes.

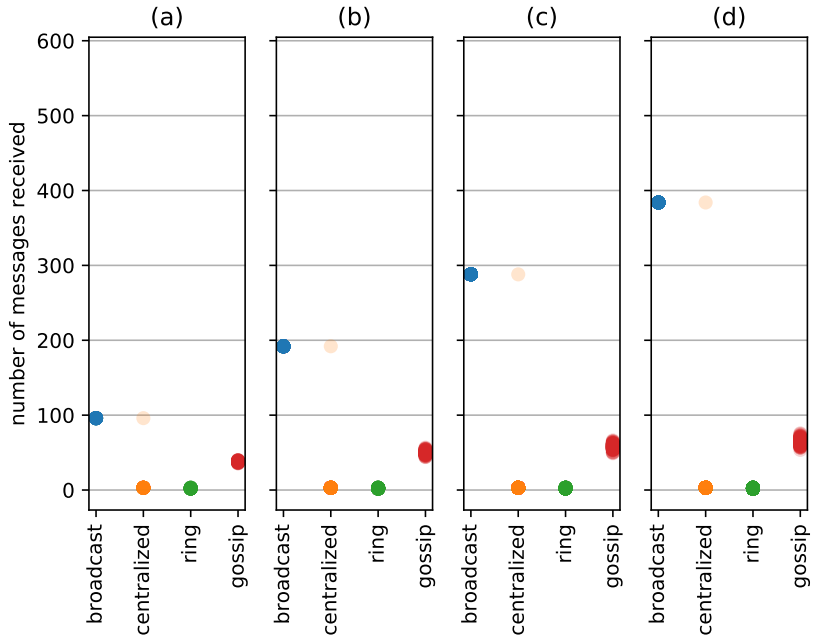

Fig. 4. Number of messages received in an agreement instance, by each process, per message exchange pattern for: (a) 49 processes; (b) 97 processes; (c) 145 processes; and (d) 193 processes.

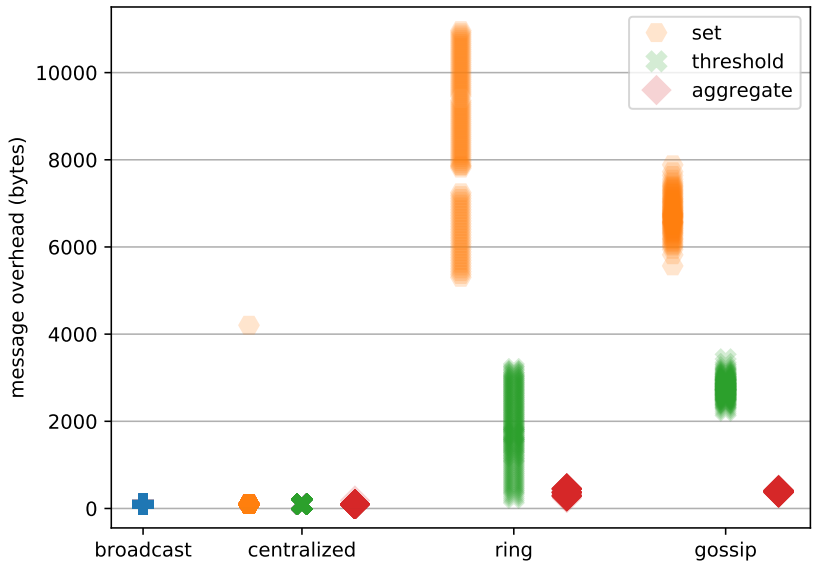

Fig. 5. Message overhead due to signatures (in bytes), averaged per process, for each combination of message exchange pattern and cryptographic primitive, for 97 processes.

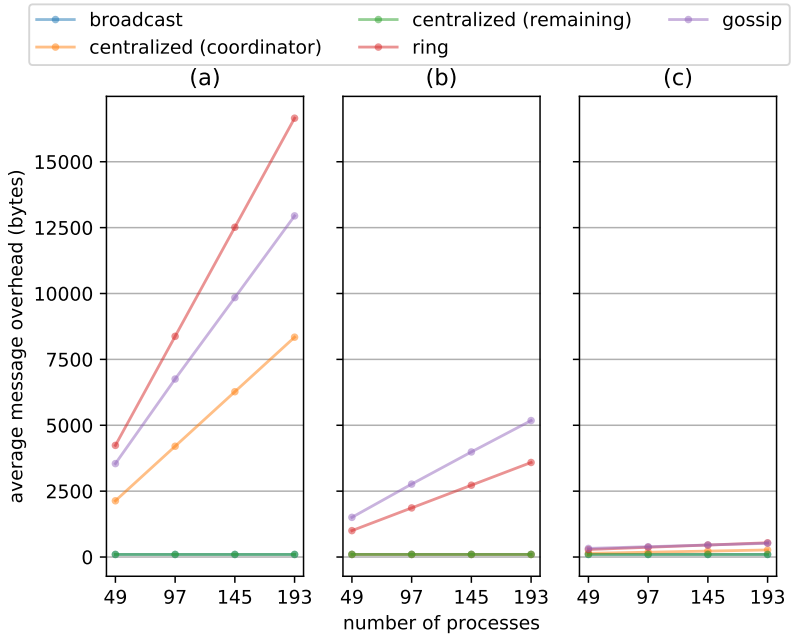

Fig. 6. Overall average message overhead due to signatures (in bytes) for an increasing number of processes, per cryptographic primitive: set of signatures (a), threshold signatures (b) and signature aggregation (c). Values for the Broadcast pattern are also presented for comparison.

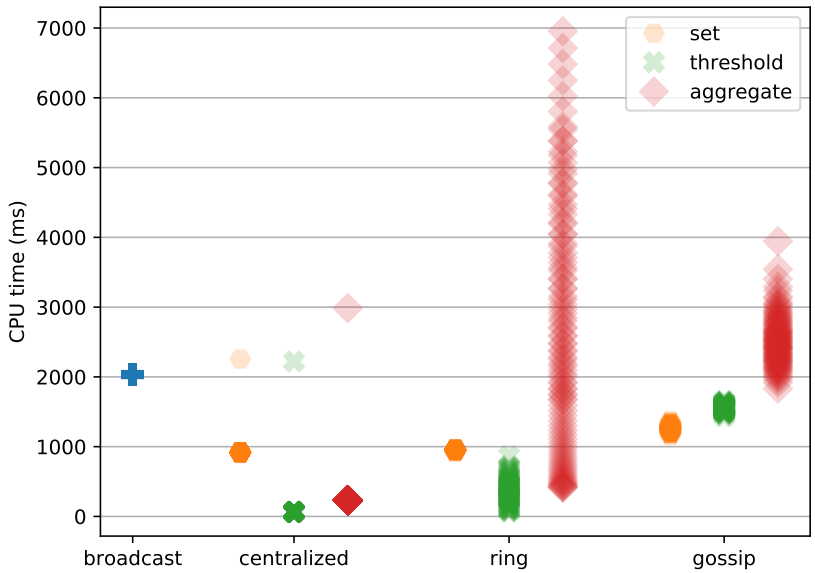

Fig. 7. Total CPU usage (in ms) for each process, per combination of message exchange pattern and cryptographic primitive, for 97 processes.

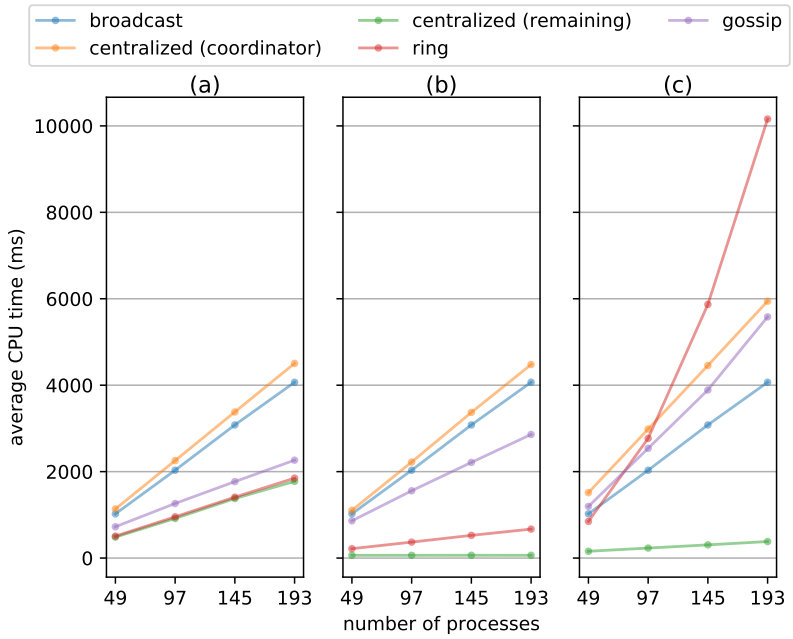

Fig. 8. Average CPU usage (in ms) per process, for an increasing number of processes, per cryptographic primitive: set of signatures (a), threshold signatures (b) and signature aggregation (c). Values for the Broadcast pattern are also presented for comparison.

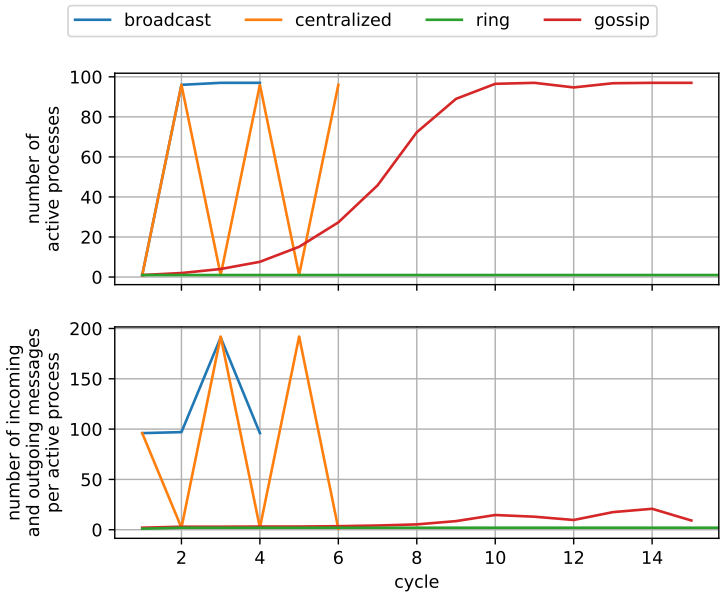

Fig. 9. Number of active processes in each cycle (a) and the average number of incoming and outgoing messages processed by each active process (b), per message exchange pattern, for 193 processes. Ring continues past 400 cycles (cropped).

mate for how long it will take for the first process to decide and also for the last to decide.

Figure 2 shows the number of simulation cycles needed for deciding an instance of the protocol, with each of the message exchange patterns. This is the number of communication steps needed for processes to agree on the next command to execute. For instance, with the broadcast pattern processes agree in four communication steps: first the coordinator broadcasts a proposal; processes then receive the proposal and broadcast its first phase vote; afterwards processes receive all first phase vote and broadcast a second phase votes; and finally processes receive all second phase votes.

Results shown in Figures 3 to 8 focus on resource usage. Note that Figures 3, 4, 5 and 7 plot a dot for the result observed in each process, showing where appropriate the dispersion of results depicted by the level of color saturation: the more overlap, the higher the color saturation. This is evident in the centralized pattern, as measurements regarding the coordinator are depicted as mostly transparent and color saturation reveals the overlap regarding remaining processes.

In detail, Figures 3 and 4 show, respectively, the number of messages sent and received for an agreement instance, for each message exchange pattern and for a growing number of processes. Figure 5 shows the message overhead in bytes due to votes carried, including the signatures in a run with 97 processes. A dot is

plotted for each process, showing the average message size for that process, which in some configurations is variable. Figure 6 then shows how the average message size varies with the number of processes in the system. Likewise, Figure 7 shows the CPU time consumed by each process. A dot is plotted for each of them, showing that in some cases the load is variable. Figure 8 then shows how the average CPU time used varies with the number of processes in the system.

Finally, Figure 9 describes how the load is distributed across different processes and across time, during the run of an agreement instance. In detail, Figure 9(a) shows the number of active processes (i.e., those that receive and send messages in that cycle) as time progresses. Figure 9(b) shows the average number of incoming and outgoing messages that are processed by each of the active processes.

5 Discussion

The considerations put forth in this section are based on the analysis of the results presented in Section 4.

Broadcast. We start by discussing the results for the broadcast pattern as a baseline, as it matches the original PBFT protocol [7]. In this pattern, each process sends and receives messages directly to and from all others. Therefore, all processes work in parallel sending and receiving n messages in each phase. Messages contain only one signature, thus the message overhead due to signatures is always the same and does not change with the total number of processes. The total CPU time is the same for all processes and increases linearly with the number of processes, corresponding to the number of messages processed. As a consequence, a decision is achieved in a small number of cycles.

Centralized. For the centralized pattern we need to make a distinction between the coordinator and the remaining processes since they behave differently. The coordinator sends $3n$ messages and receives $2n$ messages while the remaining processes always send 2 messages and receive 3 messages. The coordinator and the remaining processes alternate executions, with the latter computing in parallel. The coordinator sends and receives n messages in each cycle (high load) while the remaining processes only send and receive 1 message per cycle (low load). The overhead due to signatures in messages received by the coordinator, sent by the remaining processes, is always the same, since the messages only contain one signature regardless of the cryptographic primitive.

Initially, regardless of the cryptographic primitive used the coordinator sends proposals, which are always the same size (one signature). However, with the set of signatures it forwards $2n/3$ signatures in each phase. The total CPU time for the coordinator is slightly higher than in the broadcast primitive because of the certificates being forwarded. On the other hand, the remaining processes verify the signatures from each certificate in batch, which is faster than verifying them one by one as they do in the broadcast pattern. The total CPU time increases

linearly with the number of processes for both the coordinator and the remaining processes.

With threshold signatures, all messages sent by the coordinator contain only one signature, so the overhead due to signatures per message does not change with the total number of processes. The coordinator's total CPU time is roughly the same as in the set of signatures primitive since the benefit of creating smaller messages mitigates the drawback of computing threshold signatures. As with the baseline set of signatures option, it also increases linearly with the number of processes. The remaining processes only have to make a single signature verification thus its total CPU time is the lowest overall and remains constant irrespective of the number of processes.

Finally, with the aggregate signatures primitive, the certificates the coordinator forwards contain one signature plus info detailing which signatures have been aggregated. Aggregating signatures is a more expensive operation than creating a threshold signature thus the total CPU time for the coordinator is the highest among the centralized alternatives and it also increases linearly. The remaining processes have to compute the info to verify the aggregated signature which is slower than verifying a threshold signature but faster than verifying a set of signatures.

Ring. With the ring pattern, the protocol completes after $2+1/3$ laps around the ring which results in two thirds of the processes to send and receive 2 messages while one third sends and receives 3 messages. Processes compute sequentially which results in no parallel processing. Process load is small as each process only sends and receives 1 message per cycle.

With the set of signatures, message sizes range from 1 up to $4n/3$ signatures (two certificates) resulting in a big variation in the average size of messages among processes. The total CPU time is the same for all processes, lower than in broadcast and increases linearly with the number of processes.

Using threshold signatures, messages are smaller than if using the set primitive because when the number of signatures for a phase reaches $2n/3$, a threshold signature is created, replacing those individual signatures. The total CPU time varies per process since some processes only verify the computed threshold signatures. Despite the variation, it is lower than when using the set of signatures and also increases linearly with the number of processes, although at a slower rate.

Using aggregate signatures, messages contain up to 3 signatures plus related information, namely the processes for which signatures have been aggregated. Regarding total CPU time, there is a large variation between processes because the computational effort of the processes that send and receive 3 messages is considerably larger than that of processes that only send and receive 2 messages. Still, even among processes that exchange the same number of messages some variation occurs as those that receive a certificate from their predecessor are not required to aggregate their signatures. This makes it the worst combination of message exchange pattern and cryptographic primitive for the total CPU time since it also grows exponentially with the number of processes.

Gossip. The number of messages each process sends and receives with the gossip pattern is lower than with the broadcast message pattern and increases only logarithmically with the number of processes. The number of active processes in each cycle increases exponentially with base *fanout*. After $\log_F n$ cycles, all processes execute in parallel and each process sends and receives a small number of messages in each cycle (low load).

With the set of signatures, message sizes can grow up to $4n/3$ signatures (two certificates). Since each process sends and receives more messages, the variation of the average size of messages is smaller than in the ring pattern. The total CPU time shows a small variation between processes but is always lower than for the broadcast pattern, increasing linearly with the number of processes.

Using threshold signatures, messages are smaller because, again, when the number of signatures for a phase reaches $2n/3$, a threshold signature is created replacing these. The total CPU time also shows a small variation among different processes, being higher overall than if using the set of signatures. The reason is that it is likely that by the time some process is able to generate a threshold signature and send it to others, most of the processes will have also collected enough messages to generate a certificate themselves. This means that most processes will use CPU time to generate threshold signatures but few processes will actually make use of the threshold signatures generated by others. Nevertheless, it is still lower than with the broadcast pattern and increases linearly with the number of processes.

Using aggregate signatures, messages contain up to 3 signatures plus information on aggregation. There is a big variation among different processes regarding the total CPU time, with the average being higher than with the broadcast pattern. It increases linearly with the number of processes.

6 Lessons Learned and Future Work

Considering the results obtained with our simulation model of the core part of the protocol needed for a byzantine fault tolerant replicated state machine, we can now draw some important lessons to steer future research and development effort:

There is no absolute best message exchange pattern. The first interesting conclusion is that none of the tested message exchange patterns performs optimally in all scenarios. In fact, if processes can handle sending and receiving as many messages as the number of processes (i.e., small clusters of powerful servers), then the centralized pattern combined with threshold signatures should be the best option, since it requires exchanging the least messages and results in lower computational effort for the majority of processes, when compared to the broadcast pattern. This is the approach of SBFT [9] and HotStuff [17]. However, as the number of processes grows it becomes harder to sustain such loads. In this case, the gossip pattern with signature aggregation might be the best choice since it evenly distributes the load across servers, without the overhead of the broadcast

pattern. The ring pattern induces very high latency since there is no parallel processing. However, it might allow for high throughput if multiple protocol instances run in parallel. Moreover, there are also other patterns not included in this work, such as the communication trees employed by ByzCoin [10].

Cryptographic primitives provide a range of CPU vs network bandwidth trade-offs. The threshold signatures primitive requires a set of signatures to be forwarded until the threshold value is reached, which is a disadvantage when combined with either the ring or the gossip patterns. Moreover, if the set of processes changes, new private keys must be generated for each process to create a new master public key with which threshold signatures can be verified. In terms of computation, the signature aggregation primitive is always the slowest. This is partly due to the operations necessary for aggregating signatures and for verifying them. This means that we get a range of trade-offs between computational effort and network bandwidth, that suit different environments. Finally, we also believe that the cryptographic library is not optimized to re-aggregate existing aggregate signatures, which affects ring and gossip but not the centralized pattern [5].

Overall conclusion: The case for adaptive protocols. The results obtained thus make a strong case for adaptive protocols that can be configured to use different message exchange patterns and a choice of cryptographic primitives to suit different environment and application scenarios. Moreover, these results make a strong case for automated selection of the best message pattern and cryptographic primitive combination by monitoring the current environment. Current proposals addressing these issues are Aliph [2] and ADAPT [3] which, however, don't cover the full spectrum of options. Other optimizations can also be included in such a protocol, like recent work on distributed pipelining [16], since they are orthogonal to this proposal.

Future work. First, the proposed simulation model can be used to obtain additional results and as a test bed for the optimization of the various patterns. For instance, message size in the gossip pattern, for any cryptographic primitive, might be further reduced if one takes into consideration the destination process. For example, if a second phase vote from the destination is already known, there is no point in sending it the first phase certificate. We can also collect results for a wider range of protocol parameters (e.g., varying the fanout in the gossip pattern) and, also, assess the behavior or each combination in the presence of faults, by implementing the view change protocol. Finally, these results also pave the way for research, namely, by providing data that can be used to train and test adaptation policies.

References

1. Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., et al.: Hyperledger fabric:

A distributed operating system for permissioned blockchains. In: Proceedings of the Thirteenth EuroSys Conference. EuroSys '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3190508.3190538, https://doi.org/10.1145/3190508.3190538
2. Aublin, P.L., Guerraoui, R., Knežević, N., Quéma, V., Vukolić, M.: The next 700 BFT protocols. ACM Trans. Comput. Syst. **32**(4), 12:1–12:45 (Jan 2015). https://doi.org/10.1145/2658994, http://doi.acm.org/10.1145/2658994
3. Bahsoun, J.P., Guerraoui, R., Shoker, A.: Making BFT protocols really adaptive. In: Proceedings of the 29th IEEE International Parallel & Distributed Processing Symposium (May 2015)
4. Baudet, M., Ching, A., Chursin, A., Danezis, G., Garillot, F., Li, Z., Malkhi, D., Naor, O., Perelman, D., Sonnino, A.: State machine replication in the Libra blockchain (2019)
5. Boneh, D., Drijvers, M., Neven, G.: Compact multi-signatures for smaller blockchains. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 435–464. Springer (2018)
6. Buterin, V.: Ethereum: A next-generation smart contract and decentralized application platform (2014), https://github.com/ethereum/wiki/wiki/White-Paper
7. Castro, M., Liskov, B.: Practical byzantine fault tolerance. In: Proceedings of the Third Symposium on Operating Systems Design and Implementation. pp. 173–186. OSDI '99, USENIX Association, Berkeley, CA, USA (1999), http://dl.acm.org/citation.cfm?id=296806.296824
8. Gilad, Y., Hemo, R., Micali, S., Vlachos, G., Zeldovich, N.: Algorand: Scaling byzantine agreements for cryptocurrencies. In: Proceedings of the 26th Symposium on Operating Systems Principles. pp. 51–68. SOSP '17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3132747.3132757, http://doi.acm.org/10.1145/3132747.3132757
9. Gueta, G.G., Abraham, I., Grossman, S., Malkhi, D., Pinkas, B., Reiter, M.K., Seredinschi, D.A., Tamir, O., Tomescu, A.: SBFT: a scalable and decentralized trust infrastructure. In: IEEE Intl. Conf. Dependable Systems and Networks (DSN) (2019)
10. Kogias, E.K., Jovanovic, P., Gailly, N., Khoffi, I., Gasser, L., Ford, B.: Enhancing bitcoin security and performance with strong consistency via collective signing. In: 25th {usenix} security symposium ({usenix} security 16). pp. 279–296 (2016)
11. Lamport, L., et al.: Paxos made simple. ACM Sigact News **32**(4), 18–25 (2001)
12. Long, J., Wei, R.: Scalable BFT consensus mechanism through aggregated signature gossip. In: 2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC). pp. 360–367 (May 2019). https://doi.org/10.1109/BLOC.2019.8751327
13. Marandi, P.J., Primi, M., Schiper, N., Pedone, F.: Ring Paxos: A high-throughput atomic broadcast protocol. In: 2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN). pp. 527–536. IEEE (2010)
14. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2009), http://www.bitcoin.org/bitcoin.pdf
15. Pereira, J., Oliveira, R.: The mutable consensus protocol. pp. 218–227 (2004). https://doi.org/10.1109/RELDIS.2004.1353023
16. Voron, G., Gramoli, V.: Dispel: Byzantine smr with distributed pipelining. arXiv preprint arXiv:1912.10367 (2019)
17. Yin, M., Malkhi, D., Reiter, M.K., Gueta, G.G., Abraham, I.: Hotstuff: BFT consensus with linearity and responsiveness. In: Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing. pp. 347–356 (2019)