Poster: Coded Broadcast for Scalable Leader-Based BFT Consensus

Ioannis Kaklamanis
Massachusetts Institute of Technology
Cambridge, MA, USA
jkaklam@mit.edu

Lei Yang
Massachusetts Institute of Technology
Cambridge, MA, USA
leiy@csail.mit.edu

Mohammad Alizadeh
Massachusetts Institute of Technology
Cambridge, MA, USA
alizadeh@csail.mit.edu

ABSTRACT

With the success of blockchains and cryptocurrencies, Byzantine Fault Tolerant state machine replication protocols have attracted considerable interest. A class of such protocols that is particularly popular are *leader-based* protocols, where one server (the *leader*) is tasked with proposing and broadcasting *blocks* of new data to be applied to the state machine. Simple implementation of the broadcast requires the leader to send entire blocks to all other servers, creating a network bottleneck at the leader and reducing the system throughput as the number of servers scales. We demonstrate this effect by benchmarking HotStuff, a popular leader-based protocol, and then propose a mitigation based on coding. The key idea is to let the leader encode the block into small *chunks*, and task each server with broadcasting a chunk, thus utilizing the bandwidth of all servers during the process. We apply this idea on HotStuff, and demonstrate a 64% improvement in throughput in a deployment across 9 servers.

CCS CONCEPTS

• **Networks → Application layer protocols**; • **Security and privacy → Distributed systems security**.

KEYWORDS

erasure codes; broadcast; BFT consensus; leader-based consensus

ACM Reference Format:
Ioannis Kaklamanis, Lei Yang, and Mohammad Alizadeh. 2022. Poster: Coded Broadcast for Scalable Leader-Based BFT Consensus. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3548606.3563494

1 INTRODUCTION

Byzantine fault-tolerant state machine replication protocols (*BFT protocols*) allow a group of N servers to agree on an ordered log of *transactions*, where up to f servers may be *Byzantine* and deviate from the protocol arbitrarily. Most deployed BFT protocols are *leader-based* [2, 4, 12], where a *leader* server is periodically elected and responsible to drive the protocol for one or a few *epochs*. In each epoch, the leader proposes a batch of new transactions to be appended to the log, and broadcasts the proposal to all other servers (the *followers*). All servers then exchange votes to collectively commit (or discard) the proposal. A server must download the proposal before voting for it, otherwise it cannot check the validity of the content.[1] To ensure agreement despite Byzantine servers, the protocol must wait for enough followers to cast votes (usually $2f+1$) before committing a proposal. As a result, broadcasting the proposal is on the critical path of the protocol, and upper bounds the throughput of the system.

Existing protocols implement broadcasting in a simple way: the leader sends entire proposals directly to each follower [4, 12]. The problem with this simple mechanism is that the network bandwidth usage of the leader and the followers is *imbalanced*. Specifically, the leader needs to upload $N-1$ copies of the proposal (one for each follower), while a follower downloads only one copy. As a result, the leader, specifically its network uplink, becomes the bottleneck during broadcasting.[2] Furthermore, as N increases, the duration of the broadcast increases proportionally, causing the system throughput to decrease and limiting the ability for such protocols to scale to a large number of servers. We demonstrate this effect by benchmarking HotStuff [12], a popular leader-based BFT protocol. The issue becomes significant with recent interest in public blockchains, which call for deployments of BFT protocols across tens of thousands of nodes over the public internet [6, 8].

To allow leader-based protocols to scale, it is necessary to remove the leader as the communication hub during broadcast. While existing works on collective communication in high-performance computing and task-based distributed systems [7, 13] have studied this problem, their solutions do not apply to BFT protocols because of Byzantine servers. On the other hand, recent progress on balanced BFT reliable broadcast (RBC) and verifiable information dispersal (VID) protocols [1, 3, 11] satisfies the need, but the guarantees from such protocols are too strong. Specifically, they guarantee that all honest servers agree on the content they receive from a broadcast. In comparison, the simple implementation of broadcast in existing protocols does not guarantee agreement[3], suggesting that this property is unnecessary in this context and may introduce extra overhead. Our solution is inspired by balanced RBC and VID protocols, but we carefully design it for leader-based BFT protocols to minimize the communication complexity.

In this paper, we focus on HotStuff [12], a state-of-the-art leader-based BFT protocol for the partially synchronous network model [4]. HotStuff has a design typical of leader-based BFT protocols: the leader broadcasts the latest proposal to all followers at the beginning of each epoch. This design introduces the leader bottleneck discussed above, and we demonstrate its existence using experi-

CCS '22, November 7–11, 2022, Los Angeles, CA, USA.

ACM ISBN 978-1-4503-9450-5/22/11.
https://doi.org/10.1145/3548606.3563494

[1] Some protocols [5, 11] forgo the check on content validity by allowing invalid transactions into the committed log and deterministically removing them at a later point. Still, the protocol must ensure the content is durably stored among the servers [11]. Such protocols are not the focus of this paper.

[2] Assuming that the uplink of the leader does not have $N\times$ higher bandwidth than the downlinks of the followers. Note that the leader is frequently rotated among all servers [12] to prevent censorship, so one cannot designate one server as the leader and provision it with extra bandwidth.

[3] For example, a Byzantine leader may send different proposals to different followers.

ments on a testbed with simulated bandwidth limits.

We then propose a simple mitigation to the bottleneck: an efficient BFT broadcast protocol based on erasure coding. The key idea is to split a b-byte proposal into N *chunks*, each of $O(b/N)$ bytes, and task each server with broadcasting a chunk. Note that the communication cost is balanced across all servers. Although the leader still needs to deliver the N chunks to N servers so that they can start broadcasting, it in total sends $N \times O(b/N) = O(b)$ bytes, a cost that is independent of N.[4] To accommodate Byzantine servers who might ignore or corrupt the chunks they are responsible for broadcasting, the leader adds redundancy among the chunks with an erasure code [9]. In the general case, to accommodate up to f Byzantine servers out of N, our approach requires each node to send/receive $\frac{N-1}{N-f} \times$ the proposal size, and is therefore within 2/3 of optimal (assuming $N \geq 3f+1$). We apply this mitigation on HotStuff, and demonstrate that it alleviates the leader bottleneck and improves the throughput by 64% when $N=9$.

2 BACKGROUND

Erasure codes. Error correcting codes (ECC) are used to transmit data over unreliable communication channels. The sender encodes the message with redundant information, which allows the receiver to correct a limited number of errors and reconstruct the original message. Reed–Solomon (RS) codes [9] are a family of ECC commonly used to correct *erasure* errors, which corrupts a part of the message and make them unavailable to the receiver. (The receiver still knows *which* part is corrupted.) An $\mathrm{RS}(n,k)$ is a Reed-Solomon code which takes a message of k data symbols and adds parity symbols to make a codeword of $n > k$ symbols (chunks). The original message can be recovered from any k-sized subset of the n chunks.

Security model. We assume the same security model as HotStuff [12, §3]. Namely, the network is partially synchronous [4], and servers have direct network connections to each other. Among the $N \geq 3f+1$ servers in the system, at most f are Byzantine.

Experimental setup. Our implementation extends an opensource HotStuff state machine written in Golang [10]. We modify it to replace the direct broadcast with our coded broadcast protocol. The testbed runs on an AWS EC2 `c5.xlarge` VM. We use Linux network namespace to create isolated network stacks for each HotStuff server process, and `qdisc` to limit the ingress and egress bandwidth of each server to 10 Mbps, respectively. We also inject a 100 ms round-trip propagation delay between each pair of nodes. Each block (proposal) is 1 MB and is filled with random data.

3 LEADER BOTTLENECK IN HOTSTUFF

At any given time, HotStuff has one leader coordinating the consensus. Fig. 1 shows the PREPARE phase [12, §4.1] of the protocol, where the leader broadcasts the proposal (block) to all followers. Suppose there are $N \geq 4$ servers, the block size is b bytes, and the ingress/egress bandwidth of each server is c bytes per second. The leader needs to send $(N-1)b$ bytes, while each follower receives b bytes. As a result, the egress bandwidth of the leader is the bottleneck, and it takes $\frac{(N-1)b}{c}$ seconds to finish the broadcast. As N increases, the duration increases linearly, causing the system throughput to decrease.

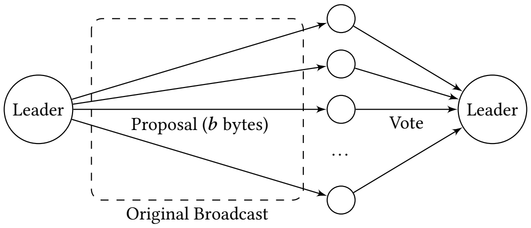

Figure 1: The PREPARE phase of the original HotStuff protocol. The leader sends a copy of the entire proposal to each follower, then waits for the votes.

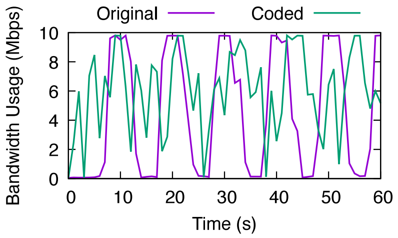

Figure 2: Trace of the egress bandwidth usage at one of the four servers running the original and the coded HotStuff protocol. In the original version, the server's egress bandwidth usage peaks when the server is the leader broadcasting proposed blocks; In the coded version, the server's egress bandwidth usage is more balanced, since bandwidth is being consumed both when the server is the leader as well as when the server forwards chunks as a follower.

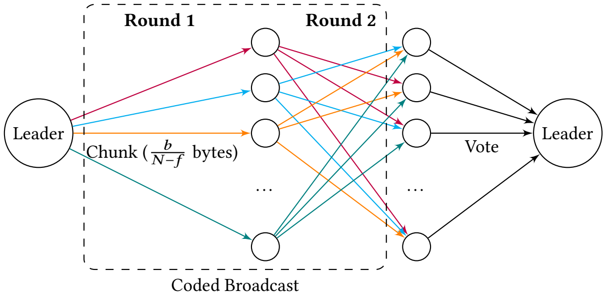

Figure 3: The PREPARE phase of HotStuff, modified to use our coded broadcast protocol (the inner box). In round 1, the leader sends a unique chunk to each follower. In round 2, each follower broadcasts the received chunk to all other followers.

The key issue here is the under-utilization of the followers' bandwidth, caused by the imbalanced network load. Fig. 2 shows the trace of the egress bandwidth usage on one server in a deployment of the original HotStuff protocol for $N=4, f=1$. The server only utilizes its egress bandwidth when it is the leader.

4 RESOLVING THE LEADER BOTTLENECK

Our design modifies the PREPARE phase of the HotStuff protocol, leaving the remaining phases unchanged. Instead of a one-round leader-to-all broadcast, our broadcast protocol has two rounds, as illustrated in Fig. 3. When ready to broadcast a proposal, the leader uses an $\mathrm{RS}(N, N-f)$ code to encode the data into N chunks, with the property that the original proposal can be decoded given any $N-f$

[4] For simplicity, the calculation does not consider cryptographic authenticators (signatures, hashes, etc.). Such costs can be hidden by increasing the size of the proposals, so that the cost of broadcasting stays dominant. Our mitigation does not increase the asymptotic communication cost of HotStuff.

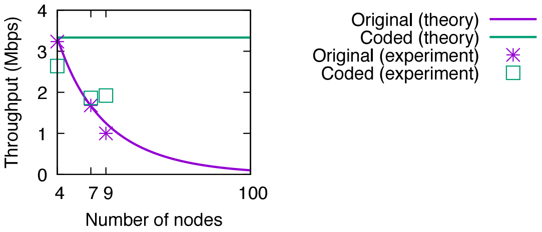

Figure 4: Throughput of the original HotStuff protocol and our modified version with coded broadcast. See §2 for details of the setup. Solid lines are theoretical numbers (see §3 and §4) assuming that block broadcast dominates the overall cost, and ignores other steps of the protocol. Dots are experimental results.

correct chunks. It also computes the Merkle root r of the N chunks, which serves as the unique identifier of the block in subsequent steps of the protocol. In the first round, the leader sends to the i-th server ($1 \leq i \leq N$, including the leader itself) the Merkle root r, the i-th coded chunk, and a Merkle proof of the chunk with regard to r. In the second round, each server broadcasts its chunk and the corresponding Merkle proof to all other servers. When a follower receives a chunk from a peer, it checks the Merkle proof against the root r it received from the leader in the first round, and accepts the chunk if the check passes. If the leader is honest, then at the end of the second round, each honest follower must have accepted $N-f$ correct chunks, so that it can decode the original proposal and proceed with the next steps per the original HotStuff protocol. If the leader is dishonest, then honest followers will fail to reconstruct a block. This is equivalent to not receiving the block during PREPARE in the original HotStuff protocol, and such followers should act accordingly. A formal security proof is ongoing work.

We now explain how coded broadcast alleviates the leader bottleneck. Consider the same setup as in §3. Each chunk is $b/(N-f)$ bytes in size, and the leader in total sends $N-1$ chunks to the followers. In the second round, each server broadcasts the chunk, in total sending $N-1$ chunks. As a result, each round takes $\frac{b}{c} \times \frac{N-1}{N-f}$ seconds, and the duration of the entire broadcast protocol is $\frac{2b(N-1)}{c(N-f)}$. (We assume the blocks are much larger than hashes, thus ignoring the cost of the Merkle roots and proofs.) Recall that $N \geq 3f+1$, so the duration is upper-bounded by $3b/c$, which is independent of N. Fig. 2 shows the trace of the egress bandwidth usage on one server in a deployment of the coded HotStuff protocol for $N=4, f=1$. The server actively uses its egress bandwidth throughout the execution, both when proposing blocks as a leader and when forwarding chunks as a follower.

To demonstrate the effectiveness of coded broadcast, we vary N and compare the throughput of HotStuff using the original and coded broadcast. Results in Fig. 4 show that the throughput of the original HotStuff protocol decreases quickly as we increase N from 4 to 7, closely matching the theoretical calculation in §3. The effectiveness of coded broadcast is weaker than expected in theory. (It reduces the throughput when $N = 4$.) We conjecture that the difference is due to the higher overhead of concurrently sending many small chunks. Still, coded broadcast improves the throughput by 64% when $N = 9$. Improving the implementation of the coded broadcast protocol and closing the gap between the theory and the achieved performance (especially for small N) is ongoing work.

5 DISCUSSION

While we focus on HotStuff in this paper, we believe that coded broadcast can benefit other leader-based BFT protocols, as long as there is a broadcast component on the critical path. Our coded broadcast protocol is identical to the first steps of the RBC protocol in [3, §3.4]. But unlike VID and RBC [1, 11], it provides no guarantee when the leader is dishonest. The lack of the guarantee does not cause problem for leader-based BFT protocols, because they must already consider Byzantine leaders [12]. In other words, the coded broadcast protocol provides identical guarantees as the direct broadcast in original HotStuff.

So far, we have assumed that all servers have the same bandwidth constraint. In practice, however, the available bandwidth may vary across location and time. An interesting future work is to adaptively adjust the chunk size (e.g., by using a rateless erasure code) for each follower considering such variation. Servers with less bandwidth should handle a smaller chunk in order not to become the bottleneck.

Another future improvement is to pipeline the two rounds of coded broadcast. Followers do not need to wait to receive the entire chunk before start forwarding it, thus reducing the broadcast duration by up to 2×.

ACKNOWLEDGEMENT

This work is supported by a gift from the Ethereum Foundation, and National Science Foundation grant CNS-1910676.

REFERENCES

[1] N. Alhaddad, S. Das, S. Duan, L. Ren, M. Varia, Z. Xiang, and H. Zhang. Balanced byzantine reliable broadcast with near-optimal communication and improved computation. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, pages 399–417, 2022.

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

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

[4] M. Castro, B. Liskov, et al. Practical byzantine fault tolerance. In *OsDI*, volume 99, pages 173–186, 1999.

[5] 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.

[6] 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.

[7] R. L. Graham, T. S. Woodall, and J. M. Squyres. Open mpi: A flexible high performance mpi. In *International Conference on Parallel Processing and Applied Mathematics*, pages 228–239. Springer, 2005.

[8] S. K. Kim, Z. Ma, S. Murali, J. Mason, A. Miller, and M. Bailey. Measuring ethereum network peers. In *Proceedings of the Internet Measurement Conference 2018*, pages 91–104, 2018.

[9] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[10] D. Shulyak. go-hotstuff. https://github.com/dshulyak/go-hotstuff.

[11] 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.

[12] 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.

[13] S. Zhuang, Z. Li, D. Zhuo, S. Wang, E. Liang, R. Nishihara, P. Moritz, and I. Stoica. Hoplite: efficient and fault-tolerant collective communication for task-based distributed systems. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 641–656, 2021.