ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

CARRIER

A superlock optimization system for any BFT SMR, evaluated on HotStuff

MASTER'S PROJECT REPORT

David Valaczkai

Supervised by:
Prof. Rachid Guerraoui
Gauthier Voron

Lausanne, 24th June 2022

Acknowledgments

I would like to thank Rachid Guerraoui for the opportunity to work on a consensus-related project. I would like to thank Gauthier Voron for his supervision of the project: the weekly meetings and interesting discussions, the answers to my many questions and his willingness to always help. I would like to thank France Faille for coordinating my project. I would also like to thank everyone else who was involved in coordinating my exchange, both at Imperial College London and at EPFL.

Lausanne, 24th June 2022

Abstract

At the heart of every blockchain lies a consensus protocol that allows the participants to agree on a single common history of transactions. Modern technologies, such as cryptocurrencies, require that the consensus commitee process millions of transactions every second. However, state-of-the-art implementations are unable to reach the throughput demanded by these applications.

We present **CARRIER**, a generic optimization for a wide range of consensus protocols. **CARRIER** is an application that takes the role of a man-in-the-middle between the users submitting transactions to the network and the committee nodes that cooperate to drive the network to agreement.

CARRIER collects transactions into batches, hashes each batch and submits the hashes to the underlying committee for processing. This significantly reduces the size of the workload on the committee as each hash can represent a batch of hundreds of megabytes of size. **CARRIER** maintains the original transactions and produces a decision that is equivalent to the decision of the committee.

This project is the first implementation of the **CARRIER** algorithm. We show that there is indeed a performance bottleneck in state-of-the-art consensus protocols using an implementation of HotStuff, one of the most recently devised algorithms. We analyze the performance of **CARRIER** on HotStuff and discuss the throughput-wise critical parts and limitations of the implementation. Finally, we lay out a roadmap for future work on this project.

Contents

Acknowledgments **1**

Abstract **2**

1 Introduction **5**
1.1 Context 5
1.2 Goals 5

2 Scalability of HotStuff **7**
2.1 Experiment setup 7
2.2 Results 8

3 Carrier Design **10**
3.1 Workflow 10
3.2 Message complexities 11

4 Implementation **13**
4.1 Introduction 13
4.2 Highlights 13
4.2.1 General structure 13
4.2.2 Cryptography and key distribution 13
4.2.3 Comparing encodings 14
4.3 Filtering already committed hashes 15
4.4 Superblock data structure 15

5 Evaluation **17**
5.1 Setup 17
5.2 Benchmark results 17
5.2.1 Robustness - Figure 5.1 17
5.2.2 Latency - Figure 5.2 18
5.2.3 Different batch sizes - Figure 5.3 19
5.3 Challenges 20

6 Roadmap for Carrier **21**
6.1 Aggregate signatures 21
6.2 Threshold signatures 22

6.3 Reliable or unreliable connection . . . 22
6.4 Improved general parallelism . . . 22
6.5 Parallel processing of nested SMR decisions . . . 23
6.6 Clearing the value store . . . 23
6.7 Routing messages to self . . . 23
6.8 Timeout on batching . . . 23
6.9 Flexible transaction sizes . . . 23
6.10 Logging . . . 24
6.11 Benchmarking framework . . . 24

7 Conclusion **25**

Bibliography **26**

A Carrier by Gauthier Voron **27**

Chapter 1

Introduction

1.1 Context

Computing systems that rely on the coordination of multiple computers, often referred to as nodes or replicas [4, 8], are becoming increasingly popular due to their successful and widespread applications such as blockchains and cryptocurrencies. A crucial property of these systems is that participating nodes all agree on a single common state. Nodes can be described as state machines, meaning that they perform operations that take them from one state to another.

In the context of cryptocurrencies, a state transition occurs when a transaction takes place between users. Nodes in the system are initially in a state where they are not aware of said transaction. Then, they follow a consensus protocol to transition to a state where they consider the transaction to be executed. Since the goal is to replicate the same state among all nodes, this is also often referred to as State Machine Replication - SMR.

Although there are important differences in complexity [1], we can consider SMR and consensus to be practically equivalent in this report.

In many real world scenarios it is also important that these systems be able to tolerate faulty or even maliciously behaving replicas. Byzantine Fault Tolerant - BFT systems in particular are able to make progress even when several replicas are malfunctioning. If there are n nodes in a BFT system, it is able to tolerate up to f faults, where $n > 3f$. The name comes from the Byzantine General's Problem.

1.2 Goals

Show scalability problem Several BFT SMR algorithms have been devised [3, 8]. Although it is difficult to draw direct comparisons between the individual implementations due to the different methods that have been used to evaluate their performances, our hypothesis is that even that state-of-the-art protocols, which present improvements over their predecessors, have bottlenecks in their performance that limit their scalability. To investigate this claim, we

discuss the results of benchmarking an implementation of HotStuff, one of the most recent BFT SMR protocols.

Present Carrier Carrier is a superblock optimization algorithm that sits on top of any BFT SMR. We present the Carrier algorithm and explain how it improves transaction throughput for the consensus system. We highlight some aspects of our implementation, aiming to explain some choices made along the way and to serve as a manual for someone looking to continue the project.

Evaluate throughput We discuss the results of benchmarking Carrier on an implementation of a state-of-the-art protocol, HotStuff. We compare the results against running the HotStuff benchmarks without Carrier. We focus mainly on throughput, which we measure in processed transactions per second.

Originally, the goal of the project was to show that HotStuff with Carrier is able to process more transactions per second than plain HotStuff. While this implementation requires some more work before it can provide the significantly improved throughput we had in mind, we have achieved some promising early results.

Our experiments show that larger committee sizes have a worse impact on Carrier's performance than on plain HotStuff's, but Carrier manages to stay more robust under high loads.

Roadmap We investigate our implementation of Carrier and take a critical look to find bottlenecks that limit the performance of Carrier. We present ideas on improvements that would make our implementation provide a higher throughput.

We conclude our work by summarizing our findings and providing guidance on how this project may be continued.

Chapter 2

Scalability of HotStuff

We used an implementation of HotStuff written in Rust by Alberto Sonnino [6]. HotStuff [8] is a leader-based BFT SMR protocol. One of the key contributions of HotStuff is that it achieves linear - $O(n)$ communication complexity in the number of replicas in the best case and quadratic - $O(n^2)$ in the worse-case scenario of the current leaders failing one after the other. Previous algorithms performed even worse than this: PBFT [3] sends out $O(n^3)$ authenticators if a single leader malfunctions, and reaches $O(n^4)$ in the cascading leader failure case. Algorithms that use threshold signature reach complexities of $O(n^2)$ and $O(n^3)$ respectively.

Hence, HotStuff is a considerable leap forward in terms of performance. Despite this, its throughput in the optimal scenario is a few hundred thousand transactions per second, or a few MB per second. This has been confirmed by experiments in the original paper of HotStuff [7]. There remains a need for high performance libraries that can be used in practice [2].

2.1 Experiment setup

We ran benchmarks on Amazon's AWS EC2 cloud computing engine. All experiments were run on c5.xlarge Virtual Machines. These VMs are equipped with 4 vCPUs of the Intel Xeon Platinum 8000 series at up to 3.6GHz with Turbo Boost, 8GiB of RAM and provide a network bandwidth up to 10Gbps. Our network usage remains below the maximum bandwidth.

We hosted one VM per replica. We also ran one client per replica on the same VM. Despite running close to each other, clients sent transactions to the public IPs of the replicas, so they still incurred real network delay. We were also more interested in the throughput and latency of the consensus itself, rather than network delays between clients and replicas. The transaction rate was divided equally between all clients.

We ran our experiments spread across four datacenters: Frankfurt (eu-central-1), North California (us-west-1), Cape Town (af-south-1) and Seoul (ap-northeast-2). We aimed for high geographic spread when selecting the regions to mimic real-life scenarios as closely as possible. Using four regions was convenient as many of our tests were run on the 4 replica setup and we

could place one replica in each region.

2.2 Results

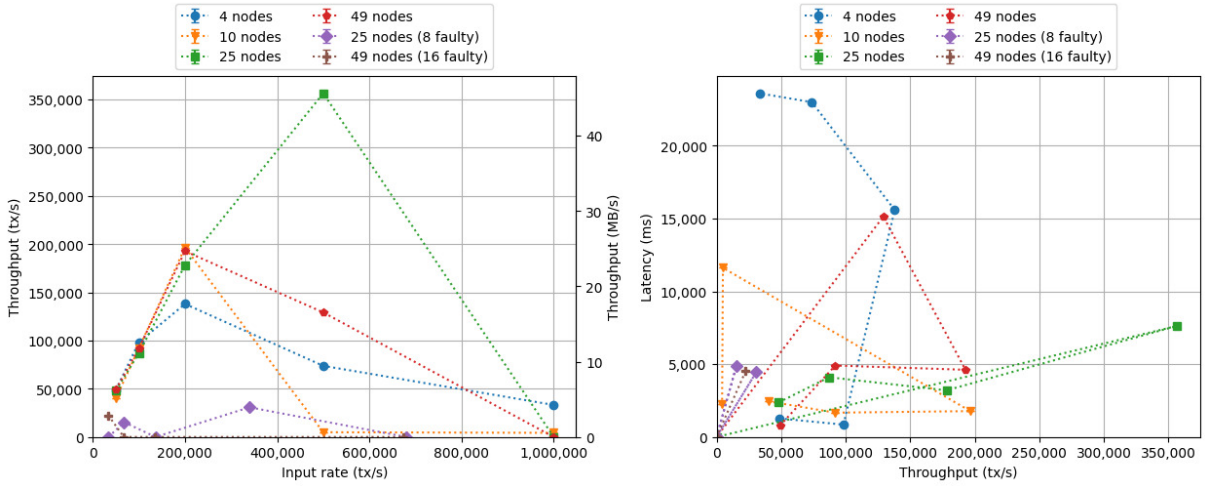

Figure 2.1: Robustness (left) and latency (right) of HotStuff

According to Sonnino's own experiments [6], the practical limit of the system on a large geographic-spread setup is around 100,000 transactions per second. This is because the mempool becomes unstable if it receives more than this many transactions.

Our experiment agrees with Sonnino's results. Figure 2.1 shows that HotStuff is able to keep up with input rates up to around 200,000 tx/s (with one exception), and throughput starts to drop once input rate increases beyond this.

When running with $(n-1)/3$ faulty nodes, the throughput penalty is huge. The [25/8] node setup managed to achieve a maximal throughput of around 75,000 tx/s, while the [49] node setup is barely able to sustain any throughput over such a geographically widespread system. We believe this is because nodes become so isolated that they have a hard time determining whether a replica is faulty and not replying or whether messages are just delivered slowly by the network.

Locally, we ran the system with an input rate of 10 million transactions per second to stress test it under extreme loads. Though the system did not crash, the nodes were unable to reach consensus on any block and yielded a throughput of 0 TPS/BPS, as expected.

In reality, the simple clients we used were unable to sustain such a high input rate. The actual number of sent transactions varied greatly depending on the system, between a few hundred thousand to just over a million. We found these numbers catching client transactions using a netcat listener and counting the bytes that came through. We did not test whether the bottleneck was the netcat listener, as we assumed it would be able to receive the incoming bytes fast enough. We also tested the speeds of 2 additional clients to the Rust one: one written in C and one written in Go. They produced input rates in similar magnitudes.

The latency of the system varied greatly, but was consistently higher as the input rate increased. When the system became completely clogged, latency is represented as 0 since no transactions went through the system, as is the case with high numbers of faulty replicas. Perhaps infinity would have been a better default value.

Hence, we can conclude that there is indeed a practical limit in the order of 10^5 on the number of transactions the system can process each second.

Chapter 3

Carrier Design

The Carrier algorithm is the work of Gauthier Voron and the original work is attached as Appendix A.

Carrier is a superblock optimization system that sits on top of any BFT SMR. The only requirement is that the SMR be able to reply with the exact same data that Carrier proposes to it. Likewise, Carrier works with any client that is able to send transactions. This makes Carrier very versatile.

Carrier refers to the entire system. Carrier process refers to a single instance of the Carrier algorithm. The Carrier system consists of several cooperating Carrier processes. Carrier consensus is the mechanism of Carrier processes reaching a decision and is not to be confused with the consensus of the nested SMR.

3.1 Workflow

Before Carrier processes are booted, it is necessary to generate a cryptographic keypair for each Carrier process. Each Carrier process is given all public keys so that they can verify authenticity of messages.

Phase 1 Carrier processes sit between the client and the replica. The client sends transactions to a Carrier process rather than directly to the node. The Carrier process buffers transactions up to a pre-defined threshold. Once the threshold is reached, the Carrier process initiates a new instance of Carrier consensus by broadcasting an InitMessage to other Carrier processes. The InitMessage contains the ID of the sender and all of the transactions up to the threshold.

When a Carrier process receives an InitMessage, it hashes its contents and signs the hash using its private key. It then broadcasts its ID, the hash and the signature to all other Carrier processes. Finally it stores the hash and all the transactions of the InitMessage in a designated store V.

Upon receiving an EchoMessage, the Carrier process verifies that the signature is valid. The

Carrier process keeps a map from each hash it has seen to list of signatures associated with that hash in a designated signature store S. It appends the newly received signature to the list associated with the hash inside the EchoMessage. Once a hash has gathered $f + 1$ signatures, the Carrier process inserts the hash and its signatures into the current superblock. Messages with invalid signatures are simply disregarded.

Once the superblock reaches a length of $2f + 1$, the Carrier process proposes it to the nested SMR protocol.

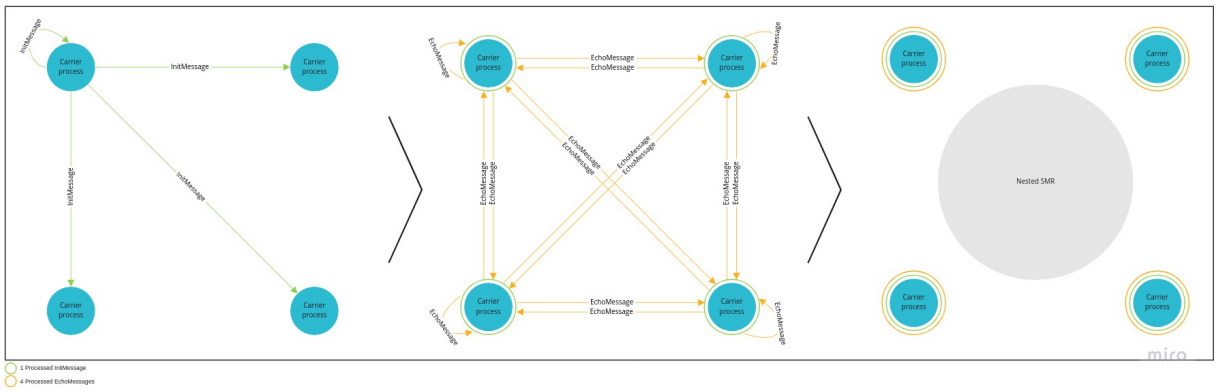

Figure 3.1: Phase 1 of the Carrier algorithm

Phase 2 The next phase begins when the nested SMR replies with a superblock that it has decided. The Carrier process looks at every hash and its signatures, verifies the signatures and checks that there are at least $f + 1$ signatures per hash. If the Carrier process has an entry for the hash in its V store, its adds the hash and its transactions to a store for accepted hashes. If it does not have an entry for the hash, it broadcasts a RequestMessage to other Carrier processes.

Eventually it receives a ResolveMessage from some other Carrier processes which will contains the transactions associated with the queried hash. Once all pending RequestMessages have been resolved meaning that the Carrier process has all transactions associated with each hash in the superblock, the Carrier process decides the superblock and the Carrier consensus instance terminates.

3.2 Message complexities

The message size complexities are the following:

InitMessage: $O(tk)$, where $t = threshold,\ k = transaction\ size$

EchoMessage: $O(1)$

RequestMessage: $O(1)$

ResolveMessage: $O(tk)$, where $t = threshold,\ k = transaction\ size$

EchoMessages and RequestMessages come with little communication overhead. ResolveMes-

sages are sent when a previously faulty Carrier process needs to catch up with the others, and their occurrence can vary greatly depending on the scenario. The critical bottleneck is the InitMessage, which must be sent with every Carrier consensus instance. Optimally we want the threshold to be as large as possible to maximize the potential of Carrier. However, we cannot allow the product $threshold * transaction\ size$ to become too large, because that would produce InitMessages so large that we would hit the limits of network propagation very soon.

Superblock: $O(n^2)$, where $n = number\ of\ Carrier\ processes$

The overhead of the superblock scales quadratically with the number of Carrier processes, which is usually equal to the number of replicas. Its size complexity can be reduced to $O(n)$ using threshold signatures. We discuss this in Chapter 6.

Chapter 4

Implementation

4.1 Introduction

We implemented Carrier in Go 1.8. Our justification behind using Go was its easy-to-use parallelism and relative simplicity for someone experience in C/Java. Although we had some previous experience in Go, we encountered a lot of new concepts during this project. In particular, building a scalable system proved to be challenging, because often a program would run fine on a small committee of size 4, but suffer from unbalanced loads or unreliable connections on larger systems.

Anything not detailed here is intended to be communicated via comments in the code.

4.2 Highlights

4.2.1 General structure

We implemented Carrier with simplicity and minimalism in mind. We deleted extraneous code often when more suble and elegant solutions were devised. The implementation of Carrier follows the algorithm written in pseudocode in the original paper as closely as possible.

Carrier processes start listening on 3 ports once booted: 1 for client transactions, 1 for communication with other Carrier processes and 1 for nested SMR decisions.

Currently, Carrier processes print out a log that contains the hashes of the superblock upon committing a decision. We wrote a skeleton function called decide() where further processing can be introduced.

4.2.2 Cryptography and key distribution

Before Carrier processes can start their normal operation, we have to generate a keypair for each process and distribute the public keys to all processes. This soon became a tedious task to perform manually, so we composed a config generator that automatically handles the

key generation and distribution, and assigns and distributed ports for each processes. First, a params file must be created. The Python scripts in the /scripts directory can be used to generate example params files. The config generator takes a params file and returns n Carrier configuration files, based on its settings. To start up a Carrier process, one must pass it one of these fresh config files.

We used the the bdn module provided by the kyber repository for Cryptography [5]. The bdn package provides threshold signatures, but this remains to be implemented in our project.

4.2.3 Comparing encodings

Initial design

When designing our implementation, we had to make a choice in what encoding we would use when serializing data to be able to send it over the network. Initially, we had a double layered encoding system:

- **Messages between Carrier processes** were double encoded to facilitate the deserializing of messages. First, the message was serialized using JSON encoding. Then, a wrapper around this message was created, that contained a Type string metadata to enable us to decode it correctly on the receiving end. The Type string, with the payload attached, was encoded a second time using Go's Gob package and finally sent to its destination. The receiving end first used a Gob decoder to decode the raw bytes into the Type string and the JSON marshalled payload, then the payload was deserialized using the JSON decoder of the correct message type.

- **Messages between Carrier and the nested SMR** were encoded and decoded only once, using the JSON codec. The superblock was the only type of message in this scenario so we did not need the Type string metadata.

The justification behind using different types of encodings was that Go's Gob package proved to be faster than JSON. Hence, we used it for messages that were sent exclusively between processes written in Go. On the other hand, we used JSON for the nested SMR messages in case we wanted to deserialize them at some point in a language other than Go.

Improved design

It became clear that the initial design was overly complex and slow. We wanted to avoid double encoding messages between Carriers, but we still needed a way to decode message into the correct struct. We had two choices: either try to decode the message into each possible Type one by one until one succeeds, or write our own binary encoding and convey the message type. We opted for the latter.

We created a tiny codec for the messages in our system, called BinaryEncoder. It supports 3 types of data: Carrier messages, superblocks and plain byte arrays. It can be easily extended to support other datatypes if necessary.

BinaryEncoder adds the only minimum necessary overhead to send messages and is around 50% faster than the previously used encoding system. To illustrate: InitMessages only have 17 bytes of overhead with our encoder.

We compared our BinaryEncoder to the JSON+Gob encoder with a few minimalistic benchmarks. We ran the benchmark for 10 seconds. On average, JSON+Gob was able to encode 49,701 InitMessages per second, contrasted with BinaryEncoders throughput of 60,166 InitMessages per second. We concluded that our BinaryEncoder provides a beneficial throughput boost and its extensibility and customizability is likely to be useful in the future.

One considerable limitation of our encoder is that it does not support flexible transaction sizes yet.

4.3 Filtering already committed hashes

The algorithm ensures that all Carrier processes propose a superblock to the nested SMR. When messages are delivered in order, all Carrier processes will propose the same superblock to the nested SMR. This is necessary to ensure that despite the faulty processes, the SMR still receives a proposal from at least one correct Carrier process. In this case, the nested SMR will decide on multiple, possibly all of the proposed values individually, effectively cloning a superblock $0(n)$ times.

The algorithm ensures that all Carrier processes propose a superblock to the nested SMR. However, in the scenario when messages arrive out of order, two Carrier processes might propose superblocks that include different hashes. The order of receiving and processing of EchoMessages determiens which hashes will end up in a given superblock. It is possible that Carrier process A receives enough signatures for hash a first, while Carrier process B receives enough signatures for hash b first. They will put the fulfilled hashes into the superblock summary, and eventually propose the different superblocks to the consensus.

To overcome this issue, we filter the already committed hashes when making a decision.

4.4 Superblock data structure

Since the order of hashes cannot be uniquely decided, we decided to use a hashmap instead of a list to represent the superblock. Maps have $O(1)$ lookup time, compared to up to $O(n)$ on lists.

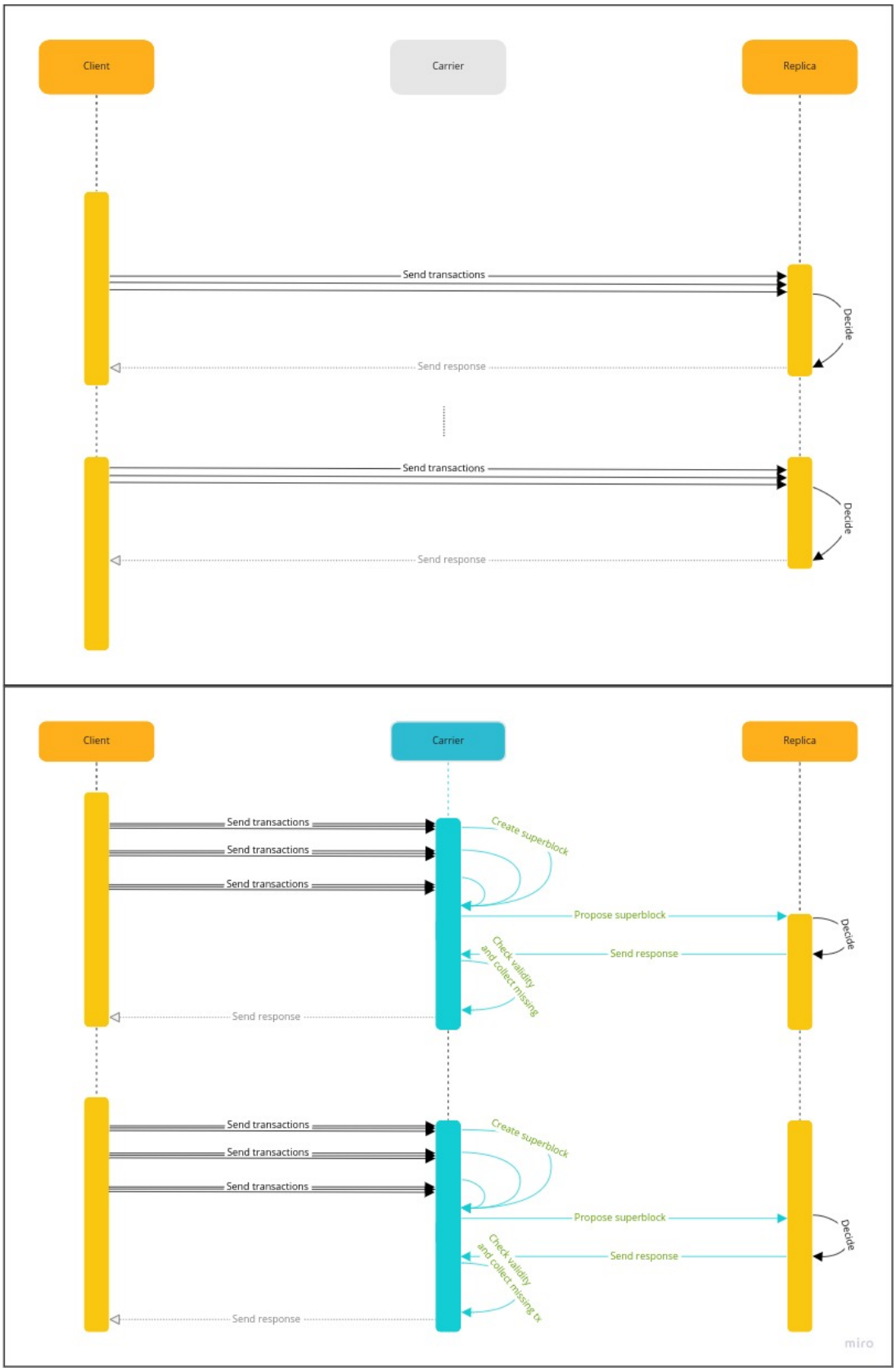

Figure 4.1: System architecture before (top) and after (bottom) Carrier. Light blue/green shows parts that we implemented during this project. Light grey represent parts that are not yet implemented and would be viable extensions to the system.

Chapter 5

Evaluation

5.1 Setup

To evaluate the performance of Carrier we used the same setup as for raw HotStuff. Additionally, we used an init-threshold value of 10,000 for our comparison benchmarks.

We extended and customized the Python scripts provided in the HotStuff repo. We added extra settings for Carrier and option to enable and disable runs with Carrier.

5.2 Benchmark results

We ran benchmarks on a range of committee sizes: [4, 10, 25, 49] nodes to see how the programs handle scaling up to many nodes. We set the transaction size to 128 bytes to ensure that Carrier would not run out of memory before the end of the benchmark. We gradually scaled transaction rate through [50,000; 100,000; 200,000; 500,000; 1,000,000] to stress-test both implementations. We ran each setup 3 times to smooth out anomalies. The figure shows the average throughput valus of the 3 runs. Each run lasted 60 seconds.

5.2.1 Robustness - Figure 5.1

In our first experiment, we wanted to find out whether HotStuff with Carrier can outperform plain HotStuff in transaction throughput.

At lower rates, plain HotStuff was able to process almost every incoming transaction at all committee sizes. At 200,000 transactions per second, interestingly, the smallest 4 node committee dropped off first. The reason for this could be that messages sent over the network got delayed due to the enormous distances they had to travel. Perhaps running the experiment more times would have evened out these results slightly.

Notably, committees of size [10, 25, 49] had 0 throughput at a rate of 1 million incoming transactions per second. As Sonnino himself noted, the mempool of his implementation becomes unstable when handling this many transactions per second. Replicas fall into a

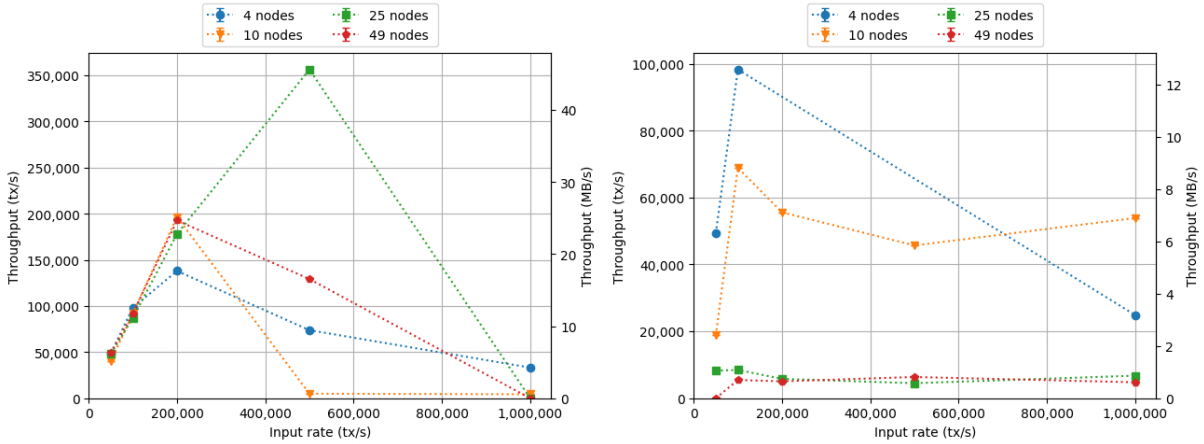

Figure 5.1: Robustness - plain HotStuff (left) vs HotStuff with Carrier (right)

bottomless pit of trying to catch up to the rest of the consensus, which is only ever pulling further ahead because of the gigantic load of new transactions.

In contrast, Carrier seems to be heavily impacted by the size of the committee. 4 Carrier processes were able to keep up with up to 100,000 tx/s, but larger committees dropped throughput to only around 5,000 tx/s at the same input rate. Judging by how plain HotStuff was able to handle these loads without problems, this suggests that our implementation is heavily CPU bound. We believe that this is caused by executing many calls to slow cryptographic operations, like Signing and Verifying. We suggest ideas on how to improve this in the next chapter.

At 1,000,000 tx/s Carrier is able to maintain a higher throughput at all committee sizes. Interestingly, it performs best on a 10 node system, processing around 55,000 tx/s. We think this is due to noise and a [4] node system would have performed best normally On [25,49] node systems, performance seems to be maintained around 8,000 tx/s throughout no matter the input rate. Notably, Carrier makes progress even under 1 million new transactions per second, contrasted with plain HotStuff, which does not. We attribute this to the superblock optimization aspect of Carrier, which enables it to process incoming transactions very quickly as executes very few operations on the transactions themselves. The consensus is propelled by the hashes of the transactions.

5.2.2 Latency - Figure 5.2

We were interested in how long it takes for a transaction to get committed under each system.

Latencies for both system are very scattered and it is not easy to spot a pattern emerging. This graph should be read by starting with the bottom left point on each line and then following the dotted connection. HotStuff produced low latencies for a committee of [4] and input rates of [50,000; 100,00] tx/s. Generally, high latencies correspond to high input rates. In these cases the replicas became overloaded and were unable to process transactions as quickly as they were fed into the system. This is similar for Carrier.

Notably, Carrier operated with much higher latencies than plain HotStuff. Latency seems to scale linearly with committee size: [4] nodes produced a latency of around 1s, while 10 nodes resulted in a delay of around 5s when transaction rate was set to 50,000 tx/s. They both scaled up to around 35s under the highest input rates. In general, HotStuff with Carrier yielded 5-10x greater latencies than plain HotStuff. We explain this with HotStuff needing to do further processing on the superblocks committed by the nodes.

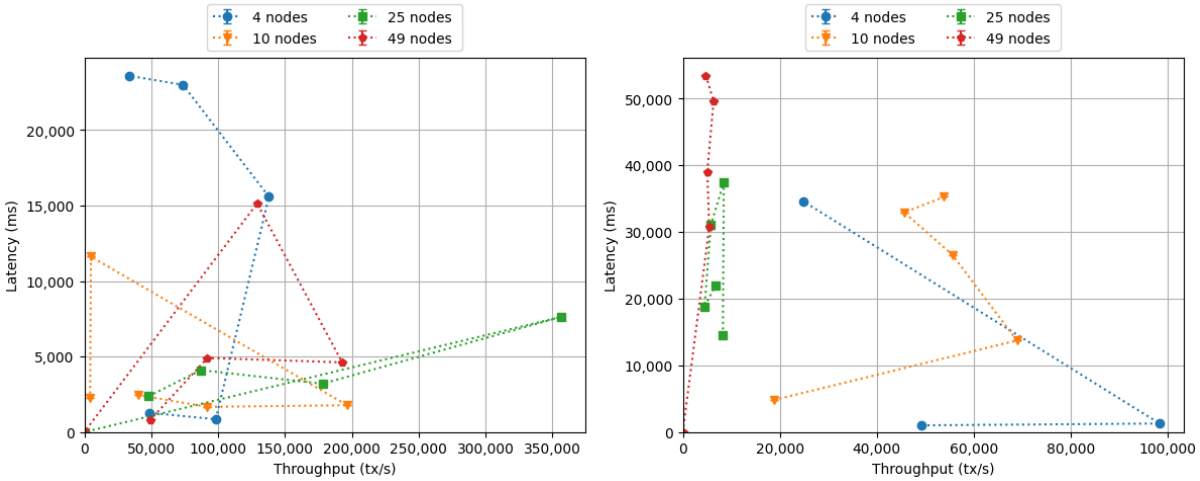

Figure 5.2: Latency - plain HotStuff (left) vs HotStuff with Carrier (right)

5.2.3 Different batch sizes - Figure 5.3

In this experiment, we were interested in how batch size (init-threshold) affects performance. We fixed the committee size to [4] as it was less troublesome to run benchmarks on smaller committees.

On lower input rates, lower batch sizes consistently produced higher throughput. However, the decrease in throughput is only logarithmic of the increase in threshold: at 100,000 tx/s, batch size 10,000 yielded almost maximal throughout, while a 10x larger batch size of 100,000 resulted in a throughput of 55,000 tx/s, approximately half of the smaller batch's.

In contrast, for higher input rates, using a batch size of 100,000 proved to be the most consistent. For an input rate of 1,000,000 tx/s, this batch size yielded the highest throughput at 42,000 tx/s. We hypothesize that using a balanced batch size between the two extremes is beneficial because fewer consensus instances are initiated than with the smaller batch size. At the same time, InitMessages did not start to become so large that they introduce a network bottleneck, as we think happens when the batch size is set to 1,000,000. In fact, InitMessages became so large and so frequent at the highest input rate that system seems to have run out of memory and produced a throughput of 0 tx/s.

Latencies are once again very scattered. We now hypothesize that this is not an error of the previous experiment, but rather the consequence of the unpredictability of the network at such large distances. Again, the general trend is that while the system remained stable, increasing the input rate produced slightly higher latencies and higher throughput until the capacity of

the system was reached, and then throughput dropped back while latencies jumped up high. We believe this is the result of the system hitting its capacity. In this experiment, Carrier was also memory bound in addition to being CPU bound.

Ideally, we should run more local experiments to compare results and confirm our hypothesis.

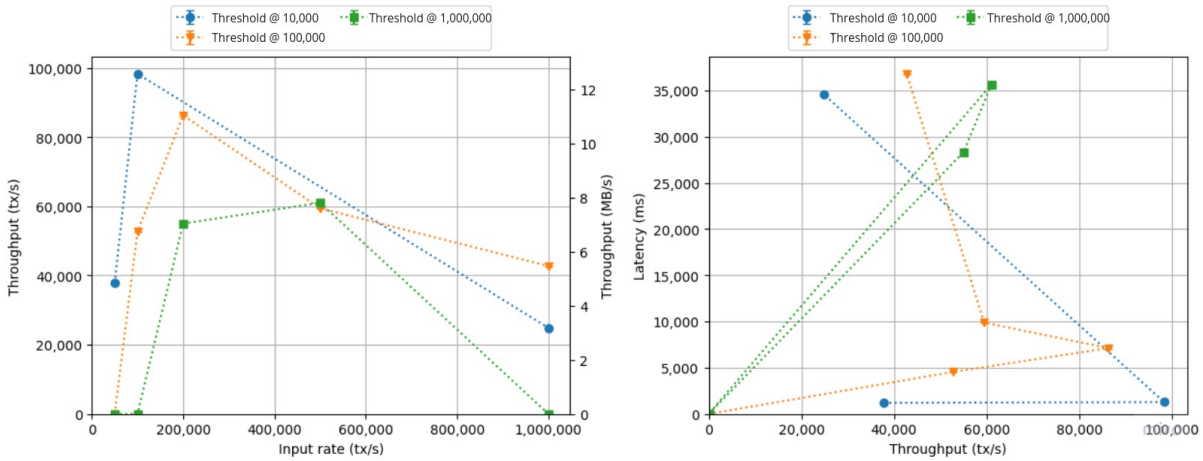

Figure 5.3: Robustness - comparing init-threshold value

5.3 Challenges

Running the benchmarks with the provided fabric scripts proved to be error-prone with large committee sizes. Often, the connection timed out or abruptly disconnected during benchmarks, so we had to repeat some runs on multiple occasions to collect meaningful results.

Being unfamiliar with the scripts and Python also proved to be a challenge. For example, there was an issue with the scripts not closing the connections after executing commands. Since unix uses file handles for TCP connections, eventually the open file limit was hit on the system and the scripts crashed. It took some debugging and inspiration from HotCrypto to solve this issue [0]. The SSH disconnection issue remains to be solved.

Chapter 6

Roadmap for Carrier

Despite our best efforts and the promising results, there remains room for improvement for Carrier. Carrier performance is mainly bound by CPU at the time of writing of this report. We had a number of ideas we did not have time to implement during the project:

6.1 Aggregate signatures

One of the major bottlenecks in our implementation is bls signing and verifying. Each call to the Sign() and Verify() functions is an expensive operation. Signing happens during (1) the processing of InitMessages, meaning that we call Sign $O(n)$ times for every InitMessage that gets sent. Verifying takes place (2) when handling EchoMessages - $O(n^2)$ times per InitMessage - and (3) when handling decisions of the nested SMR. While (1) and (2) are hard to improve, (3) presents the possibility of using aggregated signatures.

When a superblock summary is first composed, we could aggregate the signatures and add a bitmap to the superblock summary that represents which nodes' signatures are included in the aggregated signature. This would also reduce the message's size complexity by a constant factor. This is not insignificant, as it would enable us to run Carrier of hundreds of nodes in practice before the messages sizes would become too large.

During verification, we would look up the corresponding public keys of the Carrier processes identified by the bitmap, aggregate them, and use the aggregated public key to verify the aggregated signature. This would result in a single call to Verify(), improving runtime complexity from $O(n)$ to $O(1)$ assuming that aggregation is faster than verification.

Sending only the aggregated public key in the message would introduce a safety risk. An attacker could produce their own keypair, sign the message with their private key and forward their public key, disguising it as an aggregated one. There is no way for the Carrier process to verify that the public key is an approved one without the information contained in the bitmap.

6.2 Threshold signatures

Taking the previous idea one step further, using threshold signatures would produce another improvement in terms of space complexity. This would reduce the complexity from $O(n^2)$ to $O(n)$ if the signature and the combined public key are both sent in the same message. However, the information on which Carrier processes' signatures are in the threshold signature must be omitted, because even a bitmap would reintroduce a factor of n into the complexity.

Using a bitmap rather than full signatures, though only reduces the size complexity by a constant factor, is still a considerable improvement as it significantly delays the quadratic explosion of the message size when adding new replicas. In practice, this means that the system would be viable with hundreds and potentially thousands of nodes.

The advantage of threshold signatures over aggregated signatures is that they can be verified with only knowing the number of Carrier processes that signed it. It is not necessary to know exactly which process' signatures are in the threshold signature.

6.3 Reliable or unreliable connection

Currently, if a disconnection occurs between two Carrier processes or a Carrier process and its nested SMR, it does not try to reconnect. It would improve this implementation if there were a reliable connection wrapper around net.Conn that would initiate a reconnection in case anything goes wrong to make Carrier a more robust system.

Alternatively, we could switch from TCP sockets to UDP sockets. UDP introduces no overhead setup time and does not worry about disconnects. EchoMessages already act as some kind of ACK for receiving InitMessages, which could be taken advantage of.

6.4 Improved general parallelism

In the current implementation, Carrier processes start accepting Client requests before connection with other Carrier processes and the nested SMR replica are established. A broadcast worker pool of n threads is set up to process network I/O operations in parallel.

The workers block on send operations until the underlying connection comes alive. However, with the current configuration there is no timeout on the blocking and eventually all workers will block on send operations to the dead replicas. It is important to add a timeout to avoid this.

Taking the previous idea one step further, it would be beneficial to create a framework for parallelism in Carrier. Worker pool are a great tool to balance loads and spend efforts on computationally expensive tasks. We imagined that there would be a central, potentially flexible worker pool and a queue for computationally expensive or slow jobs. Blocking jobs would also have a timeout associated with them. Threads would submit expensive tasks to the worker pool to balance processing loads. Each worker would pick up a job, complete it, and then pick up another job. Operations that time out would be resubmitted to the end of

the queue and workers would be free to process other jobs, ensuring efficient parallelism in the system and preventing a complete deadlock. We believe that Carrier would be able to run faster with this setup.

6.5 Parallel processing of nested SMR decisions

Currently, a bottleneck is imposed on the processing of decisions from the nested SMR. There is only one store for accepted hashes. To process decisions in parallel, we would need to create a distinct accepted hash store for each superblock decided by the nested SMR. The store needs to live until all corresponding RequestMessages have been answered and a decision can be reached. The store must also only contain the hashes of the corresponding superblock, and no others.

6.6 Clearing the value store

Past values are currently never cleared from a Carrier processes' value store. This data adds up and can cause a Carrier process to run out of memory. For example, if we have 1MB transactions and we receive 100,000 transactions per second, we hit 1GB of stored data in only 10 seconds. The value store should be cleared periodically to prevent this from happenning.

6.7 Routing messages to self

Currently, all messages are broadcast over the network. It would be a nice improvement if Carrier processes sent messages addressed to themselves via a channel, rather than through TCP sockets.

6.8 Timeout on batching

Although not directly a performance bottleneck, currently Carrier processes will only initiate a consensus instance once the batch threshold of transactions have been hit. If the clients send transactions very infrequently, this can result in prolonged periods of idling from the nested SMR where no decision is made. It would be more ideal to have a timeout on the batching of transactions and initiate a Carrier consensus after this timeout is reached even if fewer transactions than the init-threshold have been collected.

6.9 Flexible transaction sizes

Carrier currently only supports fixed-size transactions, but this is only a limitation of the implementation. If each incoming transaction is framed with the length of the transaction, it would be possible to extend Carrier to support transactions of varying sizes. If implemented, one must pay attention to update the serialization of InitMessages and ResolveMessages, as these contain individual transactions in their payload and currently rely on the initially configured fixed transaction size.

6.10 Logging

Although zerolog is a lightweight library, logging slows down performance somewhat as does expensive I/O operations. Running Carrier with logging turned off can improve performance. Logging can be disabled in the configuration.

6.11 Benchmarking framework

Carrier is intended to work with any BFT SMR. The nested consensus acts as a "black box". It follows that we would like to test Carrier with other implementations of HotStuff, as well as implementations of other consensus algorithms.

We had the idea to create a benchmarking framework for Carrier. The framework would wrap around Carrier, and expose two interfaces: one for plugging in a client, and one for plugging in a nested SMR. Since Carrier requires the nested SMR to reply with some value, these programs would have to be adapted in some way to expose a response to Carrier. The benchmarking framework would take care of measuring the performance of the system with and without Carrier and comparing the figures.

Chapter 7

Conclusion

Our goal was to implement and benchmark a new system that helps SMRs under high loads. We have shown results of experiments that draw attention to the limitations of state-of-the-art consensus protocols. We have presented the design of the new system, and explained how it operates. We have detailed the most important aspects of our implementation. We have evaluated Carrier on HotStuff and presented the most critical findings. Finally, we proposed ideas for the future of Carrier which hopefully serves as a manual for anyone wishing to continue this project.

If we were to attempt this project again, we would make some different choices. Looking back with the experience we have gained on building a scalable system, we would aim to identify possible bottlenecks ahead of time and design robust solutions from day 0. We had parallelism in mind from the start, but a more structured system of worker threads would have improved the performance of Carrier even more.

We put our best efforts into adapting the provided Python scripts for our needs and running the benchmarks with them. Benchmarking large distributed systems proved to be challenging.

While Carrier has room for improvement, our initial implementation has shown promising results. We were able to see that there is real in the superblock optimization approach, ready to be harvested.

Bibliography

[1] Karolos Antoniadis, Rachid Guerraoui, Dahlia Malkhi, and Dragos-Adrian Seredinschi. "State Machine Replication is More Expensive than Consensus". In: 2018. URL: http://infoscience.epfl.ch/record/256238.

[2] Alysson Bessani, João Sousa, and Eduardo E. P. Alchieri. "State Machine Replication for the Masses with BFT-SMART". In.

[3] Miguel Castro and Barbara Liskov. "Practical Byzantine Fault Tolerance". In: 1999. URL: https://pmg.csail.mit.edu/papers/osdi99.pdf.

[4] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. "DBFT: Efficient Leaderless Byzantine Consensus and its Application to Blockchains". In: *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*. 2018, pp. 1–8. DOI: 10.1109/NCA.2018.8548057.

[0] Yanick Paulo-Amaro. "Minimal implementation of a smart-contract system based on HotStuff consensus and the Move virtual machine". In: URL: https://github.com/yanickpauloamaro/hotcrypto.

[6] Alberto Sonnino. "Implementation of the HotStuff consensus protocol". In: URL: https://github.com/asonnino/hotstuff.

[5] David Wolinsky and Bryan Ford. "Advanced crypto library for the Go language". In.

[7] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. "HotStuff: BFT Consensus in the Lens of Blockchain". In: 2018-2019. URL: https://arxiv.org/abs/1803.05069.

[8] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. "HotStuff: BFT Consensus with Linearity and Responsiveness". In: 2018-2019. URL: https://dl.acm.org/doi/10.1145/3293611.3331591.

Appendix A

Carrier by Gauthier Voron

0.1 Description of the Algorithm

In this section we present Carrier, a generic proposal transmission system. We remind that the goal of Carrier is to implement the super-block optimization atop of any BFT SMR. More precisely, Carrier concatenates $n - t$ proposals and submits them for consensus, effectively multiplying each consensus operation by $n - t$ times. In the implementation, a proposal is a batch of client operations. When enough clients are submitting concurrently, a proposal is several megabytes large.

A Carrier replica starts by broadcasting a proposal (line 7 in Algo 0.1). Eventually, every correct replica receives this proposal, hash it and sign it (lines 9–10) and broadcast a confirmation that they received this proposal (line 11). After receiving $t + 1$ correctly signed hash for a given proposal, a correct replica append an entry containing the hash and the $t + 1$ signatures to a super-block summary P (lines 14–17). Eventually, the super-block summary is filled with $n - t$ of these entries. The replica then proposes the super-block summary to the nested black-box SMR.

The nested SMR eventually decides a value. At this point, there is no guarantee that this value is a valid super-block summary. Typically, a malicious replica could propose a garbage value or even a super-block summary with random hashes and nothing prevents the nested SMR to decide this value. Decide garbage values are easy to handle as correct Carrier replicas simply ignore them. In the case of a correctly formed decided super-block summary, we want that correct replicas keep every hash known by at least one correct replica and ignore the other hashes. Correct Carrier replicas achieve that by verifying the signatures associated with each decided hash (line 22). If every signature is valid but the replica ignores the proposal associated with the hash, it broadcasts a resolution request (line 27). Eventually a correct replica replies to this request with the proposal associated to the hash (line 30). When a replica knows the proposals associated with every valid hash of the decided super-block summary, it decides of this list of proposals in the same order their hash appear in the super-block summary (lines 36–37).

```
1: nested ← ...                                          nested consensus
2: V ← map()                                             map from hash(v) to v
3: S ← map()                                             map from h to (sign_i(h), ...)
4: P ← list()                                            super-block summary to propose
5: D ← map()                                             subset of V for accepted hashes

6: propose(v):
7:     broadcast <INIT, v>

8: receive <INIT, v>:
9:     h ← hash(v)
10:    σ ← sign(h)
11:    broadcast <ECHO, h, σ>
12:    V[h] ← v

13: receive <ECHO, h, σ>:
14:    if verify(h, σ) then
15:        S[h].append(σ)
16:        if |S[h]| = t + 1 then
17:            P.append( (h, S[h]) )
18:    if |P| = n − t then
19:        nested.propose(P)

20: when nested decides N:
21:    for all (h, s) ∈ N do
22:        if ∀σ ∈ s, verify(h, σ) ∧ |s| = t + 1 then
23:            if h ∈ V then
24:                D[h] ← V[h]
25:            else
26:                D[h] ← ⊥
27:                broadcast <REQUEST, h>

28: receive <REQUEST, h> from p_i:
29:    if h ∈ V then
30:        send <RESOLVE, h, v> to p_i

31: receive <RESOLVE, h, v>:
32:    if h = hash(v) then
33:        V[h] ← v
34:    if h ∈ D then
35:        D[h] ← v

36: when ∀h ∈ D, D[h] ≠ ⊥:
37:    decide(D)
```

0.2 Proof of the Algorithm

We want to prove that for any BFT consensus, we can extend this consensus algorithm with a super-block optimization and the resulting system ensures the following properties:

- **Set-Agreement**: No two correct processes decide on different sets.
- **Set-Termination**: Every correct process eventually decides on a set.
- **Set-Validity**: Every value in the set decided by a correct has been proposed by a correct.

Lemma 1. *Every correct process eventually decides the same super-block summary with their nested consensus.*

Proof. When a correct process proposes a value, it broadcasts an `INIT` message (line 7) so at least $t+1$ correct processes receive it and broadcast an `ECHO` message (line 11). Furthermore, there are $n-t$ correct processes. Consequently, every correct process eventually receive at least $t+1$ `ECHO` messages for at least $n-t$ different hashes and propose a super-block summary to the nested consensus (line 19). From the termination and agreement properties of the nested consensus, every correct process thus decide the same super-block summary at line 20. □

Lemma 2. *If a correct process accepts (resp. rejects) a hash of the decided super-block summary then every correc process accept (resp. reject) this hash.*

Proof. From Lemma 1, every correct process decide the same super-block summary. Consequently, for every correct, the decided super-block summary contains the same hashes and each hash is associated with the same signatures. Additionally every correct know the same set of public keys. Consequently every correct process obtain the same result for the verification of the hashes and all take the same decision about accepting or rejecting each hash. □

Lemma 3. *No two correct processes decide on different sets. (Set-Agreement)*

Proof. From Lemmas 2, every correct process accept the same set of hashes at lines 20–27. Additionally, since we assume no collision on the hash function, the decided hashes can only resolve to the same values. □

Lemma 4. *If a correct process accepts a hash of the decided super-block summary then at least one correct knows the value associated to this hash.*

Proof. If a correct process accepts a hash of the decided super-block summary then there is a list of a least $t+1$ correct signatures of this hash (line 22). Thus, there is at least 1 correct process which broadcast this signature with an `ECHO` message (line 11). This correct process knows the associated value (line 12). □

Lemma 5. *Every correct process eventually decides on a set. (Set-Termination)*

Proof. From Lemma 1, every correct process eventually decides a super-block summary. For every accepted hash of the super-block summary, either the process knows the associated value (line 24) or, from Lemma 4, at least one correct process knows it. In the latter case, this correct process replies to the `REQUEST` message by sending the value (line 30). Consequently every correct process eventually resolve all the accepted hashes (line 35) and decide. □

Lemma 6. *Every value in the set decided by a correct has been proposed by a correct. (Set-Validity)*

Proof. Same as Lemma 4? □