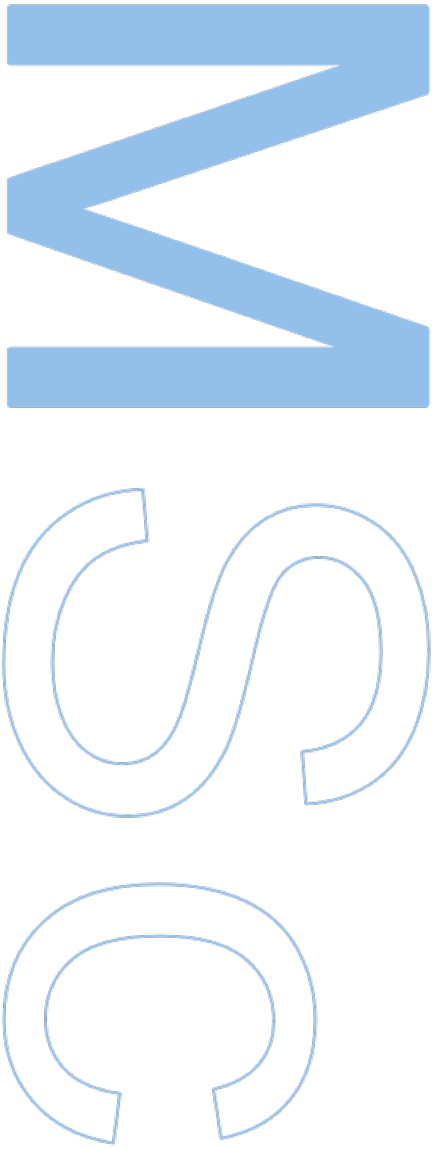
An efficient Rust implementation of BFT for supporting Byzantine Tolerant Distributed Storage

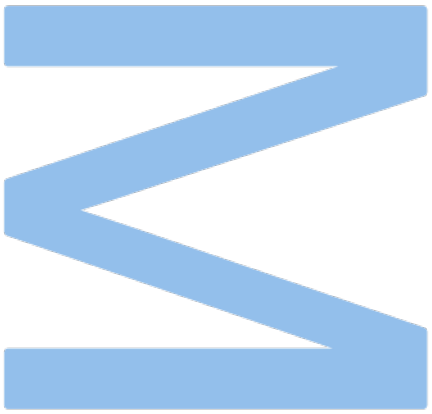


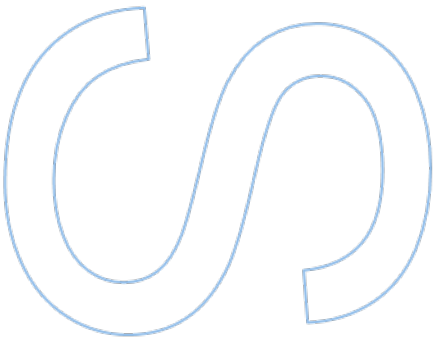
Nuno Gonçalo Neto Martin
Master's in Computer Science
Department of Computer Science
2022

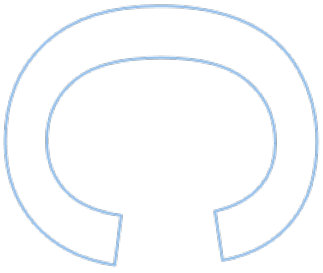
Supervisor
Rolando Martins, Assistant Professor, FCUP

Co-supervisor
Bernardo Portela, Auxiliary Professor, FCUP

An efficient Rust implementation of BFT for supporting Byzantine Tolerant Distributed Storage







An efficient Rust implementation of BFT for supporting Byzantine Tolerant Distributed Storage

Sworn Statement

I, Nuno Gonçalo Neto Martingo, enrolled in the Master Degree Computer Science at the Faculty of Sciences of the University of Porto hereby declare, in accordance with the provisions of paragraph a) of Article 14 of the Code of Ethical Conduct of the University of Porto, that the content of this dissertation reflects perspectives, research work and my own interpretations at the time of its submission.

By submitting this dissertation, I also declare that it contains the results of my own research work and contributions that have not been previously submitted to this or any other institution.

I further declare that all references to other authors fully comply with the rules of attribution and are referenced in the text by citation and identified in the bibliographic references section. This dissertation does not include any content whose reproduction is protected by copyright laws.

I am aware that the practice of plagiarism and self-plagiarism constitute a form of academic offense.

Nuno Gonçalo Neto Martingo

30/11/2022

Abstract

Over the last few decades, a large body of research was done covering Byzantine Fault Tolerance (BFT) systems.

This research has brought forward a lot of new techniques for handling byzantine faults such as block chains, State Machine Replication (SMR) protocols, chain replication, amongst others. In this dissertation, we will focus on the State Machine Replication protocol in particular.

Even with all the advancements made to BFT SMR, industry still tends to prefer CFT implementations over BFT ones as their free, open source implementations remain scarce. With this dissertation, we hope to contribute to this field by presenting FeBFT, a Rust implementation of a BFT SMR system, inspired by the approaches of BFT-SMaRt and PBFT, available to everyone under a permissive software license. The choice of Rust over other popular languages used to implement these types of systems (e.g. Java and C) allowed us to no longer depend on the garbage collector and all its performance implications without having to resort to manual memory management and all of its security issues. This is allowed by Rust's novel memory ownership model, which offers compile-time memory safety.

We then evaluate this system against the current reference implementation, BFT-SMaRt and see how it performs comparatively to it. We conclude that FeBFT has a superior throughput both with synchronous tests with large client counts and asynchronous tests with smaller client counts (but higher overall request counts). We also verify that FeBFT presents a much more stable and consistent performance in some part due to the absence of a Garbage Collector but also due to many of our architectural alterations comparatively to BFT-SMaRt.

We then describe our attempt at utilising this system in order to improve Ceph's fault tolerance resiliency. Ceph is a distributed storage system which only tolerates omissive faults (crash faults) meaning a targeted attack or an arbitrary fault could lead to information theft, loss or alteration. We attempt to improve on this fault resiliency by exchanging the CFT SMR utilised by Ceph's monitors with FeBFT giving them the ability to tolerate arbitrary faults. We chose the monitors as the target as we identified them to be a critical point for the correct functioning of the system.

Resumo

Ao longo das últimas décadas, um grande corpo de pesquisa foi efetuado a cobrir replicação de máquinas de estado tolerante a falhas byzantinas. Contudo, uma grande parte deste trabalho foca-se nos aspetos teóricos da arte e não dá muita importância a problemas enfrentados em situações realisticas de usos do sistema. Existem claro excepções, nomeadamente o BFT-SMaRt que se foca em trazer estes sistemas para "as massas".

Mesmo com os avanços feitos no estado da arte de sistemas BFT SMR, a indústria profissional continua a preferir implementações CFT a implementações BFT pois não existem muitas implementações BFT open source e ativamente mantidas. Com esta dissertação, esperamos contribuir para este campo de pesquisa ao apresentar o FeBFT, uma implementação de um sistema BFT SMR em Rust, inspirada pelas abordagens do BFT-SMaRt e PBFT e disponível para todos sob uma licença de software permissiva. A escolha de Rust como linguagem acima de outras escolhas populares (como o Java e o C) permitiu nos não depender de um *Garbage Collector* e todos os problemas de *performance* a ele associados sem ter que recorrer a gestão manual de memória e todos os problemas de segurança associados. Isto é devido ao novo sistema de gestão de memória introduzido pelo Rust que introduz segurança de memória verificada durante o passo de compilação.

Depois, avaliámos este novo sistema comparativamente a um competidor já existente, neste caso o BFT-SMaRt e verificamos como é que se comporta comparativamente a ele. Concluímos que o FeBFT tem uma *throughput* superior tanto em testes sincronos com uma quantidade grande de clientes como também em testes asincronos com uma quantidade menor de clientes (mas maior quantidade de pedidos). Nós também verificamos que o FeBFT apresenta uma performance mais estável e consistente em parte devido à ausência de um *Garbage Collector* mas também devido às nossas escolhas arquiteturais comparativamente ao sistema acima referido.

Depois, descrevemos a nossa tentativa de utilizar este novo sistema para melhorar a tolerância a falhas do Ceph. O Ceph é um sistema de armazenamento distribuído que (*crash fault*) o que significa que um ataque ou uma falha arbitrária que pode levar a roubo, perda ou alteração de informação. Nós tentamos melhorar esta resiliência a falhas ao trocar o algoritmo de replicação de máquinas de estados utilizado pelos Monitores do Ceph pelo FeBFT efetivamente dando lhes a abilidade de tolerar falhas arbitrárias. Escolhemos reforçar os Monitores em particular pois identificámo-los como um ponto crítico para o correto funcionamento do sistema em geral, que é

vulnerável a ataques ou falhas.

Acknowledgements

This dissertation marks the end of my journey as a Masters student in Computer Sciences. The execution of this work was difficult and plagued with problems and it would have been near on impossible to complete without help.

Firstly, I would like to thank my supervisors, Rolando Martins and Bernardo Portela for their extremely valuable advice, patience and knowledge. I would also like to thank Tadeu Freitas and João Soares for their help in this work, even though they were not my official supervisors.

Secondly, I would like to thank all of my colleagues at FCUP who became my very close friends over this 5 year journey. Their inspiration and support was vital throughout my journey. A special thank you to my girlfriend, who always gave me great advice and support.

Thirdly, I want to thank my family for all their support throughout my life. I would definitely not be where I am if it were not for their support, love and at the right times arguments.

Finally, I would like to acknowledge the financial support from *Fundação para a Ciência e a Tecnologia* (FCT), via the Bosch "Safe Cities" project, with reference POCI-01-0145-FEDER-046103, co-funded/funded by the *Fundo Europeu de Desenvolvimento Regional* (FEDER) through *Programa Operacional Competitividade e Internacionalização* and Portugal 2020.

Contents

List of Tables

List of Figures

Listings

Acronyms

API	Application Programming Interface
BFT	Byzantine Fault Tolerance
CAS	Atomic Compare and Swap operation
CFG	Crash Fault Tolerance
DCC	Departamento de Ciência de Computadores
FCUP	Faculdade de Ciências da Universidade do Porto
HTTP	Hypertext Transfer Protocol
IPC	Inter process communication
MAC	Message Authentication Code
RPC	Remote Procedure Call
TCP	Transmission Control Protocol
TEE	Trusted Execution Environment
TPM	Trusted Platform Module
UDP	User Datagram Protocol

Chapter 1

Introduction

The distributed systems community has long sought to improve the tolerance to faults to support critical systems. In a perfect world, this resiliency should not impact performance and scalability. However, due to well-known theoretical possibilities[24] we are limited to leverage per-domain gains that can make use of specific semantics that at most allows us to slightly circumvent these limitations. Society is becoming much more reliant on these types of systems and their usage will only increase in the foreseeable future, meaning we need to have the ability to scale these systems and assure service to everyone who needs it, wherever and whenever they need it.

There exists a large body of research to implement highly scalable, fault tolerant systems [28, 29, 33, 37, 38] focusing on tolerating benign faults (usually of the omissive type). These faults are characterised by the component not performing some interaction when specified to [45]. For a system that must be available no matter the circumstances, this level of fault protection is not enough. It leaves a clear path of attack on the system for any malicious entities and also leaves itself unprotected against many entire classes of faults.

> Arbitrary or Byzantine faults can be caused by an improbable but possible sequence of accidental events or by the meticulous action of an intentional and malicious component that is deliberately trying to defeat the system

[45].

There are many techniques existent that provide solutions to this problem since they do not make any assumptions on how a node in the network might fail [12, 15, 30, 43, 44]. Recently we have started seeing new and improved ways of handling these types of faults, namely through block chain technologies and their associated proofs (e.g. proof of work, proof of stake, etc), chain replication and State Machine Replication along with many variations of these models [12, 15, 21, 22, 30–32, 36, 41–44]. In this dissertation we will be focusing on State Machine Replication systems.

Many other attempts at implementing BFT SMR systems fail for their lack of performance,

lack of support for very large amounts of clients, high difficulty of use, rigid and sometimes monolithic design and lack of documentation. Many times we are also trapped by the choices of language of the project which restrict the services we can implement to just the one language and subsequently all of it's advantages and disadvantages [17]. Some literature also only presents a theoretical approach and therefore can't be practically used [39].

In this dissertation we will discuss the design and the implementation of a Byzantine Fault Tolerant (BFT) State Machine Replication (SMR) library, FeBFT. FeBFT implements the well established and proven PBFT [19] consensus protocol at its core along with design ideas introduced by BFT-SMaRt [15, 16] and then constructs upon them to yield a practical, performant and safe library.

We then utilize it in an attempt to introduce byzantine fault tolerance in CEPH [47], a reference distributed storage system.

This thesis has made the following contributions:

- We developed FeBFT, a BFT SMR system which targets speed and scalability in scenarios more aligned with real world expectations and challenges, along with an extremely paralellized approach to maximize performance. Novel client handling methods to better handle high client counts were developed, a fast and efficient request flow with well defined goals for each sub system allow for both low latency and high throughput and allow for maximum multi core utilization was implemented, we developed strong and weak persistent storage primitives as well as making contributions to allow for read scalability with the implementation of Followers and unordered request support.
- With Ceph, we analysed the current monitor architecture design and abstracted the necessary patterns in order to replace the Paxos based CFT SMR that was originally implemented with our own BFT SMR system based on FeBFT. We implemented a service for FeBFT that supports the operations supported by this original Paxos based system and outlined the necessary changes to make the client's access to these monitors BFT.

1.1 Organization

In Chapter 2 we present background work on BFT SMR systems which formed the basis for the implementation of this protocol.
In Chapter 3 we present the state of the art on distributed storage systems, the current options and their main characteristics.
In Chapter 4 we describe the design of FeBFT and the various contributions made by this dissertation.
In Chapter 5 we present the benchmarks we developed in order to test the systems and the results we obtained when testing FeBFT. We also develop very similar benchmarks for BFT-SMaRt in order to make a comparison between the two systems.

In Chapter 6 we present an overview of Ceph and its functioning along with the changes we devised and implemented to improve the fault resiliency of the system.
In Chapter 7 we conclude this dissertation and perform an overview on what we were able to accomplish with our work as well as presenting possible future additions we can make to the work made in this dissertation.

Chapter 2

Background Work on BFT SMR

Society is increasingly dependent on the services that are provided to them by computer systems. These computer systems however can fail or work incorrectly (either through malicious attacks or just normal failures), which might irreparably damage those that are dependent on them. To fix this, we need these systems to be highly available (with no downtime) and to always work correctly. There is a large amount of research onto highly available services that are able to tolerate certain faults in parts of the systems. This research however is more focused on tolerating benign faults in the components of the system (failure by stoppage or omission of certain steps), which are not the only faults that can occur within any given system. The components might fail arbitrarily either through malicious attacks, software errors or operator mistakes. These types of faults are called Byzantine faults and if even one of the components in a benign fault tolerant system fails in a byzantine way, then the entire system can exhibit unexpected and unpredictable behaviour. The frequency and gravity of these faults have been steadily increasing with the passing of time as software becomes more complex, systems become larger and with more components and our reliance on them is continuously increasing. In this section, we will be discussing some techniques that are designed to tolerate these Byzantine faults while maintaining the correct operation of the system.

2.1 PBFT

PBFT [19] is an algorithm for state machine replication that offers liveness and safety as long as at most $(n-1)/3$ servers are faulty in a byzantine way (in other words, it requires $3f+1$ servers to tolerate f byzantine failures) as well as not requiring any synchronous communication to provide safety, meaning it could be safely used in asynchronous systems like the internet. This means clients will eventually receive **correct** replies to their requests, even if there are f failures in the network. PBFT also introduces a novel proactive recovery mechanism that recovers replicas periodically even if there is no reason to suspect that they're faulty. This automatic recovery allows the system to tolerate any number of faults over the lifetime of the system as long as there are no more than f failures within a given window of time. This mechanism also

allows the system to detect denial-of-service attacks that are attempting to increase this window as well as detect when the state of a replica has been corrupted. PBFT also incorporates some optimisations like the utilisation of symmetric cryptography instead of public key cryptography to sign and verify messages, which was noted as a major latency and throughput bottleneck in previous systems, among others. In order to assert these properties using a Byzantine failure model that states faulty nodes may behave arbitrarily and the network that connects them can fail to deliver messages, delay them, duplicate them or deliver them out of order, PBFT makes the following 2 assumptions:

- **Bound on faults**: We assume that no more than f nodes can fail in any small window of vulnerability.
- **Strong cryptography**: We assume that the attackers are computationally bounded and therefore are unable to subvert the cryptography technologies used by the system.

The PBFT algorithm works in a primary-backup mechanism in which replicas move through successive views. In each view one of the replicas is declared the *primary* and the others are *backups*. The *primary* is responsible for selecting the ordering of operations requested by clients. This is done by assigning the next available sequence number to a request and then sending this assignment to all the backups which will then check the sequence numbers assigned by the *primary*. The *backups* must check these numbers as the *primary* might be faulty (it may assign the same number to several requests, leave gaps between the sequence numbers assigned or just stop assigning sequence numbers). When a faulty primary is detected by the backups they will trigger a *view change* and consequently, a new *primary* is chosen for the system. This algorithm ensures that sequence numbers are *dense* (no sequence number is skipped in normal operation, except for when a view-change is done where some sequence numbers might be assigned a null request, which corresponds to no operation). To order requests correctly even with failures, the system relies on quorums of $2f + 1$ replicas. Any Byzantine dissemination quorum can be used. These quorums possess two important properties, which allow them to be used:

- **Intersection**: Any two quorums of the system will have at least 1 correct replica in common. This is because if we have two quorums with $2f + 1$ replicas in a system of $3f + 1$ replicas, we know that at least $f + 1$ replicas in those quorums will be common to each other, meaning that we have at least 1 correct replica in common.
- **Availability**: There is always a quorum available with no faulty replicas. This is because in a system with $3f + 1$ replicas, we will always have $2f + 1$ replicas that are not failing and that are available to form a new quorum.

For a client to request the execution of a particular operation, he must first send a message in *multicast* to all the replicas in the system. When the replicas receive this request, they store it in their log and then execute the ordering protocol. The ordering protocol consists of three messages *pre-prepare, prepare* and *commit*. The first two are used to *totally order* the requests

sent in the current view even if the primary is faulty. The *prepare* message, along with the *commit* message, also serves the purpose of ensuring that requests stay totally order across views. When the primary receives a message from the clients, it will assign it a sequence number and then *multicast* a *pre-prepare* message to all replicas that contain the sequence number, current *view* along with the digest of the message (the message's actual content is only sent by the client and is not forwarded between replicas to save resources). When a replica receives this message, it checks to see if it's in the same *view* as the primary and if so, checks the correctness of the assigned sequence number. If this number checks out the *backup* replica *multicasts* a *prepare* message with all the information. This *prepare* message signals to the system that *backup* agreed with the sequence number the *primary* assigned to that request. Each replica will then collect messages until it has a quorum of corresponding *pre-prepare* and *prepare* messages. When this quorum is reached, the replica will have the *prepared certificate* of that operation. This certificate proves that a quorum of replicas has agreed with the assigned sequence number. Having this *prepared certificate*, the replica will *multicast* a *commit* message to signal that it has the *prepared certificate* and that it has added the messages to its log. Then, each replica waits for a quorum of *commit* messages. When this quorum is reached, the replica creates a *committed certificate* and the request is now *committed*. When a request is *committed* it means a quorum of at least $2f + 1$ replicas has agreed with the assigned sequence number and the view the request is situated in. After having this certificate, each replica will execute the operation (given all previous requests have been executed, if not it will execute them in order). After having executed the request, the replicas will send a *reply* to the client to mark the operation as completed.

This protocol only provides liveness and safety if less than $\frac{1}{3}$ of the replicas fail during the lifetime of the system. This is insufficient for systems that are long-lived since the bound of systems that fail will likely be exceeded. To tackle this issue, PBFT introduces a recovery mechanism that recovers faulty replicas. Since a Byzantine-faulty replica might appear to behave correctly, this task is done proactively for all replicas even if there is no reason to suspect they are faulty. The recovery protocol works as follows, in a very basic overview:

- The recovery must assert that every quorum certificate received by a recovering replica is backed by a valid quorum. Additionally, it must ensure that the service state the replica holds is consistent with the protocol's state, that is: For every correct replica, the value of the current state with the sequence number n must be identical to the value obtained by executing and committing the requests between a stable checkpoint h stored in the replica and n, in ascending order of sequence number (starting at $h + 1$ until n).
- Since this recovery is proactive and is done on all replicas, not just the faulty ones, recovering replicas may not be faulty and performing this recovery must not make it faulty otherwise the previous steps, which rely on the replica having a correct stable checkpoint h stored, will not work properly. We also have to assure that the replica is able to participate in the request processing protocol while it is recovering, as that is sometimes required for it to complete the recovery. However, if the recovering replica is faulty, we must assure the state that it's brought to will satisfy the required invariants and we must prevent the replica

from spreading false information.

The detection of faulty replicas and subsequent prevention of the spreading of false information is handled by PBFT[19], however, it is out of scope and therefore will not be tackled by this paper. This technique allows PBFT[19] to tolerate any number of faults to replicas, as long as no more than $\frac{1}{3}$ of the replicas fail within a small window of vulnerability.

2.2 BFT-SMaRt

BFT-SMaRt [15][40] is an open-source practical and reliable software library implementing a Byzantine Fault Tolerant state machine replication algorithm written in Java. Some of its key features are its improved reliability, modularity, ability to utilise multicore systems, support for reconfigurations with no downtime and a flexible programming interface. This library was developed because the authors found that the other alternatives (PBFT [19] and UpRight [20] at the time) were not robust enough and lacked many key features, as well as being plagued with bugs and no longer being actively maintained. BFT-SMaRt was designed to provide the following principles:

- *Tunable fault model* - *Non-malicious Byzantine* faults are tolerated by default so any errors in the delivery of messages (delays, losses and corruptions) are tolerated as well as width standing any abnormal and spurious actions from failing processes. There is also support for tolerating *malicious byzantine faults* with the use of cryptographic signatures and support for a simplified CFT tolerant mode with a protocol similar to Paxos[33].

- *Simplicity* - To maintain the correctness and simplicity of the project techniques that could have added a lot of extra complexity were avoided, even though some of the avoided techniques could have provided additional performance or resource efficiency, which could have made the project faster. However, even having made these compromises tests show that the performance of BFT-SMaRt is similar to or even better than other existing alternatives with such implementations.

- *Modularity* - BFT-SMaRt implements a modular protocol that has a well-defined consensus primitive at its core, unlike other systems which implement a *monolithic* protocol where there is not such separation of components. While these two approaches have similar performance at runtime, a modular approach will make implementing new features or changing existing ones easier.

- *Multi-core awareness* - Modern computers have adopted the multicore architecture to maximise performance. As such, BFT-SMaRt is designed to scale the system throughput with the number of hardware threads available in the system.

- *Simple and Extensible Application Programming Interface* - This library encapsulates all that is complex in *Byzantine fault tolerance* and provides a simple API that can be used by programmers to implement deterministic services.

BFT-SMaRt requires at least $3f + 1$ servers to tolerate f faults, similar to PBFT which was discussed in Section 2.1. It also supports an unbounded number of faulty clients. Moreover, since the system allows for reconfigurations of the network, the number of servers can be altered at runtime and therefore the number of tolerable faults can be increased or decreased as necessary.

In terms of internal functioning, BFT-SMaRt is very similar to PBFT. The same messaging structure and order are used, as well as the same ideas on views and leaders of the network. What BFT-SMaRt brings is a concurrent system that better utilises the more recent multi-core architecture paradigm and therefore achieves much better performance. However, there were some key issues with the separation of tasks in the protocol into an architecture that is as robust as it is efficient. The final architecture of the program separates the main tasks into the following thread model, in which threads communicate using bounded queues:

- The *Netty thread pool* and subsequent *Client Manager*, which handle all client-ordered requests, including the verification of request integrity and optional ownership. This allows BFT-SMaRt to manage hundreds of client-to-replica connections efficiently.
- We then have the *Proposer thread* which is only available in the leader replica. This thread handles assembling batches of requests and transmitting the subsequent *PROPOSE* message to the rest of the replicas. These batches are filled until either: (a) the size reaches a configurable limit or (b) there are no requests left to add. When the batch is complete, the *Proposer thread* will place the messages onto the *out queue*. The *out queue* is constantly *polled* by the sender thread pool, which will take the messages that need to be sent, serialise them, produce the MAC to be attached to them and then proceed to send them to their destination.
- The reception of these messages is handled by the *receiving thread pool*, which upon the reception of a message will verify its integrity (using the attached MAC), deserialise it and place it on the *in queue* of the *message processor thread*. The *message processor thread* is then responsible for processing them. If the request belongs to the current consensus view then it will be processed immediately, if it belongs to a consensus that is ahead of the current one, it will be placed on hold until such consensus is triggered. If the request does not fit into these 2 options, it will be discarded. When a consensus is finished (after receiving the *WRITE* and *ACCEPT* also known as *COMMIT* and *PREPARE* in PBFT) the batch is put onto the *decided queue*.
- The *decided queue* is polled by the *delivery thread*. The *delivery thread* takes the batches from the *decided queue*, deserialises all the requests, marks the consensus as finished and then invokes the *service replica* to execute the requests and generate the replies, which will then be put into the *reply queue*.

- The *reply thread* will then poll the *reply queue* and send the replies to the respective clients.
- There is also a *request timer thread* that is tasked with periodically activating and verifying if some request has remained for than a configurable timeout on the *pending requests* queue (the requests that have been received by the replicas from the clients but have not yet been processed, as in a *PROPOSE* message has not been sent by the leader). The first time this timer expires for a given request, that request is going to be forwarded to the current leader. If it is the second time the timer has expired, the current consensus is stopped and a synchronization phase is triggered (essentially a *view-change* in PBFT) to elect a new leader.

This architecture allows *BFT-SMaRt* to scale its throughput, in Kops/sec, with the number of cores the machine has mostly due to its ability to process signatures in parallel using the *Netty thread pool*, which was one of the largest performance hogs in previous works, due to the complexity of asymmetric cryptography. It also allows *BFT-SMaRt* to handle an extremely large amount of clients concurrently, something that PBFT could not do.

Similarly to PBFT, *BFT-SMaRt* also utilises some extra primitives to assert liveness and safety in the case of the general failure of over 1/3 of the network (or even total concurrent failure of all replicas in the network, unlike PBFT which only handles $\frac{1}{3}$ of the replicas in the network failing in a small timeframe). The techniques *BFT-SMaRt* initially used are *logging*, *checkpointing* and *state transfer*.

Logging consists of writing the operation into stable storage before sending the confirmation to the client, which means all operations that have been acknowledged to the clients are effectively permanent in the system even in the event of mass failure of the replicas.

Checkpointing exist to limit the size of the *log*, which can get very big very quickly, depending on the throughput of the system. A checkpoint stops the service so that all replicas are able to take a snapshot of their current state and store it in a stable storage system, which will then allow it to delete the stored *logs*. Subsequent logs build on top of the current latest checkpoint to provide replicas with the ability to rebuild the state that was left when they crashed.

State transfer allows a failed replica to get back to the most up-to-date state in the protocol. It works similarly to the recovery protocol in PBFT, by transmitting log records from other replicas if the failed replica has the most up-to-date checkpoint and by transmitting both checkpoints and log records in case the replica is very distant from the current network state.

These approaches however can drastically decrease performance. For example, *logging* can make the throughput of the system as low as the number of appends that can be made on the disk (the use of SSDs alleviates the problem, but does not quite solve it), *checkpointing* forces the system to stop until it has completed the checkpoint and *state transfers* impact the operation of the correct replicas since they need to transfer their state, clogging their network throughput. As a solution to these issues, BFT-SMaRt introduces the following paradigms:

- *Parallel logging* - Consists of logging batches of operations instead of single operations (in line with the batching approach taken by the algorithm) and processing the operations in parallel with their logging in the stable storage.
- *Sequential checkpoint* - Since BFT system only require $n - f$ replicas to progress in their executions, there are f spare replicas in fault-free executions. We use this fact to make each replica perform the checkpointing algorithm and store its current state at different times, to assure $n - f$ replicas are available to process requests.
- *Collaborative state transfer* - Typically only one of the replicas will return the full state and log, while the other replicas will send the hash of the same data to allow the recovering replica to validate the incoming data (Furthermore this approach will not work with the employed *Sequential checkpointing* since the received state cannot be directly validated with other replicas, as they can be at different checkpoints). To address this, a new protocol was devised that utilises an approach similar to the bitTorrent protocol, where all replicas act as seeders as participate in the state transfer protocol.

2.2.1 Batching approach

The batching in BFT-SMaRt is handled by the Proposer, whose sole job is to collect requests from the client queues, aggregate them into a larger batch and then propose them into the consensus. The Proposer is only available on the leader replica and it has a configurable limit batch size which will be used by the Proposer to know when to stop filling up the batch. It will also stop when there are no more requests to be collected and proposed.

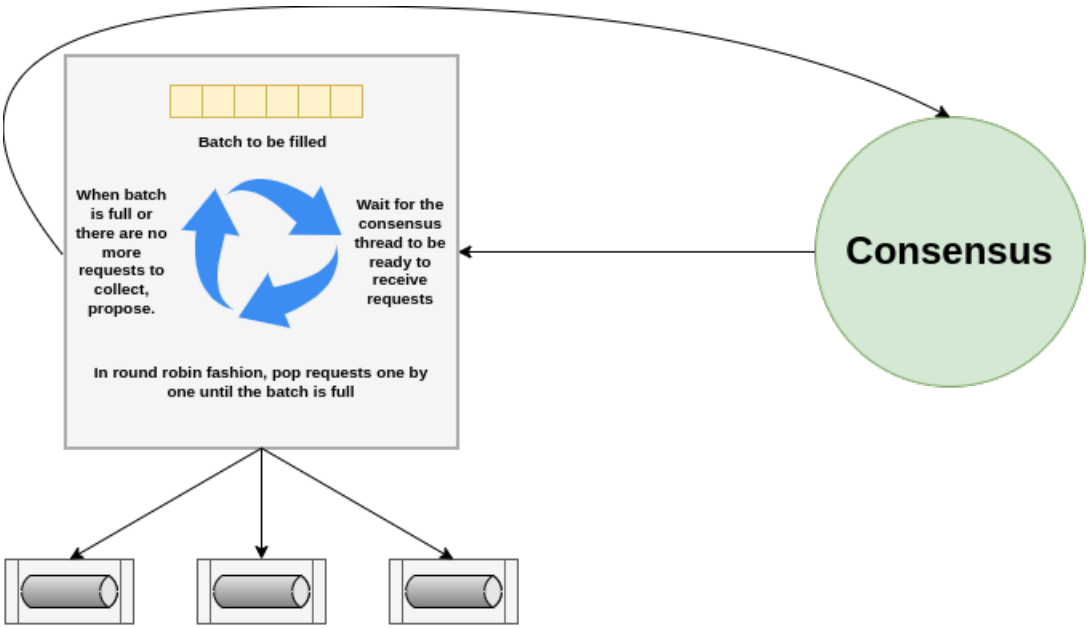


Figure 2.1: The batching approach of BFT-SMaRt

As we described in Figure 2.1, we can see that the proposer will only start collecting requests

when the consensus thread is not performing any work as that's when it's ready to receive new requests. When the proposer is triggered, it immediately tries to start filling up the batch. To do this, it will visit the client's buffers one by one, taking out one request at a time until the batch is full. This makes this process fair as if all clients have pending requests it will take from them equally (at most one more message from the client with the most messages in the batch to the client with the least messages in the batch, which is to be expected because of the last round to be done). It is not however the most efficient and concurrency-friendly way to remove requests from the client buffer, since it will require a lot of operations on the shared client queues.

This approach also mandates the existence of a limit on the number of requests each batch can contain in order to protect against a Denial Of Service attack. If there were no limit on the number of requests, we could perform the following attack:

- We know that the proposer will keep trying to populate the batch until there are no more requests to collect (while assuming that there is no limit).
- If we make the clients constantly send requests (without even waiting for the response) forever, then the proposer will keep on collecting them until it crashes for running out of memory. Even if it does not crash, the proposer will never actually propose any new batches since it will still be collecting requests from the clients therefore the system will never make an advancement.

With this attack, we could heavily reduce or even completely stop the system from performing any operations. Adding a limit on the batch size eliminates this risk but it also creates a new problem, which is the choice of batch size. If the batch size chosen is too small then BFT-SMaRt will not be able to scale properly since more rounds will be needed to process the requests. However, due to an implementation detail on the Proposer, if the chosen batch size is too large for the client amount, the batch might get delayed. This effect gets exacerbated by the fact the Proposer only runs when the consensus thread is ready for a new batch of requests (i.e., meaning it's not processing any consensus instance) so it delays the processing, resulting in added latency.

Chapter 3

State of the art on Secure Distributed Storage Systems

Cloud storage has been increasing in popularity in the last few years. It is cheap, usually reliable and decently fast. However, for specific applications that require trustworthy storage, we cannot limit ourselves to using a single cloud storage provider, as we would then be reliant on a single point of failure. A straightforward solution is to use various storage providers and combine their abilities to make sure we can handle failures to any individual provider (or even multiple failures). Various existing technologies already allow the user to leverage the availability of using multiple storage providers to maximise uptime.

3.1 HotStuff

HotStuff [8, 50, 51] is a leader-based Byzantine fault-tolerant state machine replication protocol, similar to the previously mentioned PBFT and BFT-SMaRt. It works with monotonically increasing view numbers where each view has a dedicated unique leader known to all members of the system again similar to what we saw in PBFT. It intends to address the complexity of the leader change protocol introduced by PBFT, named **view-change**. Initially, when PBFT was developed, a typical target system was composed of a few nodes like 4 or 7 deployed on local area networks. However, recent developments in state of the art and the introduction of new technologies like blockchains have renewed the interest in these areas but with a much larger scaling need. Instead of handling a few local nodes, BFT systems are now expected to handle several hundred nodes spread throughout a WAN. Since the introduction of PBFT, many BFT solutions have been designed and built around its tried and tested two-phase paradigm. However, this base protocol does not actually offer any fault tolerance when not complemented by the **view-change** protocol introduced alongside it. Unfortunately, this protocol is far from simple and incurs a significant communication penalty for even moderate system sizes. In fact, the total number of messages that must be transmitted for a given view-change in a system with n nodes can be of $\Omega(n^3)$ and even incrementing the original protocol with the utilization of

threshold signatures still requires $\Omega(n^2)$. This problem plagues not only PBFT but also every other protocol designed with its consensus protocol at its core.

Instead of following this existing protocol, HotStuff implements a new three-phase core which allows the leader to simply pick the highest *quorum certificate* it knows of. It introduces a new intermediary phase that allows replicas to "change their mind" after voting without requiring a leader proof at all. This simplifies the leader replacement protocol requiring less messages and subsequently less time at the cost of an extra messaging round for each correct consensus instance.

This simplified leader change protocol is usually found in systems built using a synchronous core, where proposals are done in pre-determined intervals Δ that must take into account worst-case scenarios of the network. Therefore, these systems miss a hallmark property of most practical BFT SMR systems, *optimistic responsiveness*.

This is addressed by HotStuff, which achieves two fundamental properties:

- **Linear View Change**: Any correct leader, once designated, sends only $\Omega(n)$ authenticators to drive a new consensus decision.
- **Optimistic Responsiveness**: Any correct leader only needs to wait for $n - f$ responses to commit it and make progress correctly.

Another benefit is that the cost for a new leader to drive the consensus is no greater than for the current leader, meaning we can withstand frequent leader changes, which has been argued is useful in blockchain systems to ensure chain quality.

HotStuff achieves this with the addition of a new phase which incurs only the actual network delays, which are generally smaller than the Δ, which is usually picked to take worst-case scenarios into account.

Each replica stores a *tree* of pending transactions as its data structure. Each node of this tree is composed from a set of transactions, metadata associated with the protocol and a parent link. The *branch* led by a given node is the path from the node all the way to the root of the tree by visiting the parent node sequentially. The protocol will monotonically grow a branch by committing new blocks. For a given block to be committed, the leader must propose it to the replicas of and system and collect a quorum of votes from $n - f$ replicas in three phases: PREPARE, PRE-COMMIT and COMMIT. This collection of votes is named the *Quorum Certificate*.

3.1.1 Phases of the algorithm

1. **PREPARE phase**: The leader starts by extending the local tree using the *CreateLeaf* function with the client commands the leader intends to propose and broadcasting the

prepare message to the rest of the replica messages to the replicas. Replicas wait for the message from the leader and upon reception, verify it and send a PREPARE vote to the leader.

2. **PRE-COMMIT phase**: The leader starts this phase by waiting for $n - ff$ PREPARE messages as replies to the previously broadcast PREPARE message. He then uses them to generate the *QC* for the *PREPARE* phase. He then broadcasts a *PRE-COMMIT* message which also contains the *PREPARE QC*. The replicas start by waiting from the message from the leader containing the *QC* and the *PRE-COMMIT* message. They validate them and if it is valid they send a *PRE-COMMIT* message to the leader.

3. **COMMIT phase**: The leader starts by waiting for the $n - f$ *PRE-COMMIT* messages in order to form the correspondent *QC*, similarly to before. Also, similarly, he then broadcasts the *COMMIT* message along with the *PRE-COMMIT QC*. The replicas wait for the leader's message. Upon reception, they validate it and send the *COMMIT* message to the leader.

4. **DECIDE phase**: The leader starts by waiting for the $n - f$ *COMMIT* messages to form the final *QC* for the *COMMIT* phase. Once obtained, he commits the commands and the corresponding *QC*. He then broadcasts a *DECIDE* message with the *COMMIT* certificate. The replicas wait for the *DECIDE* message and, upon reception, validate it and commit and execute the commands.

This basic protocol takes three phases to commit a proposal. These phases are not doing useful work except for collecting votes from replicas and they are all very similar. HotStuff improves on this by introducing Chained HotStuff, which both improves the basic protocol and at the same time makes it simpler. The idea is to change the view on every *PREPARE* phase so that each proposal has its own view. Doing this reduces the number of message types and allows for pipelining decisions.

Specifically, the votes over a *PREPARE* phase are connected in a view by the leader into a *genericQC*. This is then relayed to the leader of the following view, therefore delegating responsibility for the next phase, which would have been *PRE-COMMIT* to the upcoming leader. However, this new leader does not carry a *PRE-COMMIT* phase. Instead, he initiates a new *PREPARE* phase and adds its own proposal. This *PREPARE* phase for view $v+1$ simultaneously serves as the *PRE-COMMIT* phase for view v and so on. This is possible since all the phases have an identical structure.

With this architecture, the authors were able to develop an implementation that can match BFT-SMaRt's latency and throughput even though it uses an additional phase, allowing it to support much simpler and faster leader changes all while maintaining *optimistic responsiveness*.

3.2 Tendermint

Tendermint [9, 10, 18, 32] is a new Byzantine fault tolerant transaction ordering protocol for distributed networks. It combines the Block Chain idea with an SMR to provide a performant system capable of thousands of transactions per second on dozens of nodes geographically distributed.

Tendermint provides a form of BFT SMR that is accountable. If the safety or consistency of the network is violated, we can verify the members that acted maliciously (as long as less than $\frac{2}{3}$ of the network is malicious). The system is composed of *validators* identified by their public key. Each *validator* must have a full copy of the replicated state. They are responsible for proposing new blocks (in the cases where they are the selected leader) and for voting on newly proposed blocks by others. These blocks are identified by a monotonically increasing index called *height*. At each *height* of the system, the validators take turns proposing new blocks in rounds such that any given round has at most one valid proposer. It may require multiple rounds to commit a new block at any given *height* due to the asynchrony of the network and faults of member nodes. Each round is composed of a proposed block proposed by the leader in the round and two voting phases. For a block to be successfully committed in the system, it must go through both voting rounds with votes from at least $\frac{2}{3}$ of the network. Tendermint uses a simple locking mechanism to prevent a malicious coalition of less than one-third of the validators. A simple round-robin algorithm orders the validators. This allows only one validator to be a proposer in each round but also that every other validator knows who the proposer is for the current round. In each round, the correct proposer has a Δ of time in which he can propose his block. The validators will move to the next round if he does not propose the block in the given time frame. Proposals related to previous rounds are ignored. This easy and frequent cycling of leaders is essential for Tendermint's fault tolerance guarantees.

The system uses blocks of transactions for two main reasons:

1. **Bandwidth optimization** - Since each commit requires two rounds of voting between all validators, we must amortize this cost by fitting many singular transactions into each of the heights.
2. **Integrity optimization** - Each block stores the previous block's hash in its header, creating a hash chain. This is an immutable data structure. Enables authenticity checks for intermediate sub-states.

3.2.1 Voting phases

As we have discussed, in each round we have two voting phases. If the proposer proposes a block in the destined Δ, then the validators will validate the block and broadcast a *pre-vote* message containing the block's hash. When a validator gathers $\frac{2}{3}$ of the validator network in *pre-votes* for

a given proposed block, he has a *polka*. When a validator obtains a polka, he can now vote in the next round, the commit. To vote, he broadcasts a *pre-commit* containing the block's hash. If a validator receives $\frac{2}{3}$ of *pre-commits* for that block, he will commit it and execute it and increase the current *height* of the system.

If the proposer does not propose a block in time, the validators emit a *nil pre-vote*. If we receive more than $\frac{1}{3}$ of *nil pre-votes*, we have a *nil polka* and we can move to the next round. In case the validator can not obtain a *polka* in sufficient time, he will broadcast a *nil pre-commit*. When we receive more than $\frac{1}{3}$ *nil pre-commit*, we move to the next round.

3.2.2 Locks

In order for this protocol to function correctly, we must avoid circumstances which would justify two different blocks being committed at the same height. Tendermint solves this issue with a locking mechanism that revolves around the *polka*. In essence, for a validator to broadcast a *pre-commit* he must first have the corresponding *polka* and any validator is considered locked on the last block that he has voted with a *pre-commit*.

There are two rules for locking in this system:

1. *Pre-vote the lock* - A validator must *pre-vote* the block that he is locked on (or propose it in case he is the leader of the round). This prevents validators from pre-committing a block in a round and then parking in the polka of the next round of the same height.
2. *Unlock on polka* - A validator can only be released from the lock after it sees a *polka* at a higher indexed round than the round he currently locked on.

For simplicity, a validator is considered to have locked on *nil* at round -1 at each height so that the second rule assures us that a validator cannot commit at a given *height* until he has seen the polka for that round.

With these properties, Tendermint satisfies *Atomic Broadcast*. It is also able of identifying all faulty validators when there is a safety violation when the amount of faulty nodes is less than $\frac{2}{3}$ of the network.
However, this system relies on a synchronous core which uses predetermined slots which must accommodate the worst case time it might take the network to propagate all of the messages which we have referred to previously as Δ. This means that this system foregoes a hallmark of most practical SMR systems: *optimistic responsiveness*. This means that a non-faulty leader cannot drive the protocol in time dependent only on the current message delay. Instead, we always wait for Δ which should be set to expect the worst case scenario.

3.3 Mir BFT

Mir BFT [41, 42] is a robust BFT total order broadcast protocol aimed at maximizing throughput on Wide Area Networks targetting permissioned and Proof of Stake based permissionless block chain. It is show that Mir achieves unprecedented levels of performance without having to sacrifice latency or the robustness of the protocol. To achieve this Mir relies on a novel protocol mechanism that allows a set of leaders to propose request batches independently in parallel. The actual basic consensus protocol is based on the already proven and scrutinised PBFT protocol which we have described previously.

Mir BFT introduces many changes when comparing it to the PBFT protocol it is derived from. Namely:

1. **Request Authentication** - PBFT uses a vector of MACs, one MAC per replica of the system. Mir BFT on the other hand uses public key cryptography based signatures.
2. **Batching and watermarks** - Mir processes requests in batches, a pretty standard throughput improvement on the base PBFT protocol. However, Mir BFT always maintains the watermarks which were originally used by PBFT to boost performance. Namely, watermarks are used to represent the range of sequence numbers the leaders can propose concurrently. In Mir BFT, it is used to facilitate concurrent proposals of batches by multiple leaders.
3. **Protocol Round Structure** - Mir operates in epochs which are somewhat similar to PBFT's views. Each epoch has a *primary*, which is a node defined by the epoch sequence number deterministically utilizing a simple round robin rotation. The primary is the one who defines the the set of leaders which define the nodes that are able to propose in the current epoch. Within a given epoch we then deterministically partition batch sequence numbers across leaders such that all leaders can propose their batches simultaneously without conflicts. Some epochs have a maximum number of batches that can be executed such that epochs are called *Ephemeral epochs*. When we reach the batch count on that epoch we perform a gracious epoch change which is much faster than the view change algorithm initially presented in PBFT. Unlike *ephemeral epochs*, *stable epochs* have no limit on the number of batches. The system can only run in this mode when the number of leaders is >= to the number of *Stable Leaders* (a configurable value).

3.3.1 Leader election

As we have just seen, each epoch has a set of leaders that have the ability to propose new batches. This brings forward a few possible issues like the selection of the epoch leaders, request duplication (different leaders can include requests that are also included by other leaders, up to n repetitions n being the amount of leaders), amongst others. The selection of the leaders for

each epoch is done by the *primary* replica delegated to that epoch. This selection is constrained in the following ways:

1. If the new epoch has started gracefully then the *primary* can not reduce the amount of leaders in the epoch, he can only maintain or increase it. If the primary believes that there are more correct leaders ready to assume leadership it will increase the amount of leaders. This happens for as long as the number of leaders is less or equal than the number of stable leaders.
2. If the epoch starts ungraciously then the leader set must be reduced in size (as long as the amount of leaders is > 1).
3. The primary of an epoch must always be a leader in that epoch.

The addition of various leaders, as we have seen can introduce many problems relating to request duplication. This could be solved by making clients only sending requests to one leader, instead of broadcasting them. This, however, introduces many possible faults as the clients can be Byzantine and send their request to more than 1 leader, the leader they send to can be byzantine and not propose their request (and since no other replica knows about it, they are unable to alert the system to this behaviour). Therefore, a client must always send his requests to at least $f + 1$ replicas so we can't rely on them to do the division of requests. To address this, Mir BFT utilizes the hashes of the client requests. Specifically, it partitions the hash space of the request digests into buckets and then assigns these buckets to leaders in the current epoch. A leader can only include requests from buckets it has been assigned and if this is not followed, the leader will be treated as byzantine. In addition to this protocol, the assignment of buckets during a *stable epoch* is periodically rotated, such that leader i gets assigned buckets that belong to leader $i + 1$ (combined with modulo arithmetics). Since *ephemoral epochs* have limited duration, they also have fixed bucket assignments.

3.3.2 Epoch Changes

Gracious Epoch Change These change occur when the number of batches in an *ephemeral epoch* is completed. It is meant to be simple and efficient to allow for easy growth of the leader set. After an *ephemeral epoch* is done, the new elected *primary* will reliably broadcast the configuration for the new epoch $e + 1$. This configuration contains:

- The set of leaders for the epoch (chosen by the primary according to the previously described rule set).
- The partition of the hash space and the assignment of buckets per leader.

Ungracious epoch change These epoch changes are caused by epoch timeouts due to asynchrony or failures. This protocol is a generalisation of the view change protocol described in

PBFT. Mir supports adaptive timeouts, dependent on the current average commit rate. When a timer expires the node enters the epoch-change protocol in order to move to the next epoch. It then follows a protocol similar to PBFT's view-change protocol with a crucial addition: The amount of leaders for the next epoch will have to be lower than the number of leaders in the current epoch which will then force us to a *ephemeral epoch* as the number of leaders will obligatorily be $<$ StableLeaders.

These protocols allow Mir to provide a high-throughput robust BFT protocol at scale, capable of withstanding request duplication and request censoring attacks along with tolerating a high number of faults to the network, even if they occur to current leaders.

3.4 DepSky

DepSky [13] is a virtual storage cloud, implemented in Java, that leverages the benefits of cloud computing by combining various cloud service providers to form a *cloud-of-clouds*. This *cloud-of-clouds* allows DepSky to improve the availability, integrity and confidentiality of the information stored through replication, encoding and encryption.

The distributed system is composed of 2 parties: cloud providers and clients which can be either readers, writers or both. DepSky is made to handle storage clouds that do not possess the capacity of executing users' code, so they have to access using only the methods provided by the cloud storage service without any kind of modification. This means DepSky has to be implemented as a purely client-side software library. In order to be able to support these various different cloud services DepSky has abstracted a data model that allows them to support any provider regardless of their particular implementation. This data model contains the metadata of the file (version number, the hash of the data along with other important information like last write time, last access time, etc) and the actual data itself. This base data model is then extended and adapted to support the cloud providers that will be used.

Cloud providers have to implement five simple operations in order to be used in DepSky: *list, get, create, put* and *remove*. Since no individual cloud is trusted, we must assume they can fail in a Byzantine way. This allows the protocol to support the most general fault models and handles malicious attacks or intrusions on a cloud provider along with any sort of arbitrary data corruption caused by hardware or other types of unpredictable failures.

Readers can fail in any way (crash, fail intermittently and present any sort of unpredicted or unexpected behaviour) while writers can only fail by crashing because if the protocol supported arbitrary failures then the writers would be able to write wrong values in data units which would corrupt their state and therefore would not allow DepSky to provide the integrity it promises.

DepSky requires $n = 3f + 1$, cloud providers to handle f concurrently faulty service providers ($3f + 1$ is the minimum number of replicas to tolerate Byzantine failures in asynchronous storage systems).

The access schema used is single-writer multi-reader register as supporting multiple writers can be problematic due to the cloud providers not being able to execute code, which means they cannot verify the version of the data that's being written. To support multiple concurrent writes DepSky employs a lock/lease protocol to assert that no two writers are modifying the same data at the same time, which could cause integrity issues in that piece of data.

It has two algorithms that provide varying levels of confidentiality and fault tolerance:

- *DepSky-A* (Or Available DepSky) improves the availability and integrity of the data stored by replicating it on several cloud providers. It guarantees availability by replicating the data in a quorum of at least $n - f$ service providers, where f is the amount of provider failures the system can handle. Since the data is replicated across all those services (and we also know it has to be consistent across all the cloud providers due to the Byzantine consensus algorithm that is implemented), we only have to retrieve the data from one of the clouds, which is has to be available because $(n - f) - f > 1$, since $n = 3f + 1$. This protocol has two main limitations: Firstly a data object of size S can take up from $(n - f) * S$ to $n * S$ times the size S, which will cost on average about n times more than if it were stored in a single cloud service. Secondly, all the data is stored in *clear text* which means that there are no guarantees about the confidentiality of the information.
- *DepSky-CA*(or confidential and available DepSky) provides integrity, availability and privacy. It solves the two limitations of the previous algorithm by employing an information-efficient sharing scheme that combines symmetric encrypting with a classic secret sharing and optimal erasure code to partition the data in a set of blocks such that: $f + 1$ blocks of data are required to obtain the original data and with f or fewer blocks, we are not able to extract any information about the original data. This means that even if we have f faulty or malicious cloud providers, they are unable to obtain the data since they need another block from a non-faulty provider. This algorithm is based on *DepSky-A* and deviates from it in the following aspects: (1.) Before inserting the data into the clouds it has to be encrypted, so we first have to generate the key shares and then cypher the data so it can be uploaded to the cloud. When retrieving the data, we have to get all the key shares (we need at least $f + 1$ key shares to be able to decrypt the data) along with all the ciphered blocks so we can decrypt and read them. This also implies that the writers will have to calculate and store n digests (one for each cloud) instead of just the single one(2.) There is an additional piece of metadata stored, which is a key share. (3.) We need $f + 1$ replies instead of just the first as we need at least $f + 1$ blocks to be able to decrypt the data.

DepSky addresses four important limitations of single-provider cloud computing, namely:

- **Loss of availability** - Temporary outages of certain parts of the internet or certain data centres is a very well-known occurrence. When the data is moved to a given data centre, we are intrinsically taking a risk as that service might become unavailable at any given time be it by external uncontrollable occurrences (like natural disasters) or by targeted

attacks (like DoS attacks). DepSky addresses this by replicating the data across various cloud providers meaning if one of them goes offline, the data can still be accessed.

- **Loss and corruption of data** - By moving all of our data to a cloud service provider, we are placing trust that the service provider will back up the data regularly, use hardware that is not prone to failures and take every precaution so that no data can ever be lost. Taking into account that the cloud service providers are attempting to make the biggest amount of profit possible, this trust might not always be well placed. DepSky deals with this by employing a Byzantine fault-tolerant replication to store data on several service providers, even if some of them corrupt or outright lose data.
- **Vendor lock-in** - Due to the economies of scale, there are concerns that a select group of cloud service providers are able to become dominant and destroy the competition, creating vendor lock-in. This is compounded by the fact that the costs of moving from one service provider to another might be extremely expensive as all the data received/sent needs to be paid for, not just the data storage itself. DepSky migrates in two ways, first by not relying on a single cloud provider and therefore having the ability to distribute the data and balance the data across them. Second, it uses erasure codes to store a fraction of the total amount of data in each of the participating clouds, reducing the cost of migrating from a single provider as we will only have to migrate a fraction of the data.
- **Loss of privacy** - The cloud service provider that hosts the data has access to not only the data but also everyone who accesses it and their access patterns. The obvious solution is to encrypt the data, but when working with a distributed environment we would then have to distribute the keys across everyone that needs them increasing the chance for it to be leaked. DepSky solves this by employing a secret sharing scheme and erasure codes to avoid storing clear data in individual cloud providers.

This system does not however solve some issues that we intend to solve. Namely the write speeds are not very good as this service was designed to provide availability in geo-dispersed systems. To do this it uses replication, which means that when we want to update a given file, we will have to update that file on multiple replicas which means the write latency and the amount of information we have to send from the client is very significant. It also does not have a method for dealing with concurrent writing since it's designed for a use case where each process has its own data and no two processes are going to try to modify the same data.

In terms of performance DepSky also does not employ any type of local caching strategies (neither disk caching nor memory caching), therefore increasing access latency (as the clients **always** have to fetch the latest file version from the cloud-of-clouds) throughput as the accessing of the files is always limited by the network speed of the clients and clouds and also increasing the cost of the system since downloading data from the clouds is usually the most expensive operation. It also means that we can't preemptively cache files that we know might be accessed by the users, which would greatly reduce the access latency to the files that were correctly predicted to be needed. This lack of caching also hurts write performance, as we can't absorb the

local changes in the cached files and then perform the updates to the cloud in the background when the client is done updating the file (reducing the number of requests that we have to make to just 1 big request instead of many small requests, which allows us to take advantage of the network throughput and reduce the number of messages sent, reducing the effect latency has on our utilization). Instead, DepSky needs to push all changes to the clouds as they are being made, meaning the clients would notice a considerable performance drop when editing files that are stored in the DepSky system.

3.5 Charon

Charon [35] is a virtual cloud-backed storage system, implemented in Java and capable of storing and sharing big data in a secure, reliable and efficient way. It uses multiple cloud service providers to create an effective *cloud-of-clouds*. This allows Charon to assert no trust is placed on any one entity, no central client-managed server is required, it's able to efficiently deal with large files over a group of geo-dispersed service providers and no single set of $<= f$ (where f is the amount of concurrently faulty cloud service providers we want to support) cloud storage providers are able to access a file stored in the cloud of clouds.

Charon supports cloud service providers that are not able to execute any of their users' code which means they can only be accessed and modified using a set of functions that may be different amongst all service providers. Charon is implemented as a full client-side software and offers a near-POSIX file management interface that can be used to store and share data using a cloud-backed storage system.

It differs from the previously presented DepSky in Section 3.4 by introducing various caching techniques that allow it to improve performance by taking advantage of local storage and even RAM in the clients in junction with the back-end storage in the cloud service providers and also by the novel *data-centric* protocol to coordinate metadata writes.

The caching algorithm works by using the local disk to cache the most recently used files by the client while also using RAM caches for improved data access while the files are open. This approach allows Charon to save a lot of cost in the system as the clients do not have to download all the files they need every single time, instead, they only need to get the new versions of files that were changed versus the files cached locally, therefore, saving a lot of cost on bandwidth on the cloud service providers. Charon handles the data in fixed-sized 16MB blocks. The 16MB value was chosen as it showed the best tradeoff between latency and throughput and also because they are small enough to fit into RAM and load to/from the local disk. These blocks are encrypted, encoded and split into $3f + 1$ blocks through a storage-optimal erasure core. These split blocks are then replicated across $2f + 1$ cloud providers, in the best-case scenario. For a client to then be able to rebuild a block, he needs at least $f + 1$ of those blocks. Once a certain block is stored in a cloud provider it will never be altered (it is a **one-shot register**). Instead, whenever we want to make a change to a certain block, we upload the entire new version

of the block then change the referenced block location in the metadata to the newly uploaded block and finally we delete the old version of the block.

The novel *data-centric* protocol allows Charon to not only avoid write-write conflicts but also resist against f Byzantine faults. The data-centric design of this algorithm means it does not require the cloud service providers to execute any code as all of the logic is client-side. It works by using leases (time-based contracts) to control race conditions and version conflicts. This algorithm guarantees that only one non-faulty client can access a certain resource at any given time and that the access will have a limited duration. Each resource has three operations: `lease(T)`, `renew(T)` and `release()`. The *lease* and *renew* operations are for acquiring a new lease for T time units and renewing an existing lease for an additional T time units, respectively. The *release* operation releases an existing lease. They guarantee these properties:

- *Mutual exclusion* - There are never two valid leases for the same object at any given time.
- *Obstruction-freedom* - If a resource has no contention (no one is accessing it or trying to access it) a lease request will succeed.
- *Time boundedness* - A lease will be valid for at most T units unless renewed by its owner.

This algorithm uses a modular approach to build non-fault-tolerant *base lease objects* on top of each cloud service provider and merge $3f + 1$ of these objects to form an *f-fault-tolerant* composite lease object. This allows the usage of more efficient base lease objects (like queues, databases, etc...) and greatly improves performance. To acquire a lease, a client simultaneously calls *lease* on $3f + 1$ base lease objects and waits for $2f + 1$ successes or $f + 1$ failures, whichever comes first.

Here are the main issues Charon attempts to address:

- **Loss of availability and loss/corruption of data** - When using only a single cloud service provider, we know that there is an inherent risk that the cloud service where the data is located can become unavailable for many different reasons (the datacenter's connection can be cut, hardware might fail or the entire datacenter might collapse due to natural phenomenon). This is solved by Charon's ability to efficiently deal with multiple cloud storage providers.
- **Loss of privacy** - When using a single cloud service provider, that provider has access to all the data that is stored within it. Even if the data is encrypted by the clients the cloud service provider can see who accesses the data and their access patterns. Charon solves this by encrypting, encoding and splitting the blocks into $3f + 1$ pieces. The client then requires $f + 1$ pieces of the block to be able to decrypt the original block.
- **Controlled file sharing** - When working with Big Data sharing the information becomes quite a difficult problem. Charon solves this with a cloud-based access control protocol

that allows the owner of the file to decide its permissions. This in turn allows the users to control who can access their information (or if it's meant to be private). Because this protocol is cloud-based it does not require any trust in the Charon clients.

This system solves a big part of the problems that are present in distributed systems as we have just seen, however, there are still a few problems with its approach. It requires extra resources for managing the metadata and its lease protocols, it doesn't handle mixed workloads (many clients reading and writing concurrently in the same place) very well as well as being optimized for Big Data and therefore has lacklustre performance when the data it is handling does not fit into its ideal moulds.

3.6 SCFS

SCFS [14] (*Shared Cloud-backed File System*) is a cloud-backed file system written in Java with reliability, durability, privacy and file sharing as its main purposes. It utilizes various eventually consistent cloud service providers to provide strong consistency and a near POSIX file API. Using various cloud service providers allows SCFS to not trust any singular cloud provider with the security, availability and integrity of our data.

SCFS provides a pluggable backend that supports various types of cloud storage providers even if they have different storage layouts and access patterns. SCFS is a fully client-side software as these cloud service providers are not capable of executing any user-given code and they are only accessible via the provided set of functions from the service provider.

SCFS integrates classical ideas like consistency-on-close, separation of data and metadata and also contributes with some novel cloud-backed storage techniques:

- **Always write/avoid reading** - Always push the local writes to the cloud but if possible (if there are no new versions in the cloud), read the file from the local storage.
- **Modular coordination** - SCFS uses a fault-tolerant coordination service instead of the more usual locks and metadata management. This has the benefit of assisting the management of consistency and sharing while having the modularity needed for this instance to allow different fault tolerance tradeoffs to be supported.
- **Private Name Spaces** - A new data structure was used to store metadata information about files that are private to a given user (not shared). These files are treated as a single object in the storage cloud, thus relieving the coordination service from keeping information about these private files (that by their own nature cannot be subject to write-write conflicts) and therefore improving the performance of the system.
- **Consistency anchors** - This novel mechanism allows SCFS to assert strong consistency instead of just eventual consistency which is typically *unnatural* for the majority of

developers. By using this mechanism, SCFS can provide a near POSIX familiar API for accessing the file system without having to modify the back-end cloud services.

SCFS splits its architecture into three separate main components: the *backend cloud storage* that stores the file data, the *coordination service* to store and manages the metadata and the *SCFS Agent* that implements the API for the clients to interact with the system.

This separation of parts allows for a lot of flexibility. An example of this is that the separation of file data and metadata allows for parallel access to files in parallel file systems. It also allows us to interchange the implementation of given sections of the program without having to re-engineer the entire code which gives a lot of possibilities for the end user to choose how he wants each part to function so it can best fit their needs.

A *coordination service* was chosen for metadata storage for three main reasons: Firstly they offer consistent storage and can thusly be used as *consistency anchors*. Secondly, they implement complex and effective complex replication protocols that ensure *fault tolerance*. Thirdly, these systems implement operations with *synchronization power* which can be used to implement basic functionality needed for an operating system like locking a file to prevent *write-write* conflicts.

File data is maintained not only in the cloud but also locally to each client in a cache, like the program referenced in Section 3.5. This strategy allows it to save a lot of money as it can avoid reading from the cloud (unless there is a new version of the file available) as well as improve the performance and the availability (even if the cloud-backed storage is not online, if the files are available locally they can still be accessed).

The implementation of SCFS allows for various modes of operation that allow us to customize it based on the consistency and sharing requirements of the users. The first mode of operation, *blocking*, is the mode that has been described until now and provides strong consistency.

The second mode, *non-blocking* is a weaker version of SCFS in which closing a file does not block until it is completely uploaded to the clouds. In this mode, the metadata is updated and the associated lock is only released after the upload has been completed thusly maintaining mutual exclusion. This mode of operation allows for improved performance at the cost of reducing the durability and consistency guarantees.

The third and last mode of operation is the *non-sharing* mode. This mode is useful for users that do not need to share files. It does not require the use of a *coordenation service* as there is no possibility of having *write-write* conflicts. Instead, all of the metadata is saved in a personal namespace.

3.7 MetaSync

MetaSync [26] is a distributed, secure and reliable file synchronization service implemented in Python that uses multiple clouds as untrusted storage providers. To assure global consistency

is achieved across all the storage providers a novel variant of Paxos *pPaxos* was developed to provide efficient and consistent updates on top of the unmodified APIs provided by the cloud storage providers (MetaSync does not require the back-end services to execute any custom code). It also introduces a novel stable deterministic replication algorithm that allows it to minimize the reshuffling of already replicated objects when there is a reconfiguration of the back-end cloud service providers (such as increasing capacity or adding/removing a service).

MetaSync allows users to synchronize files without having to expose themselves to any fragility, unreliability or insecurity associated with any single cloud service provider. By using multiple cloud service providers it is able to guarantee confidentiality, integrity and availability without having to trust any of the individual cloud service providers.

In order to support any number of cloud service providers (along with the ability to add support for any novel ones) MetaSync provides an abstraction for the file data and metadata that can then be implemented to support any different storage layout or access pattern that the storage provider might require. It is assumed that service failures are independent, each service provider implements its own API correctly and complies with the guarantees that are provided and that communications between the clients and the service providers are secure. It can handle up to f crash failures until it loses its guarantees. This means that by default MetaSync is a *crash fault tolerant* software. We will also consider some extensions that allow up to f cloud service providers to have faulty implementations of their API, be actively malicious and other types of unpredicted faults (these extensions are *Byzantine fault tolerant*). The main goals of MetaSync, taking into consideration these threat models, are:

- **No client-client communication**: Clients can coordinate through the synchronization services without requiring any communication among them. That is, clients are not required to be online concurrently for them to achieve synchronicity.
- **Availability**: Files are always available for reading or writing, even if there are f concurrent failures (either *crash faults* for the default implementations or *byzantine faults* for the extensions mentioned above).
- **Confidentiality**: No f faulty cloud service providers can gain access to any data stored on the service.
- **Integrity**: No f faults in the underlying services can result in a loss or corruption of any files.
- **Capacity and performance**: The service should improve performance compared to single cloud-backed storage services by combining the capacity of the underlying services.

MetaSync consists of three major components, namely the *synchronization manager*, the *storage service manager* and the translators.

The *synchronization manager* ensures that every client has a consistent view of the synchronized files by using the novel *pPaxos CFT* algorithm. The *storage service* implements a

deterministic, stable mapping that allows the replication of the file objects with minimal shared information. The *translators* implement optional modules for encryption and decryption of file data, integrity checks for retrieved objects and many more as the translators are built on top of a modular API that allows the user to create new extensions.

The management of files and their versions is handled in a similar way to how Git handles them. Objects are identified by the content of their hashes to avoid redundancy. Directories form hash trees (data structures similar to Merkle trees), where the root directory's hash is the root of the tree. Each individual file is split into smaller objects called *Blobs* to maintain and synchronize large files efficiently. In MetaSync, there are three types of objects: *Dirs*, *Files* and *Blobs*. Each of these files is uniquely identified by the hash of its content appended to the type of object it is. *Dir* objects contain the names and hashes of the files contained within them. *File* objects contain hash values and offsets of the *Blobs* that make up the content of the file and the *Blobs* are the objects that actually store the content of the files.

In addition to this object storage system, there is a separate system to maintain the metadata in order to provide a consistent view of the global state. This metadata is split up into two different types: *shared metadata* which can be accessed and modified by every client and *per-client metadata* which can only be modified by the owner of the file. The *per-client metadata* is pretty simple as it cannot suffer from any type of *write-write* conflicts. *Shared metadata* however is a lot more complicated. When we want to alter a piece of *shared metadata* that can be accessed and altered by multiple clients and still maintain a consistent view afterwards, all clients must agree on the sequence of updates that were applied to the state. However, clients do not have direct communication between them so they must rely on the storage services to achieve a consensus. To achieve this without running any custom code in the cloud service providers a novel consensus algorithm was devised named *pPaxos* (passive Paxos).

In the classic *Paxos* algorithm, there are two types of clients: proposers and learners and a type of backend service, the Acceptor. It is the job of the *acceptors* to prevent inconsistent proposals and to retry failing proposals. The state only advances when a majority accepts a given proposal. It is not feasible to assume that the backend services that are going to be used implement the *Paxos* acceptor algorithm, so we will only require them to provide an *append-only* list that is capable of *atomically* appending an incoming message to the end of the list (this capability is either readily available or can be easily layered on top of the provided storage services). This list allows backend services to act as *passive acceptors* instead of the regular *acceptor*.

pPaxos utilizes these *passive acceptors* in the following way: Instead of the acceptors deciding what proposals are going to be accepted or not, this action will be performed by the proposers. Each of the clients keeps a structure for each backend service that contains the state it would possess if it ran the *Paxos* acceptor protocol. For a client to send a proposal it must perform the following steps:

- Send a `PREPARE` to every backend service with the proposal number. This message will be

appended at the end of the list contained in the *passive acceptor*. The proposal number must be unique.

- The client will then verify whether the proposal was accepted by fetching and processing the list of each of the *passive acceptors*. It will abort the proposal if any client inserted a larger proposal number in the log.
- Similarly to *Paxos*, the proposer will then propose the highest numbered proposal that was accepted by any *passive acceptor* as the new root. It will send this value as an `ACCEPT_REQ` to every *passive acceptor* so it can be appended to its log. This value is committed if there is no higher-numbered `PREPARE` requests in the log.
- When the new root is accepted by the majority, it can conclude that the new value has been committed and thusly stop trying. If the operation fails, the client chooses a random exponential back-off and tries again with a larger proposal number.

Chapter 4

FeBFT

FeBFT is an efficient BFT SMR middleware library, descendent from the aforementioned protocols PBFT and BFT-SMaRt. It follows the assumption of $3f + 1$ nodes in order to tolerate f faults. These n nodes are responsible for replicating the state of an abstract service that the library user defines. The faults tolerated by this system are of byzantine nature meaning up to f nodes can fail in any arbitrary way (malicious attack on the system, software bugs, bit flips and any other possible fault on top of the usual crash fault tolerance guarantees).

One of the features that make this library interesting is that it combines the memory safety of garbage-collected languages such as Java (which is the language of choice of the previously mentioned system BFT-SMaRt) with the performance and low overhead execution of directly compiled languages such as C (which was the choice for the PBFT implementation) and C++. We are able to provide these guarantees because of the programming language of choice for this library, Rust. Rust is capable of providing memory safety guarantees at compile time due to its Ownership model of handling memory management which removes this responsibility from the programmer and therefore allows us to write provably correct memory-safe programs without the developer having to manually prove and audit the code for memory leaks. We will discuss more this system further ahead.

Another feature provided by this memory model is the ability to detect at compile time various very common concurrent programs such as race conditions, use after free, amongst others. It also endorses the utilisation of a concurrency model based on message passing across various threads instead of utilising shared memory alternatives. This is a much safer way of programming concurrent programs compared to shared memory. However, it does still support the more normal shared memory model with the utilisation of access control primitives (like mutexes, read-write locks, etc) or with the *unsafe* keyword. Unsafe code allows the developer to go around the compiler constraints which means that he is required to assert code correctness. This can allow us to achieve higher performance levels if we know what we are doing but also opens up possibilities for concurrency or memory management errors.

These properties allowed us to compose a system that provides these principles:

- *Modularity* - FeBFT implements Mod-SMaRt, a modular SMR protocol. This protocol diverges from classical monolithic SMR systems like PBFT by separating the consensus primitive from the replication logic, effectively making them easier to comprehend. Additionally, FeBFT further attempts to improve on this by simplifying the process of switching between different dependencies or swapping out entire sub parts of the system without having to do significant code overhauls.
- *Extensibility* - FeBFT also set out to provide a clean-slate platform for the development and research on BFT SMR systems, deviating from the JVM ecosystem that was introduced by BFT-SMaRt. Due to this, FeBFT's code should be extensible in a number of different languages, with extraordinary simplicity.
- *Ease of Use* - End users of FeBFT should not have to spend much time studying and analysing the entire library in order to use it. The amount of boilerplate and library-related code that the user needs to implement should always be kept at a minimum, allowing the user to focus on his service and his service alone.
- *Performance* - FeBFT should have great throughput in most situations without ever having to put the integrity and robustness of the service in jeopardy. This is assisted by the choice of language since it does not require a garbage collector nor an interpreter to translate byte code into machine code. It is also assisted by the extremely parallelized design of FeBFT, which attempts to make use of the entire underlying system.

Utilising Rust also provides us with a much higher level of inter compatibility with other languages like C and C++ which themselves are then able to interact an enormous amount of other languages which provides the user of this library with a lot of flexibility compared to previous systems effectively locked the programming language choice to the language used to implement the system.

4.1 Replica node design

The internal design and implementation of the consensus, state transferring and leader changes are very similar to the protocol implemented by BFT-SMaRt (which itself derived from PBFT). It is composed of various modules which each correspond to a particular part of the sub-protocol.

- **Executor** - The service implementation that should be implemented by the end users which sits on top of FeBFT.
- **Consensus** - Is responsible for deciding the ordering of the client requests.
- **State Transfer** - Delivers state and log updates to replicas that either have suffered a fault or just drifted out of sync with the quorum.

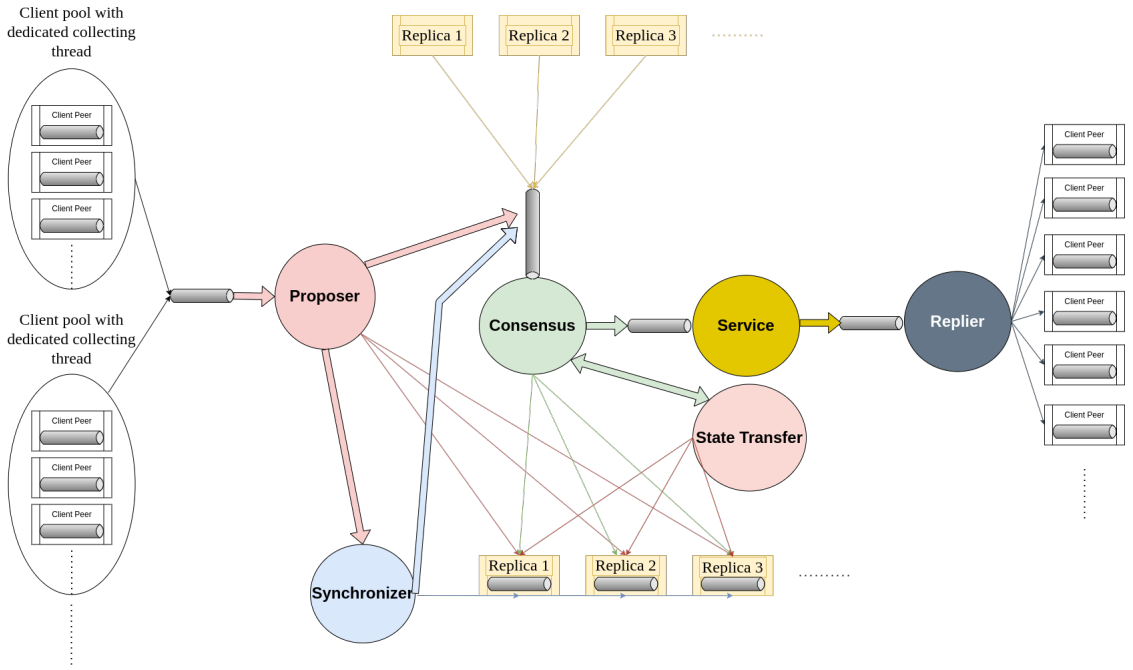


Figure 4.1: The design of FeBFT, providing an overview of the flow of requests through the system.

- **Networking** - Responsible for connecting and sending/receiving information from peer nodes.
- **Synchronizer** - Handles faulty leaders and orchestrates different consensus instances, derived from BFT-SMaRt's Mod-SMaRt.
- **Proposer** - Collects client requests from the client pools, aggregates them into batches and proposes them to the Consensus.
- **Reply Thread** - Responsible for delivering the replies generated by the service to the clients.

There are also some other parts of this system that do not appear in this diagram since they are smaller parts of the overall SMR design but they will be described further in Chapter 4.

This BFT SMR system is meant to facilitate future research work by using it as the base for more complex, byzantine fault-tolerant services. Furthermore, since it is based on already proven work, it eliminates the need of having to prove each of the system's sub-protocols, since there already is literature proving them[40] [15]. We utilized this so we could better focus on actually improving the architecture around the base protocols in order to facilitate further usage by other developers and most importantly in order to make a better scaling, better performing protocol.

As we discussed previously, we wanted to make FeBFT as parallelised and as performant as possible. As shown in previous work [11][23], speedups with increasing number of threads are subject to Amdahl's law [27] [25]. So, our objective was to have very well-defined roles for each

of our threads, roles which have been carefully thought out so we can minimize shared memory locations so that each worker is not creating performance penalties for the other workers. To achieve this, we followed Rust's message passing [34] design to implement our parallelization, by utilizing channels to pass information between each of the threads. This allowed us to get very good performance and CPU utilization since each of the core parts of the program is able to run without disturbing any of the other parts. The utilisation of channels also provides us with another advantage versus utilizing other architectures, as it provides buffering to the interactions between each of the components. For example, if there is a spike of operations done and the Executor cannot keep up to that level, the channel can act as a buffer so when the system's load is reduced it is able to process the built-up requests, without ever having to reject requests or slow down the consensus. It is naturally able to absorb times when demand is above the expected maximum. This comes of course at the cost of some RAM and the actual size of the buffers should be adjusted to fit the available memory of the system.

However, one possible argument that could be made is that it would not be able to properly capture and utilize the full performance potential when executed on potent hardware (since the amount of threads is already decided by the architecture and does not change with the number of processors available). To combat this drawback, this design was combined with a customisable thread pool (that is not represented in the picture) to handle more CPU-intensive tasks that are not as bound to the actual logic of the middleware and therefore would just slow down the threads of whichever component was responsible for performing them. Tasks like signing/verifying message signatures and creating speculative messages so the consensus thread is not slowed down at the creation of the messages, amongst other varied uses. Utilizing a thread pool for this type of work allows us to scale even better since we have the core defined threads that always execute the same amount of work (since the amount of consensus instances that can be executed in a given time period is physically limited by the latency of message delivery, as we will see with further detail in Section 4.3) while being able to scale with the amount of available processing power to handle having to sign and verify many more individual requests.

State transfers, consensus and synchronizations all run on the same thread, which is the main replica thread. This thread stores the current phase of the algorithm which is used to decide which state machine should be triggered in order to advance the consensus. Thusly, we know that there will never be any type of concurrent iteration with these 3 separate state machines. This means that we do not have to account for multiple threads trying to access and modify the same data so we do not need to protect the data of these 3 modules with synchronization primitives. Instead, they are all safe to be accessed from each other for whatever is necessary. This also does not hurt the performance of the system at all because if the protocol that needs to be run is the state transfer protocol then the system is not in the condition to run the consensus protocol anyways, so having it on a different thread would just be another hassle that we would have to deal with that would yield no performance benefits. The same can be said for the synchronization protocol. The latter does however have another part that is handled separately from the actual synchronization protocol.

4.1.1 Consensus

The consensus module is responsible for handling the consensus messages received from other replicas as well as the proposed batches by the proposer. It maintains a state machine to keep track of the current consensus instance being decided, which stage of the decision it is currently on and also generates and dispatches the consensus messages that are required for the system to advance its state (e.g. Broadcasting the Accept message in response to a correct Pre-Prepare request send by the leader). As we have also seen, relying on the synchronicity of messages sent by the network is not feasible since any message can get lost, reordered or delayed for an indefinite amount of time (which is a very large reason for the appearance of this type of BFT SMR system). Apart from requests sent by clients, the messages relating to the actual consensus protocol must be processed in a specific order. Particularly, messages of the same type do not have to follow any restrictive ordering (for example, the ordering of accept or commit messages is indifferent to the result of the consensus) while different types of messages do have to follow a particular ordering (e.g. commit messages must be processed when the consensus state machine is already in the commit phase and not while he is in the accept phase). Another possible issue caused by networking problems is processing messages that pertain to consensus instances that have either already been decided or that we have yet to reach.

Therefore, the consensus module is also responsible for handling the correct ordering of the consensus messages so they can be processed correctly without causing issues. To do this, the *TboQueue* was implemented. This queue is aware of the current consensus sequence number and phase and uses that information to order and decide which message the consensus thread will process next.

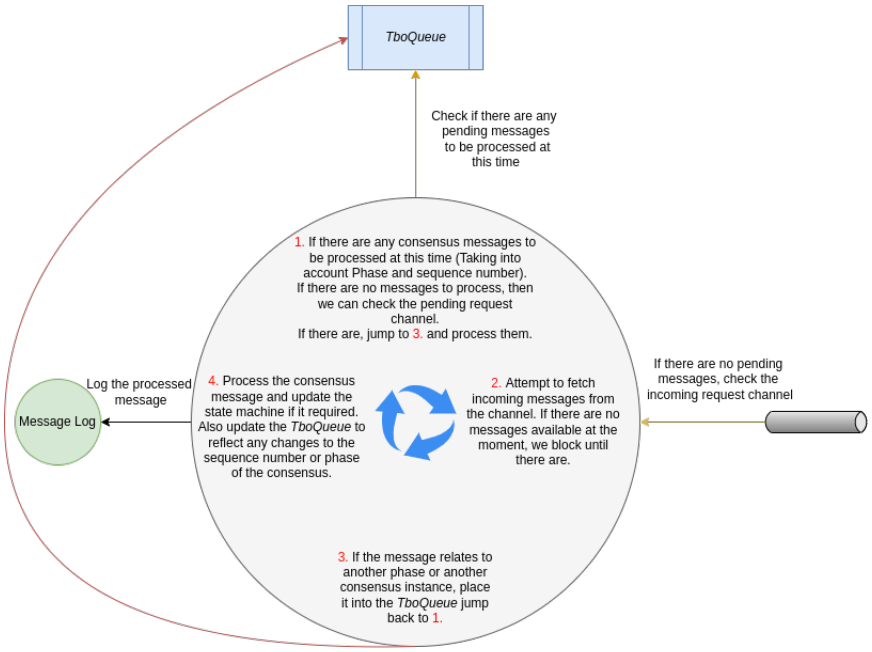


Figure 4.2: An illustration of the basic consensus module functioning.

This is illustrated by Figure 4.2, which demonstrates the basic functioning of the Consensus module. We do not go into more detail on how the state machine operates and what phases there are and which messages must be sent because, as we have already mentioned, the design of the state machines was based on already proven designs. In the case of the consensus, the implemented state machine is directly derived from the PBFT state machine, even keeping the same phase and message names. Another detail we are able to see in the Figure is the fact that every message that gets processed by the consensus must then get stored in the message log. This is so we can not only rebuild our own state in the case of any problems but also so we can help other replicas recover from faulty positions so they can be part of the consensus again. We will discuss more about how this logging works in Section 4.6.

The entire log is contained within this module, since it (along with the state transfer module) is the only one that will actually need access to it. The consensus module also handles requesting and managing checkpoints from the service when necessary in order to limit the size of the log such that long-running systems do not run out of memory. Because of this, the log and the consensus module were designed in a way that allows for unrestricted access (without having to resort to synchronization primitives such as locks) since the only components that will be accessing these modules are the Consensus and State transfer modules.

4.1.2 State Transfer

The state transfer module is responsible for repairing the state or completely integrating a new replica into the quorum. This protocol is conducted by a replica dubbed *leecher* which fetches state and operation logs from *seeder* replicas. The *leecher* then takes these states and applies the operation logs in order to obtain the same state as all the other quorum replicas. The actual state is fetched from seeders with older checkpoints, while seeders with more recent checkpoints will only send validation data so the *leecher* can verify the state it was sent is correct. The basic idea of CST is for *leechers* to get at least f+1 validators in intermediate checkpoints, taken by executing the received client requests in the log entries, thus yielding the same state as the *seeder*.

4.1.3 Synchronizer

This module is responsible for performing view changes (basically leader changes) and when required, summoning the state transfer or consensus protocol. It is also responsible for discovering faulty leaders, by having timeouts on requests it receives from clients. Every request received by a non-leader replica starts a timer that only gets cancelled when the request is received in a valid PrePrepare message from the leader. If the leader is faulty and is omitting requests or delaying the operations then this will trigger a timeout on the request.

It is based on BFT-SMaRt's protocol, which is similar to PBFT in the number of messages and exchange pattern utilized, with three distinct messages exchanged to perform a view change: STOP, STOP-DATA and SYNC. One of the key differences to the PBFT protocol is the fact

that they require two timeouts on given requests in order to actually launch a view change. The first timeout will just lead to a request forwarding from the replica that has timed out the request to all other replicas present in the quorum. What this does is protect the system against malicious clients that were attempting to attack it, using an omission attack. This attack consists of sending a request to only a partial set of replicas in the quorum without containing the current leader. Performing this attack without BFT-SMaRt's first timeout forwarding could lead to clients being able to selectively choose the leader or just DoS the service by causing it to constantly change views, therefore, disturbing its performance.

4.2 Client design

Similarly to a great deal of other BFT SMR systems, FeBFT clients do not perform any critical operations to the functioning of the system. Instead, they are just limited to sending requests and delivering replies to the user in an RPC-type fashion.

Clients have the ability to call the remote service asynchronously or synchronously, depending on the needs of the user. The client requests work with sessions. Each session can only have one concurrent request but each client can have multiple concurrent sessions, meaning we are able to effectively perform multiple concurrent requests. This library makes use of the *async/await* features for client requests and performing a request returns a future that can then be awaited by the user in order to obtain the reply (when it is received).

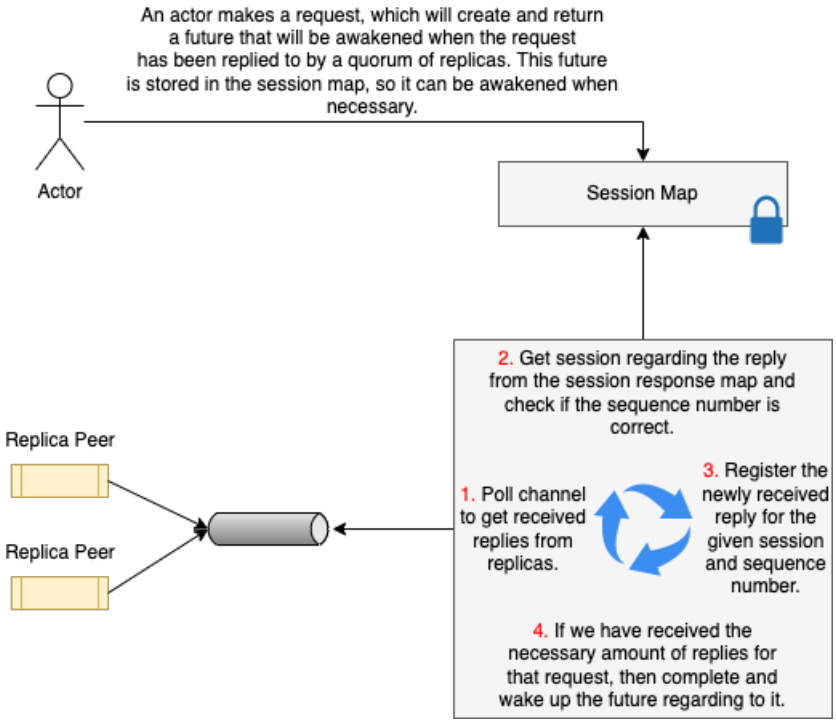


Figure 4.3: The background architecture that exists so end users can have a clean and simple API

As Figure 4.3 shows, there is quite a bit of background operations going on in order to provide this clean and simple interface to the end user. Namely, we have a message processing thread which is responsible for taking the replies received from the replicas (handling of replica connections and other network-related information will be discussed in Section 4.5). This is complemented by a shared session map, where the client will place the response map corresponding to the session. We do not use the sequence number as part of the key because as we saw each session can only have one concurrent request. We do still store the sequence number for the request so we can verify if the reply is for the correct request. When we have received a quorum of identical replies for a given request, we mark it as ready and if the client has already *awaited* the request, the task that is responsible for it will be awakened. If the client has not *awaited* the request then when he does, it will be instantly responded to as we already have the cached response.

We also implemented a callback-based system which utilises the same infrastructure as the asynchronous implementation we have just seen. The difference is that instead of storing the *Waker* related to the section, it will instead accept a function that takes the response as an argument which will be called as soon as the request has been replied to. This function call is executed on the thread pool in order not to disturb the execution of the message processing task, which is key to the client's performance.

4.3 Improving batching

To understand why the performance of FeBFT was so poor, we must look at how the State machine algorithm works. We know, as we have seen in the background for PBFT in Section 2.1, that for a byzantine SMR algorithm to reach a consensus, we require 3 rounds of messaging between the quorum of replicas (pre-prepare, prepare and commit). This means that we are physically limited by the inherent latency of sending network messages to other computers (which exists even in local networks, observed in this case to be around $75 - 100\mu s$ dependent on many factors which can make it a lot worse).

In practice this means that the algorithm is theoretically and unavoidably limited to around 4444 consensus instances per second due to this factor (taking into account the 3 necessary message rounds at a $75\mu s$ latency). In practice, we observed this number to be closer to $1250 - 1750$ instances per second, wildly dependent on many factors. We cannot fix this as it is impossible to increase the speed at which the requests travel through the cables and this problem will only be exacerbated if we are attempting to deploy a system where the replicas in the quorum are geographically distant (which makes complete sense, as we are designing a system that is meant to tolerate failures, so placing all the replicas in the same data centre or even the same geographical region leaves us open to consequences from natural disasters, power outages, etc). In this case, the latency increase is felt for each of the 3 messaging rounds, so in reality, increasing the latency between the replicas by 5 milliseconds will lead to an increase of 15 milliseconds for each consensus instance which means that we will only be able to perform a

theoretical 66 consensus instances per second (disregarding processing time and other variables, just if we were only limited by latency, in reality, this number will be quite a bit lower), compared to the 1250 rounds locally. We can quickly see this becomes a very large limiting factor even with small increases in latency.

To mitigate this issue, we must make the main limiting factor of scalability something that we are actually able to control and scale. In this case, we might not be able to reduce the amount of time the messages take to reach their destination but we are able to increase the bandwidth at which they are sent. This means that we should point towards bandwidth being our limiting factor and not latency which in turn indicates that we should not depend on the number of consensus instances we are able to decide in a given time period and instead rely on how many individual requests we can fit into each of these consensus instances.

4.3.1 FeBFT batching approach

Similarly to BFT-SMaRt, FeBFT also performs batching of requests to maximize the number of operations that it can perform in a given time frame. This is done because the amount of possible consensus decisions that can be made in a given time frame is physically limited due to the necessity of having 3 messaging rounds. The sending of these messages contains an inherent latency that cannot be removed. This latency is exacerbated greatly if the replicas are placed in geographically distinct locations (the further away the more effect this will have on the number of consensus decisions that can be made). This is because the speed at which information can travel is naturally limited to the speed of light. The ability to maintain performance even with distant replicas is extremely desirable for systems which require such strict consistency and fault tolerance since having all the replicas of the system placed in the same geographical area opens the possibility for natural disasters, power outages or other problems that only affect that region to cause the complete failure of the system. So, we cannot rely on the number of consensus instances done per quantity of time for the scaling of operations performed. Instead, we must rely on having as many operations as possible in each of these consensus instances since we can increase the bandwidth of the connection between quorum replicas (but we cannot reduce the latency).

Batching of requests can be one of the largest influencers on the performance of the overall SMR system. If we make batches that are too small, we may be leaving a lot of performance on the table in terms of total operations per second while if we wait too long in order to reach a given target batch size we can increase the latency of the request-response times to levels that are not optimal. In order to achieve a balance, FeBFT chose an approach that is meant to be dependent on the number of requests that are being made at the time. That means if we have a low amount of concurrent requests, the batches should be small and we will not wait long before proposing a new request batch to maintain the low latency. When we have a high amount of concurrent requests, we should make larger batches in order to take advantage of the aforementioned advantages larger batches bring. This approach relies on the randomness of the

task scheduler and on the concurrency caused by incoming request tasks all attempting to push their requests into the next request batch to be proposed.

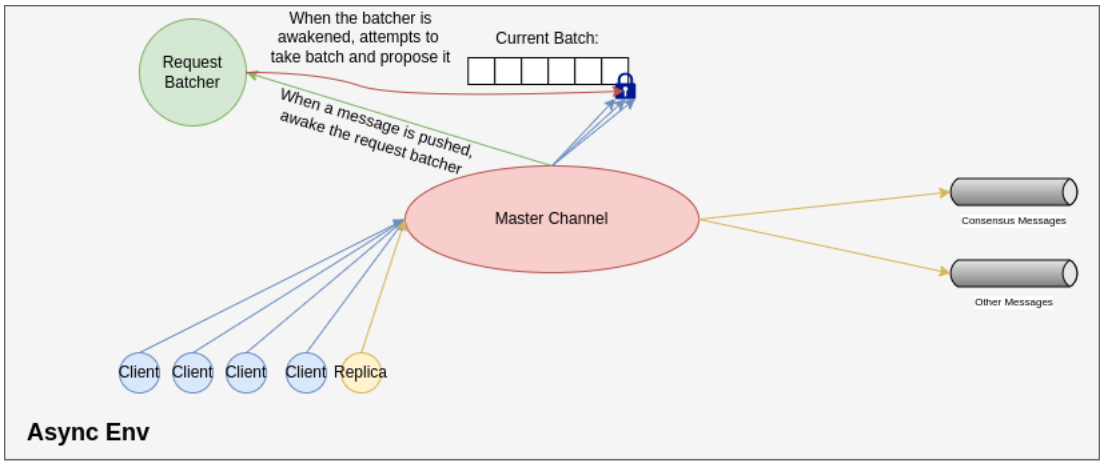


Figure 4.4: The original approach for batching requests in FeBFT

As is pictured in Figure 4.4, we can see that client requests (that follow the blue path) try to acquire the lock to the batch and push their request into it. There is no attempt to cache requests at the client connection level, instead, they immediately try to push the request into the global batch vector. After they have done this, they will attempt to awake the request *batcher*, so it can come acquire the lock, take the batch away and then send this newly created batch to the master channel, so it can be processed in the consensus task. The idea is that the more clients are concurrently attempting to push requests into this Vector, the more time the request *batcher* will take to acquire the lock to the current batch since so many client requests are also trying to access it to push their own requests. This effect should even be amplified with the number of cores we decide to assign to the asynchronous environment as in reality we can only have as many concurrent requests being pushed into the batch as the number of cores assigned to the asynchronous environment. We can also see that if there is little concurrency on this mechanism, the request *batcher* can quickly acquire the lock without much contention and create and send the batch which means the latency is also reduced.

After the message has been created, it is then sent to the master channel. The consensus thread then collects this message, reads all of the requests contained within and stores them in a log (composed of a hashmap). When the consensus is ready to receive a new batch, it will take all the messages from the log and propose them into a new batch.

As we will see further ahead, this approach yields some interesting results and a consistent performance but with some throughput issues. This is discussed in Section 5.2.

We can however observe some architectural problems from this approach. The fact all client requests are immediately sent through the master channel means that it will create a very large point of contention. When we want to maximize the scalability provided by utilizing multiple core architectures, we want to minimize these points of contention since they hurt performance

heavily.

Another point of possible performance loss is the amount of unnecessary context switching that is introduced with having the context thread continuously extracting small batches just in order to add them to another hash map, as well as wasting time that could be necessary for the processing of important consensus messages.

4.3.2 Purpose built data structures to maximise batch size

In an attempt to fix the issue of small batches, we started by implementing some purpose made concurrent data structures that were designed to maximise the utilisation of the natural concurrency caused by the many clients all attempting requests at the same time. In order to do this, there were a few data structures implemented and benchmarked so we could decide which was the best-suited data structure for our use case.

Before we go further into each data structure we must first discuss some common topics that are used in all of the subsequent items.

- ***Exponential backoff*** - Backoff means that we will backoff the active wait for a small amount of time (usually just yielding execution). The amount of time that we attempt to sleep is randomised. The sleep time must be randomised because if we have a situation where many threads are accessing the same critical area causing problems for execution and we set all the threads to sleep for the same amount of time they will all wake up at the same time and we have not solved the issue. Making these times random fixes these issues as they will not come back to execution at the same time. The exponential portion of the statement means that for every time we go to sleep, the max range on the potential sleep time will be multiplied by 2.
- ***Cache Padding*** - A field being wrapped in a CachePadded means that the wrapped variable will not share it's cache line, independent of the size of the variable (for example if we have a 64 bit variable in a 128 bit cache line, the final 64 bits will be filled with junk). We utilise this because when we are working with highly concurrent workloads, multiple variables sharing the same cache line can lead to terrible performance as whenever one of those variables is updated by any thread, the entire cache line will be invalidated and subsequent data reads to a variable that was not changed but shared the line with a variable that was changed will require us to go to RAM and fetch the latest value for the entire line. By assigning an entire cache line to a single variable we prevent this from happening and therefore increase performance.

4.3.2.1 Bounded lock free mpmc blocking queue

This data structure is based on an implementation for a queue that is able to efficiently handle multiple producers and multiple consumers[46]. It's not lock-free in the official meaning of

the word, instead, it was just implemented with atomic read-modify-write operations without recurring to mutexes at any point along with exponential backoff to prevent active waiting. It is a bounded blocking queue which means that if we try to enqueue when the queue no longer has any space, it will block on the operation until we have a slot open for us. Likewise, if we try to remove from the queue when the queue is empty, it will block until there is an element ready for us to remove it.

This design allows us to enqueue and dequeue elements from it with a single compare-and-set operation (no amortisation, just a single CAS operation). It performs no dynamic memory allocation/management during its operation and it also isolates the data accessed by consumers and producers (they do not touch the same data while the queue is not empty) which means the queue is handled with a head and a tail, where producers only interact with the tail while consumers only interact with the head of the queue.

The data structure and its members can be seen in Listing 4.3.

```
pub struct LFBQueue<T> {
    head: CachePadded<AtomicUsize>,
    tail: CachePadded<AtomicUsize>,
    array: Box<[StorageSlot<T>]>,
    capacity: usize,
    one_lap: usize,
}

struct StorageSlot<T> {
    sequence: AtomicUsize,
    value: UnsafeCell<MaybeUninit<T>>,
}
```

Listing 4.1: Data structure of the Lock Free Bounded Queue

Both the head and tail are cache-padded atomic unsigned integers (size depends on the architecture of the underlying system).

The head and tail of the queue store not only their current index in the queue but also the current lap (To allow both the head and tail to wrap around the array).

Each storage slot also contains a sequence number which is also an atomic unsigned integer. This number indicates to us if that slot is ready to be read from / written to. If the slot equals the *tail*, this slot will be the next slot to be written to while if it equals the *head + 1* this node should be the next one to be read from.

We then have the underlying data store (a simple array, meaning this is a bounded queue). This doesn't require any synchronisation because we are guaranteed by the queue and dequeue operations (and the use of atomic operations and variables) that no two different threads are ever able to access the same memory position at the same time.

This data structure however only has the ability to queue and dequeue 1 object at a time so how can it be useful for our purpose which is to accumulate requests and then get them in a large batch of elements instead of just one by one?

To solve this problem, we introduce a novel dequeue method, which has the ability to dump all of the requests in the queue with a single CAS operation, like if it was the regular dequeue method.

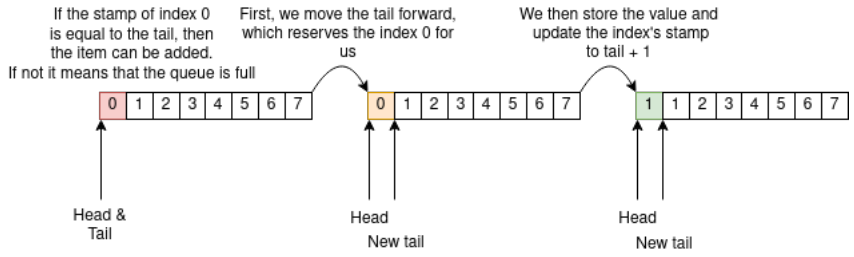


Figure 4.5: The steps of the enqueue procedure

Enqueue operation As we can see in Figure 4.5, the enqueue procedure is pretty simple.

We start by extracting the current index of the head (basically removing the lap from the head value) and getting the slot at that index. We then get the stamp for that slot. If $head + 1 == stamp$, then the slot at that index is ready to be written to and we can proceed. There are two reasons why this can return false, namely:

- There could be a dequeue operation still ongoing that is not done utilising that slot.
- The queue could be full.

If we have the ability to proceed, we will move the head forward one slot (utilising a compare and swap operation with *head*, to make sure another worker didn't come in the meantime and steal our slot). This operation effectively "reserves" the slot for our usage because since we use atomic CAS operations, we know that only we will possibly have control over that index. We then populate the index with the value, update the stamp and the operation has been concluded.

Dequeue operation As we can see in Figure 4.6, the dequeue procedure is also pretty simple.

We start by extracting the current index of the tail (basically removing the lap from the tail value) and getting the slot at that index. We then get the stamp for that slot. If $tail == stamp$, then the slot at that index is ready to be read from and we can proceed. There are two reasons why this can return false, namely:

- There could be a enqueue operation still ongoing that has not yet placed the item in that slot.

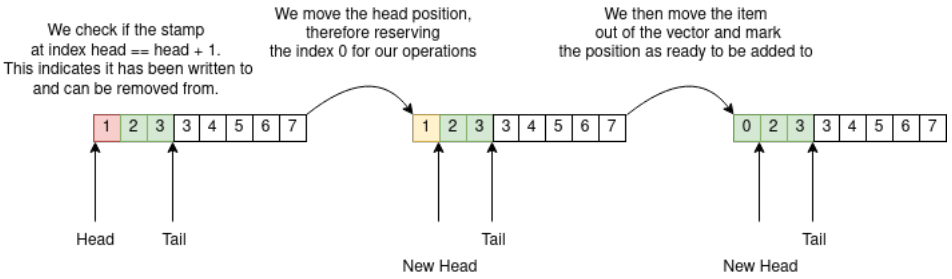


Figure 4.6: The steps of the dequeue procedure

- The queue could be empty.

If we have the ability to proceed, we will move the tail forward one slot (utilising a compare and swap operation with *tail*, to make sure another worker didn't come in the meantime and steal our slot). This operation effectively "reserves" the slot for our usage because since we use atomic CAS operations, we know that only we will possibly have control over that index. We then remove the value from the slot so we can return it to the actor and update the stamp of the slot.

Novel dump dequeue method In a very basic explanation, what this method does is move the head to the same position as the tail, effectively marking every position in between as ready to read from. As described above, we have a sequence number in each storage slot that indicates to us if that particular slot is the next one to be read from / written to. Now, by doing this large shift of the head of the queue we would be breaking the invariant so what we do is *pretend* that we moved the head one by one for the current worker and then we extract the value from each of the slots and mark them as ready for writing. Now since this does not require any more accesses to the head/tail global variables, we know that this does not add any concurrency strains on it.

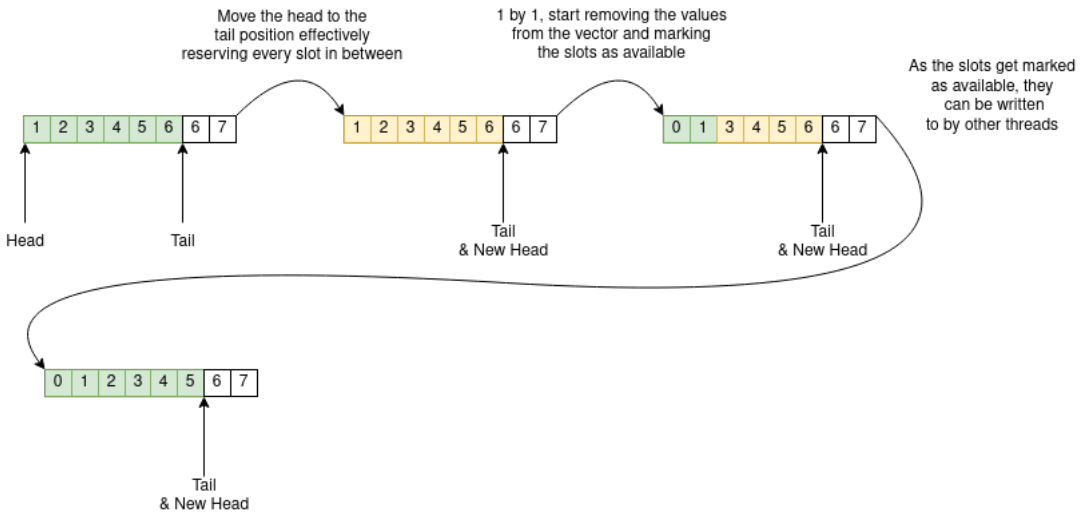


Figure 4.7: The steps of the dump procedure

There are some potential issues with this dump approach to dequeuing multiple elements in a single CAS operation. Namely, since we want to set the head equal to the tail this means that we first have to load the current tail position and only afterwards can we set the head equal to it, which is not an atomic operation (the composition of two atomic operations is not atomic) and therefore leaves an opening for changes in the head and tail in the meantime.

To prevent this possibility, the following steps were taken:

- We know that both the head and tail only move forward, never backwards and that the head will never overtake the tail and the tail never laps the head (we have to look at the array as a circular array where after we reach the end we circle back to the first position).
- We read the current head position of the queue into *prev_head*.
- We read the current tail position of the queue into *tail*.
- We perform a compare and swap on the current *head* with *prev_head* as the element to compare and *tail* as the new element. If it fails, update *prev_head* and *tail* with the new values and repeat until success is found. If it succeeds then we know that the *head* is now pointing to the *tail*, and every element between *prev_head* and *tail* can be removed safely.

What are the possible occurrences with this scenario between the reads and the CAS operation?

- A dequeue operation is made and the head is moved forward up to n slots. This means that the CAS operation will not be successful and we will retry the operation.
- An enqueue operation is made the tail is moved forward up to n slots. This means that we will not capture these new elements that were just added, as *tail* still contains the old value. However, we do not intend to assert that we capture *every* element, as that would require total locking of the queue until the operation is completed. In these moulds, this behaviour is totally acceptable and causes no issues.
- In the case where both operations are performed, then we will fail since a dequeue operation causes it to fail the CAS operation.

Therefore, this method yields a correct and safe amount of elements to remove from the queue.

4.3.2.2 Mutex based bounded queue

A simple array wrapped with a Mutex with passive wait utilising condition variables compared to the active wait of the previously mentioned concurrent queue. The data structure with all it's members can be view in Listing 4.3.

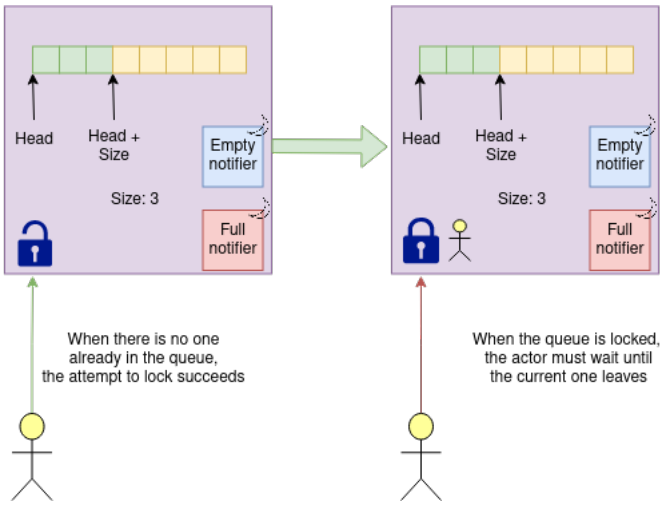


Figure 4.8: The basic functioning of the Mutex based queue

We can see in Figure 4.8 that the struct is composed of the capacity of the queue, a mutex that encapsulates all the queue information like the array, head and current size. We do not need to store the tail position as that can be obtained by summing the size to the current head of the list.

We then have the condition variables for when we are trying to enqueue/dequeue elements when the array is full/empty, respectively. These are used for performing blocking operations on the queue (enqueuing a full queue or dequeuing an empty queue, for example) where we have to wait for other actors to perform actions before we are able to perform our own actions. When an element is added to an empty queue, the $empty\_notifier$ condition var is signalled and when an element is removed from a full queue, the $full\_notifier$ is signalled. This can be seen in Figure 4.9.

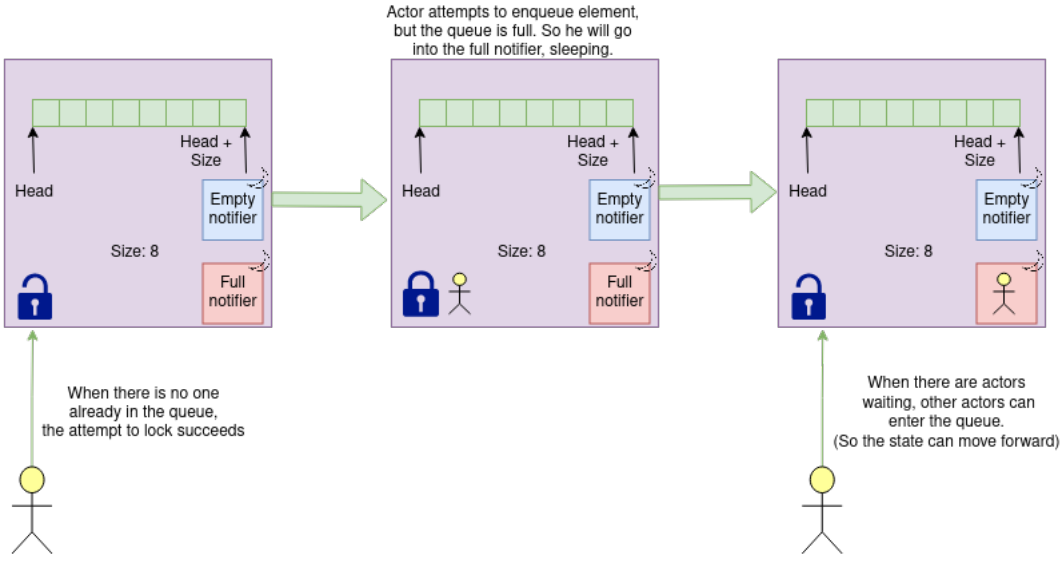


Figure 4.9: The functioning of the queue in situations where the actor has to block

We also have the option to utilise exponential backoff to perform active waiting for a little bit instead of going to sleep immediately after failing to dequeue/enqueue an element from an empty/full list. This could be useful for situations where we are working with a situation where operations are very frequent so we know beforehand that the amount of time the thread is going to be waiting is so small it is not efficient to send the thread to sleep and force a context switch.

In this data structure we do not utilise the CachePadded wrappers because all of the information is contained within a Mutex, which will always force a complete re-fetch of all of the contained values from RAM before allowing the process to access the critical area.

Enqueue and Dequeue operations Since this queue is based on a simple Vector wrapped with a mutex, the operations are basically the locking procedure shown in Figure 4.8 followed by a push/pop operation, respectively.

Dump Operation The dump operation is a bit more complex as we wanted to make it as quick as possible and with as little time within the mutex as possible to prevent us from slowing the advancement of the queue.

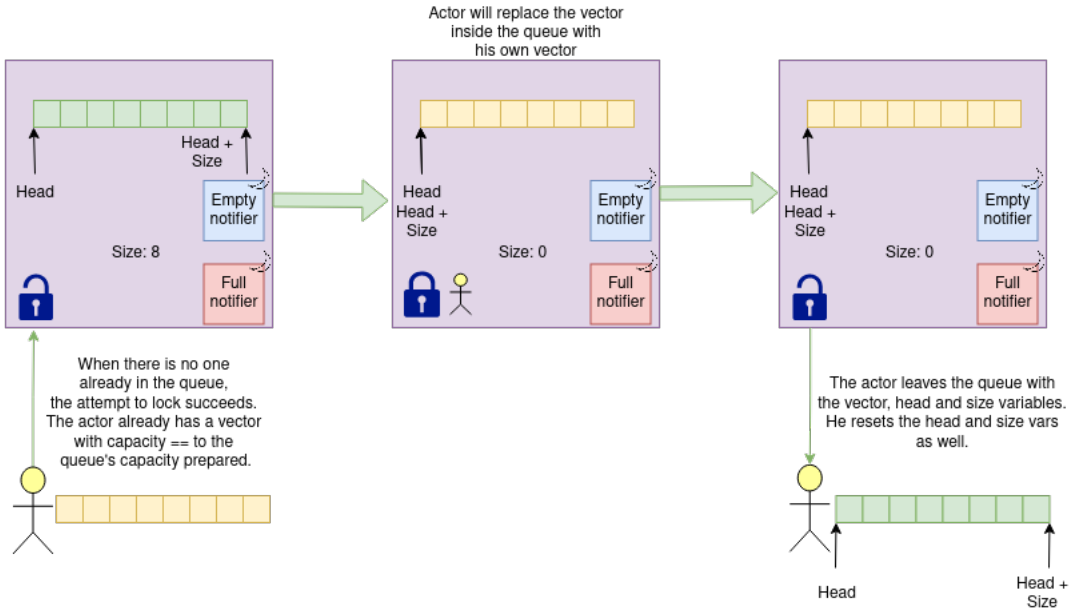


Figure 4.10: The functioning of the queue on dump operations

As we can see in Figure 4.10, the dump operation starts with the actor allocating a vector of the same size as the internal vector of the queue. This is done while he is outside of the Mutex so we do not make the queue wait for the operating system's memory allocation, which can take a long time in certain conditions. Therefore, we only try to acquire the lock when we have all of the necessary pieces to perform the operation. When we are ready, we acquire the lock, replace the vector within the queue and take the head and size variables (and replace them with 0). We now have all the elements outside of the Mutex and can do whatever we want with them without

worrying about slowing down the progress of other actors.

4.3.2.3 Lock free rooms based bounded queue

This concurrent queue was specially thought out and designed with the exact purpose of maximizing the concurrency effect of having multiple clients performing requests at the same time.

Like the data structure mentioned in Section 4.3.2.1, this data structure is not completely lock-free, it just utilises a combination of atomic read-modify-write operations and active waiting with exponential backoff.

The rooms approach means that we have various types of operations, each corresponding to a given *room* and we allow any number of concurrent operations within a given *room*, but we only allow one *room* to be accessed at any given time. This means that if there are many clients performing requests and enqueuing messages we will not be able to enter the dequeue room, leading to large accumulations of requests.

This, in combination with the fact that it is a bounded queue (meaning it has a limited number of slots and when it fills up we are no longer able to enqueue requests so the room will become empty and allow us to dump all of the messages), leads to this data structure being able to very efficiently and effectively accumulate requests when there is a very high amount of actors enqueuing while also not harming the latency of request response when we have a small number of clients and therefore little concurrency on the queue (which means we have opportunities to change room and therefore empty the queue).

The structure of the queue can be seen in Listing 4.4.

As we can see, the data structure is composed of the *array* which is a fixed-size Vector. To keep track of the current *head* and *tail* position we utilise cache-padded atomic unsigned integers.

We then have an atomic boolean variable that stores whether the queue is currently full. This variable exists to prevent us from ever joining the enqueue room just to see that the queue is already full, which would make us have to exit the room again. With a very high number of concurrent requests, this entering and leaving in the enqueue room can cause issues as it can delay the amount of time that we take to join the dequeue room (in reality, there can be situations where we would not ever be able to join the dequeue room). By employing the *is_full* variable to "cache" the current state of the queue, we are able to avoid joining the room when we already know the queue is full and therefore prevent this problem from happening in the first place.

Lock free rooms The main factor for the performance of this queue is the efficiency of the Rooms implementation. Therefore we needed a lock-free implementation that used as few atomic operations as possible. Our solution is an implementation that is capable of entering/leaving

rooms in a single CAS operation. We demonstrate how we were to achieve this in Listing 4.5.

We first start with a State for the current room. We can either have a vacated state (no rooms are currently occupied) or we can have a room that is occupied with n current participants. However as we are able to view, the state is not utilised anywhere in the actual Rooms structure. This is because we condense the State enumerator into a single, cache padded, unsigned 64 bit variable by using the first 32 bits to represent the currently occupied room and the last 32 bits to represent the number of workers currently within the room.

When we wish to join or leave a room, we start by loading the current state, perform the necessary changes if it is a valid operation and then perform a compare and swap operation on the room state. If it is not a valid operation we will use exponential backoff to continue trying to perform the operation. This practically active wait can sometimes be unnecessary however, for example when we have many workers in a room, we know that it might take a large amount of time for all of them to leave it and open the opportunity for us to enter the room. This is what the variables *listener* and *event* are for. When we have been trying to enter the room for an extended amount of time (while using exponential backoff) and we know there are many workers currently within the room, we enter a sleep state which will be awakened whenever the state of the Room is changed to *FREE*.

Enqueue Operation The enqueue operation is derived from the operation described in Section 4.3.2.1 with some large tweaks, given the properties provided by our Rooms implementation. Namely, in that implementation we saw the necessity of a *stamp* on every slot of the queue to know if the slot was ready to be written to, we will see later on why this is not necessary here.

As is visible in Figure 4.11, the enqueue operation begins with the actor attempting to join the room correspondent to the enqueue operation. If he is not successful, then two things can happen:

- If the flag *backoff* is **true**, then the actor will follow the exponential backoff idea. If, after a given number of attempts the acquisition is not successful then the actor will go to sleep until the occupied room is vacated.
- If not, then the actor will immediately go to sleep until the occupied room is vacated.

If he is successful, then we will proceed with the operation. We start by moving the tail forward one position, therefore reserving the position for ourselves (since the head is an atomic integer). We then populate this slot and leave the room.

Notably, this operation lacks the *stamp* utilised in Section 4.3.2.1. The *stamps*, as described before, existed for two reasons:

- There could be an ongoing dequeue operation that had already moved the tail but had not yet removed the element from the vector. This is no longer an issue since, given our rooms

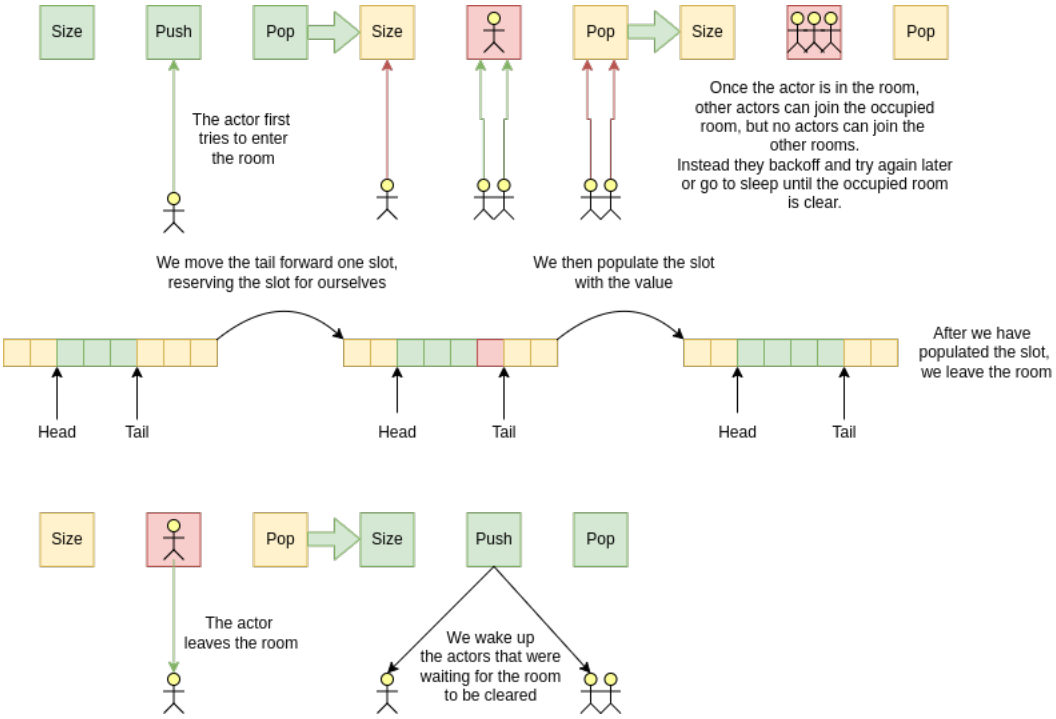


Figure 4.11: The functioning of the Rooms based queue on enqueue operations

architecture, we cannot have concurrent enqueue and dequeue operations and we only leave the room when we have finished placing the item inside the queue, so this situation is not possible.

- The queue could be full. This method solves this because we now have another way of checking for this (even without the *is_full* cache variable). Again, since we can't perform dequeues and enqueues concurrently, we know that we if the head is currently being moved, the tail cannot be altered and vice versa. So, it's possible to load the tail variable and compare it to the head to see if the queue is full and if it is not we then perform a CAS operation on the tail to move it forward. If the tail has moved, the CAS operation will fail and we will try again. If it has not, then we know the queue is not full and we can push the value safely.

Dequeue Operation Similarly to the enqueue operation, this is also derived from the dequeue operation visible in Section 4.3.2.1.

Like we can see in Figure 4.12 the dequeue operation, similarly to the enqueue operation, begins with the actor joining the corresponding room. The steps taken after attempting to join are similar to the ones described in Section 4.3.2.3.

After entering the room, we start by moving the head forward one position utilising an atomic operation, effectively reserving the taken slot. We read and remove the value from the given index and leave the room afterwards.

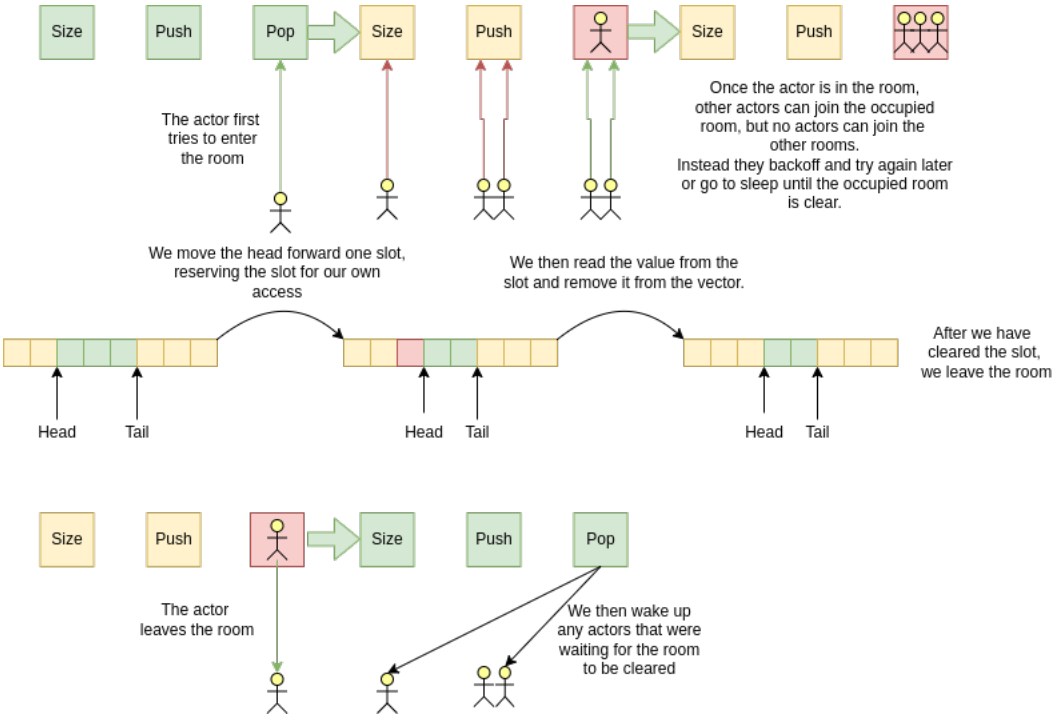


Figure 4.12: The functioning of the Rooms based queue on dequeue operations

Similarly to the enqueue operation, described in Section 4.3.2.3 we also do not require the use of *stamps*, for reasons very similar to the ones described above, notably:

- There could be an ongoing enqueue operation that has not yet placed the value onto the slot. Again, the design of the Rooms allows us to assure that this never happens, when an actor is able to enter the dequeue room all enqueue operations will have been completed.
- The queue could be empty. Again, similarly to the enqueue operation, we know that we cannot perform dequeues and enqueues concurrently and vice versa. So we know that the tail position is going to be fixed when we are performing the dequeue operation. This means we can load the head value, compare it to the tail to see if the queue is empty and then utilise atomic CAS operations to move the head forward. If the head has moved, the CAS operation will fail and we will try again. If it does not fail, then we know the queue was not empty and we can safely remove the value.

Dump operation This queue, like the previously referenced queues, also has a custom dumping method that allows us to get all the elements currently in the list by performing a single CAS operation (setting the head to the tail) and then removing all of the elements from the array, starting from the returned previous head value. We can find a visual representation of this operation in Figure 4.13.

This dump method is based on the dump method seen in Section 4.3.2.1 with the absence of *stamps*, which we have seen why are unnecessary in the previous operations.

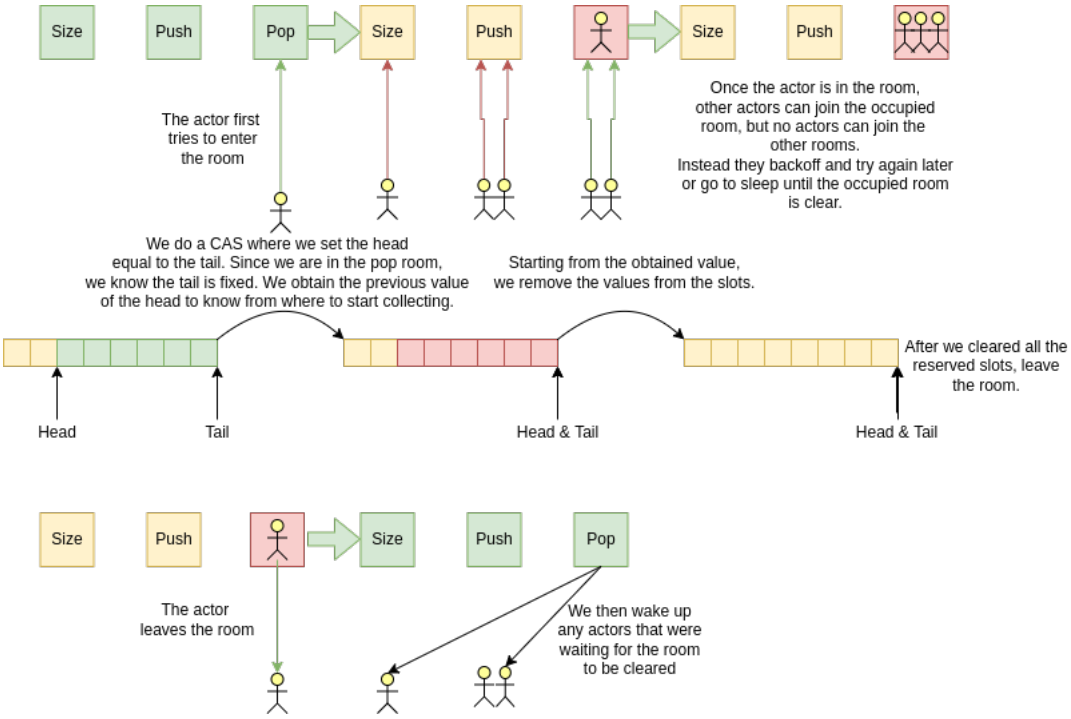


Figure 4.13: The functioning of the Rooms based queue on dump operations

So, we start by loading the current *tail* value and setting the *head* to the *tail*. We know the tail is fixed since we are inside the dequeue room so that will not be an issue. The only possible situation is the *head* being moved. This also does not present any issues with our approach since the operation of setting the $head = tail$ is atomic and returns the previous value of the head (so there is no possible way another actor could have executed a head movement while we are performing our own). If the previous head value is equal to the tail then the queue is empty and no items are returned. If not, we start removing elements at the returned (previous) *head* value until we reach the tail.

We did not go with an approach similar to the one shown in Section 4.3.2.2 for a few reasons, namely:

- In this approach, we can have multiple concurrent actors dequeuing from the same vector. This is because of the atomic operations we perform on the *head*, which assert us that no other actor will be told to access that slot. However, if we switch out the entire vector, we could be destroying currently ongoing dequeue operations that were still not done accessing the vector. This could be quite the opening for consistency and safety issues.
- We could have also gone with a separate room for dump operations where we could maybe have fewer issues with the approach mentioned above. However, we believed that this approach would also not solve the aforementioned issues so we settled on this design, which we had already proven was correct and functional for the previous data structure.

4.3.2.4 Comparative results between the presented data structures

After implementing these structures, we then had to find a way to evaluate them and decide which of the data structures was the most fitting for the necessities that we have. To do this, we developed a benchmark that attempted to mimic the conditions these data structures will be used in. The benchmark we developed consists of the following:

- We have a configurable number of producers that are constantly attempting to push messages into the channel, which play the role of the SMR clients.
- We have a single consumer, which attempts to remove requests from the channel using either the pop operation to remove one at a time or the novel dump operation we introduced in our developed data structures.
- The producers produce a set number of *items* and the consumers must consume all of the items that the producers have produced.

We then measure how long it took to pass all values from the producers to the consumer, the number of iterations done per unit of time and for the benchmarks we used the dump method in we also measured the number of requests returned by the dump method.

Firstly, we will analyse the results for the normal *pop* operation, which removes a single element from the queue.

We ran the test with 10000000 requests, using the above-described bounded queues with a capacity of 10000. Table 4.3.2.4 shows us each of the implementations' behaviour when run with 2, 4, 8 and 16 producers and a single consumer (to match our use case of batching). In the left cell, we see the time it took to run the entire test. In the right cell, we see how many iterations each of the producers did per millisecond on average.

Table 4.3.2.4 shows that LFBQueue is by far the best scaling queue for single operations. This is due to a few things:

- It completely isolates the tail from the head of the queue, meaning we can have concurrent push and pop operations without affecting each other's performance.
- It only needs one CAS operation in order to reserve the target slot for a given worker, minimising the probability of a concurrent worker screwing up the operation midway, forcing us to have to restart the whole operation. This means that there is always at least one worker advancing even when there is a very high amount of concurrency.
- Exponential Backoff is employed for when there is an extremely high amount of concurrency. Because this technique utilises randomness to decide the amount of time it should sleep (with the longest possible time increasing exponentially) it allows operations that are all

executing in phase (all executing concurrently) to go out of phase and therefore achieve better performance.

We then have the LFBRoomQueue, with a pretty bad performance. This is to be expected since we need every single producer to leave the push room so we can enter the pop queue. This means that the more producers we have, the harder time we will have of entering the room meaning we have to wait for the queue to fill up so it stops allowing producers into the room. When the queue is full we are able to freely enter the room and pop a value, however, as soon as we are done with that, it will instantly wake up all of the producers meaning they will all enter the room (even if there is only one space for values) and the problem gets compounded.

We then have the basic mutex-based queues which as we saw in Section 4.3.2.2 have two possible modes of operation. We can either use backoff or not. We also compare these two modes in this benchmark. As we can clearly see, the backoff enabled mode is much faster than the regular mutex mode. This is because as we have seen with the LFBQueue, exponential backoff allows us to avoid having to perform passive waiting where it would then have to wait for the wake call. Instead, we can perform active waiting without the hassle of extremely high CPU usage combined with much higher concurrency leading to more collisions and data races, which means fewer threads are able to advance their state effectively since they are all stuck fighting each other for control. We can see that the Mutex queue with no backoff handles scaling of the workers extremely poorly, even more poorly than the rooms-based queue. This also makes sense since the Mutex, even when the queue is full, is constantly being locked by producers, just so they can see that the queue is already full which then slows the progress of the consumers and means we can process fewer operations.

In Table 4.3.2.4 we have described the results of our benchmarks while using the dump method on the consumer side of things. This means that on each run of the consumer, he will take all of the requests that were contained in the buffer at the time. In the first column for each thread count run we have the time taken to consume all of the produced requests, in the second column we have the amount of iterations done per millisecond second (it/ms), in the third column (the one in yellow) we have the average batch size that was withdrawn with each consumer iteration and in the fourth column, we have the largest batch produced in the entire test. As we have mentioned, our purpose is to determine which of these implementations provides the best ability to maintain an equilibrium of batch size and low latency. We are looking both at the speed at which it can complete the low producer benchmarks (which indicates its ability to maintain low latency when working with a small number of clients producing requests) as well as the combination of time taken and batch size produced for the benchmarks which have a larger amount of producers. In the latter situations, we do not actually need the time taken to be the smallest, since as we have seen, there is a limit on how many batches the system can process in a given time frame. Therefore, our goal is to find a balance, where the data structure in question might not be the absolute fastest but produces fewer batches for the same amount of requests meaning the quorum takes less time to process (since the time taken to process a batch of 10000 requests is majored by the 3 rounds of messaging).

Thread counts	2 prod/1 cons		4 prod/1 cons		8 prod/1 cons		16 prod/1 cons	
LFBQueue	2.03 s	3306 it/ms	2.45s	1654 it/ms	2.54s	923 it/ms	3.14s	386 it/ms
LFBRoomQueue	10.10s	661 it/ms	17.42s	237 it/ms	36.25s	67 it/ms	98.84s	12 it/ms
MutexQueueBackoff	4.19s	1611 it/ms	5.58s	729 it/ms	9.39s	253 it/ms	17.84s	67 it/ms
MutexQueueNoBackoff	8.46s	788 it/ms	22.8s	438 it/ms	79.14s	28 it/ms	172.83s	6 it/ms

Table 4.1: Personalized data structure benchmarks

Threads Count	2 / 1				4 / 1				8 / 1				16 / 1			
LFBQueue	2.39	2788	1	1016	2.46	1749	1	1174	2.55	934	1	1287	3.02	394	1	10000
LFBRoom	5.53	1213	4	632	5.66	710	9	1430	4.68	504	7	10000	6.23	193	178	10000
MutexBackoff	4.19	1611	1	4066	3.87	1046	3	1497	4.53	523	5	10000	6.13	196	10	10000
MutexNoBackoff	8.46	788	3	883	8.19	505	4	1199	18.67	119	5	1468	15.4	76	10	2870

Table 4.2: Personalized data structure benchmarks using dump polling

As we can see the fastest queue is still the LFBQueue, for the reasons that we have described above. However, as we have seen, we are not particularly looking for the fastest queue available and indeed, when we look at the average batch sizes we can clearly see that the rise of proposer threads is not being accompanied by the rise in batch sizes, as we can see that the average batch size is 1 in all the benchmarks performed.

We then have the LFBRoomQueue which as we saw in the previous test was one of the worst performers due to the reasons aforementioned. In this case, we see a massive boost in performance because the usage of the dump method fixes the problem that we described, where the consumer has to wait for all proposers to leave the room and then when he is able to join the room he only takes one element. When he takes that element, the publishers will be awakened and will all join the room only to see that there is only one spot available and they all have to leave the room again. When we use dump, this no longer happens because by clearing the entire queue, we are then allowing the proposers to join and actually have space to push their requests into meaning they didn't join the room for anything. The result is a queue that maintains its performance even with increasing the number of producers and has a comparable performance to the fastest queue (only around two times slower). It does however have a key, very important difference which can be observed in the average batch size numbers. We can see that right at 2 producers, the average batch size is 4x larger than the batch size of the fastest queue. Not only this, the size of the batch shows a trend to increase as it increases from 4 to 9, then goes down a bit to 7 and then we have a huge increase to an average batch size of 178.

In the third row of the table, we then have the simple Mutex-based queue with Exponential Backoff enabled. As we can see, it displays performance very similar to the Rooms-based queue in terms of speed of execution and scalability with increasing thread counts. When looking at the average batch size also shows some promising characteristics. We can see that with little concurrency (with 2 producers) the average batch size is 1 however when we start increasing the concurrency we can see a trend to start increasing the average batch size as we get 3, 5 and 10 respectively. This is to be expected since Mutexes treat producers and consumers equally (they have the same chance of joining the mutex if they request access at the same time). This combined with us raising just the producers means that the consumer is going to have a worse time attempting to acquire the mutex which is exactly what we want when there are a lot of clients producing requests. In the fourth row, we have the simple Mutex-based queue without Exponential backoff. The numbers are significantly better than in the regular pop test, however, they are quite worse than the same mutex-based queue but utilising Exponential Backoff. This combined with average batch size numbers similar to its sister implementation meant that we didn't consider this option.

Having analysed the results of each of the queues and taking into account the needs that this queue had to comply with for their use case we will now choose the best option. The queue we chose was the Rooms based queue for multiple reasons. Namely:

- The LFQueue is very fast to complete all of the operations but generates batches that are

so small that it could take as many batches as overall requests to perform them. This, in combination with our knowledge of how BFT SMR systems work makes this not the optimal option.

- The Mutex-based queues, even though have very similar performance aren't nearly as effective at producing very large batches when there is a lot of concurrency when compared to the Room-based queue.
- The Rooms based queue presents good enough performance in terms of speed while providing the best ability to utilise the concurrency provided by having multiple clients performing requests simultaneously.

4.3.3 Client pooling and request collection and aggregation

One of our largest concerns was being able to handle an extremely large quantity of clients without sacrificing latency or throughput while attempting to fully utilise the underlying compute resources of the replicas (vertical scaling). In order to achieve these objectives a novel peer-handling method was devised and implemented for FeBFT. Its purpose is to split the workload of collecting client requests, aggregating them all while also not going too overboard on the latency of the requests. A sketch of the design of the system can be seen in Figure 4.14.

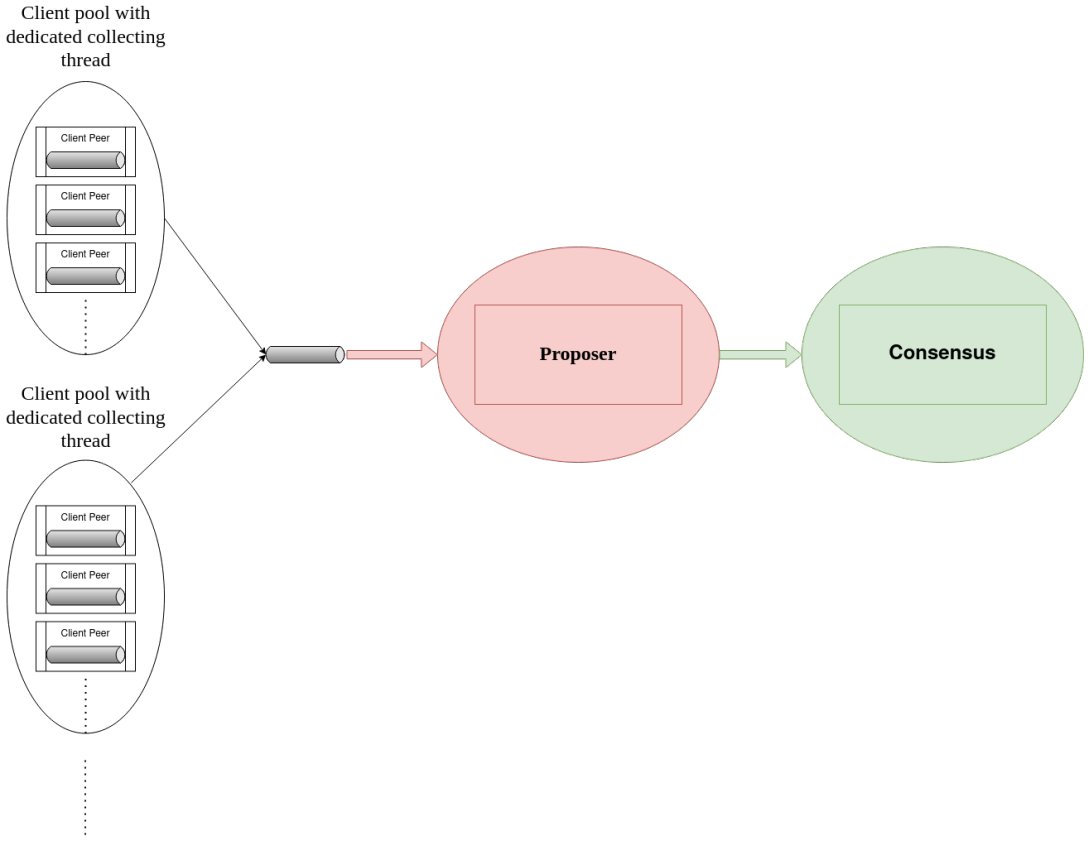


Figure 4.14: The architecture of the client pools.

Originally, as we described in Section 4.3.1, FeBFT's client handling was done by directing all requests into a master channel which the replica would then poll. This meant that this channel had a very high amount of concurrent operations which we know causes data races and overall slowdowns in the performance of the system. Our purpose was to compartmentalise the data as much as possible to allow the CPU to waste little to no time in repeating operations that failed due to high concurrency or having to constantly go to the main system RAM in order to refresh the values that are being concurrently updated.

To achieve this we introduced an individual request buffer for each client peer. This buffer is then populated by the network layer when the clients perform their requests. These requests must then be collected, aggregated and proposed to the SMR algorithm so they can then be decided and executed. However, the collection of these requests can become quite tricky when we are dealing with a very large amount of clients since it has to go through all of the clients while collecting them. Our solution is to split up these clients into separate pools and each pool will have its own dedicated request collection thread, which goes through all clients in its pool, collects the requests, aggregates them and then propagates them to the next layer. We then have configurable values for each of these collection threads that tell them how to behave (target aggregated request vector sizes and sleep times between request aggregations to prevent unnecessary usage of the CPU). However, having multiple collection threads could also lead to concurrency issues if they all tried to accumulate onto the same batch. Our solution to this was to have each of the pools separately aggregate requests so they can work effectively and then introduce another layer, responsible for receiving these partial batches and aggregating them.

This new layer (identified in Figure 4.14 as the Proposer) receives these partial batches and then aggregates them into a single arbitrarily large vector. It will continue doing so until the consensus thread is ready to receive a new batch of requests to propose at which point the proposer will create the Pre-Prepare request and broadcast it to the quorum so it can then be processed (it will also broadcast to itself directly into the consensus thread so the state machine can also be advanced on the leader).

This design allows having a very configurable and tunable model for any possible use case. If we want to handle a few clients that produce many requests we can configure the system to have a few clients per pool and with large target aggregated vector sizes and sleep times (giving the clients time to fill up their buffers before going back in to collect them). On the other hand, if our use case is having many clients that produce sparse requests, we can configure the system to have many clients per a given pool with small target vector sizes and also little sleep time (to reduce the latency of the requests, as if we used large target vector size/sleep times, we would be stuck waiting for the requests to arrive instead of processing the requests that we already have).

This architecture brings another great advantage over the architectures of BFT-SMaRt and PBFT. It is able to process and aggregate requests while the consensus instance is running and therefore have those requests prepared for when the consensus layer is ready to receive them, reducing the latency and therefore increasing the number of decisions that can be made per unit of time while also being able to aggregate and propose much larger batches, leading to

more operations per each of those consensus decisions. In BFT-SMaRt each of the clients also has its own dedicated buffer to store pending requests, but the proposing thread will only start collecting them when the consensus thread is ready to receive them (so it effectively is like the proposer and consensus thread were one, since the proposer will only operate when the consensus is not operating and vice-versa). This difference is already noticeable with our local network testing, as we will show further ahead but it should also be even more noticeable in a scenario with geographically distanced replicas as the client pools and the proposer will have more time to work and propose larger batches of operations, compared to the approach BFT-SMaRt took.

This also allows us to fix another issue compared to BFT-SMaRt's proposer design: limited batch sizes. This issue arises from the design choice mentioned above where the proposer will only collect requests when the consensus instance is not deciding anything, so if we choose an unlimited batch size, the proposer could just continue collecting requests for an undetermined amount of time (if the clients are making enough requests to keep the proposer busy). Even choosing just a large batch number will make the collection take a very long amount of time, all of which is completely wasted since the consensus thread is not doing any work.

Because of FeBFT's novel client pool and proposer design, we are able to have an unlimited batch size. This is because we aggregate requests while the consensus is working and as soon as it's prepared to receive a new batch of requests, the proposer proposes the requests that it has aggregated in the meantime (since the last consensus decision), so little to no time is spent with the consensus layer idling, waiting for a request batch from the proposer. This allows us to handle peak loads very effectively even when the system is not tuned for large request batches. This also means that the system takes advantage of situations where latency is larger to create significantly larger request batches (as the proposer has more time to aggregate requests) which means that it will be able to utilise the network bandwidth to its favour instead of relying on latency which we mentioned to be one of the most important factors in the scalability of state machine replication algorithms in Section 4.3. .

4.3.3.1 Per client request collection

A large part of this efficiency also comes from how we actually collect requests from each of the client buffers. In BFT-SMaRt the proposer collects requests 1 by 1 from each of the clients in a round-robin fashion until it either reaches the targeted batch size or the clients have no more requests to collect. This is not only inefficient as we must do many rounds (and if there are clients with empty buffers we will have to iterate over them time and time again until we have a batch with the targeted size) and at each of these collections, we are interacting with a queue that might also be accessed by the networking thread as clients send their requests which leads to race conditions and slowdowns for both the network layer and the proposer (and since the consensus layer is waiting for the proposer to finish collecting requests, it is directly impacted by this). Our architecture already naturally mitigates all of these issues by its tiered approach, meaning that even if the collecting thread of a client pool would get stuck collecting

from a certain client because the client's buffer is also being accessed by the networking layer, our proposer would be able to proceed as normal.

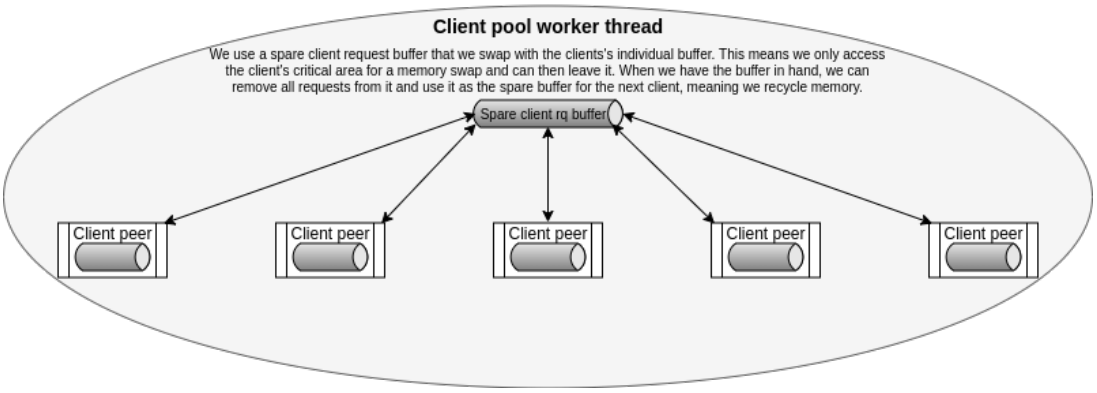


Figure 4.15: The functioning of the client pool worker.

However we also did not want to put any unnecessary delays or slowdowns on our networking layer, so we devised a method to be able to collect the requests from the client buffers while also minimising the amount of time we spend in critical areas. We know that each client has his own request buffer, which contains the requests sent by the client which were not yet collected. Our solution is to switch this buffer with a blank one, meaning we are only accessing the critical area for a memory swap operation and we are then able to leave it and process the requests in the client pool thread without causing any slowdowns to the networking layer. Another advantage of this is that we can then use this now empty client request buffer as the new buffer for the next client, therefore avoiding having to allocate/deallocate any memory during this whole operation, meaning we can spend the majority of time actually collecting requests instead of waiting for operating system calls for memory allocations or having to slow down the networking layer.

4.3.3.2 Proposer functioning

We have seen how the requests get from the network all the way through the client pool collectors until they are delivered to the proposer. We will now see how we engineered the proposer to maintain the least amount of latency while also maintaining great batching abilities.

As we are able to see from Figure 4.16, the proposer is composed of an infinite loop. The first thing we do is check the shared channel with the client pools for any new partial batches passed along. The way we handle this is different depending on whether the replica is currently the leader or not.

- If we are not the leader, we will block on the queue until a new partial batch is delivered. When it is delivered it is sent to the synchronizer which keeps track of performing timeouts and subsequent view changes. So if we are not the leader, the proposer does not need to create batches (as we do not perform proposes) and therefore only performs the job of keeping track of requests.

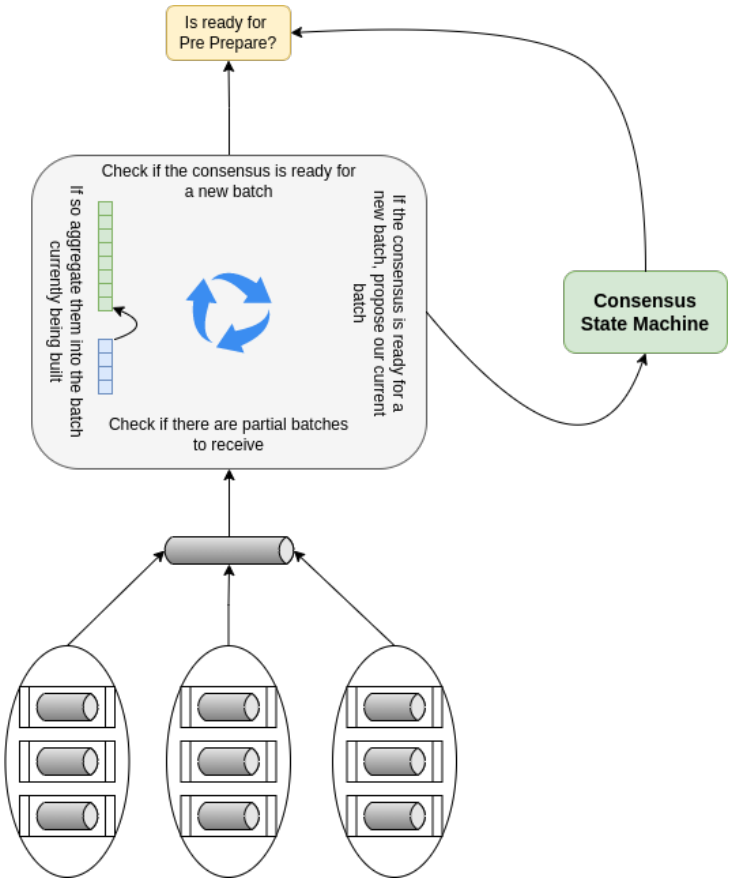


Figure 4.16: The design and functioning of the Proposer.

- If we are the leader, we will not block on the queue. Instead, we check if there are any elements available. If so, we take them and move to the next step. If the channel is empty, we will skip straight to the third step which is checking if we are able to propose a new consensus instance.

To know if the consensus is ready for a new request batch we utilise a simple Atomic Boolean shared across both the consensus thread and the proposer thread in combination with a Mutex which stores information about the current view and operation sequence number, necessary for creating the batch for the consensus state machine. This requires a single CAS operation, meaning if it is not ready we do not even have to access the Mutex.

This design is capable of completely occupying a thread since it has no positions where it sleeps or waits for any other input, which is less than optimal. To address this, we utilised yield instructions to reduce CPU usage. However, since this part of the system is so important for operation latency, we wanted to make sure we kept the amount of time sleeping very short so the CPU usage is still high.

4.3.3.3 Separation of the client and replica request flow

Along with the change to the client handling method, we also altered the way requests sent by replicas get handled. We had to alter the way replica requests are routed and handled because the latency introduced by the client pooling workflow was affecting the already latency sensitive consensus protocol.

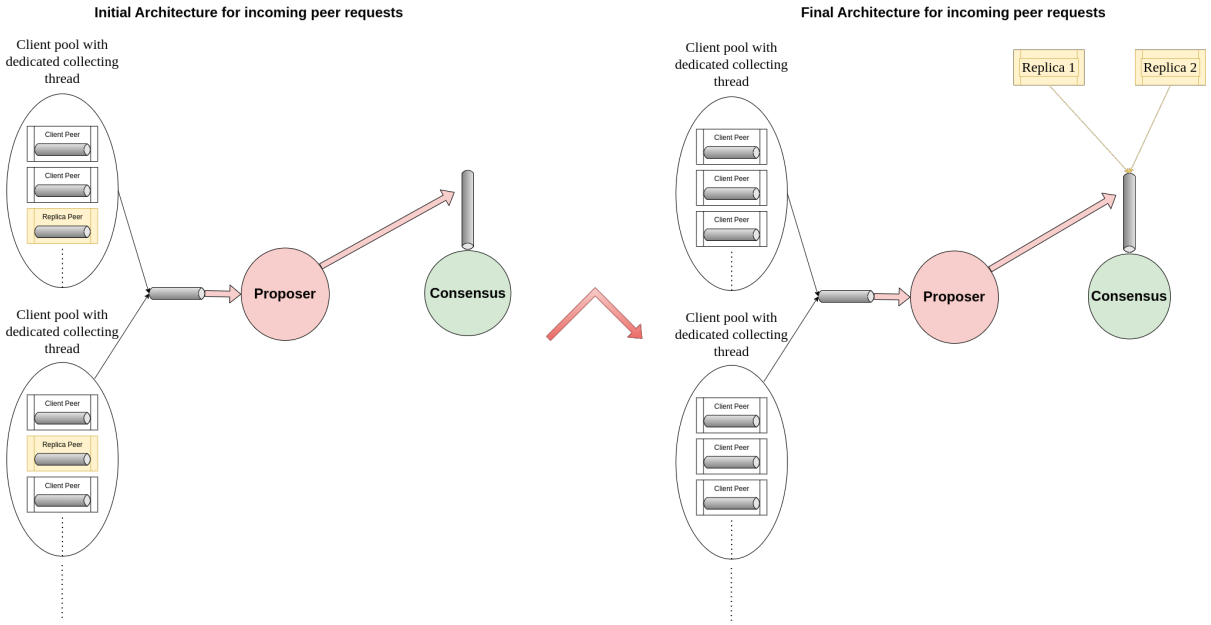


Figure 4.17: The incoming replica and client request flow.

To remedy this situation, we made the network connections between two replicas insert the requests directly into the consensus layer's incoming queue, skipping the client pooling and proposer processing as shown in Figure 4.17. This also means that replicas do not need to have a request buffer, instead, they use the consensus incoming queue as a shared buffer for all the replicas. In principle, this shouldn't cause many issues with concurrency and even space in the queue, since the amount of replicas is in principle small (much smaller than the number of clients) and the consensus layer does not do much processing on the requests so it should be able to keep up with the incoming requests from the replicas.

4.4 Alterations to the runtime

Initially, the entire FeBFT project ran on an asynchronous runtime (which could either be the standard libraries asynchronous runtime, *async_std* or *tokio*, an open source community-developed runtime). The asynchronous runtime is an event-driven, non-blocking IO platform and is responsible for scheduling Tasks and handling IO events delivered from the operating system. It is capable of scheduling N tasks on M processors (multiplexing) which inherently provides vertical scalability to our architecture. It is also very easy to use and very flexible.

At first glance, this seems like a good option for our purposes as we are developing software that is IO-bound and event-driven (Our replicas only respond to requests, they never create or perform their own requests). However, after a more in-depth analysis we quickly found several glaring problems that severely hampered FeBFT's performance, namely:

- Asynchronous runtimes are meant for IO-bound applications where each Task spends most of its time waiting for IO, not actually performing any sort of work, meaning it is not very indicated for most of our use cases.
- Tasks are meant to be lightweight, non-blocking units of execution, similar to threads but managed by the asynchronous runtime instead of the OS scheduler. This does not fit the requirements of many of our components, which require blocking because many times it's worth it to block on a given operation (like waiting for the output of another given thread) rather than having to context switch the task out of execution and then back into execution again (In the meantime the task gets put at the back of the execution queue and spends a lot of time without making any progress).
- Tasks are scheduled *cooperatively* instead of the more common *preemptive scheduling*, so they must indicate to the asynchronous runtime when they are ready to be stopped. This is done whenever an *await* was called, which effectively told the asynchronous runtime the task is ready to *yield*.

All of these factors effectively mean that tasks are constantly getting context switched in and out of execution (many times also switching the processor that executes it in the meantime) which means that important tasks might be prevented from making progress while they are waiting for a processor to be executed in. It also causes a lot of cache misses as the tasks are constantly getting moved around, preventing the CPU from developing a good cache locality and therefore forcing our program to read from RAM much more often which significantly slows down the operation.

As a result of these factors, we decided to stop using the asynchronous runtime across the entire project and instead leave it just for what it is designed to handle, IO-bound, non-blocking tasks, in particular, network tasks. To replace this runtime, we started using dedicated OS threads instead of tasks, which allow us to block when we believe it to be worth it, preventing the runtime from having to context switch the execution unit and lose a lot of computation time while doing so. It also allows us to utilise the processor's cache much more effectively as the threads are not getting moved around nearly as much, allowing the cache locality to tend to an optimal state, instead of having to be reset every time a blocking operation must be performed.

4.5 Network connection multiplexing

As we have mentioned in Section 4.1, FeBFT relied on Asynchronous runtimes for practically everything from network multiplexing to handling the actual state machines required for the

service. As we have also seen in Section 4.4 this was not the particular best approach to handling multi-core support and actually utilising the underlying CPU power.

These issues with the asynchronous runtime got to the point where it made us question whether it was the best option for connection multiplexing (which as we saw was one of its main usage cases). This made us think of possible alternatives and implement and test them in order to discover which is the most performant way of handling the connection multiplexing.

We will document all of the choices along with their problems and our thoughts as to why they were occurring in the upcoming sections.

4.5.1 Asynchronous runtime

We will first start by describing the original network layer design of the original FeBFT project, which relied on the asynchronous runtime for connection multiplexing and utilised asynchronous I/O with the underlying TCP sockets and the runtime for task scheduling.

An image illustrating the architecture can be found in Figure 4.18.

We allocate an asynchronous task for every peer incoming connection which receives, deserializes, verifies and then delivers the message to the specific peer's buffer (in the case of clients they each have their individual queue and in the case of replicas they deliver messages straight to the Consensus layer ingest buffer) as detailed in Section 4.3.3.

The sending of messages was handled quite differently. The initial handshake and connection establishing was done using a single asynchronous task but from then on the socket is stored in a map, associated with the corresponding peer id and wrapped in a Mutex.

We then had a dedicated thread pool for the CPU-intensive tasks required to prepare the messages for sending over the network such as signing, hashing and serializing the messages.

In Figure 4.18 we can imagine the thread pool as the message producer since the original message producer gives the job to the thread pool and the thread pool will then process the requests and create the task with the request already ready to be sent.

When we wish to send a message to a given peer, we first run the message through the thread pool and once the message is ready to be sent over the network, we spawn an asynchronous task which will be responsible for retrieving the corresponding socket from the map, acquiring the mutex and then sending the message.

We encountered quite a few performance issues with this approach, namely with request throughput as the number that we were observing were nowhere near the numbers of similar BFT SMR systems like BFT-SMaRt.

We believe these problems were mainly related to how the sending of messages was being done. With the method described here, the number of tasks that exist at any point is directly

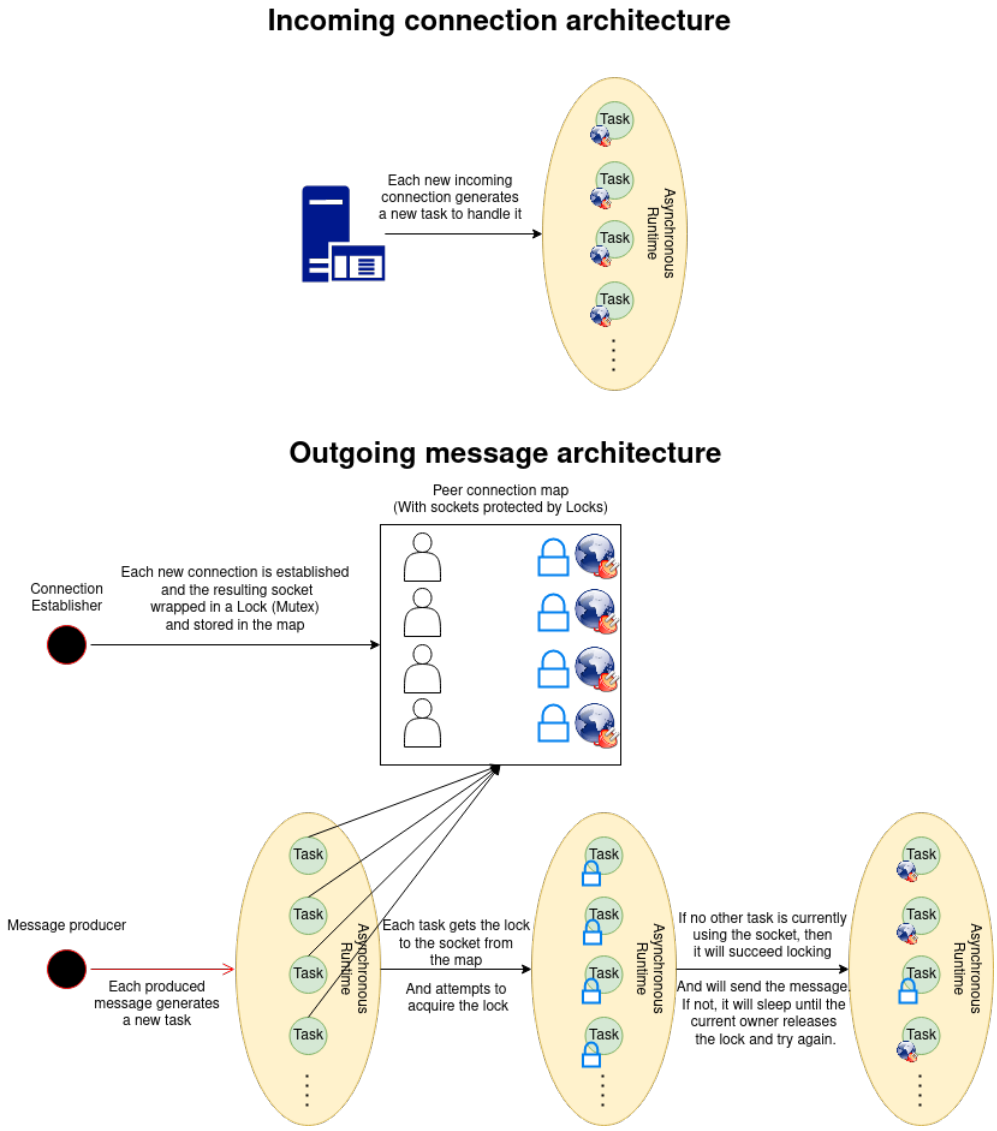


Figure 4.18: The incoming replica and client request flow.

dependent on the number of requests being sent (So even if we only have a few clients, if they are sending many concurrent requests, can create a lot of send tasks). All these tasks will be trying to acquire access to the Mutex that protects the sending socket, meaning this might become a very large point of contention, which can lead to a lot of tasks being awakened every time the mutex is released (so they can get their opportunity of acquiring it). This causes a lot of context switching, which in turn leads to poor CPU utilisation because the CPU is busier performing context switches than performing actual useful work.

Another possible cause is that we believe the asynchronous IO to be slower, both in terms of raw throughput and in terms of raw latency, concerns which were then supported by some performance tests which compared it to synchronous IO. These concerns in particular led to the development of the next solution.

4.5.2 Threadpool based model

Our second option was to stop utilising the asynchronous IO APIs which would also allow us to stop depending on the asynchronous runtime. Instead, we utilised regular standard library synchronous sockets and regular OS threads.

Figure 4.19 attempts to illustrate this design.

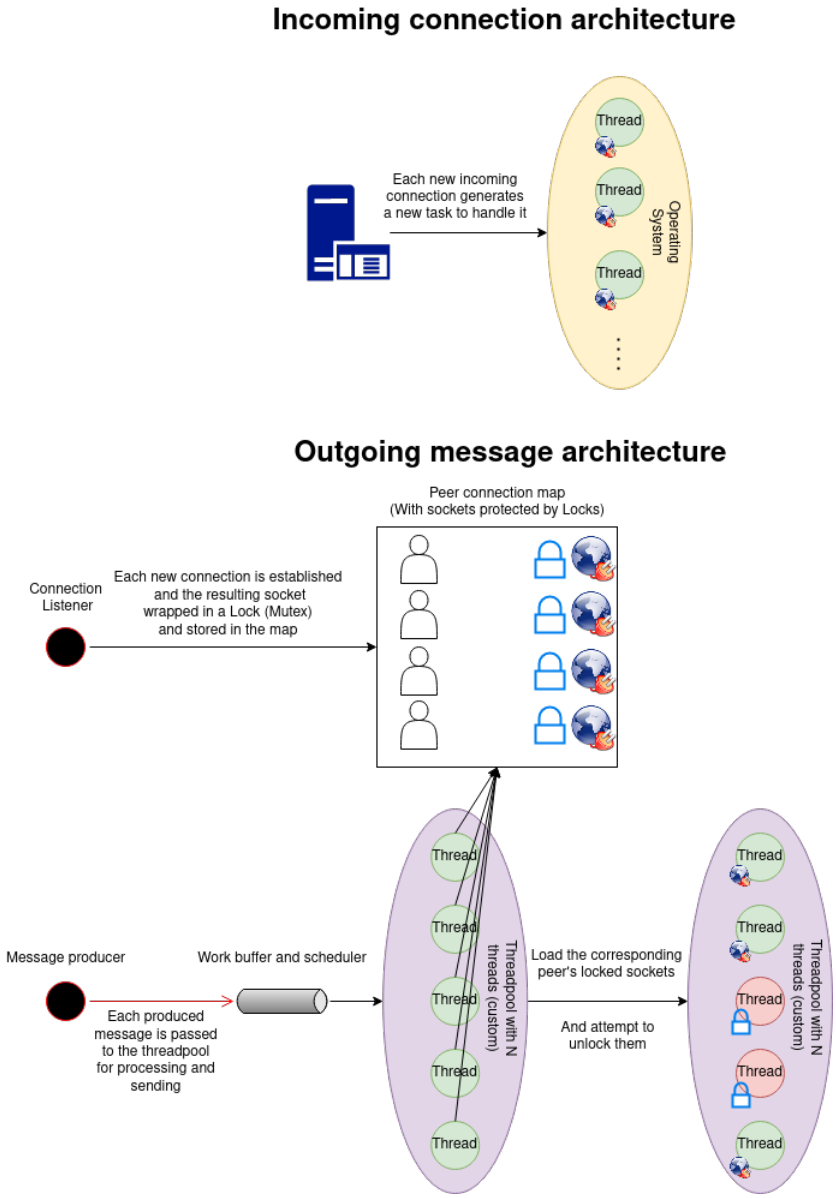


Figure 4.19: Design of the thread pool based model for network multiplexing

For the reception of messages, we kept the one task per peer connection design model and changed the execution units from asynchronous tasks to OS threads.

We did however change the way messages were being sent. The model of not having any dedicated threads for sending to a particular peer was maintained and we kept the map of peers

and their respective sockets wrapped by a Mutex.

We moved from having the thread pools only perform the CPU-intensive tasks of preparing the messages for network dispatching to having the thread pool handle all of the work since the instruction of sending the message is given to the actual message being sent over the network. We believed this would solve a few of our issues, like:

- The number of tasks scaling with the number of messages that were being sent, which can be arbitrarily large. We believed this problem would be solved because of the natural way thread pools handle their work. Even if we present M jobs to the thread pool, it will only utilise the N workers that we initially set out for it to have so this would naturally fix this issue.
- Also thanks to this previous point, we believed this alternative would fix most of our CPU under-utilisation thanks to context switching again because the number of threads that were concurrently executing was limited to N.
- This also led us to believe that our issue with concurrency at the socket level, where tasks would have to wait in order to be able to access the sending socket due to the high amount of tasks all trying to use it and again a lot of CPU time is wasted in these acquisition attempts.

Initially, the inherent problems with this approach were not obvious to us, but they soon became such. With a more in-depth analysis we can see that there are a few very large issues with this approach that lead to severely hampered performance numbers, namely:

- Firstly, the use of a thread pool combined with the use of blocking IO lead to plenty of wasted CPU time, since a thread belonging to a thread pool will only initiate a new job when it completes the current one. This behaviour was fine when the pools were only being used for CPU-intensive work as it meant the threads were always busy and actually performing useful work. However, when the networking IO workload was introduced this was no longer the case. Instead, the threads in the thread pool spent most of their time idling, waiting for the OS response to system calls instead of actually performing useful work, which severely limited our potential performance.
- Secondly, this option didn't actually solve the issue of creating a large concurrency problem at the peer socket level. This is because, if the conditions are right, all the jobs currently being executed by the thread pool can be destined to the same user (there was nothing that prevented it, in fact, with the way we ordered the sending of replies to the user to allow us to only flush the socket once even if we had sent 100 messages to the same user made this very likely to happen). In these conditions, we would have a complete disaster. We know that only one worker would be able to access the sending socket at any given time, so the other $N-1$ workers would all be idling, waiting for access to the socket. If this

was not bad enough, we still maintain the issue where the worker blocking on the socket causes that worker to idle, until the OS has responded to the system call. So in reality, it was possible to have all N workers occupied while only being able to send 1 message at the same time. (This issue can is represented in Figure 4.19 by the red threads, which are blocked waiting for access to the peer's socket, leading to wasted CPU time).

These factors led us to the conclusion that this was not the ideal solution. This leads us to the next solution.

4.5.3 Per peer sending thread

This design was the result of a complete remodel of how we handle message sending. As we saw from the previous designs, creating an execution unit for every single message sending has several drawbacks from generating a lot of context switching due to the unreasonable amount of tasks to creating a great deal of concurrency at the peer socket level, leading to lost performance while waiting for access to the Mutex. Our idea to resolve this was to make the amount of execution units scale with the number of connected peers, by giving each peer his own dedicated sending task (or thread, depending on the IO API the user decides to utilise). In this model, each of these tasks consumes messages from a queue and whenever we wish to send a message to a peer, we place the message on that peer's queue and then let his dedicated sending thread to handle it for us. With this, we no longer have to wrap the sending sockets in mutexes, since only the given peer's sending task will have access to it, allowing us to reduce contention and therefore increase the ability of each sending task of performing work continuously.

This architecture can handle us utilising both regular green threads (OS threads) as well as running this on the asynchronous environment. We implemented both these options as we wanted to see if the problem was only with the architecture of the networking or if there was also some underlying performance bottlenecks being caused by the asynchronous runtime which is why we will refer to each of these threads or tasks that handle request reception/dispatching with a given peer Execution Units (EU) instead of their actual implementation (since both are implemented). We will analyse the performance differences further ahead in Section 5.4.

We then evaluated a few approaches on how to handle the signing and serialization of the messages which we will also discuss here in greater detail.

An overview of the architecture can be found in Figure 4.20.

Peer sending threads tasked with the CPU-intensive work Initially, we wanted to compartmentalise all the processing for messages into each of the peers' sending threads. So whenever we wanted to send a message, we placed the unserialized message into the peers' queue and then allowed his dedicated execution unit to process and send the message. This immediately raised some concerns over its performance, as we had many cases where we were broadcasting

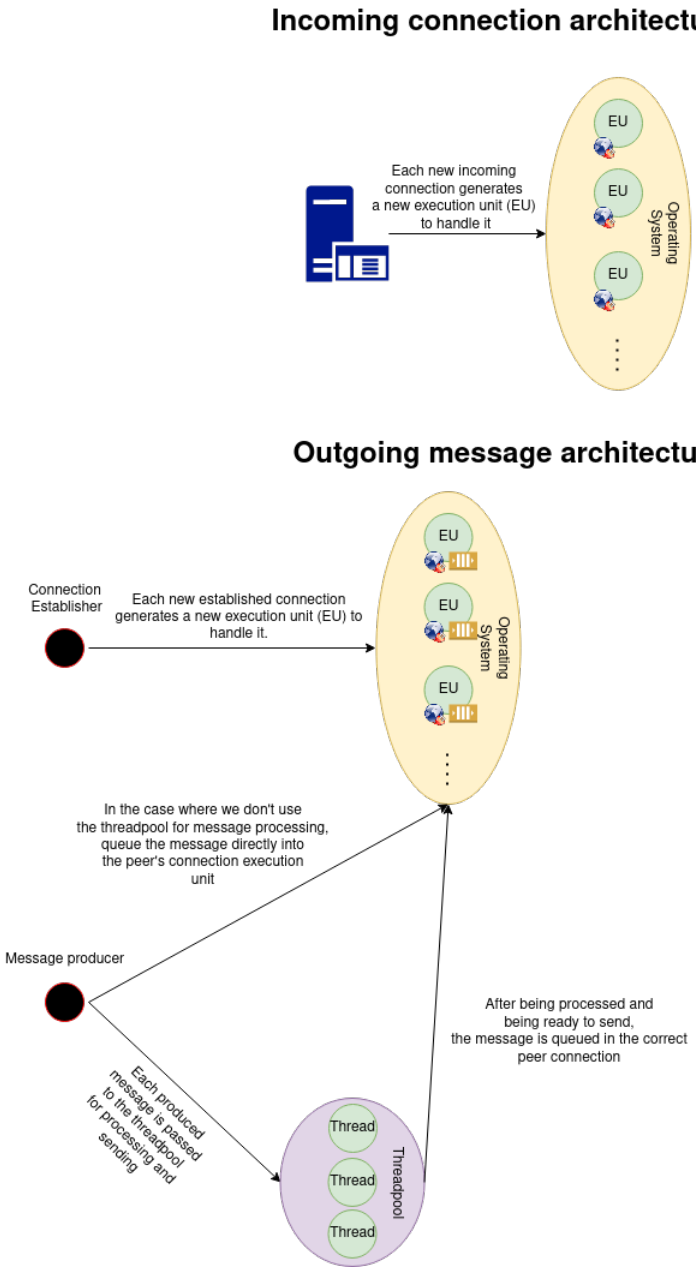


Figure 4.20: Design of the per-peer connection thread architecture

the same message to a group of N peers and with this system we would have to serialize and sign the message N times instead of just signing and serializing once and then only sending the serialized data. In this situation Rust's ownership model did not assist the situation at all because it did not allow us to share the same copy of the message across all of the peers we wanted to send it to (which makes sense since the compiler would not know when it could release the memory safely since it had to wait for all peers to have received the message), instead, we had to clone the message N times which could lead to severe performance hits as well if the request was large (very possible given our design choice to opt for a few large requests instead of many smaller requests). So not only were we potentially multiplying the work we had to do to

sign and serialize the messages, but we were also adding a new workload of cloning the messages so they can then be distributed across the peer-sending threads.

Threadpool for CPU intensive work, dedicated thread per peer for sending Our second option was to resort to the original design, where we used a dedicated thread pool for these CPU-intensive workloads. This fixed our work multiplication problem since we are able to first utilise the thread pool to sign and serialize and only afterwards, once the message has been processed, pass the raw message into the destination peer-sending threads so they can be swiftly sent.

We believe this to be a good setup for our current design since we keep the workers of the thread pool busy performing actual work while delegating the IO waiting to threads that can be easily context switched (and that only block the sending to its designated peer, allowing us to still concurrently send as many messages as peers we have, which was the limit on the previous designs as well since the sending sockets were always protected by Mutexes).

With this architecture we found there to be some factors that helped us boost performance significantly, namely:

- By making the amount of execution units scale with the number of connected peers instead of with the amount of messages being sent, we were able to greatly reduce the strain on the schedulers and subsequently also greatly reduce the amount of context switching that must be done.
- By introducing buffers to hold messages that have not yet been sent, we are able to better handle peers that have high latency or that are slow to respond as we do not get stuck with lots of sleeping tasks that are waiting for the current worker to finish sending messages. Instead, when the worker is done sending the current message, he can just get the next one from the buffer again reducing context switching and contention in Mutex acquisitions and increasing the locality of information, which is better for cache performance.

This design however is not the optimal solution for this problem. We can find some issues with it:

- Having a thread for each connected peer still leads to scalability issues, since we have no bound on the number of connected peers. This is not good for the CPU scheduler, which will still have to perform a lot of context switches, even though it's to a lesser extent than with the previous approaches.
- We can't control the amount of processing power we want to give to networking, since we can't specify how many threads we want the networking layer to utilise (since they scale with the amount of clients). This, again, might lead to us overloading the CPU with too many threads, causing the important threads like proposers, consensus and executors to not progress as quickly, therefore degrading the overall performance.

The optimal solution is something that we will discuss in the following section.

4.5.4 The holy grail

Optimally, we want a design that is capable of, with a reduced number of threads (certainly not a number that scales with the number of messages or clients), multiplexing all of the connections effectively.

A framework of this kind could have a design of the following genre:

We designate a thread as a connection acceptor, which handles new peer connections (This could also utilise a thread pool model or similar to allow us to establish many connections at the same time). We would then have another pool of threads responsible for actually handling the sending/receiving of information, which would work as follows:

- Each thread would be assigned a group of network connections that must be handled by that particular event thread.
- Each of the threads will host an endless event loop, which uses the *epoll* API to know which of the connections handled by it is ready to be read from/written to. After obtaining the list of connections that are ready, it performs the necessary operations and goes back to blocking on *epoll*.

An architecture of this type would fix all of our issues in the following ways:

- It would fix our thread scaling problem, as the amount of threads could be defined at the beginning and only the defined amount of threads would ever be utilised.
- Given that it fixes the thread scaling problem, it would also fix the excessive context switching problem, since we could define a relatively small number of threads (in line with the number of processing cores available for networking), preventing the CPU from being flooded. Fixing the context-switching problem means that the CPU is able to actually spend time performing useful work, instead of being busy moving threads in and out of the cache.
- Taking into account that we are able to manually define the number of threads that are dedicated to processing connections we are able to plan out how the CPU cores will be divided which would allow us to not overdo it and flood the CPU (which would then lead to the excessive context switching problem).

APIs that perform this sort of tasks are readily available in most languages (Java has Netty, C++ has ASIO, etc) however there exists no such platform on Rust, so we were left scrambling for alternatives, leading to the previously mentioned designs.

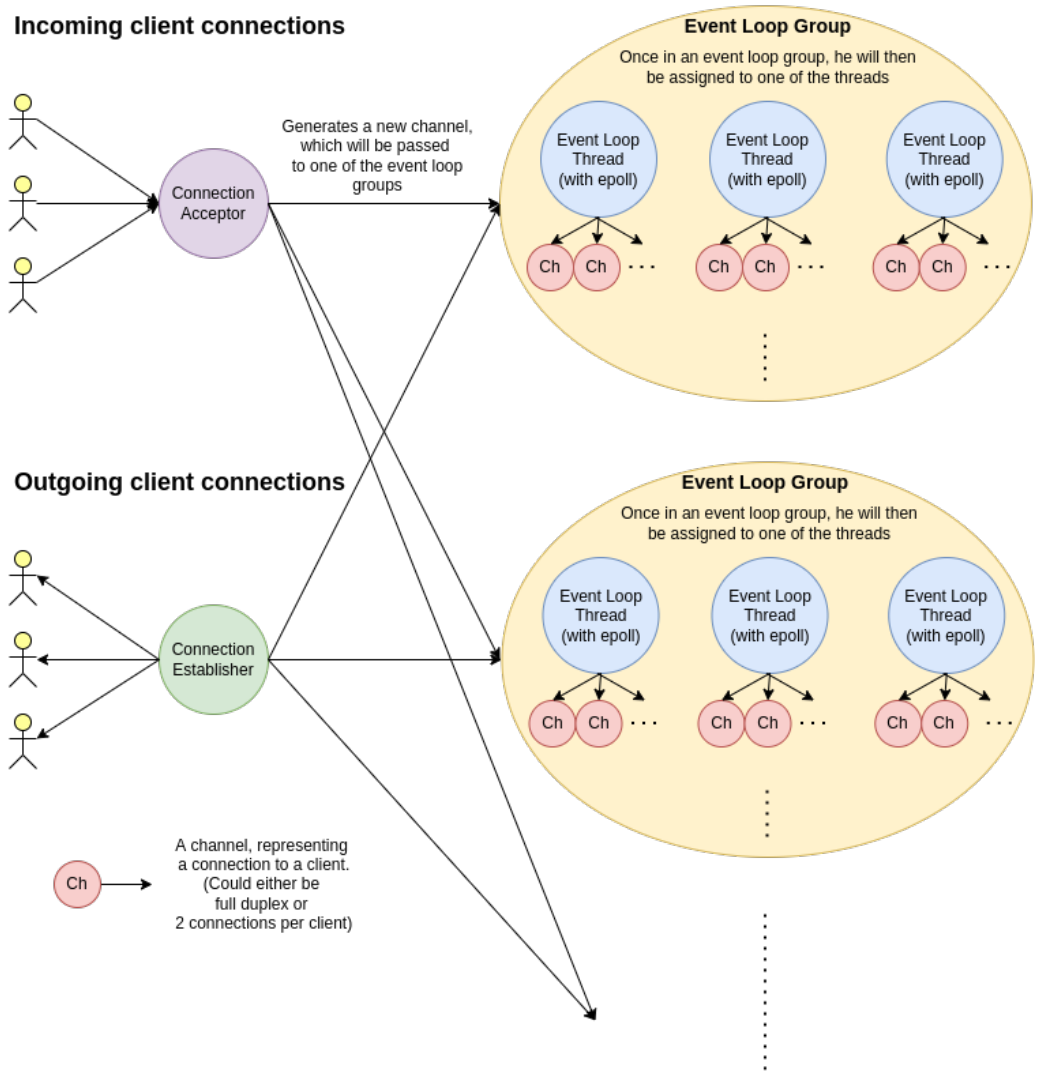


Figure 4.21: Rough design of the ideal network multiplexing architecture

4.6 Persistent storage

Initially, FeBFT was not made to have a persistent storage layer. In theory, this is not necessary for a byzantine fault-tolerant state machine replication algorithm because of a few factors:

- Naturally, a BFT SMR system provides tolerance of f byzantine faults, meaning that, at any point in time, the system will only provide guarantees if $<= f$ replicas fail at the same time.
- Given this knowledge, we know that the system does not, theoretically, have to provide any guarantees for when the system has more than f failures concurrently. Therefore, we can assume that, at any given point in time, there will always be at least $2f + 1$ replicas available with the up-to-date and correct system state so we do not need to persistently store all of the information, which could slow down the performance of the system.

- Even if we have more than f faults, as long as we have a group of replicas $> f + 1$, we can easily rebuild the current state on the faulty replicas by utilising our state transfer protocol.

However, it still remains important to persist the state of the system for a number of practical, real-world applications which require many replicas in a system (if not all) to be powered down simultaneously in order to install an update, for example. There could also be scenarios where all replicas shut down for any given reason (if they are installed in the same data center and there's a power delivery issue, for example). In general, there are a lot of possible scenarios where we are required to power down the entire cluster meaning the theoretical guarantees do not always apply.

In order to solve this, we implemented persistent storage for FeBFT. To do so efficiently and by losing the minimum possible performance, we devised an asynchronous way of storing requests while still maintaining certain guarantees on the persistence of the operations.

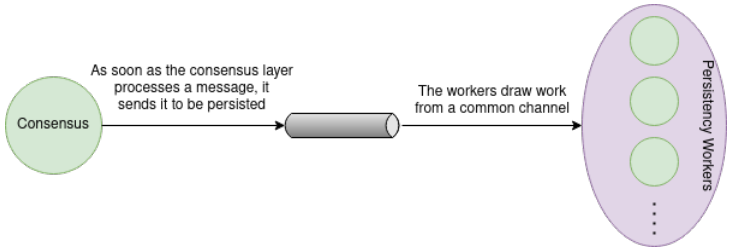


Figure 4.22: Base design of the persistent storage workflow.

As can be seen in Figure 4.22, this solution is composed of a configurable number of workers and a multi producer multi-consumer channel, where information to be persisted is placed by the producers and is afterwards consumed by the workers and placed onto the respective storage. Optionally, alongside the information meant to be persisted, the producer can also include a callback (a function reference) that will be called as soon as the information has been successfully persisted onto storage. This feature is used to build the separate modes of operation of this persistent storage.

Underneath the hood of this system, we use RocksDB[4]: an embeddable log-structured persistent key-value store.

We chose RocksDB as the data store for this project for many reasons, such as:

- RocksDB is capable of providing ACID operations on arbitrarily-sized byte streams, meaning it can fit any kind of data types comfortably.
- It supports fast lookups by utilising indexing along with range scans and other very useful data retrieval methods.
- It's very performant on fast storage flash based and can be tuned to support high random-read workloads, high update workloads or a mixed case (in our case we want it to be tuned

for high update workloads).

- It can handle concurrent write and read operations without requiring any sort of locking which is very useful for our multiple-worker design.
- Amongst many other reasons that made it the obvious choice, to the best of our knowledge.

Given this base, we then built two operation modes on top of it: Strict mode and Optimistic mode which intend to provide a different level of theoretical and practical guarantees about the functioning of the system.

4.6.1 Strict persistency mode

In this mode, for a request to be executed by a replica, it must first be persisted in storage.

With this, we intend to give a few guarantees on the functioning of the system:

- When a given operation is passed to the executor, it has already been persisted and therefore completely recoverable even if all of the replicas fail simultaneously. This means that whenever a client receives a reply for a given request, there is almost no chance (unless all replicas suffer faults that destroy their persistent storage) that request is going to be backtracked or forgotten.
- Even if the persistent storage finishes performing store operations in different orders than the decided order of the consensus (in the case where we have more than one persistent storage worker), the executor must receive the operations in the order decided by the leader and represented by the sequence numbers.

As we have mentioned, we wanted to provide these guarantees while minimising the performance impact on the system, so we had to pick an architecture that allows the consensus layer to continue deciding batches of operations even if we are currently working on persisting requests. To do so, we needed a way to keep a queue of requests that have been processed by the consensus layer but have not yet been persisted. This queue must maintain the original ordering of the requests so that even if a request n gets persisted before another request $n - x$, it must wait until all requests $n - x, n - x + 1, ..n - 1$ are persisted and executed before n itself can be executed.

Our solution was to add another dedicated thread named *Consensus Backlog* that receives the requests that are processed by the consensus layer along with the messages that must be persisted so the request is considered ready for processing (not all messages have to be persisted, just the Pre-Prepare request, $2f + 1$ Prepare requests and $2f + 1$ Commit requests (enough to form the certificate for that particular consensus sequence number)).

We believe adding this new thread makes sense because firstly, most of this thread's time will be spent sleeping waiting for input from the persistency layer or the consensus layer so there is

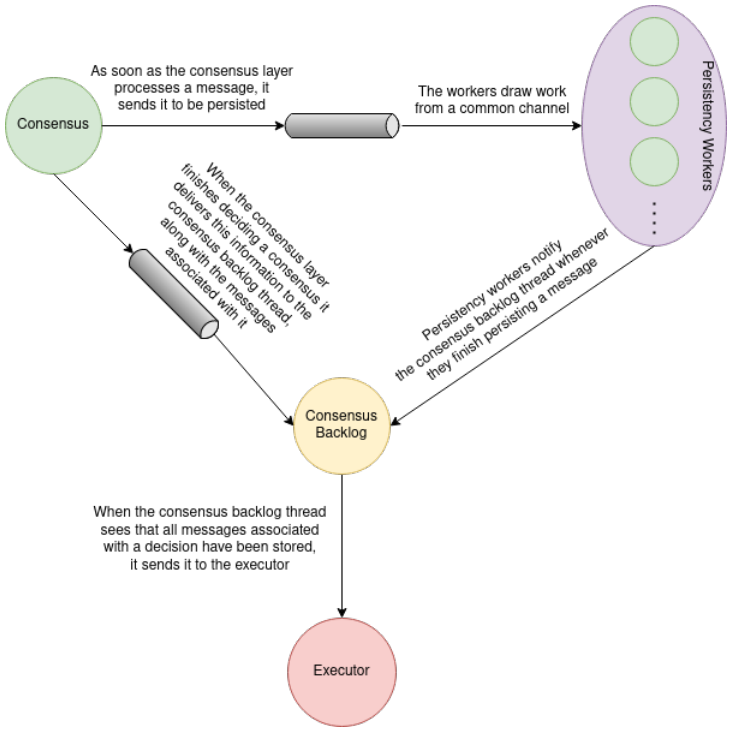


Figure 4.23: Strict mode design of the persistent storage workflow.

not a lot of performant computing performed by this thread. Secondly, we can move the waiting for the persistency operations to this new thread, therefore allowing the consensus to continue to process requests. We only have to stall the consensus thread when the buffers are full.

As is described in Figure 4.23, the consensus backlog receives from the persistent workers' notifications warning it that a given message has been persisted into storage. If the messages are related to the current waiting consensus decision then it will see which messages are still missing and if there are no messages left to persist, it will send them to the executor. If the message does not pertain to the current batch, it will be placed on a map so that when the corresponding consensus instance comes along it can be dealt with accordingly.

In reality, this means that the peak sustained performance, over a long enough span of time (until the consensus backlog buffer is full), is in fact limited by the speed of the underlying storage medium because at that point it will only accept new requests when the existing requests have been persisted. However, when we want to provide the guarantees that this mode of operation provides this was always going to be a large limitation.

What we are able to do is support the number of operations that the underlying datastore is able to handle and add use the backlog's buffer to handle bursts. These bursts will get stored in the buffer and we can then process them when the service has a point in time where there are fewer operations than what the datastore can handle.

We believe this to be a very good setup since in the real, practical world, it's very rare for a system to be at 100% load all the time since then it would not be able to handle any type of

increase due to peak / low usage hours or new users. Systems that have to be at 100% all the time probably need to be redesigned or scaled in order to handle the required traffic. Combining it with a buffer of an appropriate size (which should be chosen based on factors like how much RAM we have available, how many peaks/drops there are in a given time frame and how big they are) can lead to a system that even without being able to actually handle the number of operations needed at the peaks can spread out the load into times where it is not as busy. The underlying data store is of course probably going to be the limiting factor in most occasions but with recent advances in flash-based memory technology, we have seen a great increase in speed and capacity accompanied by a great drop in price, which our middleware system can take advantage of.

4.6.2 Optimistic persistency mode

With Optimistic mode, we intend to relax the restrictions quite a bit in order to allow the performance to not be limited by the underlying datastore's speed. We do not intend to provide any additional guarantees (besides the default BFT SMR-provided ones) on the functioning or consistency of the system. Essentially, this mode serves more as a utility to the system than an addition to the default properties.

In this mode, we do not make any requirements on whether the request must have been persisted before it is passed along to the executor and subsequently answered. This means that a request can be executed before it has been saved in persistent storage, which means that if the conditions are right and all replicas crash before the latest requests have been persisted (if just one replica was able to persist it before crashing, the request will be available, since the messages we save compose the necessary *quorum certificates* for it to be admitted by all correct replicas).

So effectively this mode intends to provide a way for replicas to not have to obtain the entire state whenever they crash or need to be restarted, which saves a lot of work from having to be done by the other replicas in the quorum (sending the checkpoint and the message log). It also allows us to, with some extra steps like disabling incoming requests and then waiting for all the pending requests to be persisted, restart all the replicas in the quorum which is useful for updates, maintenance, etc.

4.7 Rust's memory ownership optimisations

When working on this asynchronous persistent storage feature, we encountered a few notable issues caused by some particular Rust characteristics which could potentially hamper performance significantly if the right conditions were met.

In general programming languages follow two options for managing the memory of the program. These are:

- ***Garbage Collection*** - With garbage collection, the language itself provides a runtime which analyses what memory is currently being used and subsequently what memory is not being utilised. It can then use this information to free the non-utilised memory so it can be reclaimed and reused. This option requires us to have an underlying runtime which is responsible for calculating the reference trees (the tree that maps all of the alive object references, starting from the beginning of the program) and then actually cleaning out the memory. Having to do such a thing requires a computing amount proportional to the number of objects currently in memory (so programs with high memory footprints will generally take more time to be garbage collected) and actually stopping the execution of the program, so we can accurately calculate the tree. This can cause complete temporary stalling of the program, leading to performance degradation. It does however make life much easier for the programmer and accelerates the development of the program. These approaches also allow us to protect against many memory attacks (buffer overflows, stack overflows, etc), making developing safe programs a lot more easily along with offering some protection against memory leaks (it is not complete protection though, the programmer can still cause memory leaks if he makes certain mistakes). Some examples of languages that utilise this approach are Java, Go, Python, etc.
- ***Manual Memory Management*** - In this option, the task of managing the memory utilised by the program falls on the programmer himself. There is no runtime or garbage collection of any sort and no protections against most memory attacks or memory leaks. This means that this mode adds no overhead and requires no runtime, meaning performance offered by this mode is not hampered in any way (however this mode does not guarantee performance, instead it is up to the programmer to perform good memory management and use safe practices to defend against attacks). This mode is extremely error-prone because humans are not perfect and even experienced programmers might make mistakes which can lead to security and performance problems. Some examples of languages that utilise this approach are C, C++, etc.

Rust chose a novel approach to memory management. It is dictated by a set of rules named Ownership. As the name implies, memory is managed through a system of ownership along with a set of rules for the compiler to check. If any of the rules are violated, the program will fail to compile however after being compiled these rules will not affect the performance of the program in any way.

These rules are as follows:

- Each value in Rust has an *owner*.
- There can only be one owner for a given object at any given time (but the owner of a value can be subject to change).
- When the owner goes out of scope, the value will be dropped.

The base rules only account for full values, however sometimes we do not need to take ownership of an object, we just need to read its value for some operation. This is where *references* and *borrowing* come into play. A *reference* is similar to a pointer in that it's an address that can be followed to the data it's representing (which is owned by some other scope). It differs from a pointer as a reference can never be invalid, it always points to a valid value.

However, references do not allow us to alter the underlying value they just allow us to read from it. This is where *mutable references* come into play. *Mutable References* allow the programmer to modify the contents of that value. They do have a very large restriction though: if there is a mutable reference to a value, there can be no other references to that value. This restriction allows us to control the number of modifiable references to a given value a compile time which means we can actually prevent data races at compile time. This is a great leap forward in terms of security and consistency for multi-threaded programs which utilise shared memory principles.

This design even with all of its advantages for security, performance and memory management does come with some significant drawbacks which were quite noticeable when we were working on the persistent storage system. When we want to send the messages and executor checkpoints (checkpoint contains the full state of the system at the time it was taken) to the persistent storage layer for them to be stored we faced an issue. The owner of the messages (and checkpoints) is the consensus layer and we need the consensus layer to actually keep that ownership because these messages are going to have to be stored in the consensus layer until they are ready to be passed along to the executor, at which point it will take ownership of the messages so they can be executed.

Borrowing and *references* also face some restrictions when being used in conjunction with multiple threads. This is because references are not like pointers, they must always point to a valid value, as previously mentioned, so thanks to our ownership model we can have a situation where thread A creates a value V and then passes a reference $\&V$ to another thread B. Initially, this might seem like a valid program, however when we take into account this model we scenarios like the one in Figure 4.24 where the following happens:

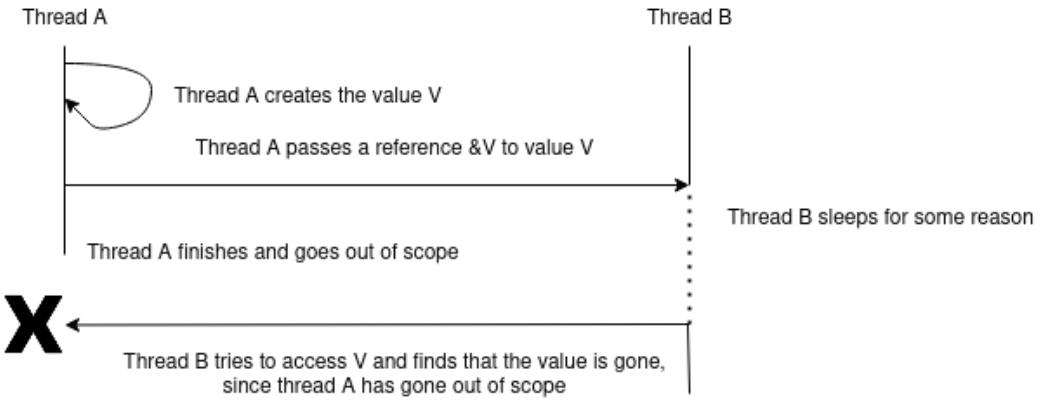


Figure 4.24: Scenario where passing references across threads without control goes wrong

- If the thread A (the owner of the value) goes out of scope, the value V is going to be deallocated.
- Due to basically random thread scheduling by the OS, there are no guarantees on when thread B is going to be executed (and it might only get executed after thread A goes out of scope) so consequently, there are no guarantees on the validness of the reference $\&V$ that is in thread B.
- Therefore the compiler does not allow for these things to happen as the liveness of the value cannot be guaranteed at the compile time.

These restrictions presented themselves as a challenge when we started working on the persistent storage layer. Since we wanted the actual IO work to be done in separate threads so we do not lose any compute time on the consensus layer, we needed a way to pass the messages to the persistent storage while still maintaining the ownership of the message in the consensus thread. We need to maintain ownership in the consensus layer because the messages have to be kept at least until we pass them along to the executor for execution.

So we were faced with a problem: We needed to pass the messages to the persistent layer while also maintaining ownership in the consensus layer (and as we have seen, passing references is not a solution).

To solve this we initially did the obvious thing that Rust already suggests for these situations: Cloning. With cloning, we effectively copy the entire value into a new variable which can then be passed to wherever we want to. For our performance tests, which presented messages with no size and the executor had no state (they were just meant as a test for how many messages could be passed through the system) this approach presented no issues while also doing exactly what we intended. However, when we started to abstract our train of thought and started considering systems with actual states and requests with sizes we quickly started to identify many situations where this approach could hamper performance and cause issues with resource usage, namely:

- Since the original owner of the information is the consensus thread, the cloning of the messages and application states would also have to be performed there. This is a non-issue when we have 0-byte requests and no application state, however, if we have a situation where the state of the application is several gigabytes in size (or even larger) we can start having several issues with performance. Firstly cloning this object requires us to allocate a similarly sized chunk of memory which can take an indeterminate amount of time (or even crash the program, if there is no memory available). Secondly, after allocating the memory we would still have to copy over the entire value which again can take a large amount of time.
- If the state of the application is large enough, having multiple copies of the same state in different parts of the system is not only inefficient and slow but also a humongous waste of resources.

Therefore, we decided we needed a new alternative that solved all of the above-mentioned issues. Our solution was to use *Reference counting*. *Reference counters* work similarly to C++'s *shared_ptrs*. That is, instead of having a single owner for a given value, we can have an arbitrary number of owners and the value will only be disposed of when all owners go out of scope. Because we are working in a multi-threaded environment, we need to use the *Atomic Reference Counter* or *Arc* to guarantee consistency on the counting of owners. However it was not that simple, since we know that there can only be one mutable reference simultaneously for a given value and since an *Arc* value represents ownership of the value, *Arc* requires us to protect the underlying value from multiple threads attempting to modify it at once.

To do this Rust uses automatic traits called *Sync* and *Send* which the compiler derives based on the elements of the value if possible or it can be explicitly added by the programmer if he is sure of the safety of operation (manually adding this is an **unsafe** operation, so it relies on the programmer to assure that nothing bad can happen). We will not speak further on these traits since they go a bit out of the bounds of what we want to discuss. In summary, since we knew that the messages and states are immutable, we were able to wrap them with an unmodifiable type, which implicitly guarantees that the underlying value is safe to be accessed from any number of threads (immutable data is by definition thread-safe).

With these changes, we were able to avoid cloning these potentially huge chunks of memory and instead, we replaced them with a simple atomic increment performed by the *Atomic Reference Counter*.

4.8 Observer Clients

Observer clients are a novel client type whose purpose is to monitor for updates of the consensus state machine. They receive information that is related to the consensus state alone, they do not process any information related to the service implemented. This means that they mainly receive information that is relevant to the consensus state machine namely:

- Current state.
- Current view number.
- Current sequence number.

We also provide them with some other information not directly related to the state machines but also relevant to the function of the SMR protocol, like:

- Checkpoint starts and ends.
- Changes in the phase of the SMR. For example when we performing view changes and state transfers.

These clients allow us to monitor the health of the system, whether all replicas are maintaining synchronized consensus state machines and what the current state is amongst other statistical monitor aspects. It can also be easily extended to support more operations, more information and other aspects.

Since this was only a monitoring and "extra" part of the system (it is not necessary for the correct operation of the general system) we wanted to make sure the design was completely modular and this entire module could be entirely disabled without the user having to perform any code modifications. Also, we wanted the performance of the system to not be affected in any way shape or form, so we had to make sure that all the work done towards these clients was handled completely differently from the rest of the system (which also fits into the modular design approach we mentioned earlier).

Figure 4.25 aims to describe the design we took when implementing this system on the replica side. We created a new module name *Observer* with a corresponding dedicated thread and incoming message buffer. This buffer, like in previous instances of use for similar buffers, is essential to assure that the core replica systems never get at all slowed down because they are waiting for the Observer module to complete processing the previously received message. Adding this intermediary buffer allows the observer module to maintain a backlog of messages left to process (although this backlog shouldn't exist, since there is little processing done on the messages by the Observer module, mainly just dispatching the correct messages to the observer clients). In this image, we can also see another characteristic of this system which is that not all clients are Observers (represented in the image by having the observer module only contact a subset of the connected clients). Instead, in order to become an observer a client has to go through a "registration" process that we will discuss further on in this section.

Another thing we can also observe is that each replica independently notifies its connected Observers of any events that are occurring in that particular replica. There are no messages traded between replicas to know what to send to the Observer clients, instead each replica is responsible for itself. We will also discuss further ahead how this is handled and how we are able to utilise this feature to our advantage to give us more detailed information and a better look at the overall health of the system.

Moving on to how we implemented this on the client side, we can see what the final client architecture looks like in Figure 4.26. We followed the same principle of isolation that we set on the replicas, especially because clients have even more possibilities than the replicas. We can have the observer part completely disabled, we can have clients that are not observer clients but can become so and we have observer clients that are actively listening to update events from the replicas. It also gives us a look at how the observer events are delivered to the service application implemented on top of the clients.

The delivery of observed events on the clients works through callbacks. This means that the service registers a callback (which in this case is a simple function that accepts the event that has occurred as an argument, along with some other information that we will discuss further

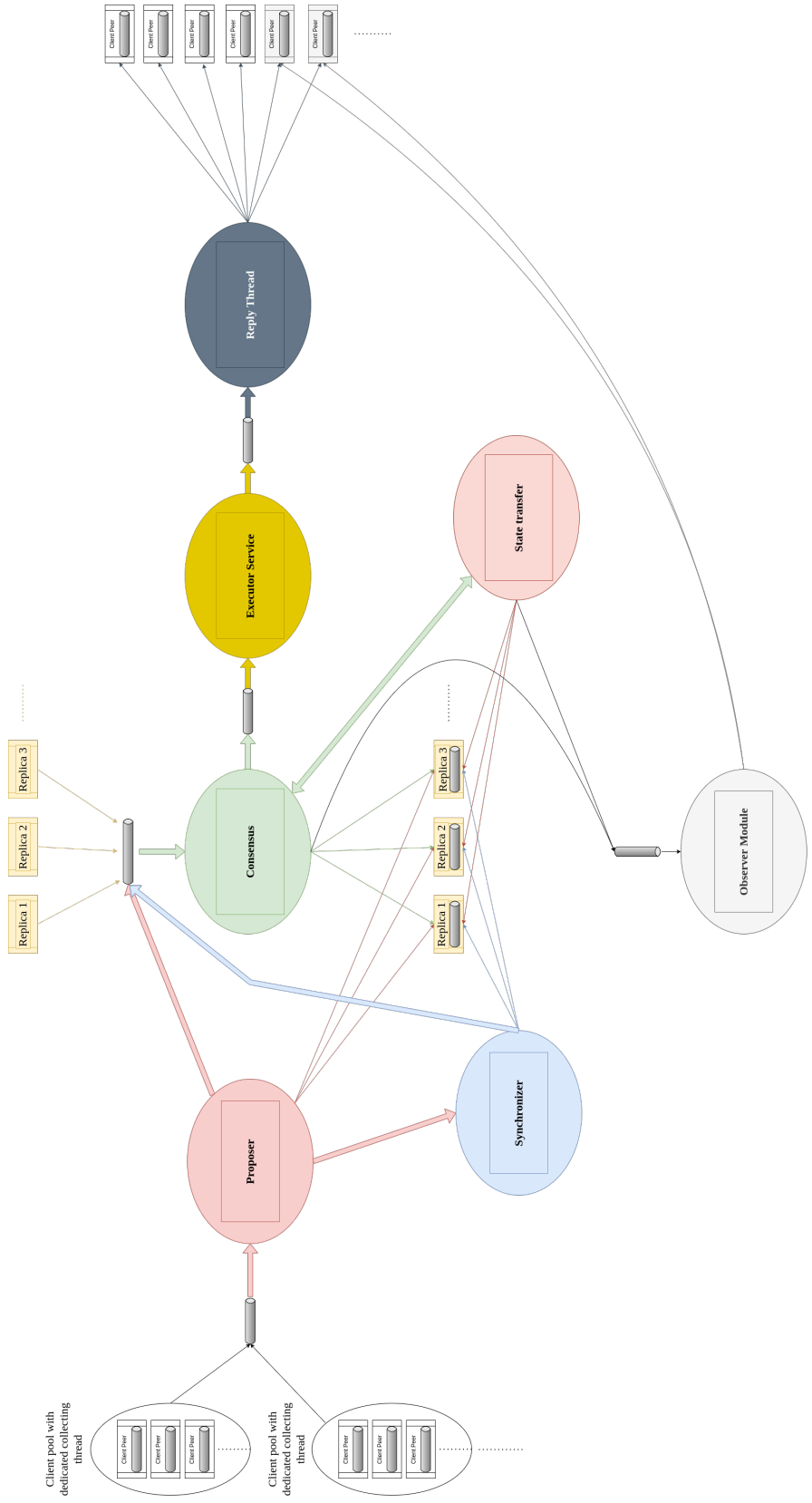


Figure 4.25: FeBFT's architecture with the observer module added on

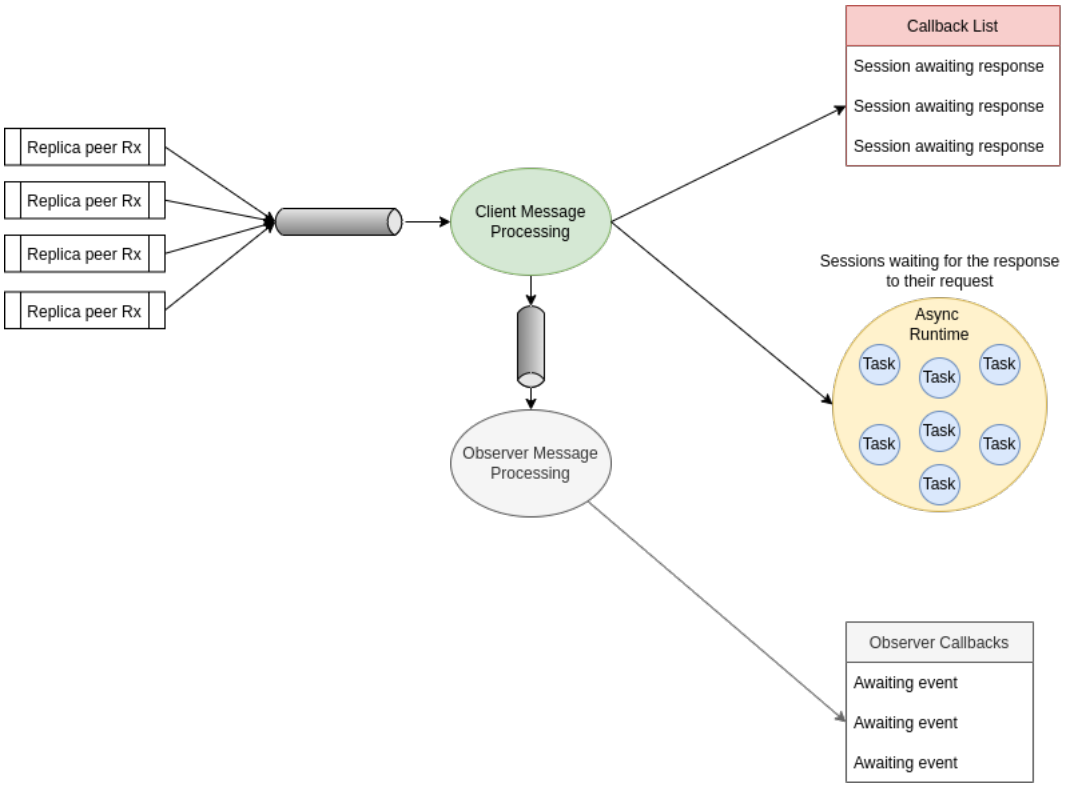


Figure 4.26: The client design with the newly added observer clients

ahead) which will be called as soon as an event is delivered to the observer from a replica. These callbacks are called from the same thread that initially processes the observer messages, so the service implementation should take care of what gets executed in this thread because it might block the execution and reception of other messages, which in turn could potentially lead to a loss of performance (if the time taken is too high and can't keep up with the number of observed events).

Initialising an Observer client As we have seen, not all clients are observer clients and they have to go through a process in order to become one. With the current implementation, any regular client can initialise its observer module and become an observer client however this can be easily changed (given we introduced a separate module with a dedicated thread that handles observer-only events, we can add some sort of authentication or requirement without slowing the core SMR systems). In order to register a client as an observer, he must first broadcast an *ObserverRegister* message to all replicas so they register the client as an active observer. We then have to wait for a number of responses from the replicas until the observer is officially registered. After the observer is officially registered and ready, we return a handle for the service to be able to register callbacks.

Handling the independent replica reporting As previously mentioned, each replica is responsible for notifying the observers of its own state, there is no inter-replica communication

relating to observer states being passed so the client has to be able to handle it. This is great for our monitoring needs and target because we are able to individually monitor certain replicas for failure or for inconsistencies. However, how are we able to distinguish correct updates from observations made by faulty replicas? To handle this, we made clients wait for $2f + 1$ similar event observations before delivering that update as the correct update. This information is given when we are invoking the callbacks to deliver the state updates so the service can correctly identify when the observed event can be taken as a correct update of the quorum's state machine or as the faulty individual state of any given replica.

4.9 Read scalability

Initially, FeBFT did not handle operations that do not require any particular ordering to be executed. This type of request allows our system to execute a lot more requests since they do not have to go through the consensus protocol to be ordered, they just have to be executed.

This means that unordered requests have the potential to scale our performance significantly since they are not bound by the physical limitation of how many consensus instances can be executed in a given time frame (a limitation imposed by the various rounds of messaging). Instead, they are only bound by the networking capabilities of the replicas and the executor's ability to execute those requests.

4.9.1 Unordered operations

Unordered operations work similarly to ordered operations except they do not have to go through the consensus layer of the SMR protocol. This is because, as the name states, they do not require ordering. This, as mentioned previously, means the amount of unordered requests that can be executed by the SMR protocol is bound only by the number of requests the network can receive/reply to and the number of requests the executor can handle.

As we can see in Figure 4.27, most of the components of the SMR protocol do not partake in the execution of unordered requests. The components that do partake in the execution of unordered requests are the coloured components so in this case:

- The client pools.
- The proposer thread.
- The executor.
- The reply thread.

The client pools are responsible for retrieving the requests from the client queues, similar to

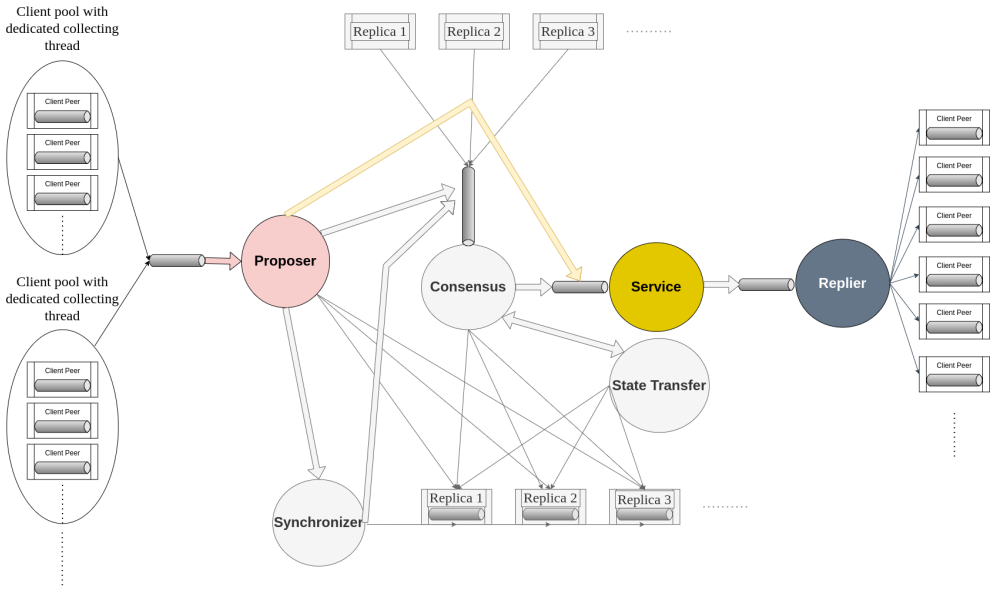


Figure 4.27: Diagram showing the simplified path unordered requests go through

their task on regular ordered requests. Again similarly to ordered requests, we then pass the requests to the proposer layer which will be responsible for processing the actual requests.

The proposer is responsible for aggregating the requests into batches so they can be then passed along to the executor. Initially, it might not make much sense to aggregate the unordered requests since as previously mentioned we are not limited by the number of consensus instances that can be decided in a given time frame. However, we believe that the batching actually helps performance, even if it somewhat harms latency. The reason for this opinion is that inserting and removing elements from a channel takes time and since the channel is actually being accessed by more than one thread, there has to be synchronization between them. Our thought is that we want to minimise the number of operations on the channel and maximise the amount of work done by each of them (with the same work of synchronization we can pass 1 operation or a batch of 10000 operations). Instead of performing these operations on a shared channel, we are performing them on a local vector which is much more efficient. We then have a configurable target batch size and timeout and whichever is triggered first will cause the batch of unordered requests to be passed along to the executor. These parameters can then be altered by the user depending on his requirements for the service.

We do recognise that this could also lead to some underutilisation of the executor service while the proposer is still waiting for enough requests to form a batch of the target size or waiting for enough time to pass so it triggers the timeout, time which can be wasted if the executor does not receive any ordered requests to execute in the meantime (or if he is not yet done executing the previous batch of unordered requests).

The executor then actually executes the requests and produces the corresponding replies which are then passed to the reply thread so they can be delivered to the clients.

Also, since these requests do not pass through the consensus layer, they do not get put into persistent storage. There is no problem with this since unordered requests are not meant to perform any alterations to the state of the application (that's why they do not have to be ordered, if they performed any alterations we would need to know the order so we can reproduce them and get the same result on all replicas) and we, therefore, do not need to remember them so they can be executed if the state is lost, since they have no impact on the said state.

In terms of fault tolerance for unordered requests, we must analyse them as the unordered requests that they are.

We have two possibilities for handling fault tolerance on unordered requests:

1. ***Crash fault tolerance*** - In order to ensure crash fault tolerance on unordered requests, clients need only communicate with a single replica while also employing a timeout on the request (or if we want to handle crash fault tolerance without having to wait for the timeout, we can broadcast the request to $f+1$ replicas). When we receive the first response, we can instantly provide it to the client as we do not need to wait for any other replies (crash fault tolerance assumes that a replica is either working and online or not working and offline, there cannot be a replica that is online and replying to requests that is replying to them wrongly).

2. ***Byzantine fault tolerance*** - Byzantine fault tolerance is quite a bit more complicated due to the pure nature of unordered requests. As we have stated (and the name itself states) there is no guaranteed ordering for when these requests get executed so it's possible that even if a request is broadcast simultaneously to n replicas, we can receive n different replies as a result, all while having 0 faulty nodes, just as a result of the ordering of execution. This problem gets compounded if the data being accessed has very high liveness (it is being very frequently updated while we are performing the unordered requests). This information does not alter the fact that to assure protection against byzantine faults, we cannot trust the response of any f replicas, instead, we must have equal responses from at least $f+1$ replicas. So, to perform an unordered request with byzantine fault tolerance, we must first take into account whether the data that we are accessing is live or is not currently being modified. The number of replicas that the client should attempt to contact will vary greatly with this information and if the data's liveness is very high and we intend to maintain byzantine fault tolerance it's sometimes worth the overhead of performing an ordered request over having to perform a possibly unending amount of unordered request rounds. So the implementation for these requests has to have a lot of input from the service implementations in order to have the best performance possible. If the data is not very live, we can point to a number of replicas close to $f+1$, like $2f+1$. We then wait for the responses and when we receive $f+1$ equal responses we deliver the response to the client. If we do not receive enough equal responses we then have two possibilities: We retry the request as an unordered request (while also possibly contacting more replicas, if possible) or by moving the request to an ordered request, as previously mentioned. There

should be a limit on how many retries on the unordered request are made since spamming the replicas with unordered requests is much less efficient than just performing an ordered request even when we take into account the overhead of having to order the operations.

4.9.2 Quorum Followers

In the previous section, we saw the first step to augmenting the read requests' performance by separating them from the ordered requests and making them skip the entire SMR protocol so we aren't limited by the physical amount of consensus instances that can be done. However, we are still limited by the number of requests that both the networking and the executor can handle.

Similarly, we are also tasking the replicas that are a part of the decision quorum with the load of receiving, processing and responding to unordered requests. So if the quorum replicas were already reaching their peak load on networking and request execution (they were not being limited by persistent storage or consensus layer bottlenecks, which is what unordered requests are meant to avoid) adding the overhead of unordered requests on top of it would just cause the quorum replicas to be overloaded and therefore reduce the number of ordered requests that it can execute which is extremely against what we intend to do in this system.

So what we want is a way to scale the amount of unordered requests our system can handle without hurting the quorum replicas' ability to process ordered requests. We do not intend for this solution to be costless as if the user wants low cost he can still maintain the base $3f + 1$ quorum replicas and deal with the performance limitations associated with that choice (we can't keep the cake and eat it). We want a solution that gives the user options as to how to scale and distribute the load of the system in order to handle larger workloads (in this case read focused workloads). We believe this to be very important for several possible applications of our software, namely databases that might require an ability to scale the number of requests that it can handle at a moment's notice both geographically and in terms of performance.

Our solution to these requirements was the introduction of **Followers**. A **Follower** performs a similar role to a replica except it does not participate in the quorum responsible for deciding the ordering of requests. They perform a subset of the operations a normal replica. Namely, they listen to and process messages sent in the quorum however they do not actually respond to them (therefore they do not partake in the consensus). They also have the ability to perform state transfers (as recipient and as a provider) which will also allow them to perform some important tasks that we will talk about further down.

Followers introduce an ability that, to the best of our knowledge, does not yet exist in any other BFT SMR system and that introduces some very interesting characteristics for many uses. Namely, since they are capable of executing unordered requests, we have given our system the ability to horizontally scale the number of requests the system can handle. This can be visualised by Figure 4.28. As is visible, normally there are no interactions between followers, instead, followers receive all of their updates from the quorum replicas and then just communicate with

clients. This however does not really tell the entire story of how the communication is handled between the quorum replicas and the followers.

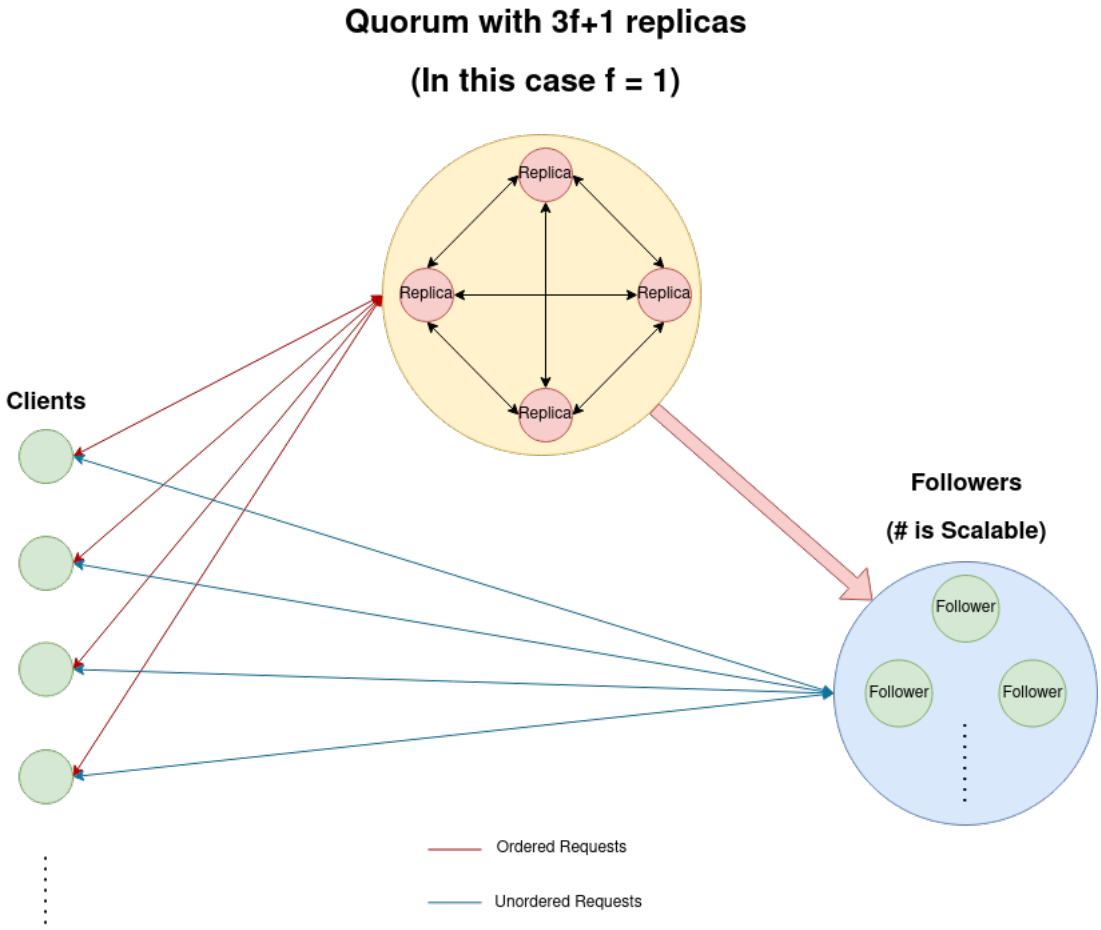


Figure 4.28: Illustrative design of the quorum and followers and how they interact

Request propagation between quorum replicas and followers We know that in a given BFT SMR system, the leader is responsible for aggregating client requests into *batches*, assigning them sequence numbers and then broadcast the Pre-Prepare request to all other quorum replicas in the system. This Pre-Prepare request contains all of the client requests in that batch so it can be of a very large size, depending on the batch size. This is exactly what we intended to do, as explained in Section 4.3, however, it does lead us straight to a pretty big possible bottleneck in our system. This bottleneck is directly related to the size of the Pre-Prepare requests and the fact that they have to be sent to every single quorum participating replica in the system so it will push the leader's upload bandwidth to the absolute maximum possible. Now, if we also task the leader as the sender of these Pre-Prepare requests to all of the followers we can quickly see how the horizontal scaling promises quickly go down the drain since we would have to also scale the leader's upload bandwidth very significantly (and since any replica can be the leader of the system, depending on the current view, all replicas that participate in the quorum would have to be prepared to do this) if we do not want to lose any performance. To solve this, we came

up with a way of propagating these requests without adding all this extra strain on the leader. The leader still broadcasts the Pre-Prepare request to all the other quorum replicas but does not broadcast it to the followers. Instead, the other replicas in the quorum (which only need to send broadcast small prepare and commit messages, so they do not stress their upload as much) are responsible for forwarding the messages sent by the leader to the followers.

While developing this approach, we discovered a potentially big problem with the fault tolerance of this system. If the quorum replica responsible for sending to a certain follower is faulty and doesn't send them, the follower would have no way of knowing this. This is because of two things:

- Clients do not broadcast their ordered requests to the followers, so the followers never perform timeouts and thusly can never know if the replica that was assigned to send them the requests is faulty.
- The replicas that are a part of the quorum and that do perform timeouts to determine when the leader is faulty are not watchful of the forwarded requests by other requests since they do not receive them.

So, in order to tolerate f byzantine faults on the replicas of the quorum we choose $f + 1$ quorum replicas to send the Pre-Prepares to a given follower. We do not require the follower to wait for all of the messages because as soon as he receives the first request he can verify the signature and assert that it has been sent from the leader. We only have to send the $f + 1$ messages because there are f replicas that can be faulty and not send the message, however, thanks to public key cryptography it's impossible for a replica to forge a request.

Consistency in Followers As we have seen, followers are not part of the quorum and therefore the quorum can advance without the "confirmation" that the followers have also received and processed the requests. This is by design since we do not want to slow down the quorum's decisions due to the followers as the propagation of messages to them and actually between them can take quite some time. Therefore, since the quorum does not wait for the followers, we say that they are *eventually consistent*. In other words, if no new consensus instances are processed, followers will eventually converge on the last updated value. How long the propagation takes is highly dependent on how it's architectured which will be discussed in Section

Using followers to restore the state of faulty/new replicas or followers Usually, when a replica crashes and has to restore its state, it will request it from another quorum participating replica (or replicas, depending on the BFT SMR system). As we have just previously seen, the quorum replicas require their upload speed for quorum-related purposes, for keeping followers up to date and to reply to clients. Given this, we realised that the followers did not have nearly the same strain on their networking as a regular quorum participating replica (which at any point in its lifespan could be tasked with being the leader or with having to forward the Pre-Prepare

requests sent by the leader) since they only had to receive the updates from the quorum and reply to the requests their clients made. So, given this information, we can conclude that utilising the followers in order to help the state transfer protocol would be very beneficial to the performance of the system. It also means that we can bootstrap new followers with almost no cost to the quorum replicas (except for the fact that the follower will also have to receive the status updates but we will see more about further on), giving us the scalability properties we desired to obtain.

Using followers as live backup replicas Given that followers share the state machines with regular replicas (along with sharing the underlying storage design and most of the actual code), we can theorise that they are able to be used as *backups*. That is, when we are able to detect a fault (notably most byzantine faults are not detectable, so in reality, this is mostly applicable for crash faults) we can take the faulty replica out of the quorum and swap a follower into its place without having to perform any state transfers or other updating procedures. This would technically allow us to provide a more robust fault tolerance model since when we are able to detect the faulty replica, it can be instantly fixed and the system goes back to a more stable state (for example if $f = 1$ and we have a replica that has crashed, then no other replicas can fault however if we swap in a correct follower then we get back the tolerance to a fault). We did not implement this, since it requires a system with support for live reconfiguration of the system which at the time, FeBFT does not support.

Handling received ordered requests In the current implementation, any ordered requests made to a **Follower** are dropped, since they are not meant to handle ordered requests as this job is tasked to the quorum replicas. It is possible that allowing followers to forward requests to the quorum replicas might allow for better overall performance if there are dedicated channels of communication for replica networking, separate from the channels used to communicate with clients. If the bottleneck of the system in question is the networking bandwidth of the client-facing entry point, this solution might help alleviate the problem by providing more entry points for the client's requests. Since we believed this to be a bit of a fringe case that will rarely be encountered, we decided it was not worth the extra work and resulting complexity of functioning and therefore did not implement this.

```
fn dump(&self, vec: &mut Vec<T>) -> Result<usize, QueueError<T>> {
    let backoff = BackoffN::new();
    let mut prev_head = self.head.load(Relaxed);
    loop {
        let tail = self.tail.load(Relaxed);
        match self.head.compare_exchange_weak(prev_head, tail,
                                               SeqCst,
                                               Relaxed) {
            Ok(_) => {}
            Err(curr_head) => {
                prev_head = curr_head;
                backoff.spin();
                continue;
            }
        }
        if prev_head == tail {
            return Ok(0);
        }
        let mut collected = 0;
        loop {
            if prev_head == tail {
                break;
            }

            let head_ind = self.extract_index_of(prev_head);
            let head_lap = self.extract_lap_of(prev_head);
            let slot = unsafe { self.array.get_unchecked(head_ind) };
            let sequence = slot.sequence.load(Acquire);
            if prev_head + 1 == sequence {
                let new_head = if head_ind + 1 < self.capacity() {
                    prev_head + 1
                } else {
                    head_lap.wrapping_add(self.one_lap())
                };
                let elem = unsafe { slot.value.get().read().assume_init() };
                slot.sequence.store(prev_head.wrapping_add(self.one_lap()),
                    Ordering::Release);
                vec.push(elem);
                collected += 1;
                prev_head = new_head;
            } else if sequence == prev_head {
                atomic::fence(SeqCst);
                backoff.spin();
            } else {
                backoff.snooze();
            }
        }
        return Ok(collected);
    }
}
```

Listing 4.2: Dump method of the Lock Free Bounded Queue

```
pub struct MQueue<T> {
    capacity: usize,
    array: Mutex<QueueData<T>>,
    full_notifier: Condvar,
    empty_notifier: Condvar,
    backoff: bool,
}
struct QueueData<T> {
    array: Cell<Vec<Option<T>>>,
    head: usize,
    size: usize,
}
```

Listing 4.3: Data structure of the Mutex Based Queue

```
pub struct LFBRArrayQueue<T> {
    array: UnsafeWrapper<Vec<Option<T>>>,
    head: CachePadded<AtomicUsize>,
    tail: CachePadded<AtomicUsize>,
    rooms: Rooms,
    is_full: AtomicBool,
    capacity: usize,
}
```

Listing 4.4: Data structure of the Lock Free Rooms based Queue

```
enum State {
    FREE,
    OCCUPIED {
        currently_inside: u32,
        room_nr: u32,
    },
}
struct Rooms {
    state: CachePadded<AtomicU64>,
    room_count: u32,
    listeners: AtomicU16,
    event: Event,
}
```

Listing 4.5: Data structures relating to the Lock Free Rooms implementation

Chapter 5

Evaluation

5.1 Testing methodology and data collection

In order to test the throughput of our BFT SMR system, we developed a series of tests meant to stress the quorum replicas. The idea of the tests as well as the parameters used in them will be described in Section 5.2.

These tests measure various indicators that allow us to quantify the overall performance of the system such as operations per second, batch size, consensus latency, among others. Usually, when we want to measure the amount of operations that such a system can process we check on a regular period how many operations have been done in that period and then perform some calculations in order to obtain the operations per second number. However, due to several difficulties in BFT-SMaRt we were not able to employ this style of performance measurements. The difficulties are entirely related to the existence and utilization of a garbage collector. As we have seen repeatedly throughout this dissertation, BFT-SMaRt's choice of language means it requires a garbage collector to manage its memory. The GC might, at any point in time, stop all threads from execution so it can clean up the unused memory without any type of warning and for an indeterminate amount of time (and in the case of our stress tests these stops could take more than 5-10 seconds at a time). Therefore we can clearly see any approach based on the previously described system would yield measurements with large gaps in between them which would lead to possibly incorrect measurements.

To remedy this, the original developers of BFT-SMaRt changed how they collected data to be dependent on the amount of operations performed by the system instead of a time based metrics approach. It works by measuring how long it took to perform X operations and then with that information it would calculate the amount of requests performed in that time. In the testing we created for FeBFT we followed this same principle as we needed to have comparable testing setups and data collection so we could correctly compare the two systems. However this approach is also very useful for actually hiding the fact that no operations were performed in a given time period, since it will always provide a number > 0 of operations performed per second

even if for a number seconds the system performed 0 operations. This meant that the final average of operations performed per second is going to be heavily skewed since it has little data points that reference the time where the system processed very few operations (since to provide measurements the system has to process operations) while having a large number of data points from when the system is working correctly and at full performance.

To address this issue, we decided to implement a system in our data processing and analysis tools that aggregates all of the readings in buckets based on the time that they were made on, calculates the average number of readings in non-empty buckets and then utilizes it to fill the empty buckets with measurements of 0 since during that period of time the system effectively did not perform any operations. We run each of the testing scenarios at least 10 times and aggregate the results in order to obtain a consistent and reliable view on the performance of the system. We then perform the mean and the standard deviation of each of the buckets in order to plot the results and obtain the graphs we use in the upcoming results.

5.2 Initial FeBFT performance

As we have seen in Chapter 4, FeBFT is a descendent of PBFT and BFT-SMaRt that aims to address a lot of disadvantages with the design and implementation choices made by their developers.

We will now analyse the performance of this original architecture. To determine the performance, we utilised the following benchmark which was engineered to test the maximum performance of the system:

- We have a set number of clients, 1000 in this case.
- Each client only has 1 session active, which means that each client can only perform one request at a time.
- Clients perform a set number of operations.
- When a client is done performing the operations, it dies.
- Operations do not actually do anything, they are a simple blank request. This is because we do not intend for the executor to present a bottleneck we only want to test the performance of the actual abstract SMR system. Logically a poorly designed Service will limit our performance.

In this case, all requests made have a size of 1 byte (excluding the header, which is a fixed size, this is only referring to the payload). This is again because we are not interested in stressing the network bandwidth. Instead, as mentioned, we only intend to discover the absolute maximum performance that the batching and proposing system can achieve.

All tests were performed on a LAN cluster consisting of 5 identically specced machines all connected via a 1Gbps network. The machines have the following configuration:

- 2x AMD EPYC 7272 12-Core 2.9GHz with SMT enabled (Simultaneous Multithreading). This means it has 24 logical threads. In total with both CPUs we get 48 logical threads and 24 physical processing cores.
- 256GB of RAM
- 256GB boot NVME SSD
- Ubuntu 20.04

We assigned one replica per machine and the remaining machine as the client machine. This means we have 4 replicas, which gives a fault tolerance of $f = 1$ (given $n = 3f + 1$).

We then collected various statistics like operations done per second, average batch size, various message latencies, etc.

This benchmark is an identical copy of the benchmark tests done in order to test the performance of BFT-SMaRt. We will also present results for BFT-SMaRt so we can compare them to see which has the best performance.

As we are able to see from Figure 5.1, FeBFT handled 36881 ± 6281 operations per second on average. The 95% confidence interval is $36709 - 37053$ operations per second. To put these numbers into perspective, we also run this same test with BFT-SMaRt in Section 5.4.3.

We can see that BFT-SMaRt's numbers for this test are higher, maintaining an average of around 50000 operations per second. This shows a large gap in performance. We immediately realized that there had to be something wrong with our library, because having such a difference meant that it was due to large implementation or design issues.

Initially, we thought that this might be due to the way we were storing messages in the log. This concern came up since the original developer of the library had indicated to us that from the profiling he had done he was able to see that there was a very large bottleneck due to the way the operations were being stored. As we have seen from the BFT SMR protocols mentioned in this dissertation, the system works by storing a log of the decisions that have been made by the quorum (log which gets reset whenever we perform a checkpoint). This log is what is going to be utilised when we need to perform state transfers and other SMR related logic.

The message log in question was a single HashMap. Each individual client message is placed onto this hash map upon reception of the request. These messages will be taken from this client requests map when they are proposed by the leader and will then be moved to another message log (also a HashMap) in combination with the prepare and commit messages. We can see how this leads to a lot of accesses of the hash maps which in turn leads to a lot of hash operations which can be very expensive in terms of CPU. However, when we put the actual number into perspective

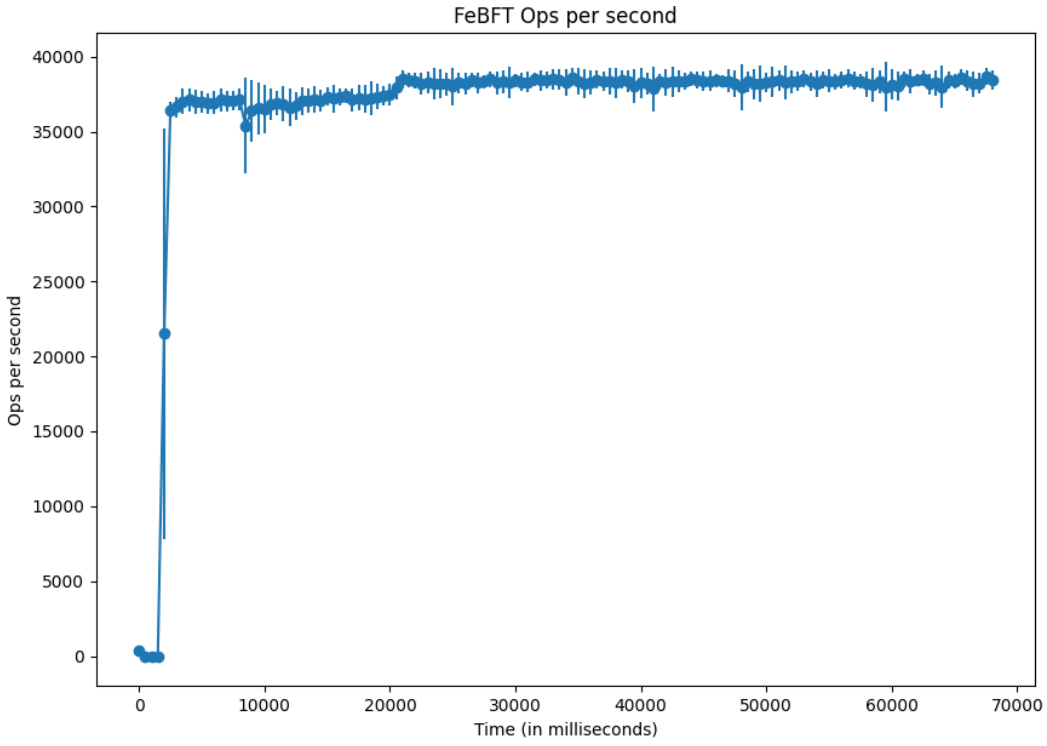


Figure 5.1: The operations per second of the original FeBFT SMR system

we were quickly able to realise this could not be the cause of the performance problems. This is because even though hashing operations are indeed expensive and we perform multiple accesses to the hash map for each consensus instance that is processed (along with multiple accesses per each individual message as well), the amount of actual operations being processed is completely negligible and cannot possibly be the cause of such a large bottleneck. So we continued in our pursuit of the culprit for these results.

We then looked at the batch sizes for FeBFT, which we can see in Figure 5.2. As is visible, the batch size varies a lot but tends to keep itself around the 40 requests per batch mark (supported by the average batch size of 43 requests per batch). Comparing this to BFT-SMaRt's batch size (which averages at around 317 requests per batch) we can see that our batch size is quite small. If we take the total number of operations per second and divide it by how many requests are contained in each individual batch we get the amount of consensus instances that are being performed per second (on average).

So doing this math for BFT-SMaRt yields us:

$$\frac{50000}{317} \simeq 157 \tag{5.1}$$

While doing the same operation for FeBFT gives us:

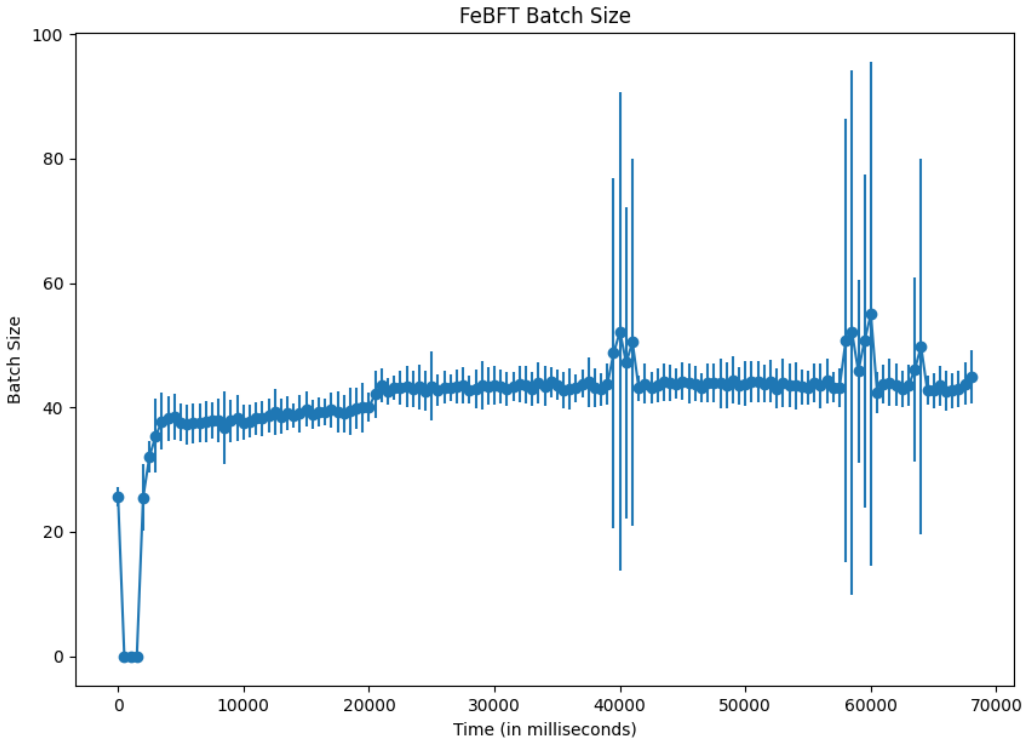


Figure 5.2: The size of the batches in the original FeBFT SMR system

$$\frac{37400}{40} \simeq 935 \tag{5.2}$$

So, as we can see, BFT-SMaRt is actually doing a lot less consensus instances per second than FeBFT so the problem is clearly not the amount of rounds we can perform in a time frame. We can clearly see the issue is related to the amount of requests that we are squeezing into a each of those batches. As we saw in Section 4.3.1, FeBFT relies heavily on concurrency and the unpredictability of the OS scheduler in order to build large request batches. After viewing these performance numbers, we decided to give some thought as to why this could be happening and why the batching was working so poorly. We were using 1000 concurrent clients, so there is no possible way that the problem was due to the lack of concurrency, since we could completely saturate all of the cores of the replica with reception of requests, meaning adding more clients will not increase the amount of concurrency that we will experience at that point. So since we can't increase the concurrency any further and currently the batching performance is not good, we had to find a way to improve this system.

5.3 Performance of the overall algorithm with the new Data Structures

After having analysed the performance of each of the implementations and chosen the one we wanted to use, we then implemented the necessary changes and proceeded to test the performance of the system with the newly made changes.

To do this, we utilised the same testing setup and the same benchmark that was described in Section 5.2.

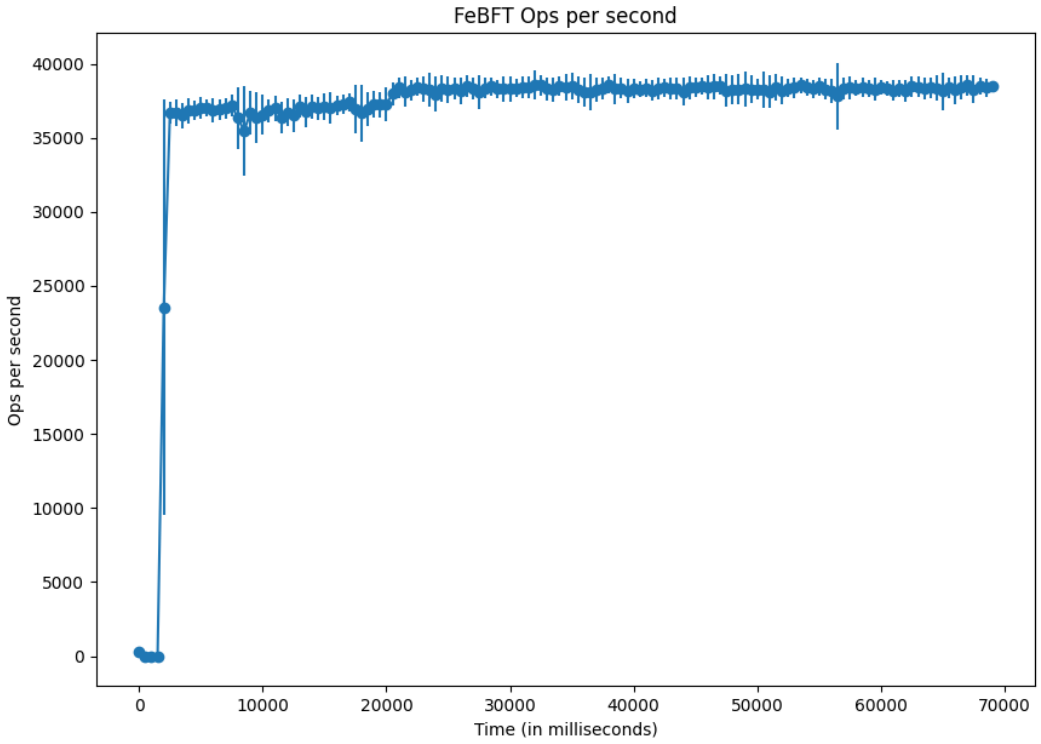


Figure 5.3: The operations per second of the FeBFT SMR system with the new data structures in place

As we can see from Figure 5.3, the overall performance of FeBFT seems to have maintained the same meaning the changes we made did not affect the performance

Similarly to the operations per second, Figure 5.4 shows that the average batch size has not changed much which makes sense given the operations per second didn't change either.

This is quite surprising to us given the information and explanation that was provided to us as to why the original developer for the observed performance was related to the data structures used. We proceeded to search for points of contention or bottlenecks that might be leading to performance loss. In particular we went on to completely alter the architecture of the system as

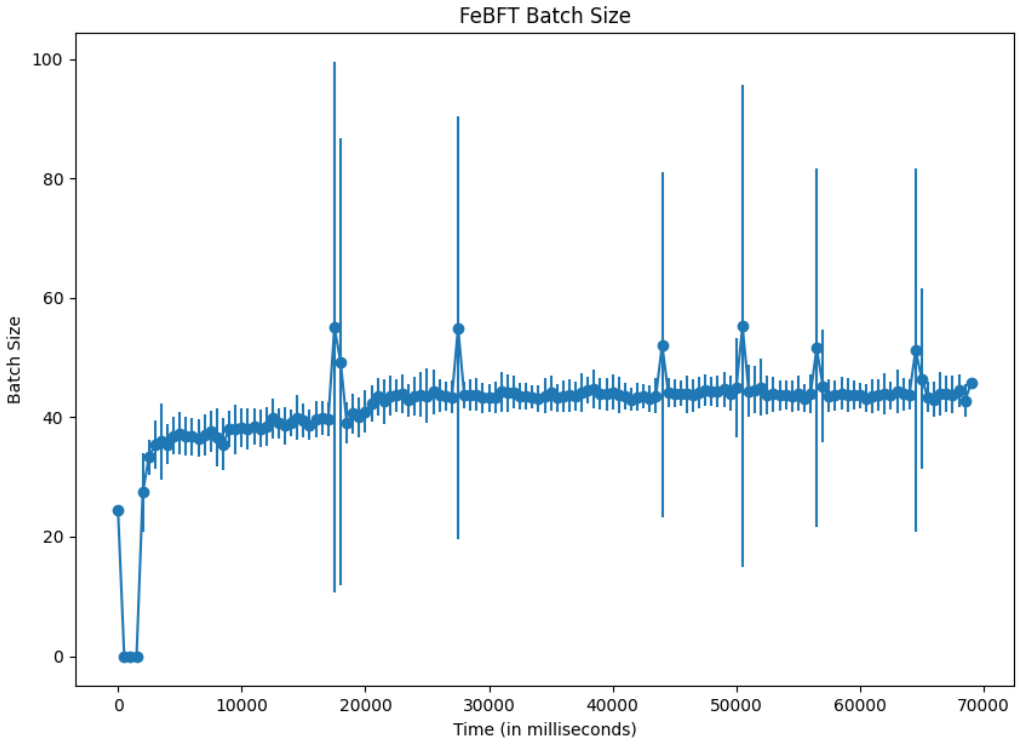


Figure 5.4: The batch size of FeBFT with the new data structures in place

described in Chapter 4.

5.4 Final performance analysis

We used 2 sets of tests in order to test the performance of the new changes to the architecture and then separately to each of the networking approaches we developed in order to know which was the highest performant, why and how we could possibly improve on it.

The testing machines used are the same as the testing setup described in Section 5.2.

The first test used was the same test described in Section 5.2, utilising synchronous clients that only perform one operation concurrently per client. This allowed us to test the performance initially but then we started to hit a few roadblocks. Notably, we weren't being able to increase the amount of operations per second the system was processing no matter how many clients we attempted to add. In some situations we were even seeing a decline of operations per second when increasing the amount of clients.

Why we believed the client operations were limiting performance Since increasing the client count was not resulting in any increase of the performance of the system we started

attempting to discover what was the bottleneck in the test that was not allowing the operations per second to increase. Our most prominent idea (which was later confirmed by the performance of our second test) was that this issue with performance was being caused by an excess of threads which was then overflowing the runtime (in situations where we were still utilising it) and the OS scheduler (in the situations where we had already changed the networking system). This makes a lot of sense because for each of the clients we have to run 1 thread per sending connection to replicas (so in this test it's 4 threads), 1 thread per receiving connection (again, 4 threads in this test) and 1 thread for message processing. Each client then had a thread dedicated for producing requests and waiting for their response. So if we have 1000 clients, we would need a total of 10000 threads/tasks. That high of a number of threads means that the CPU will start to struggle to advance the state in many of those threads. It would also have to spend a large part of its time waiting for context switches to finish, effectively wasting lots of very important CPU time.

In order to resolve these issues, we had to find a way to reduce the amount of threads that we were using. Taking into account the way FeBFT clients operate, we knew that the only way to do this was to reduce the amount of clients that were used in the test. However, how are we able to reduce the amount of clients while still increasing the amount of overall operations per second?

To achieve this, we had to make each client send more than one request concurrently. This is because, when the clients are performing requests synchronously they are very limited by the latency and processing time of the quorum (they can only start the next request when they receive the response to the current one). By allowing clients to perform many requests concurrently, we are able to reduce the effects of this issue. FeBFT already have the ability to perform concurrent requests, utilising various sessions. Each session can only have 1 concurrent request, but we can have up to $2^{32} - 1$ sessions at the same time meaning we get a lot of room for concurrent requests. Each of these sessions can only have 1 request concurrently but it can be reutilised for many requests.

So we designed our second test with the purpose of actually stressing the FeBFT quorum and getting a better insight into how many operations FeBFT actually withstands.

This test consists of the same principle, a group of clients making requests to the server, except now instead of having many clients performing 1 requests we have few clients performing many requests concurrently. In principle this should mean that our issue with excessive thread usage should be somewhat mitigated. It is not entirely migrated since we still used a decent amount of clients. We didn't take the number of clients down to as few as possible because of a couple of reasons:

- If we went too low on the amount of clients then we would not be able to actually utilise the full performance potential of the machines being used.
- Second and most influential, we wanted to have a semi-realistic scenario where we could check the performance of our batching approaches and overall management of clients.

Performing this test with an unrealistic, very small, amount of clients makes no sense since it's more than likely to not be the use case of this middleware.

Each client can perform at most n concurrent requests. When they have already sent n requests without receiving a response, they shall wait until they have received responses to some of their requests, after which they can send requests again until they reach the limit and so forth.

We will now see the various tests we conducted, their results and the conclusions that we were able to draw from each of them and what changes ensued.

5.4.1 Performance with the client pools

As we have seen, one of our first steps after we saw the custom data structures did not work as intended was to rework how the requests were received, processed, aggregated and the subsequent batch of requests created. We did this with the introduction of client pools and the separation of client and replica request flow, with the introduction of synchronous connections between replicas (as opposed to the asynchronous connections originally used) as we can see in Section 4.3.3 and further.

In order to test the performance of the system we utilised the original microbenchmarks benchmark which relied on synchronous requests with the same setup as described in Section 5.2.

In Figure 5.5, we can see the number of operations per second of the system averaging at 28728 ± 2880 operations per second. The 95% confidence interval is $28672 - 28735$ operations per second.

The performance is pretty similar compared to the previously demonstrated scenarios, maintaining a pretty stable amount of requests executed (as demonstrated by the 95% confidence interval). The spiking nature of the graph is natural due to the way this test works. As we have seen, the test is synchronous so the clients have to wait for a response before sending their next response. This leads directly to the behaviour we are seeing in this test, which is that the performance follows a high and low pattern where a low point in the operations per second means that a lot of clients were still waiting for their response and when those clients get awoken to receive the response, they will wake up and all send their requests, leading to another peak. But now all those clients are sleeping and waiting for a response to their request, so we get a drop in performance. But then this drop in performance gives an opportunity for those sleeping clients to wake up and send their request and we go on and on like this forever.

However, we developed the client pooling design and implemented with the purpose of improving the replicas' handling of client requests to promote the creation of larger batches of requests, which as we have seen are key to the general scalability of the overall system.

Figure 5.6 shows that the batching being done is actually not that high when taking into account the amount of operations per second that the system is processing which tells us that our

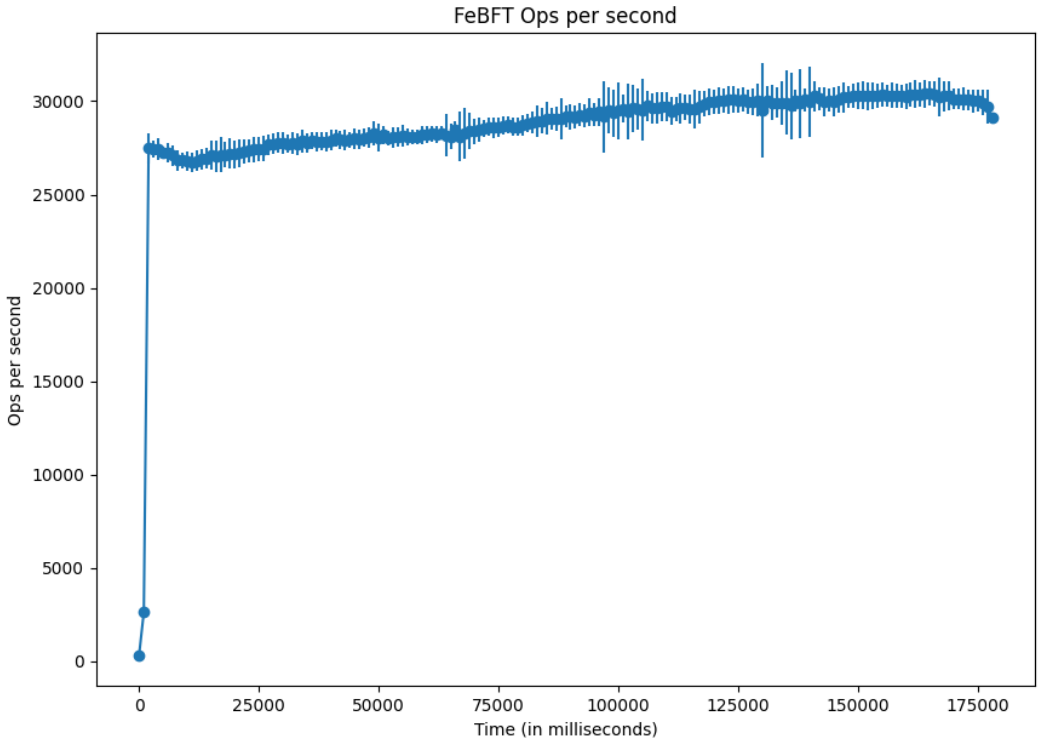


Figure 5.5: The operations per second of FeBFT after introducing client pools

change to synchronous channels, along with the redesign of the client requests and replica request handling (which was initially done all in the same place) reduced the latency of the system quite significantly and the system was now able to process more consensus instances. However this wound up being a side effect of separating these elements and reducing the contention along with compartmentalising the work that was being done (the creation and proposition of batches having it's own dedicated thread instead of being shared with the main consensus thread) and was not really our intended target as we have seen in Section 4.3 we do not want to optimise for lowest possible latency (however if we can get it it is always a good bonus, which we have). Notably though it showed us something more: there was a bottleneck somewhere in the system. We noticed that the CPU usage in the clients was not very high (maxing out at around 50% average utilisation with almost no hotspots that could be indicative of a significant bottleneck) and the usage in the replicas was even lower (with around 20% CPU utilisation) which would indicate that there is still a lot of performance left on the table.

To investigate this issue further, we re ran this same test with 2000 clients and with 750 clients with no other further changes to the testing setup. We discovered that all the tests had near identical performance where we should have seen an increase of 2x and .75x respectively if the client testing setup was scaling as it was supposed to do. Another even weirder aspect was the the CPU utilisation also remained almost the same for all 3 tests which should make no sense since we were running wildly different client counts. The expected behaviour would

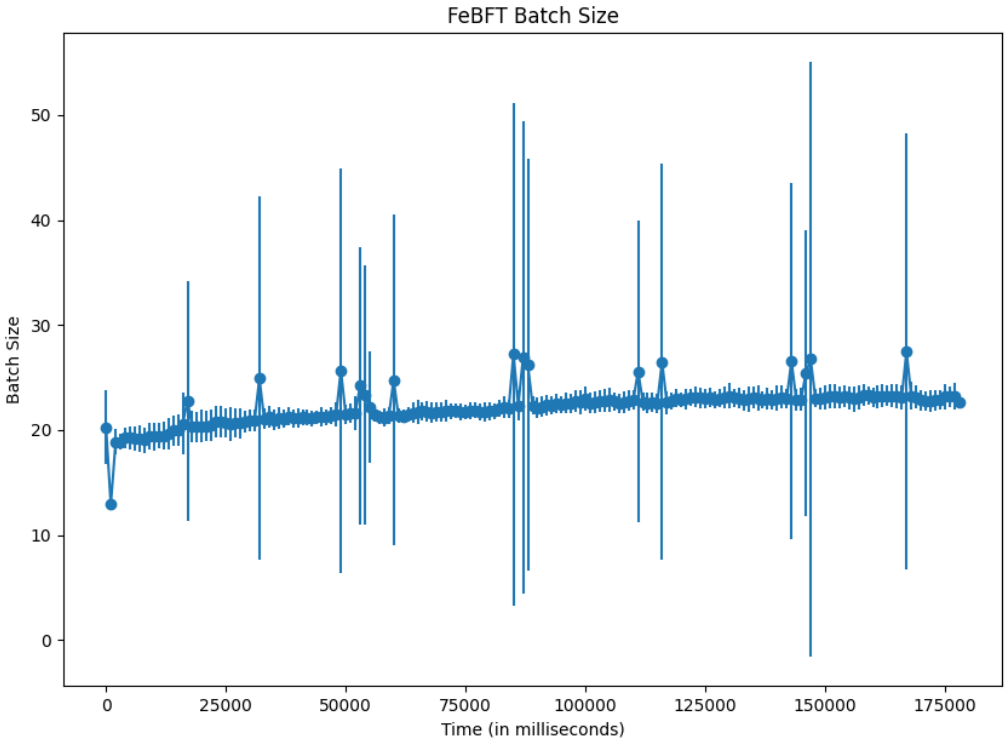


Figure 5.6: The batch size of FeBFT with the client pool design

have been for the CPU usage to max out and from then on not produce any more requests per second. What we were seeing is that even with the CPU under utilised, we were not being able to produce any more requests. From here we started looking at how the actual clients were engineered and how that could cause issues when running many of them in a single box. We observed that they utilised 1 asynchronous task per connection to each of the replicas along with also spawning a task on every client message that was being sent to the replicas, another task for processing the messages that were received from the replicas and another task to actually produce and order the execution of the requests. Taking into account clients each connect to 4 replicas, we clearly saw the possible issues with the approach. The overuse and over reliance on the asynchronous tasks meant that the CPU was spending almost half of its processing time on switching between tasks to process.

We decided to better investigate the asynchronous runtime, how it operates and what we could do to fix this issue in Section 4.4.

5.4.2 Performance with various network multiplexing implementations

As we have discussed in Section 4.5 the results of the tests we performed led us to believe that the asynchronous runtime was a large contributor to the issues. As such we proceeded to remove

the usage of the asynchronous tasks in FeBFT. To do this, as described previously we re designed FeBFT to perform a more incisive separation of responsibilities, assigned each its own dedicated OS thread and created a better separated pipeline for requests following the message passing approach Rust suggests for multi threaded workloads, which we have described in Section 4.3.3 and onwards.

So our logical next step was to also rework how the connections are handled by FeBFT. Like we saw in Section 4.5.1, originally FeBFT created a task for every outgoing message, along with a task to handle each incoming connection meaning it scaled very poorly when we increased the client count meaning we can have bottlenecks preventing us from testing our performance any further. Similarly, we then went on to describe and discuss other methods that were developed to handle this exact situation along with some of their potential issues and their cause. We developed two additional methods of multiplexing the connections while also describing the architecture of a third one that we did not have the time to implement.

As we have seen, from the previous Section 5.4.1 round off to about 30000 requests per second in the synchronous tests. So our target was to beat that number of operations per second while maintaining the same number of clients and the same testing structure and idea.

We also made some other changes compared to the previous iterations of this program, particularly in the way we handled the forming of batches in the Proposer. In the initial tests we just allowed the service to run without any control of any kind. That meant that as soon as the consensus was ready for a new batch, the proposer would propose it no matter what size it currently was. This lead to the boost in consensus decisions per second as we saw but, like we also described then, this was not our goal. Our goal was to make the scaling point the batches that were being generated, not how many instances of the consensus we could run in a given amount of time because as we have described in Section 4.3, when we want to geographically distribute this system, the batching potential will be the determining factor to the ability of the system to scale performance. Therefore we introduced a minimum batch size in combination with a timeout of course, to prevent situations where it's endlessly waiting for more requests to fill the batch. We tweaked the values of these parameters until we found a good balance, which turned out to be a batch size of around 500 with a timeout of around 5000μs. This lead to a reduced amount of consensus instances per second but lead to an overall good increase in performance when combined with the other changes we have detailed for this test, as we will now see.

We will start by detailing the performance of the multiplexing method described in Section 4.5.3 while utilising normal OS threads and TCP sockets, we will see the performance results of using this same architecture but with the asynchronous environment later on.

5.4.2.1 Synchronous Microbenchmark results

We first have to detail some changes to the way the data collection was performed. In the previous tests, we saw that batch size never really passed 40 operations per batch and overall

performance tended to be around 30000 operations per second. In these tests, we were taking these measurements on every 2000 operations done which worked out just fine since it took several rounds of consensus to actually achieve this number, which allowed us to actually measure how long it took to perform those 2000 operations. When we started investigating with these new multiplexing methods however we started to run into some issues with this approach. The main issue is that we were getting batches of requests that were so large that we started getting some issues. These issues existed because if a batch was larger than the operations per second spacing that we were using, it would register that it did all those operations in a couple of microseconds (it did not take into account the time it took to actually achieve the consensus) which would give us some ridiculously high numbers that were totally ruining the statistics of our tests. Therefore we had to extend the amount of operations between each measurement quite significantly, which lead to less measurements and a greater variation between each of them which is clearly visible in the graph compared to the previous tests. Another very important factor to this jittering is the batch size which we analyse how it affects it further ahead when we are analysing the batch size results.

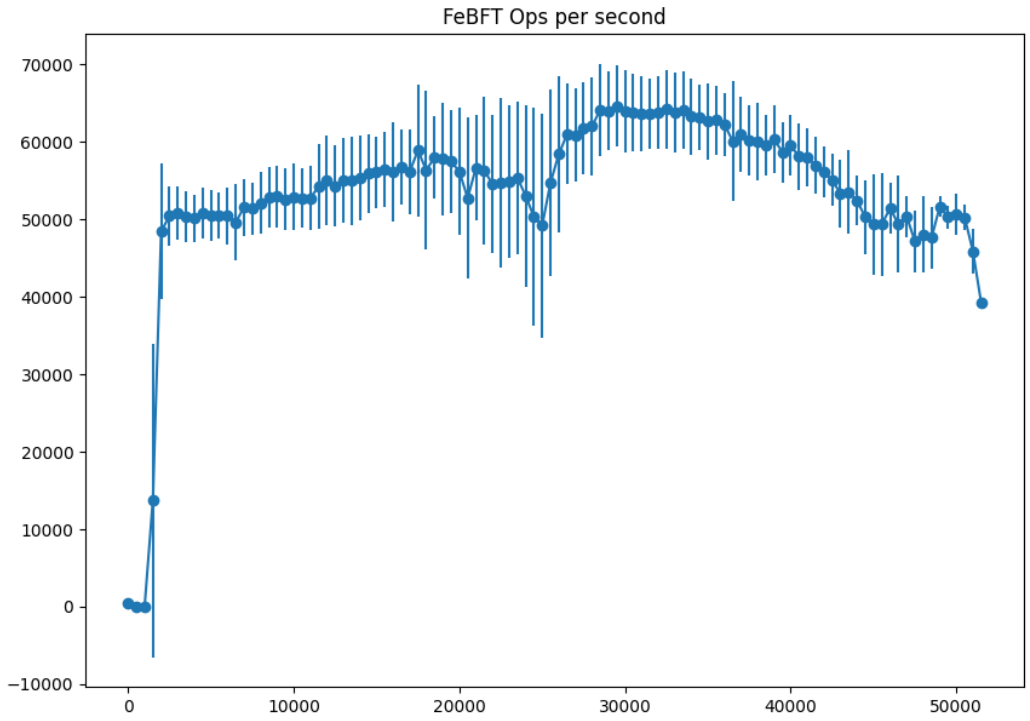


Figure 5.7: The operations per second of FeBFT when utilizing per peer sending

In Figure 5.7 we can observe that the performance is averaging 55277 ± 11894 operations per second. However, we can see that these measurements have a very high standard deviation, which means that the average is not that accurate. The 95% confidence interval is $54951 - 55603$ operations per second. As we have just explained, the very large variance in measurements is

due to the compound effect of the nature of this test (as we have seen in the previous section it's normal for this test to have performance dips and highs such as this) combined with the larger measurement interval we implemented, meaning these two things add up to one another to give us the performance we are witnessing.

Analysing this data we can clearly that the performance is far higher than what we were previously seeing meaning we were indeed correct when assuming that the networking (particularly at the client side) was causing a very large bottleneck that was keeping our performance down. By re engineering the handling of messages to the design explained and justified in Section 4.5.3 (particularly the handling of sending messages, which previously scaled the amount of created Tasks directly with the amount of messages sent and was therefore very bad for our scaling necessities) we were able to achieve much higher performance. However, we still had to see if this improvement was due to latency advancements or if we actually found a way to properly aggregate requests, leading to large (and therefore scalable) batches.

For this, we must analyse the average batch size which can be found in Figure 5.8.

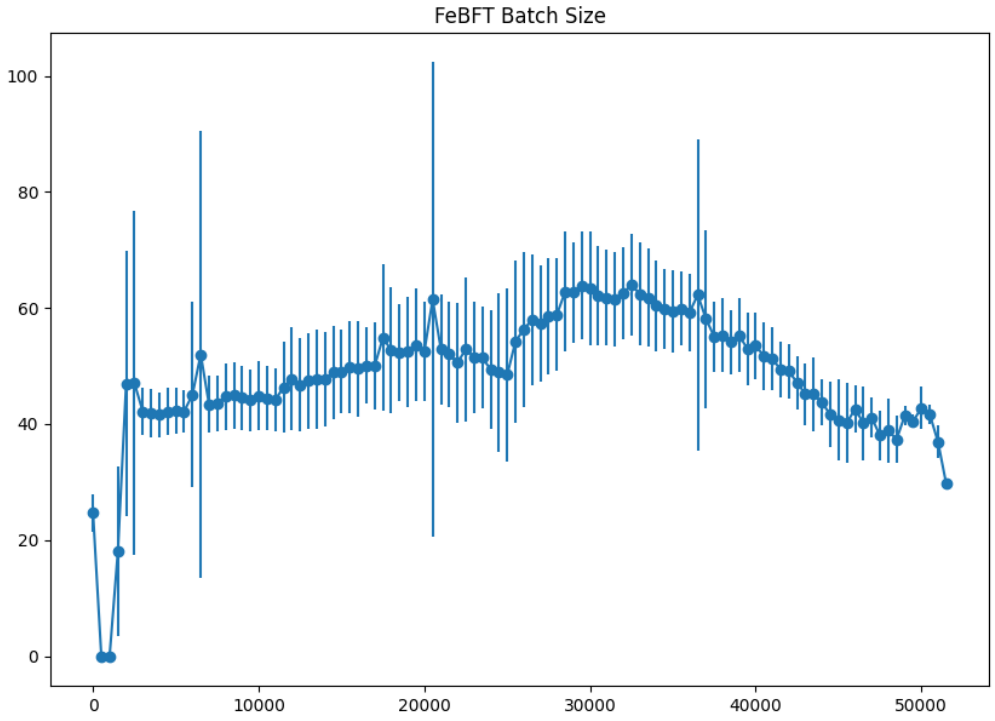


Figure 5.8: The average batch size of FeBFT when utilising per peer sending

The average batch size we observed here was 50 ± 15 requests per batch. Comparing the results of Figure 5.8 with the results found in previous tests, like in Figure 5.6 we see that the batching remained pretty consistent with the previously shown tests, raising slightly along with the overall system performance. We believe this is purely due to lack of client performance, as

running that many clients is exerting a huge load on the client machines and we are wasting a lot of very important CPU time context switching between all of them. We believe this is the case because as we will see in the following asynchronous tests the batching of requests is not the problem, as it was very effective.

5.4.2.2 Asynchronous Microbenchmark results

After seeing the very encouraging results for the synchronous benchmarks, we decided to see if this was the highest possible performance we could obtain from this system. To do this, as we have described previously we developed a new benchmark that relied on having a lot less clients than the regular Microbenchmarks test but each of the clients performed many concurrent requests. As we have seen, having an excessive amount of threads allocated at the same time causes a lot of performance issues with in the underlying computer since the OS cannot possibly schedule that many tasks in such few cores. If that weren't enough, the cost of constantly changing the thread that is being executed is extremely high (both in terms of the time taken to context switch and in the loss of locality caused by switching a threads' executor) which causes performance limitations and under utilisation of the underlying system.

In this test we had the same issue with the measurement intervals as we had in the previous test so we again had to increase it in order to get real numbers instead of just infinities because of divisions by zero. This again means that we will have less measurement points which in this case did not cause jitters due to the actual nature of the test which no longer relies on clients having to receive a response to their requests in order to proceed. Instead clients are allowed to have many concurrent requests awaiting a response meaning they can keep a much more constant, stable load on the system which should allow us to get much more fine measurements and sustained performance results.

This test was ran with 100 clients each executing a maximum of 400 concurrent requests.

In Figure 5.9 we can see all these changes definitely paid off along with confirm our suspicions that the bottleneck to the performance of the system was in the clients and not the replicas since this test changed nothing about the replica's approach to the reception and handling of requests and yet we saw a near 100% improvement on general performance. The average performance of the system in this test is of 136091 ± 29236 operations per second. The 95% confidence interval is $134607 - 137576$ operations per second.

We can also see that the jittery behaviour has disappeared from the test, due to the reasons we have already mentioned previously.

After having seen that the overall performance of the system we were half expecting the batch size to have only gone up by around 142%, with the number of consensus instances maintaining stable effectively keeping in line with the observed performance. However we could not be further from the truth, as Figure 5.10 shows quite clearly. We can see that we had an average batch size of 18430 ± 2119 requests per batch which is an increase of nearly 300x! This is exactly the

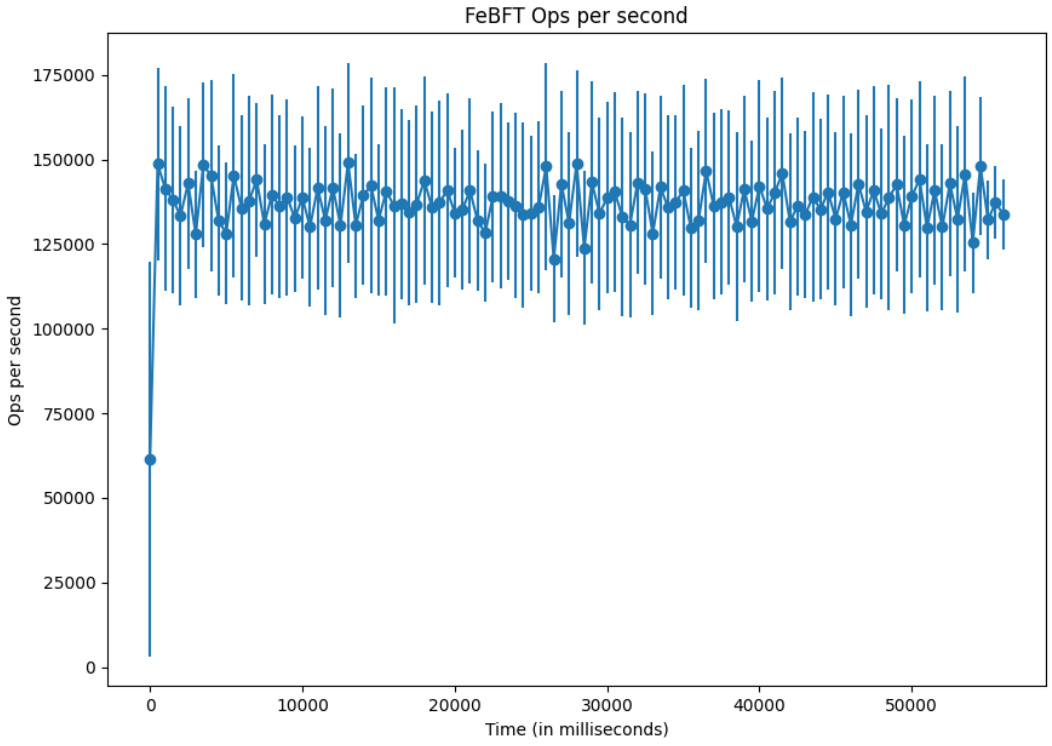


Figure 5.9: The operations per second of FeBFT when utilising per peer sending

behaviour that we had intended for this system, with the batch size increasing and the amount of consensus instances not having to be as high. We obtain this phenomenon without having to alter any of the timeout and minimum batch sizes that were previously mentioned because of a natural phenomenon with the network speed. To understand it we have to look at the testing setup and also at how the protocol works:

- We know that the machines are all connected through a 1Gbe LAN connection.
- We also know the leader has to broadcast the Pre-Prepare message (which contains all the requests of the current batch) to all other replicas of the quorum. This of course takes time, dependent on the network speed available between the replicas.
- We know that our Proposer is capable of working while the consensus thread is also waiting for messages or performing work, meaning it can keep collecting and aggregating requests during all this time.

Taking all of this into account we see that in this case the system comes to a graceful balance, where the speed of the network means that sending the batch containing n requests takes time, which gives this time to the Proposer to prepare a larger batch for when the system is ready to receive it. This is what we believe happened for the batch size to grow to 300x while the

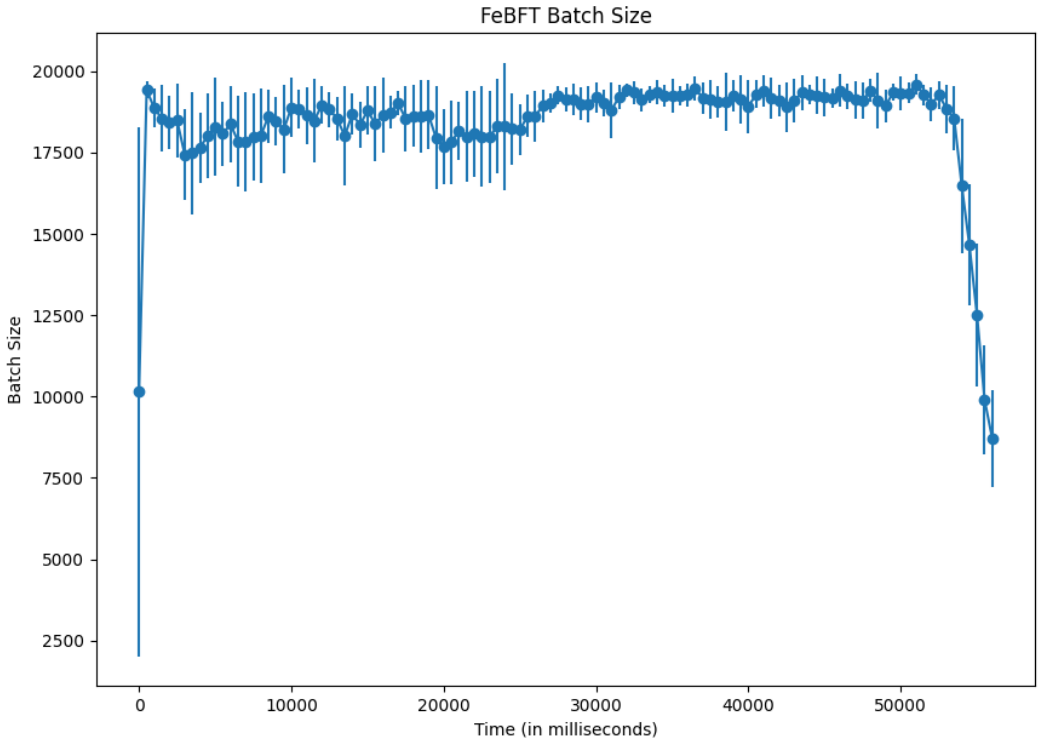


Figure 5.10: The average batch size of FeBFT when utilising per peer threads

overall operations only grew by around 142%. This is completely intended and follows the exact behaviour we desired in order to assert scalability in any and all situations this middleware might be used as described in Section 4.3. If we were to increase the network speed without altering the amount of requests the clients are producing, the expected response of the system would be to reduce the batch size since the network no longer takes as much time to send it which would lead to more consensus instances but overall the same performance which means we would effectively get reduced latency. This effectively means the system depends on the **network speed** instead of the **latency**, which is perfect because we can feasibly scale the speed of the connection between two places but we cannot reduce the amount of time that it takes to get for information to travel across space-time.

After analysing the results of this test we again noticed (similarly to previous tests) that the CPU usage of the clients was still not very high, which meant we were still not completely utilising the performance potential of the client machine. To confirm this, we again reran the test with more clients (175 in this case, instead of the 100 previously) and with less clients (75 instead of the 100) and we found a similar result to the results presented. The effective performance of the system was not changing (in the case of the 175 clients performance barely increased at all, while when we lowered the client count it decreased but not proportionally to the amount of clients we removed) which meant to us that the bottleneck described in the previous tests was not yet solved and subsequently our performance numbers were still being effectively hampered

by the clients' ability to produce enough requests to saturate the replicas.

5.4.2.3 Testing with Asynchronous runtime

After having tested the new connection multiplexing module of FeBFT we observed that the performance had improved significantly with the changes done. As we described in Section 4.5.3, we altered the design of network connections in order to make the amount of tasks created scale with the amount of connected peers instead of scaling with the amount of requests made (which was causing lots of issues with performance). As we saw in that Section, the architecture is agnostic to what types of execution units and socket APIs we utilise. In the previous test we utilised OS threads and standard library sockets for communication and since we saw such great performance gains, we decided to also run this experiment with the Tokio sockets and Tasks to see if the issue was solely due to the architecture of the program or if the usage of the Asynchronous runtimes and asynchronous sockets also played a role in the poor performance we were seeing previously.

Our goal is to isolate the performance of just the asynchronous runtime in order to determine if it is indeed a source of issues with the overall performance of the system. Because of this, we chose the asynchronous test since it showed to be the most capable of exerting a stable, consistent stress on the system at the highest rates of operations per second. We maintained all the other parts of the test exactly the same including the amount of clients and concurrent requests.

As we can observe in Figure 5.11, the performance of the system while utilising the asynchronous runtime libraries for Sockets and Tasks is on average 42565 ± 33374 operations per second. The 95% confidence interval is $41203 - 43927$ operations per second.

As we can see, the overall performance of the system has taken quite a hit by exchanging the regular standard library sockets and threads with asynchronous runtime Tasks and sockets. Comparing the two averages 42565 vs 136091 operations per second, we can see that there was a 219.7% drop in performance. This test allowed to confirm the suspicions that we had throughout the implementation of this system, as described in Section 4.4. We believe this to be because of the asynchronous runtime's inability to utilise active waiting even when it is the best option at that time instead heavily favouring the passive waiting approach, as whenever tasks reach an await they will go into sleep even if the next message was just a fraction of a millisecond away. This means that these tasks will then need to be context switched out and back into execution, meaning we have wasted a lot of CPU cycles not performing work. This is particularly noticeable in this test because of the way the requests are produced and then delivered. As we have described earlier, this test consists of having a relatively small amount of clients performing many concurrent requests. Like we also said, each of the clients has a single thread whose purpose is to produce requests and then pass them to the FeBFT client so they can then be broadcast to the replicas (this step is where our network multiplexing comes in). As we have seen, this architecture consists of having one execution unit per each connection (outgoing and incoming). In the case of the outgoing messages, each of these execution units has a queue meant to hold

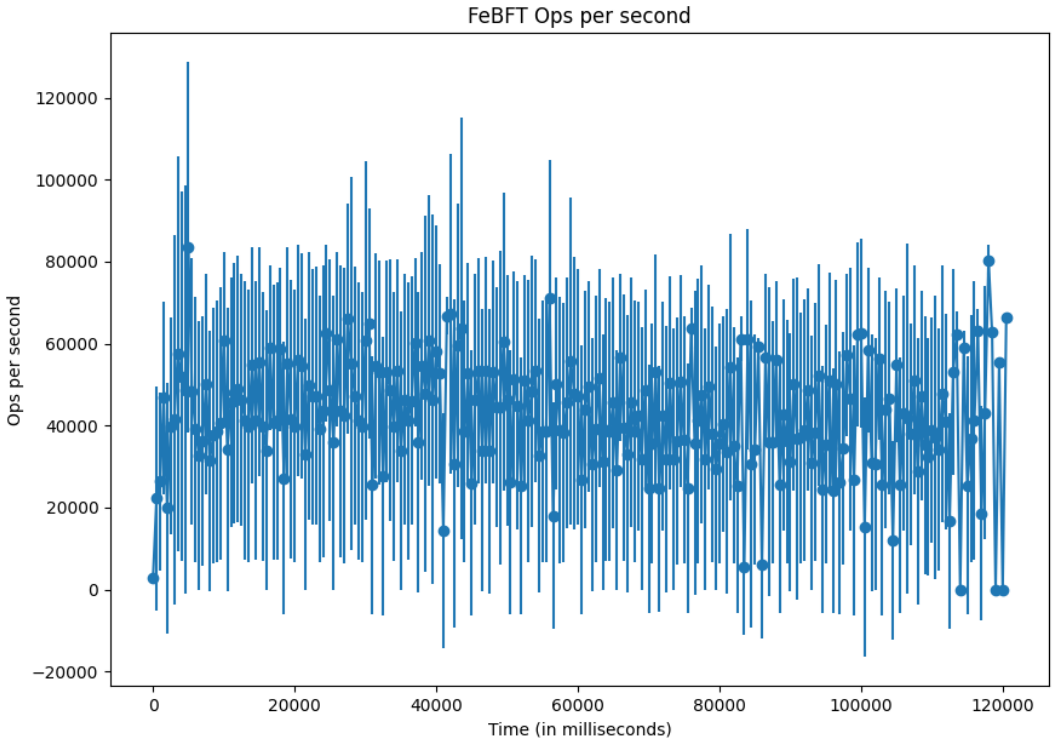


Figure 5.11: The operations per second of FeBFT with the new architecture and while using the asynchronous runtime

messages that are through the connection handled by the respective execution unit.

Another possible issue is that, as described in Section 4.4, the asynchronous runtime is meant for small, IO bound tasks that will execute for minuscule amounts of time. Therefore, using this library in combination with this test will highlight the worst possible performance, since this tests focuses on few clients with very high utilization in each of them.

Now, we have already mentioned that each client has a thread responsible for producing the actual requests. Given the way the OS handles threads, it's very likely that this thread runs until there are no more requests to produce (already occupied the concurrent request quota). This means that there will be many requests made in quick succession, which will then all be delivered to the networking layer of the protocol. If this layer is utilising regular OS threads, then their ability to wait actively means that these network execution units will keep being executed until they have delivered all of their messages. While if we are utilising asynchronous sockets and queues, it's very likely that the tasks will go into sleep immediately instead of waiting for the upcoming requests, meaning we waste a lot of performance as we have already seen.

The batch size averages out at around 2219 ± 2067 requests per batch.

This is inline with the performance drop of 219.7% that we observed with the operations

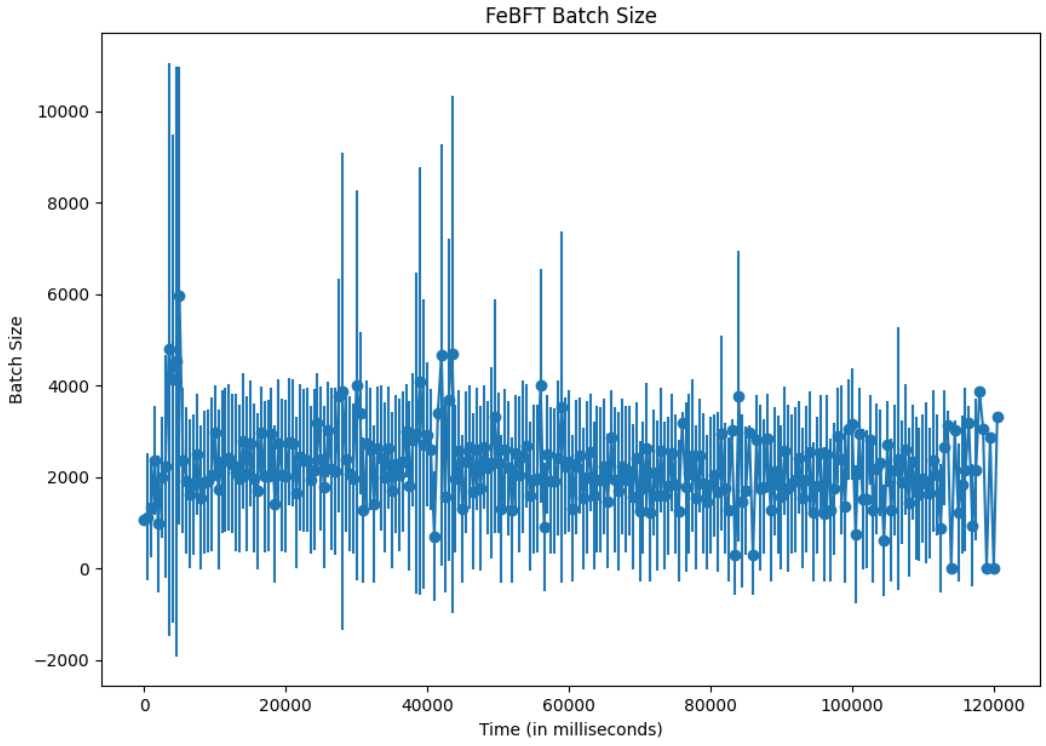


Figure 5.12: The average batch size of FeBFT with the new architecture and while using the asynchronous runtime

per second. This can be easily explained by the clients simply not proposing enough requests because we have actually seen this same architecture produce larger batches (and thusly more operations per second). This came as no surprise and further backed the idea that we haven't actually been able to test the effective peak performance of the replica architecture because we do not have enough computing power to power the clients and produce enough requests to fully load the replicas.

5.4.3 BFT-SMaRt performance

In order to provide a point of comparison, we also implemented the tests described in Section 5.4 in Java, utilising BFT-SMaRt as the BFT SMR middleware.

The initial microbenchmarks that we utilised to benchmark FeBFT (in Section 5.2) was actually based on the microbenchmarks found in BFT-SMaRt. Therefore it was quite simple to create a very similar test, with just some changes in the way we present the information to provide some extra information like timestamps, ID of the node, etc. This test too had a large number of synchronous clients that waited for the response to the current request before issuing a new one.

Our asynchronous test required some additional implementation details on our end. Namely, we moved from using the synchronous API to the callback based asynchronous one which was also provided by BFT-SMaRt. Since we also utilised callbacks in the FeBFT test, this made it quite easy to translate what we had implemented in Rust into Java.

We utilised the exact same 5 identically specced machines, connected via a 1Gbps network. The machines as stated had the same configuration as the ones described in Section 5.2. The Java version used was Java 11 (openjdk 11.0.17 2022-10-18) on all machines.

As we have seen in Section 4.5 we believed the multiplexing and handling of the network connections was severely bottlenecking our system. This was later confirmed when we performed our performance tests in Chapter 5. BFT-SMaRt utilizes Netty for its network management which provides it with great performance and scalability and effectively solves this problem for them. Since in Rust we have no such alternative, it would be natural that BFT-SMaRt had better performance since it is more capable of utilizing the performance available to it.

5.4.3.1 Performance in the synchronous test

Again, similarly to the previous tests we performed on FeBFT, this test also featured 1000 synchronous clients, all attempting to propose requests to the replica. The batch size used for this test was 500.

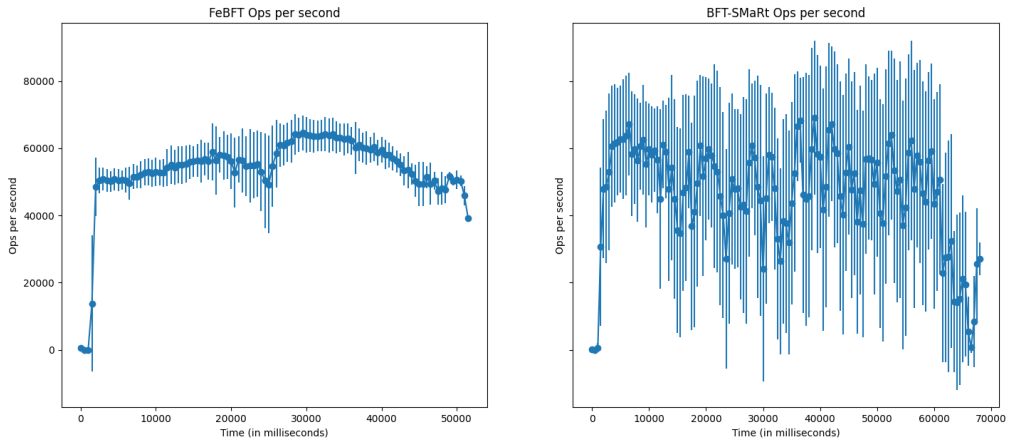


Figure 5.13: Operations per second of BFT-SMaRt in the synchronous test, side to side with FeBFT's ops/s

The average operations per second of this implementation is of around 49219±30523 operations per second. Similarly to FeBFT's results, we also have a very high standard deviation because of the nature of the test as we have described in the performance benchmarks done on FeBFT in Section 5.4. The 95% confidence interval is $48466 - 49973$ operations per second.

Comparing these results with the results we obtained from FeBFT in Section 5.4.2 we can see

that FeBFT has quite better performance in this test. In fact, comparing the 55277 operations per second achieved by FeBFT we can see that it is a 12.3% improvement on the BFT-SMaRt results we obtained.

Another very interesting thing we can see in Figure 5.13 comparing to Figure 5.7 (FeBFT's results in this test) is that in BFT-SMaRt we have much lower lows of operations per second comparing to FeBFT. Like we explained before, this test is prone to this kind of behaviour but not really to the level that we are seeing here. We believe this is not due to a design fault or implementation error, but the pure choice of language itself. BFT-SMaRt is implemented completely in Java which means that it is going to have a garbage collector running in order to keep the memory usage under control and clean up all the unused memory. We believe that this is causing the protocol to lose a lot of performance, since the GC has to completely halt every thread running in the JVM in order to build the reference tree and determine which objects are no longer being utilised.

This is further verified by the batch size numbers, viewable in Figure 5.14 which we can observe to never actually drop that much, especially comparing them to the operations per second graph. This indicates that the issue with the performance drops is actually that the SMR is not processing any consensus instances for a while due to garbage collection, not that it is producing smaller batches.

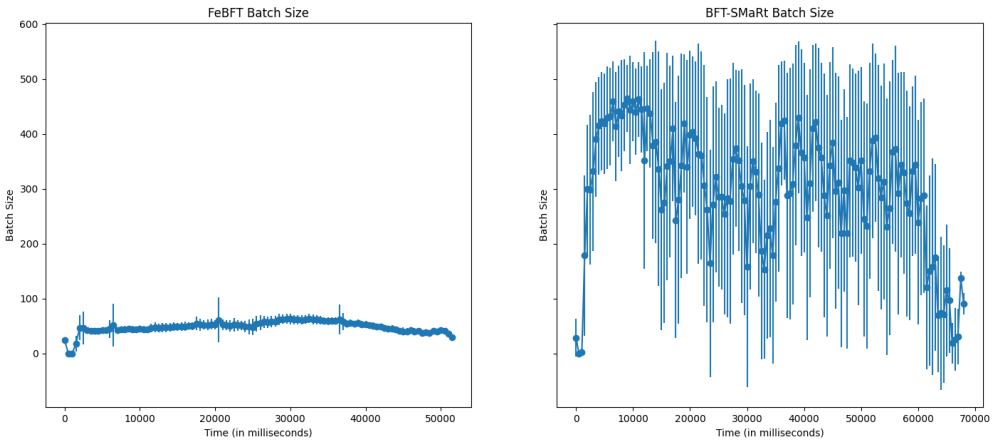


Figure 5.14: Average batch size of BFT-SMaRt in the synchronous test, side to side with FeBFT's batch size

Another thing Figure 5.14 shows us is the artificial limit on the batch size imposed by BFT-SMaRt since no produced batch goes over the batch size of 500 requests per batch.

This is due to the way the proposer in BFT-SMaRt works. Like we described in Section 2.2.1, the batching in BFT-SMaRt relies on having a limited batch size in order to prevent Denial of Service attacks. Like we have also spoken in 4.3, the ability to produce batches efficiently and the ability to scale the number of operation per batch depending on the current load of the system is extremely important in order to keep latency low when there is low system usage while

also being able to increase the amount of operations per second (even if it means sacrificing a bit of latency). This mandatory restriction harms this ability, which means that in our opinion, BFT-SMaRt is not as scalable as FeBFT.

This is because as we explained in Section 4.3.3, FeBFT does not need to impose a ceiling on the amount of requests that will be part of the batch, instead it has the ability to naturally scale the performance depending on the current demand on the system.

5.4.3.2 Performance in the asynchronous test

In order to test the full performance potential of BFT-SMaRt in order to compare it to FeBFT, we also wrote a version of the asynchronous benchmark for BFT-SMaRt as we have previously said. For these tests we used a batch size of 10000. Initially, we wanted to test the performance of both FeBFT and BFT-SMaRt with 150000 operations per client (so $150000 * 100$ operations in total) but we found that BFT-SMaRt was not capable of withstanding so many operations and wound up crashing and failing to complete the test. Therefore, we had to lower the operation count to half of that (75000 requests per client) in order to obtain valid data from BFT-SMaRt.

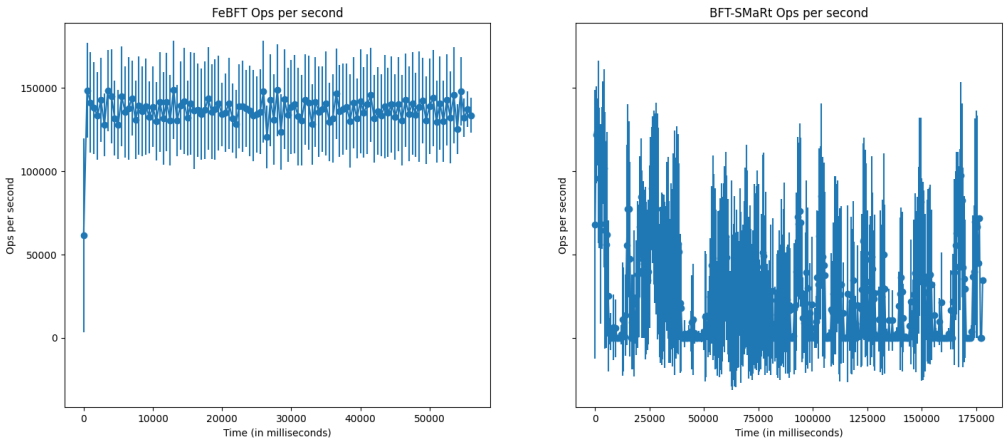


Figure 5.15: Operations per second of BFT-SMaRt in the asynchronous test, side to side with FeBFT's ops/s

The average operations per second is of 30537 ± 49653 ops/s. However, as we can see from Figure 5.15, there are many peaks and drops comparatively to the results obtained by FeBFT on the same test. Comparing this to the results of FeBFT 134091 ops/s, we see an decrease of 339.11%. As we saw in the previous test, which was prone to jittery behaviour, we believe that BFT-SMart's choice of language could have a role to play in the large jittering of operations we were seeing (which was quite larger than what we saw in FeBFT).

Taking a closer look at the presented graphs in Figure 5.15 and Figure 5.16, we can clearly see very large sections of time where BFT-SMaRt has close to or even 0 operations per second performed. These gaps are due to Java's garbage collector swooping in and stopping all threads

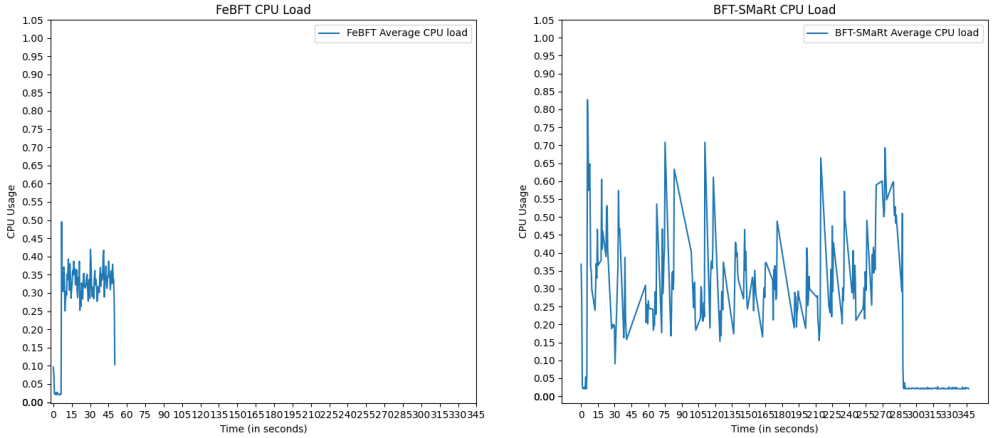


Figure 5.16: The CPU usage in the client machine in the asynchronous benchmark for BFT-SMaRt, side to side with FeBFT's CPU utilization in the same machine

(including the measurement thread of CPU usage, which causes the gaps in the CPU graph) in order to collect the garbage. The way the operations per second are measured actually helps hide this fact and actually helps BFT-SMaRt hide this lack of performance in the ops/s averages reported as described in Section 5.1. In reality, the system frequently halts completely and goes for very large stretches of time without executing a single request (we can see an example of this from second 30 to around second 50, where the system did not process a single request in that 20 second interval). If we took just the averages of the measurements (without our changes to rectify these errors), the average operations per second would be 97462 which is more than three fold our measurements.

These observations are also very much supported by the sheer amount of time the test took to run. In FeBFT we can see the test completes in around 60 seconds (100 clients performing 75000 operations each) and has an average operations per second of 136000. In fact, taking these numbers we can see that: $136000 * 60 \simeq 75000 * 100$ meaning the time taken to perform the operations is in line with the time it would take the replicas to perform that many operations at 136000 ops/s average. Take this math to the BFT-SMaRt side of the test and this becomes very unclear. The test takes around 185 seconds. Taking this time and using the average we calculated just from the measurements (without our adjustments), we can see that $97462 * 185 \neq 75000 * 100$, not even close to it. We can also see that the operations per second measurements with our alterations fall short of the expected number but comparing to the 97462 ops/s, it is more than an order of magnitude closer to the expected number of operations done.

As is visible in Figure 5.17, the RAM usage of FeBFT is quite significant but also quite more stable than the RAM usage of BFT-SMaRt. These high readings can be explained by the fact that in FeBFT we are collecting the RAM usage of the entire system while in BFT-SMaRt we are only collecting the RAM usage of the JVM itself. To make things even worse, the fact we are using at least two threads per TCP connection (incoming and outgoing) means that we are

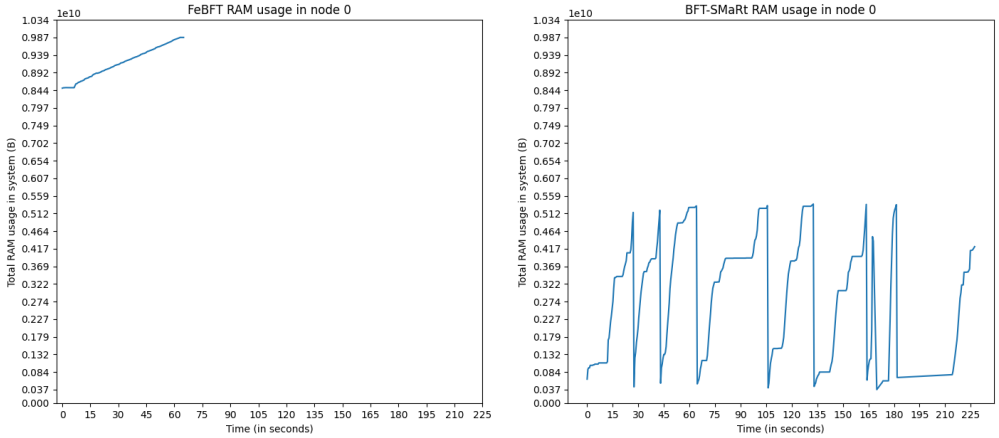


Figure 5.17: The RAM usage in the leader machine in the asynchronous benchmark for BFT-SMaRt, side to side with FeBFT's RAM utilization in the same machine

going to utilize a very high amount of RAM just on that alone. This would be fixed by our ideal network multiplexer, described in Section 4.5.

Therefore, in these graphs it is more important to analyse the evolution of the RAM usage than the actual RAM usage itself. When analysing that aspect, we can see that BFT-SMaRt is wildly variant while FeBFT maintains a steady rise in usage, consistent with the storing of the processed requests in the message log. The variance of BFT-SMaRt can be explained quite simply by the nature of the language chosen to implement it. Java, as we have seen, relies on a Garbage Collector for unused memory management, so what we see is that the RAM starts rising uncontrollably (with the "trash" of requests that have been processed but the objects relating to that have not yet been collected) until the garbage collector steps in and clears all of that memory. We can see that the actual RAM usage is not very high when we exclude the garbage that is yet to be cleaned. The fact that we are using a lot of memory (natural due to the nature of the test) means that these garbage collection stops are going to be enormous (as we saw one taking 20 seconds in Figure 5.15) and therefore lead to complete stops for long periods of time which is inadmissible for a system that is meant to be highly available and performant.

The batch size graph, visible in Figure 5.18 displays an image that is consistent with what we have seen in the operations per second graph, viewable in Figure 5.15 which has many spikes and drops in performance. This batch size is reflective of the lack of requests that were being received by the replicas, caused by the performance issues with the clients' garbage collector which we discussed earlier.

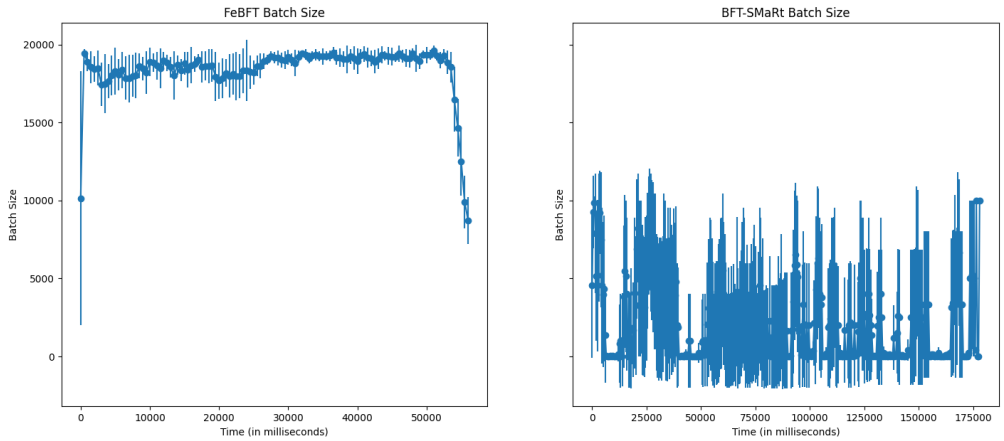


Figure 5.18: Average batch size of BFT-SMaRt in the asynchronous test, side to side with FeBFT's batch size

Chapter 6

Improving Ceph's Resiliency

6.1 Ceph Overview

Ceph [47] is a distributed file system that is focused on delivering excellent performance, reliability and scalability. To achieve these goals, Ceph maximizes the separation between data and metadata management by replacing regular allocation tables with a pseudo-random data distribution function named CRUSH [48] which is designed for heterogeneous and dynamic clusters of unreliable storage devices along with dynamically distributed metadata management along with employing RADOS [49] to automatically manage data storage and replication.

Due to this extra level of separation, this system is composed of more components than what were used to seeing in the previously discussed storage options. Instead of the more common 2 types of members in the system (client and server), Ceph has 5 types of members:

1. Object Storage Devices (OSD).
2. Metadata Servers (MDS).
3. Ceph Monitors.
4. Ceph Managers.
5. Clients.

The OSDs are responsible for storing all of the raw block data (they also store the metadata, but we will see why and how later). The Metadata Servers are responsible for managing all metadata and processing all of the operations related to the metadata (for example, *ls*, *stat*, etc). They will then store the raw metadata data in the OSD cluster as if it was regular block data. The Monitors are responsible for one of the most important parts of this whole system. As we have just seen, Ceph employs CRUSH to calculate where a given block of data is going to be located in the system. However, this algorithm requires knowledge of the current components

that are a part of the system (in particular the OSD map, which contains all currently available OSDs). Monitors are responsible for keeping this map available and up to date for all clients to access. The Managers are responsible for keeping track of statistics, logging and other management-related responsibilities.

The clients are used to actually access the data stored on the system. They interact with almost all other components of the system, namely:

- Communicate with Monitors to obtain the current state of the system and up-to-date maps on which MDSs, OSDs, etc are currently online and operational.
- Communicate with metadata servers to perform metadata operations.
- Communicate with OSDs to perform data lookup and storage.

We will now see a more detailed explanation of each of these components and how they operate.

6.1.1 Object Storage Devices

An OSD is a simple storage device with a CPU, a network interface card, a local cache and an underlying disk or RAID setup. They replace the traditional block-level interface with one that can read and write to much larger named objects because of each OSD's individual capabilities to distribute low-level block allocations to the storage devices themselves, while also not requiring a lot of work by the client as he does not have to calculate these allocations. Ceph utilizes the OSDs abilities to distribute the complexity of data accessing, update serialization, replication and reliability, failure detection and recovery. This means that OSDs have to possess the capability of executing code, meaning that not all cloud service providers can be simply and directly integrated into the cluster.

Ceph must manage all of the data across a constantly evolving cluster of OSDs in a way that maximizes the utilization of available bandwidth and device storage resources. To avoid imbalance (where recently joining nodes are mostly idle and empty) load asymmetries as well as maintain availability and integrity of the stored data Ceph deploys two systems: CRUSH and RADOS.

CRUSH [48] (Controlled Replication Under Scalable Hashing) handles the mapping of the objects into *placement groups* (PG) using a simple hash function with a bit mask to control the number of groups in the network. It then assigns each *placement group* into an ordered list of OSDs which will store replicas of the data using a pseudo-random data distribution function that can be recalculated by any client, OSD or metadata server using just the *placement group* and a short description of the OSD cluster meaning this operation is completely distributed and effectively solves the two problems of where should the data be stored and where is the data located. The amount of replication and even the quantity of data that is sent to each individual

OSD will be dependent on *placement rules*. By design, small changes in the topology of the OSD cluster have very little effect on the PG mappings in order to minimize data migration due to device failures or cluster expansion. However, there may be large topology changes via large expansions or failures. To handle these scenarios, Ceph utilizes RADOS.

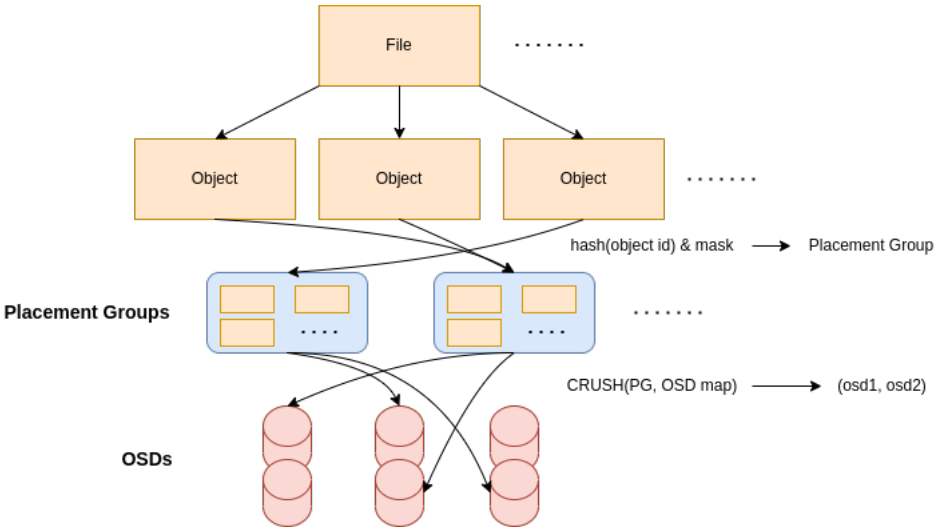


Figure 6.1: Illustration of the steps taken by a client to know which OSDs to contact, using CRUSH after having obtained the OSD map from the Monitors.

Figure 6.1 details the steps taken by a client when he wants to obtain a given file after he has already obtained the current up-to-date maps from the monitors. Firstly clients calculate the placement group of the object they want to store/retrieve. We will then use the CRUSH algorithm to calculate the OSDs corresponding to that placement group. When the client wants to retrieve the object, he will communicate with the first OSD in the PG. If that OSD does not respond (crashed), he will move on to the second one and so forth. When the client wants to store an object, he will only contact the one OSD (in case the first one is offline, the second one and so on). The OSDs are then responsible for the replication of the data, as we will see in the RADOS section. This means that this protocol is not tolerant of byzantine faults, since clients will always only contact a single OSD so if that OSD is faulty, the client will never know.

RADOS [49] manages its own replication of data using a derivation of primary-copy replication and also disassociates synchronization from safety when acknowledging updates so we can have both low-latency updates for synchronous applications as well as well-defined data safety. This means that after the client has the correct *placement group* for the data and has calculated the OSD list, he will send the file to the primary OSD (the first non-failed OSD in the list) which will then be responsible of replicating that data to the replica OSDs. When the replicas have received the update and have stored it in their memory, they will return an *ack*. After the primary has received the *ack* from the replicas, it sends it to the client allowing synchronous POSIX calls to return. However, the update is not complete yet, after the replicas are done writing the updates to their disks they return a commit to the primary. When the primary receives all the commits it then sends a commit request to the client, marking the update as completed and

committed. This method spares the client of the complexity and resource utilization surrounding the synchronization and serialization process between the replicas, shifting this burden onto the OSDs which are assumed to have better networking capabilities and resources. This process is illustrated in Figure 6.2.

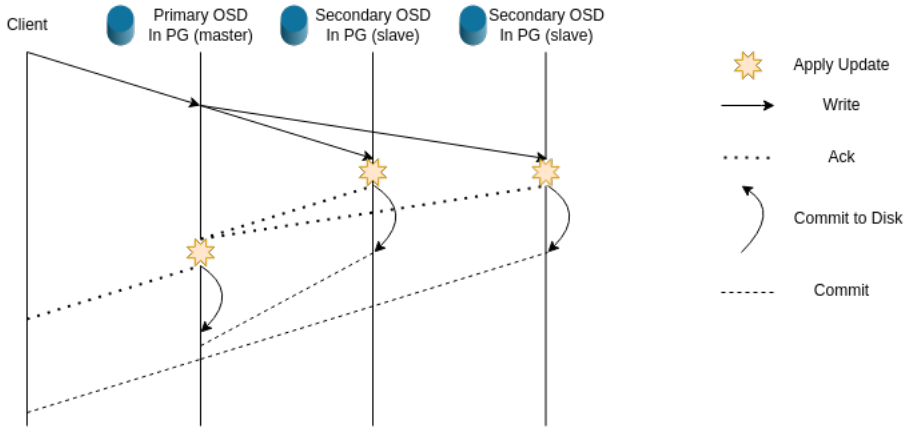


Figure 6.2: Illustration of the steps taken by RADOS in order to persist client information.

RADOS, as was mentioned previously, also handles failure detection and maintains availability and integrity in these cases. Some failures, like disk errors or data corruption, can be self-reported by the OSDs and handled accordingly while failures that make the OSD unreachable on the network require active monitoring by the system. Because of this, every OSD monitors the peers that it shares *placement groups* with. If any OSD is unresponsive, it's initially marked *down* and any responsibilities assigned to it will be moved on to the next OSD in the *placement group*. If the OSD does not quickly recover, it's marked as completely down and is removed from the data distribution and another OSD will join each *placement group* the deceased OSD was part of to re-replicate its contents. By marking the OSDs as *down* and only after an extended period of downtime *out*, RADOS avoids initiating data re-replication due to systemic problems. This distributed fault detection allows it to get fast fault detection without putting too much pressure on each OSD by forcing it to check every other OSD and also resolve the occurrence of inconsistency with central arbitration.

RADOS however can only support the detection and subsequent repair of crash faults, it is not capable of detecting and handling byzantine faults.

6.1.2 Metadata servers

User interactions with the file system namespace are handled by the Metadata server cluster. Since metadata operations often make up to half of the file system workloads and lie in the critical path, making the metadata cluster as performant as possible is critical for overall performance. Managing metadata also presents a critical scaling challenge in distributed systems unlike capacity and file I/O rates, which can scale almost linearly with the number of OSDs available, operations involving metadata require a greater degree of interdependence which makes

scaling while maintaining consistency quite the challenge. The first step Ceph takes to fix this is to have very small file and directory metadata, which consists of the directory entries (file names) and the inodes (which take a constant 80 bytes). Unlike conventional file systems, no file allocation data is needed since that is substituted by CRUSH. This allows the metadata servers to efficiently manage a very large working set of files by having them in memory independently of the actual file's size, which allows the server to maximise locality and therefore performance. Ceph also utilises Dynamic Subtree Partitioning to allow each metadata server to handle its own part of the namespace tree, allowing each part of the namespace tree to be handled by a certain metadata server, which further maximises locality and cache efficiency.

Although the MDS cluster's ideal run conditions would have it satisfy all requests from its in-memory cache, metadata updates must be committed to the disk for safe persistent storage. To do this, Ceph employs a set of large, bounded, lazily flushed journals which allow each metadata server to quickly dump its updated metadata to the storage cluster in an efficient and distributed manner. These logs will also absorb repetitive metadata updates so when the old journal entries are flushed to the long-term storage, many are already obsolete. The recovery of metadata servers can be done by reading the journals stored in the OSD clusters which will recover the critical contents of the failed server's in-memory cache and therefore recover the file system state. This allows Ceph to take advantage of the best of both worlds: stream the updates to the OSDs in an efficient and sequential manner while also processing most of the requests from its in-memory cache, which provides the best performance.

The Dynamic Subtree Partitioning algorithm works by adaptively distributing cached metadata hierarchically across a set of metadata servers. Each server measures the popularity of metadata within the directory hierarchy using counters with exponential time decay. Any operation done will increment the counters hierarchically from the affected inode all the way to the namespace tree root, which provides each server with a weighted tree that describes the current load distribution. The values for each server are periodically compared and in case of imbalances, subtrees with appropriate size are calculated and assigned to the MDSs so the workload can remain evenly distributed. In cases where this algorithm cannot cope with a certain hot spot, that metadata subtree might be distributed across various metadata servers. This will sacrifice some locality in the caching and also disallow critical opportunities for efficient prefetching and storage but it will also allow the system to disperse potential hot spots and flash crowds and achieve a balanced distribution (*e.g.*, many opens can be easily distributed across by replicating the metadata across various servers while a heavy write workload will get their contents hashed by the file name across the cluster).

6.1.3 Ceph Monitors

The primary role of a Ceph monitor is, as we have seen, to maintain a master copy of the cluster map and they must also provide authentication and logging services.

The cluster map is a composite of maps including the monitor map, the OSD map, the

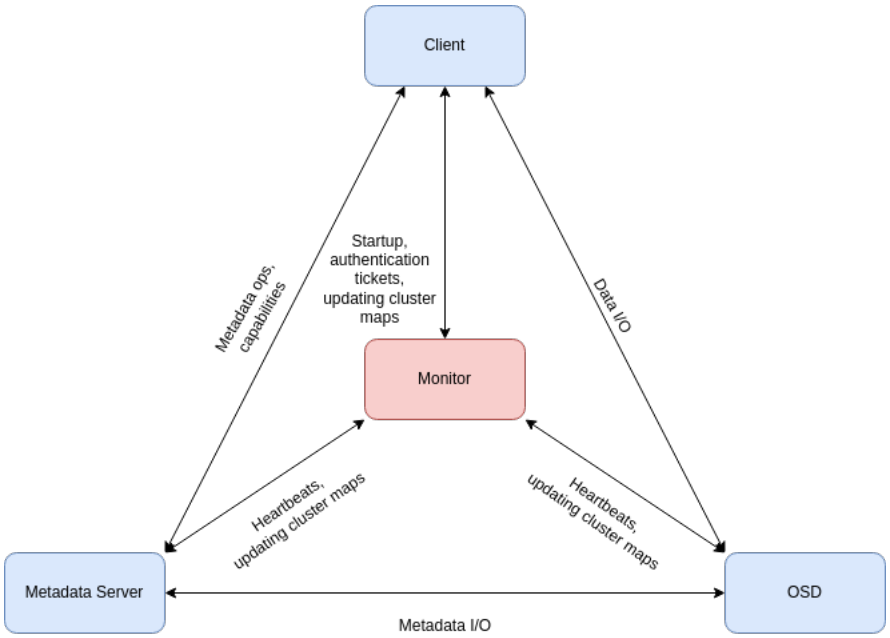


Figure 6.3: Illustration of the communication between various ceph components. Monitors communicate with all other ceph components.

placement group map and the metadata server map. It tracks a number of important things such as:

- Which processes in the Ceph Storage Cluster are *up* and running or are *down*.
- The current state of placement groups (*active* or *inactive* and *clean* or some other state).
- Other details related to cluster states and cluster health such as total amount of space, space used, among others.

As we were able to see in Figure 6.3, any meaningful changes to the state of the cluster are going to be transmitted to the Monitors (like OSD Daemons going down, placement groups falling into degraded states). The Monitors will then update the cluster map to reflect these changes.

Ceph imposes strict consistency requirements for Monitors when discovering other Monitors within the cluster. Whereas clients and other ceph daemons utilise the Ceph configuration file to discover monitors, monitors discover each other by utilising the monmap. This is done to avoid configuration file errors that could potentially break the cluster (like typos). Since monitors use the monmaps for discovery and these same maps are shared with clients and other daemons, the monmap provides monitors with a strict guarantee that their consensus is valid.

Cluster updates are incremental and thusly we keep a history of all the prior cluster states. Each version of the map is called an *epoch*. This history allows Monitors that have an older version of the monmap to be able to incrementally catch up to the current state of the cluster. If

the Monitors were to use the configuration files instead of the monmap, additional risks would be introduced due to the config files not being automatically updated and distributed across the quorum. This means monitors might utilise older versions of the file, by accident or lapse of the administrator and therefore fail to recognise a monitor, fail to be a part of the quorum, amongst other potential issues.

Ceph requires monitors to support ACID transactions in order to prevent them from hosting and serving corrupt or incomplete versions. In order to support this, it utilises RocksDB [4], the same embeddable persistent key-value store we utilised in FeBFT and discussed with greater detail in Section 4.6.

Ceph clients are required to connect to and exchange information with Monitors in order to write/read any file from the Ceph Cluster due to the function of CRUSH and RADOS, as we have just seen. Ceph has the ability to run in a single monitor configuration, however, it does introduce a single point of failure into the system as well as a possible large point of contention (since every single client has to connect to and obtain information from that single monitor in order to perform any type of operation on the cluster). Ceph mitigated this by introducing a way to host a scalable number of monitors which fixes our single point of failure and our inability to scale. However, as we have seen, clients obtaining wrong cluster maps would lead to wrong results from the CRUSH algorithm, which in turn could lead the client to unavailable or faulty OSDs which could lead to loss of performance or even loss of information. So, we need all the monitors in the cluster to be strictly consistent. As we have seen updates to the cluster map are incremental so we need a way to orchestrate the ordering of operations in all monitors such that in the end, all monitors have the same ordering of alterations to the map, leading to the same state. How were they able to achieve this?

OSDs follow a master-slave model, where the OSD contacted by the user (usually the first OSD of the placement group) will be responsible for replicating the data onto all of the other OSDs in the placement group, taking the role of master. This approach is fine in that situation since if the first OSD of the PG is not online, then the client will just contact the second one and the problem is migrated and handled with success. If the OSD crashes before replicating the data, the client would not get a reply and would know something went wrong, so he could direct his request to the next OSD in the PG. It is also fine in terms of ordering of operations since ceph works with *one shot blocks*, that is, after a block has been written it will never be altered, instead, a new block with the changes will be uploaded. Therefore there are no issues with the ordering of operations causing conflicting states.

However, this design would not be ideal for the monitors because it leaves the door open for situations where the master would crash and leave the slaves without updates and therefore stale information. It would also leave the entire cluster unprovisioned of monitoring since when the OSDs, MDSs and Managers attempt to contact the master Monitor and it's offline, contacting any of the others would be useless since only the master can execute requests. If, in order to fix this issue, Ceph's developers took a page out of the OSD design and allowed any of the Monitors to be master whenever he receives new information it would have left Ceph vulnerable to issues

with the ordering of operations causing inconsistent states across the Monitor cluster. So the developers of Ceph chose to use a custom crash fault-tolerant state machine replication algorithm derived from Paxos [33] to orchestrate the ordering of operations across the cluster.

We will now discuss how this was done and the general architecture of the ceph monitors.

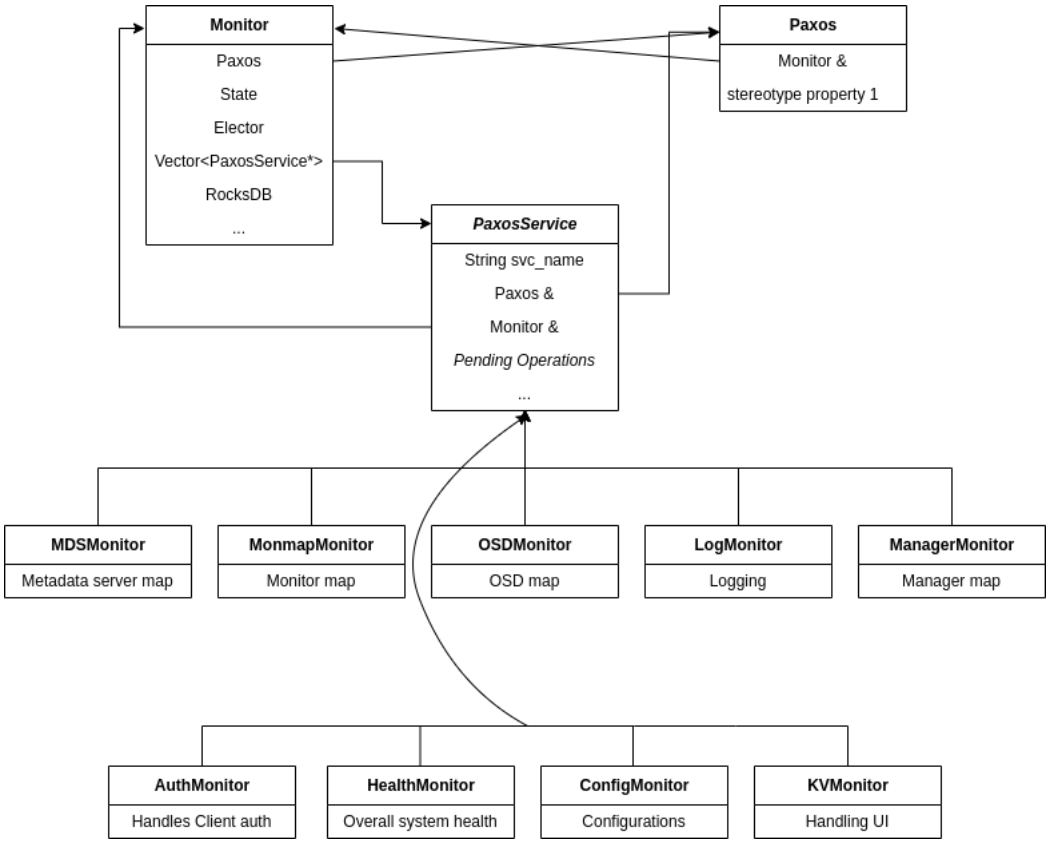


Figure 6.4: Illustration of the initial Ceph Monitor architecture

As is visible in Figure 6.4, we can see that the Ceph Monitor is "separated" into 3 classes. *Monitor*, *Paxos* and *PaxosService*. These 3 classes are responsible for handling the monitor behaviour, Paxos logic and service base logic amongst other things.

Paxos Library based on the Paxos algorithm, with some alterations. It is meant to be a simple replication substrate on which other services can be built upon. It stores data in a key-value format, utilising RocksDB for ACID transactions. This means that if we have that the last committed transaction is 4, then we will have version 4 in our store and no subsequent versions, while if we have that the first transaction is 1, then our lowest version is 1 and no less. Each of these versions is a blob of data, incomprehensible to Paxos, produced and delivered to Paxos by the services. Paxos will then join all the information from all services, check the current version and attempt to propose a new consensus instance. The information will only be persisted on the disk when it has been accepted by the Paxos quorum. This handles most of the logic required for Paxos like achieving quorums and proposing values amongst other logic required by the state machine replication. However, it does not handle all of the logic required for the correct

behaviour of the SMR. Namely, things like election logic and the maintenance of the quorum participants are handled by the Monitor class.

Monitor The top-level Monitor class. Handles leader elections for Paxos, maintaining and updating the quorum (adding/removing Monitors, as necessary), special modes of functioning for specific situations (for example the crashing of a large part of the cluster), synchronization for when we need to restore the monitor's state or restore the cluster from a failed leader situation (where a leader change has to be done) as well as communication with other monitors, task orchestration and lots of other small logic for the functioning of the monitor. It is also responsible for directly initializing certain values to the data store so the services can all run without problems (since it requires the first version to already be present in order to function correctly).

PaxosService The *PaxosService* class is the base, abstract, class upon which all services managed by the Monitor are built. It allows one to easily obtain an association between a Monitor and a Paxos class, in order to implement a Service. It indicates which methods must be implemented by each Service in order to be usable with Paxos as well as providing some base utility methods that all services will need. Then, as Figure 6.4 shows, we have the derived classes of this class which are the actual implementation of the services.

In general, this design is quite monolithic. We can clearly see that the Paxos implementation is deeply integrated with each of the parts of the architecture. *Monitors* have a very active part in the orchestration and management of the quorum, Services depend heavily directly on Paxos (however to a lesser extent) and the Paxos implementation is quite bare and reliant on the functioning of other parts of the system instead of being handled as it's own "separate" self-contained, self-controlled state machine replication algorithm. Given our objective was to replace this crash fault-tolerant, paxos-derived, state machine replication algorithm with our own byzantine fault-tolerant algorithm, it meant that we had to go back to the drawing board on the architecture of the system and figure out a way to try to abstract the SMR algorithm into its own, inter replaceable package. We then found some other significant design choices that were going to make the work of transforming Ceph Monitors into a Byzantine fault-tolerant system quite a bit more difficult.

Firstly, as we were able to view with the architecture that we just presented, the Paxos-derived system has a significant alteration to the way it functions. In the original Paxos proposal, we are able to identify two distinct members of the system: Replicas and clients. As such, Clients were the ones responsible for delivering operations to the algorithm and replicas were responsible for guaranteeing the correct ordering of request execution. So the original Paxos design, clients will broadcast their request to all the replicas in the quorum but only the leader will attempt to propose these requests into the system. However, as the design in Figure 6.4 indicates, there is no such distinction between replicas and clients. Instead, all replicas are clients and all clients are replicas. That is, the replicas themselves are the ones that generate the requests and are also the ones responsible for proposing and executing them. Now, as we have seen in Paxos there

is always a leader, which is the one responsible for proposing the new requests into the system so not all replicas are able (or allowed) to propose new items. In the original design of Paxos, this is handled because clients broadcast their requests to all the replicas, so the leader will of course also receive them. In Ceph's implementation, their solution was to forward any request a non-leader Monitor wanted to perform to the current leader and the leader would then propose it.

Secondly, because the whole system was only made to tolerate crash faults, they implemented a lease system, which allowed replicas to know when it was safe to read the latest value from the local storage. Like in the previous point, usually in order to perform a request, Clients broadcast the request to all replicas and then wait for $f + 1$ replies, to assure that a quorum of replicas has received and persisted the request. Indeed, when we are working with crash fault tolerance this is not necessary since if we are performing a read request, we know that as soon as we receive a reply we can utilise it (since there is no possibility of the replica being compromised and answering with wrong values). However in Ceph's design, there are no clients, so replicas are given the freedom of not even having to forward requests to the leader, they can just perform them locally.

Thirdly, Ceph clients only connect to one monitor and only perform queries on that given monitor. When we only want to ensure against crash faults, then this situation can be managed and when we detect faults in Monitors, we can just connect to another Monitor.

Fourth, similarly to ceph clients, other ceph components only connect to a single monitor to provide it with information like heartbeats, cluster map updates, amongst others. So the single monitor that is the recipient of the information is responsible for (and is trusted to do so) propagating this information across the entire cluster.

Fifth, many features of the Ceph Monitors relied on having state updates from the quorum for situations like reading while the monitors are in the middle of deciding a new version or when the monitor in question does not have the most up-to-date state. The way Ceph does this is through callbacks which should be triggered as soon as the event that initially held up the operation is completed.

6.2 Road to byzantine fault tolerant Ceph monitors

We will now see how we overcame the aforementioned issues and how we attempted to integrate FeBFT into Ceph and add Byzantine fault tolerance to the Monitors.

Firstly, we had to find a way to integrate the Client and Replica division that is present in FeBFT. As we saw in Ceph Monitors, each monitor is in itself a replica and a client at the same time. However, in FeBFT, requests are only made by clients and then batched and proposed by the leader. To accommodate this architecture, we made each Ceph Monitor host both a replica and a client. FeBFT Clients maintain their behaviour of broadcasting requests to all of the

replicas. As a result, Ceph monitors no longer need to forward any requests they want to make to the leader directly. Instead, they will use the FeBFT client to perform them. We then allocate a thread to run the replica's main loop and then utilise the client whenever we wish to perform any type of request. We use the unordered request capability of FeBFT to run read requests and normal ordered requests for write operations.

Secondly, we had to find a way to redesign the architecture described in Figure 6.4. This is because, as we discussed in Section 6.1.3, the original Ceph Monitor architecture is quite monolithic, mostly due to the high interdependency between the Paxos and Monitor classes, meaning we can't just replace the Paxos class with our own SMR system as we also have to account for the changes that would have to be made to the Monitors. Additionally, we also had to discover in which ways the services utilised the state machine replication and the monitors and how we could adapt that to include as little code change as possible since each of the services is a very complex, independent system which if we needed to modify could take a lot of time. However, we also didn't want to just discard the Paxos SMR algorithm as it requires developers to then maintain two separate versions of the Ceph monitor. Our solution was to find a way to abstract the state machine replication algorithm from the Monitor architecture. To do this, we first analysed how the Paxos algorithm relied on the logic that was kept in the Monitor section so we could then figure out the perfect point to separate the two parts.

Ceph Monitor and Paxos interdependence The first, very noticeable and very glaring issues were that a lot of the Paxos logic was actually handled in the Monitor class. Namely:

- Elections.
- Paxos quorum information and state.
- Request forwarding to the leader (Monitor class handles all communication between Monitors).
- Synchronization (State transfers).

We had two choices on how to handle this situation:

1. We could move all of this logic into the Paxos class. This option, architecturally, was the option that made the most sense. It would allow us to have a completely self-contained, separated state machine replication portion of the code which would handle all of the logic and could logically and "easily" be replaced with another SMR implementation, which was our end goal. It would also make for a more readable, understandable implementation since we would no longer have to document the complex interactions between the Monitor and Paxos classes which as we have seen are highly dependent on one another. However, it also presented some quite complex issues in the implementation portion. Doing this would require very deep knowledge of the interactions between the monitor and Paxos as

well as of the actual Paxos implementation itself. Since we had no one with expertise or knowledge of the inner workings of this protocol, we knew this option would require a lot of try-and-fail attempts, many potential issues and a lot of time which was not particularly abundant.

2. Instead of abstracting just the Paxos class into a generic SMR class, we could also abstract the Monitor class and then have a separate Monitor for each SMR system we wished to implement. This option in terms of architecture is not ideal, since we still have a great interdependence between two objects that should ideally be completely separate for our intents. It also makes implementing more alternatives harder and more complex, as well as requiring more work to actively maintain. It does, however, solve our most pressing issue, it requires much less in-depth knowledge about the implementation details and can be done quite quickly and without having to refactor too much code. The amount of additional work required to maintain is also not that large, since all of the common code between the various Monitor implementations is abstracted to the base class and therefore will only need to be updated in one place.

We wound up choosing the second option since we were already running pretty low on time and did not have assistance from anyone who was already knowledgeable in the subject.

As we can see from Figure 6.5, with our architectural changes, we were able to incorporate FeBFT into Ceph's monitors without having to completely rework all of the services (which was one of our main objectives) and without having to refactor Paxos in order to include all of the logic that was previously contained in the Monitors (as well as all of the other items that would have to be refactored to account for it).

The next problem we wanted to fix was how to handle Ceph Monitors being both a replica and a client. To accommodate this we initialise both a replica and a client in each of the monitors and then allocate a separate thread dedicated to running it and we use the client for all the monitor's needs such as proposing requests and reading from the service. We could have run the replicas separately and only use a client instance within Ceph since we do not actually use the replica's reference for anything other than running it, but then we would have to start/stop two services instead of just one and we would also not have abilities like only starting the replica after the initialisation steps for the monitor had been taken as well as being able to actually use the monitor's data store instead of using FeBFT's. We will see why this was very important later on.

Following this, we wanted to address the issues relating to the system's design towards crash fault tolerance and the following optimisations for better performance. In specific, the way the leasing protocol works and how ceph clients only connect to one ceph monitor.

Leasing Protocol The leasing protocol was built in order to reduce the latency of accessing the storage. When a Monitor has a valid lease (which can only be assigned by the leader) he can read directly from his local store, avoiding having to perform a full commit. This not only

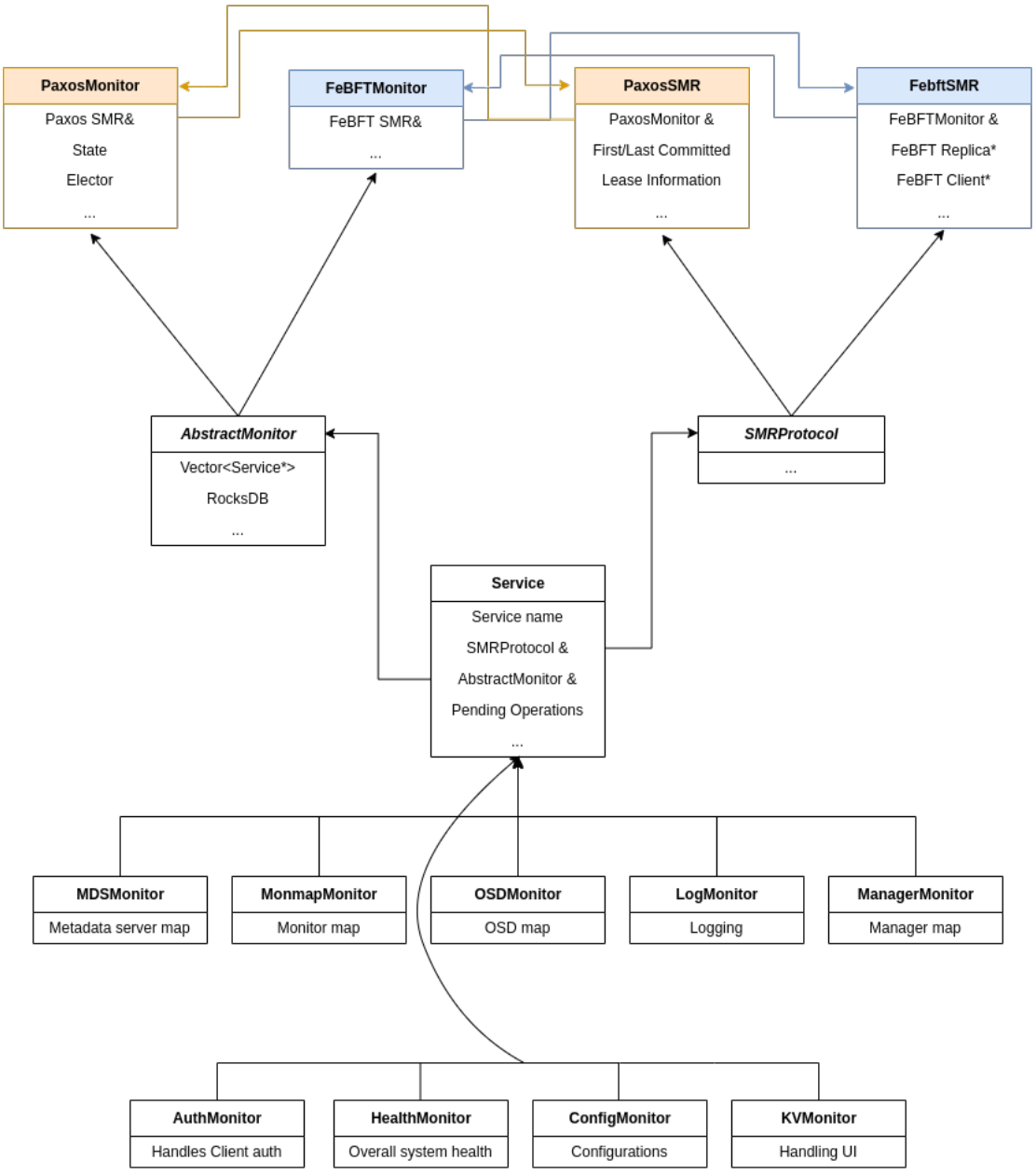


Figure 6.5: Illustration of the modified Ceph Monitor architecture

reduces the latency but also avoids having to overload the quorum with commit rounds. However, this poses a lot of problems when we try to stricken the requirements of Byzantine fault tolerance. As we know systems that only tolerate crash faults only have two possible states for each of the participants: active and correct or crashed (unavailable and incorrect). There is no state where the participant is active but is not correct, which is possible when we are working with byzantine faults. In byzantine faults, we cannot trust the saying of any f nodes, as they can be faulty and respond with incorrect answers (either for malicious purposes or because of natural phenomena like bit flips or data corruption). So, we know that this type of leasing protocol does not fit the needs of our BFT system, because we are always dependent on the correct functioning of our own device, which we know cannot be taken for granted. Our own local storage could have

suffered a random bit flip or even malicious manipulation and therefore cannot be trusted. We require at least $f + 1$ equal answers to be sure that the information is not incorrect in any way.

Ceph client connections This problem fits exactly into the same scope as the previous *Leasing* issue. As we have seen with that situation, we know that when we want to protect against byzantine faults, we cannot depend on any f participants to provide us with the correct answer as even if they are active and do reply to our requests, we cannot tell if they are currently faulty and therefore providing wrong information. Ceph clients only connect to one Ceph Monitor, so if the monitor has any type of fault while reading from the SMR system and replying to the ceph client the client is left without any kind of way to check the validity of the answer. Even if the Ceph Monitor acts as a client and reads from the SMR correctly (without utilising the leasing protocol) there is absolutely no guarantee that no faults occurred in the time between receiving the response from SMR, processing it and delivering it to the Ceph client. To fix this, we need to make the Ceph Clients connect to at least $f + 1$ monitors and wait for at least $f + 1$ equal responses.

Ceph component connections Similarly to the clients, we also share the issue of relying on a single point of failure and again, to fix the issue we should be connecting to at least $f + 1$ monitors so there is no possibility of any given f byzantine faulty monitors fail and do not propagate the information.

These changes however are not particularly straightforward. As we have mentioned, in Ceph the monitors behave both like replicas (storing state and performing the relevant changes) and as clients (performing ordered requests, based on information that was given to them by other parts of the ceph system) as well as performing unordered requests to read information from the SMR system so they can respond to the actual Ceph Clients. So we see that there is actually a duplication of efforts since the Monitors have to perform read operations on the SMR in order to respond to requests made by the actual Ceph Clients so if the ceph clients connect to multiple monitors the effective amount of requests made to the SMR is multiplied. So in order to fix this we have a couple of options:

- We could make the ceph clients connect to at least f monitors and allow each monitor to read from its local storage and then allow the ceph client to perform response matching and assert which response is the correct one. In terms of ease of implementation, this would be the most attainable option since it would (in principle) only require us to rework the clients a bit to allow for multiple concurrent connections and then use some hashing algorithms to verify the responses we obtained matched up and were correct. We would not have to redesign the way ceph's monitors' services work and interact with the SMR storage since they would be able to be based on the same leasing system. However, this alternative also made it a lot harder to handle the issues with the other ceph components, which often require alterations to the state following information received as we will see further ahead.

- We could make the Ceph Clients actually act like regular SMR clients and perform the requests directly to the replicas, effectively skipping over the Ceph monitor interactions. This would require an incredible effort to completely remake the clients and would also require all of Ceph's monitor services logic to be moved to the clients or to the FeBFT service, effectively moving operations that should be general to both Paxos and FeBFT into areas where they would have to be maintained completely separately. This would effectively ruin the efforts made beforehand of attempting to maintain ease of maintenance for developers that came afterwards, as they would have to maintain both a Paxos and a FeBFT version. It would also require the rewriting of tens of thousands of lines of code, which would have to then be retested for correctness and like mentioned maintained completely separately from the Paxos version of the software. This would, however, be the optimal solution in terms of correctness since as we have seen in all of the other BFT SMR systems we described, they all describe an architecture of replicas that do not actually perform requests instead they are just responsible for processing and answering requests from other clients. It's also the option that fits best with the very similar issue with other Ceph components, which we will discuss further on.

In the end, we were forced to attempt to implement the first solution since as we have seen, the second solution would require an impossible amount of work for our time frame.

The possible solutions to the ceph component connections are a bit more complex since they provide cluster map updates and other information that needs to be incorporated into the SMR state. This means that if we propagate the same alterations to various monitors and then each of those monitors tries to propose it, we could get duplicate requests and therefore inconsistent states and runtime errors amongst many other potential issues. This means that the option of allowing the components to connect to at least f monitors and multicast their requests so the monitors can then receive and propagate them is quite tricky as we would then have to add another layer of checks to make sure we didn't propose the same request twice, which would add another layer of verification making it slower and more error-prone. So, the only possible solution we found was to make the ceph components act as FeBFT clients and broadcast their requests to all of the replicas in the quorum. This would require us to move all of the Monitor logic of handling these requests and performing the necessary alterations on the quorum state into FeBFT's service, meaning (like the previously mentioned solution for ceph clients) we would have to effectively maintain two completely separate versions of Monitors, not to mention the initial huge amount of implementation work that would be required in order to implement and test all of the logic for all of the services in ceph's monitors into rust and FebFT. This required a lot of in-depth knowledge and time which we did not have.

6.2.1 Implementation details

We will now discuss the actual implementation details of the aforementioned architectural changes, namely how we integrated FeBFT, a Rust BFT SMR system into Ceph which utilises C++, as

well as some of the several issues we faced while doing so and how we resolved them.

Firstly as we have seen in Section 6.2 we started by altering the architecture of the Ceph monitor to include two new abstractions that would allow us to incorporate FeBFT into Ceph's monitors and not have to completely remake or redesign any of the services that they provide, which are each very complex and specialised. This involved the creation of two abstract classes:

- *AbstractMonitor*
- *SMRProtocol*

We then took the regular, Paxos Monitor and analysed it from top to bottom in order to figure out what were the parts that were general to both Paxos and FeBFT and which parts were specific to Paxos and therefore had to be relocated to the PaxosMonitor, instead of remaining in *AbstractMonitor*. We also had to move some of the features that were not completely related to Paxos, but at the same time also didn't fit without BFT design into the PaxosMonitor like:

1. *Scrubbing*: Scrubbing is used to check if all of the monitors were storing identical contents (as situations like bit flips, data corruption, amongst others can happen randomly). This is not particularly needed when we were utilising FeBFT since it's already a byzantine fault tolerant system, meaning it's already built to handle situations like that naturally and already includes the necessary measures to identify and fix the situation.
2. *Stretch mode*: Stretch mode is designed for ceph clusters where a significant part of the cluster is stuck behind a single network component for example a single cluster distributed across multiple data centres and we want to tolerate the loss of an entire data centre (so a very significant part of the cluster). We didn't include this in FeBFT because it was heavily designed to work with a CFT mode of operation and would require a lot of adapting in order to function with FeBFT.

We also analysed the Paxos implementation and found all of the necessary methods for the functioning of the Monitor and abstracted them into the *SMRProtocol* class. This class is more an interface than an abstract class since it only contains method definitions that the SMRs have to implement.

We then started integrating FeBFT into Ceph. Our first step was to start working on a service that was capable of handling the needs of Ceph Monitor's SMR. To do this, we first analysed how the monitor and services utilised the current Paxos-based SMR algorithm. We discovered that the SMR was utilised for data storage and that it performed absolutely no operations on the data that it was provided with. It treats the data it receives as a blob of data, incomprehensible to the SMR. The job of the SMR is to associate the values provided to it by the various services with versions such that the ordering is consistent across all of the monitors in the cluster. This allowed us to not have to make a service that was very intricate, since we could treat all of that data as a simple blob in Rust.

6.2.1.1 Ceph's FeBFT Service

Given this information, we started implementing the FeBFT service with the basic operations that could also be found in Ceph's Paxos SMR, namely:

- Read a given version from the service.
- Read the latest version from the service.
- Write a new version into the service.
- Delete a version from the service.

We then had to start satisfying some of the more intricate requirements that were honoured by the Paxos SMR data store that was used, most significantly the ability to process transactions atomically instead of treating transactions as a set of separate operations. For this, we had to implement a new type of request, a transaction, that contains an array of requests. This transaction has to be dealt with atomically so that means if a part of the transaction fails or if the system crashes before the full transaction is completed and persisted, no part of the transaction will remain. Because of how we handled the transactions (treating them as a single request in FeBFT, instead of separately) we naturally get this behaviour (due to the natural properties of FeBFT) where the client must receive $2f + 1$ replies in order

After having completed the basic operations in Rust, we then had to find a way to utilise FeBFT within C++. This is possible because both languages utilise the LLVM compilation strategy, so we can generate static and dynamic libraries from Rust that we can then utilise inside our C++ project. We also had to build an interface between the C++ code and the Rust code [7]. To do this, we declared a number of external methods and data structures that we would then be able to utilise in Ceph. We then utilised cbindgen [5] to generate the C header file from our Rust library, which we will then copy and utilise in the Ceph monitors.

To handle generating the libraries from the Rust, importing and linking them into the C++ project automatically, we utilised a CMake tool called corrosion [6].

Transitioning memory from Rust to C++ and vice versa One of the big problems with utilising Rust code in C/C++ and vice versa is that Rust has a completely different memory management concept that does not play well at all with C/C++'s manual style of memory management. As we have already seen in Section 4.7, Rust has a particular style of compile-time automatic management named the Ownership model. To handle this issue, we had to be very careful with the way we shared memory between Ceph and FeBFT, taking care of removing the memory from the ownership scope of Rust, while also taking care to always return the memory to Rust so it can be disposed of properly (Rust classes have a specific layout which is not compatible with C++, so freeing memory allocated by Rust in C++ could bring some issues) while also taking these cares when working with memory allocated in C++.

Persistent data storage in FeBFT Initially, we wanted to utilise FeBFT as a completely "separate" program, where the only point of interaction between the two was through the FeBFT client stored in the Monitors. Under these pretences, we developed a data structure that followed the necessary structuring for Ceph Monitor data which followed the normal structure utilised in RocksDB. The basic structure can be seen in 6.2.1.1. We have an outer Map which means to represent the way RocksDB partitions the data into Columns and then the inner map which stores the actual data stored in that particular column. The data is indexed using a Binary Search Tree and we store the data itself as a blob of bytes exactly like we saw was handled by Paxos in Ceph Monitors.

```
BTreeMap<String, BTreeMap<String, Vec<u8>>>
```

Unfortunately, we quickly realised this was not going to work properly when taking into account how the Ceph Monitors were set up. This was because the services (and the monitor itself) require an already populated data store with the default data (like the monitor map, which as we saw in Section 6.1 is how the monitors know who to connect to). This is done on the initial startup of the monitor, where the monitor is run with the flag *mkfs*, which indicates to it that it should initialise the necessary data points for the correct functioning of the monitor. This initialisation is done directly into the RocksDB data store, skipping over Paxos entirely (which also makes some sense, since the information stored within the data store is also required for the correct functioning of the Paxos quorum). This, as we can clearly extrapolate, is not compatible with our initial approach of compartmentalising FeBFT and only accessing it through the client. The most obvious solution to us was to make FeBFT read from and write to the actual underlying RocksDB data store of the monitor. This solution did not present itself without its problems though. Namely, how were we going to use this C++ API in Rust, where we had no way of performing any method calls on it? Our solution was to create a C interface which consisted of method references to C++ methods that accepted the RocksDB data structure and the necessary arguments to it and then executed the operation in C++. In Rust, we then kept these method references so we could call them when we needed to perform any operations on the data store.

This also offered another potential benefit (theoretically) which is the ability to transition from Paxos to FeBFT and vice-versa. Implementing this would require some modifications to the BFT system to handle the absence of quorum certificates for all of the requests performed while in Paxos mode we were not done because this feature was not part of the scope of this dissertation.

Issues with utilising RocksDB both in FeBFT and in Ceph Monitors When we wanted to start compiling and testing Ceph with the new changes, we ran into a problem on the last step of linking, where we linked the Rust service with the C++ project. The issue was related to having multiple function definitions relating to RocksDB since both FeBFT and Ceph's

monitors depended on RocksDB for its data storing techniques. Because Rust is so recent and its interactions with C/C++ are even more recent and very sparsely documented, we were unable to find a way to solve this issue while maintaining the RocksDB dependence on both sections of the project. The solution that we did find in order to successfully compile and start testing it was to introduce a compilation flag that allowed us to opt-out of the RocksDB dependency in FeBFT. This means that we do not get any persistence in FeBFT, but due to FeBFT's modular design, we could implement this with another data store such as LevelDB [2] which would effectively solve the issue.

Handling with the required state updates As previously discussed, we know that Ceph Monitors rely heavily on receiving state updates from the quorum through callbacks in order to execute code at the correct time. FeBFT did not have any way for us to get notifications about the state of the quorum either through the replicas (which do not have any outward-facing API's, as they are not meant to be accessed anyways) or through the clients. In order to address this issue, we could introduce a way to deliver these updates either through the replicas or through the clients and then introduce a way to get these updates and actually call the C++ callbacks relating to Ceph. Since we wanted replicas to continue to be a "black box" with no access to them (since that's the client's job), we decided the best way was to provide this information on the client side. As such, we introduced a new type of client, built on top of the infrastructure provided for the regular clients named the Observer Client, which we discussed in Section 4.8. We created a C function reference type in Rust that accepts a callback as an argument and triggers the callback when called. We accept this function reference as one of the arguments needed to initialise the FeBFT client. When a callback is created by the monitor, it is passed along to the FeBFT client so it can then be triggered by the observer client.

Chapter 7

Conclusion

This dissertation focused on the development and contributions made to the Byzantine Fault Tolerant (BFT) State Machine Replication (SMR) middleware FeBFT as well as utilising it in order to improve the fault tolerance resiliency of the distributed storage system Ceph.

FeBFT was implemented in Rust and attempted to improve upon already existing BFT SMR systems available for public use both in terms of performance, code maintainability and resource utilisation. Another extremely important goal was to make FeBFT as easy to utilise as possible for its consumers (developers who implement their services utilising this library).

Having chosen Rust as the programming language of this middleware gave us many advantages versus other popular choices like Java [1] or C. Namely, Rust's memory management primitives mean that not only do we not require a Garbage Collector but we also do not have to perform manual memory management. Instead, we have compile-time proof that our program does not have any sort of memory access or management bugs which means our program is protected against the most common cause of security exceptions. This choice was recently supported by the NSA's recommendation to stay away from manual memory management [3]. Another great advantage of this approach is that we are able to provide incompatibility with a great number of other languages which are also based on LLVM, like C and C++ (and subsequently all languages that these languages are able to interact with).

We then discussed how to obtain the performance targets that we wanted to hit in order to effectively improve on the already existing system, how to make the overall system architecture more scalable in more real-world facing scenarios and also some quality-of-life improvements for the end user. We faced many challenges when attempting to obtain our desired performance namely with the usage of the asynchronous runtime, the inefficient batching of client requests and the networking performance difficulties. We discussed solutions that we implemented in order to fix the aforementioned problems as well their positives and negatives. In the case of networking performance in addition to solving the issue, we presented a theoretical architecture for a solution that could yield even better performance that we were not able to implement in the scope of this dissertation.

We proved that it is indeed possible to build a general-purpose, robust BFT SMR library in Rust with our presented implementation. We also provide an easily extensible architecture with a very simple RPC-like approach to the interface (as well as a secondary callback-based approach for more advanced users). We were also able to beat the performance of the protocols this system was based on like BFT-SMaRt by providing a stable, consistent performance which we believe is being bottlenecked by our client machine's processing power. This performance is only possible by the lack of garbage collector, which can severely harm performance. However, the properties of Rust allowed us to do this without incurring on any of the memory safety that a manually managed memory is vulnerable to, like with PBFT.

We then went on to discuss Ceph, a distributed storage system which is designed to withstand crash faults while still maintaining almost linear horizontal scalability and how we were going to attempt to improve its resiliency. Namely, we wanted to make the Ceph Monitor cluster tolerant to Byzantine Faults. As we mentioned, the monitor cluster is responsible for delivering extremely important information about the current active OSDs, MDSs and other system components. An attack or fault to this cluster could mean the clients receive malicious or just wrong information which can lead to various issues like information leaking from the system or just being lost. This made us believe it was the first and most important part of the system to be protected. We spoke about how we had to change the architecture of the Ceph Monitors in order to support exchanging the original CFT SMR protocol with our new BFT SMR protocol, FeBFT and all the issues that originated from these changes. We also presented some other issues with Ceph's design that could make it vulnerable to byzantine faults and how we believed they could be fixed.

Although having implemented all of these changes to Ceph's Monitor architecture and making the required service for FeBFT to be integrated into it, we ran into many issues and wound up facing issues with the initialization of the Monitor which were preventing the correct functioning. We were not able to resolve these issues in the timeframe required for this dissertation but we are in the process of contacting the Ceph Monitor maintainer and requesting his assistance in order to fix these issues and prove FeBFT can be utilised within Ceph to provide Byzantine Fault Tolerance.

7.1 Future Work

Our advancements to FeBFT although complete, opened a lot of doors open for potential future work to further advance and improve FeBFT.

7.1.1 Utilising Followers as Collaborative State Transfer helpers

As we described in Section 4.9.2, followers also follow the quorum and keep the state of the quorum in order to respond to unordered requests. As we also saw, they do not necessarily follow the quorum's decisions at the speed the quorum is making them (as we do not guarantee strict

consistency, only eventual consistency). This is because they do not partake in deciding any ordered operations, instead, they just listen and follow the decisions taken by the quorum.

However, they are still perfectly capable of helping the quorum replicas perform state and log transfers, leading to less load being put on the quorum participating replicas. If the replica performing the state transfer is also a follower, then all of this can be done without even contacting the quorum participating replicas, meaning we can potentially add an infinite amount of followers without harming the performance of the ordered operation execution at all. If it is not a follower then it will still have to check with the quorum replicas to assert it has the most up-to-date state. Even with this extra check, we can still prevent the overwhelming majority of traffic required to transfer the state by diverting the recovering replicas to the followers.

7.1.2 Utilising Followers as Live Backups

Again, as described previously in Section 4.9.2, followers keep the state of the quorum. They keep all of the necessary certificates in order to assure the current state they have is correct and it follows the same design as a regular quorum participating replica.

So it's perfectly feasible and actually quite simple to make a system that is capable of taking a replica that is currently designated as a follower and transforming it into a quorum participating replica. We would have to introduce a system that is capable of handling live reconfigurations.

Whenever the system detects a failure in the replicas that are participating in the quorum, it can remove the failing replica and use one of the followers to restore full failure tolerance capabilities to the service. The follower would have to make sure it is up to date with the quorum replicas before actually participating (as a recovering replica would also have to do) but since it sets off from a state which is more than likely very close to the current state of the quorum, we can potentially save a lot of bandwidth and time, meaning the system is left vulnerable for a much shorter period.

7.1.3 Scalable request propagation amongst followers

When we initially thought of the idea for followers, we wanted to implement something that would give an ability for our system to scale horizontally very easily in order to maximise the number of operations the system can execute (even if they are only read operations). However, another one of our primitives was that we did not want to hurt in any way the performance of the ordered request system.

As discussed in Section 4.9.2 the effect on the leader's performance if he had to transmit the pre-prepare messages to all followers and how we mitigated this by making the replicas that are not leaders responsible for propagating the pre-prepare messages throughout the followers. This partially solves the issue because we can scale the number of replicas in the quorum but there is always just 1 leader, effectively giving us some room to grow.

This is not however the most effective and efficient way of doing this. So we thought of another way to propagate the requests throughout the network with maximal scaling efficiency and if possible even latency efficiencies. What we thought of was a hierarchical tree-like system, where the quorum replicas only broadcast to the first level of the hierarchy which would then be responsible for propagating the requests throughout the rest of the hierarchy of course always taking into account the fact we can have faulty nodes in the system.

This would allow the followers to be laid out by data centre, cluster and even rack for maximum bandwidth and latency efficiency.

7.1.4 Implement support for trusted components

There is plenty of literature discussing the implementation of trusted components in BFT SMR systems [12]. Usually, this literature is aimed at reducing the overhead that is usually associated with byzantine fault-tolerant systems. This is because we know that the trusted components can only fail by crashing (so we do not have to account for their byzantine faults).

This can be done by using the trusted components for reducing implementation costs, communication steps, execution delays or even reducing the number of replicas needed to safely replicate a BFT system, allowing the number of replicas required to withstand f faults to be as low as $2f + 1$, compared to the usual $3f + 1$. Depending on the approach, the trusted subsystem can include entire virtualisation layers, centralised configuration services, trusted logs or can be as small as a trusted platform module, a smartcard, etc. A key characteristic is that subsystems with less trusted computing bases are less likely to fail arbitrarily.

7.1.5 Automatic client session handling

As we spoke about when describing the benchmarks we developed for the system in Section 5.4, we discussed how the clients worked and how they rely on sessions in order to perform multiple concurrent requests since each session can only have 1 pending request at a time. This means that the developer that is implementing his service on top of FeBFT is going to have to be responsible for managing the client sessions and how the requests are performed concurrently, which will increase the development time and complexity.

We had an idea that would help mitigate this situation and facilitate the development for developers that wish to implement their service with FeBFT. This can be done by introducing ways to automatically handle client sessions in order to facilitate developers implementing concurrent requests in their services. This can be done in two ways (or both even, to provide as much choice as possible to the users):

- ***Static session handling*** - The first possibility (and the simplest one) is to statically allocate n sessions as soon as the client is initialised. From then on, the client would be

limited to this number of sessions as no more would be allocated. This means that each client will be limited to n concurrent requests and any extra concurrent requests would have to wait for a session to become available before being performed. The implementation for this would be quite easy as it could just be a bounded queue, where all available sessions are placed. Whenever a session is used it gets withdrawn from the queue and when it's ready to be reused it is placed back into the queue.

- ***Dynamic session handling*** - The second possibility is a bit more complex, however, its implementation is in reality quite similar to the one we just described. Basically, we would handle the sessions similarly (with a bounded or unbounded queue), where we initially allocate n sessions. The difference is that whenever we want to perform a request and there are no sessions currently available, instead of waiting, we would just create a new session and add it to the "pool" of sessions that can be used by the client. Similarly, when the session is done being used it gets placed back into the queue for reuse. This means that there is no practical limit on the number of requests that can be performed as a new session can always be allocated.

Either of these solutions would improve the usability of FeBFT clients significantly by abstracting a large part of client management and allowing service developers to focus on the service they are developing instead of having to focus on managing the middleware library.

Bibliography

[1] Java programming language. https://www.java.com/en/.

[2] Leveldb: Leveldb is a fast key-value storage library written at google that provides an ordered mapping from string keys to string values. https://github.com/google/leveldb.

[3] Nsa guidance on protection against memory safety issues. https://www.nsa.gov/Press-Room/News-Highlights/Article/Article/3215760/nsa-releases-guidance-on-how-to-protect-against-software-memory-safety-issues/.

[4] Rocksdb: A persistent key value store for fast storage environments. http://rocksdb.org/.

[5] cbindgen creates c/c++11 headers for rust libraries which expose a public c api. https://github.com/eqrion/cbindgen.

[6] Corrosion, formerly known as cmake-cargo, is a tool for integrating rust into an existing cmake project. https://github.com/corrosion-rs/corrosion.

[7] Using rust code inside a c or c++ project. https://docs.rust-embedded.org/book/interoperability/rust-with-c.html.

[8] Ittai Abraham, Guy Gueta, and Dahlia Malkhi. Hot-stuff the linear, optimal-resilience, one-message bft devil. *CoRR, abs/1803.05069*, 2018.

[9] Yackolley Amoussou-Guenou, Antonella Del Pozzo, Maria Potop-Butucaru, and Sara Tucci-Piergiovanni. Correctness of tendermint-core blockchains. In *22nd International Conference on Principles of Distributed Systems (OPODIS 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[10] Yackolley Amoussou-Guenou, Antonella Del Pozzo, Maria Potop-Butucaru, and Sara Tucci-Piergiovanni. Dissecting tendermint. In *International Conference on Networked Systems*, pages 166–182. Springer, 2019.

[11] Thomas E Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[12] Johannes Behl, Tobias Distler, and R. Kapitza. Hybrids on steroids: Sgx-based high performance bft. pages 222–237, 04 2017. doi:10.1145/3064176.3064213.

[13] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: Dependable and secure storage in a cloud-of-clouds. *ACM Trans. Storage*, 9(4), nov 2013. ISSN: 1553-3077. doi:10.1145/2535929.

[14] Alysson Bessani, Ricardo Mendes, Tiago Oliveira, Nuno Neves, Miguel Correia, Marcelo Pasin, and Paulo Veríssimo. Scfs: A shared cloud-backed file system. pages 169–180, 01 2014. ISBN: 9781931971102.

[15] Alysson Bessani, João Sousa, and Eduardo Alchieri. State machine replication for the masses with bft-smart. pages 355–362, 06 2014. doi:10.1109/DSN.2014.43.

[16] Alysson Neves Bessani. Chapter 4 a guided tour on the theory and practice of state machine replication. 2014.

[17] Stephen M Blackburn, Perry Cheng, and Kathryn S McKinley. Myths and realities: The performance impact of garbage collection. *ACM SIGMETRICS Performance Evaluation Review*, 32(1):25–36, 2004.

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

[19] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20:398–461, 11 2002. doi:10.1145/571637.571640.

[20] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Michael Dahlin, and Taylor Riché. Upright cluster services. pages 277–290, 01 2009. doi:10.1145/1629575.1629602.

[21] Bram Cohen and Krzysztof Pietrzak. The chia network blockchain. *vol*, 1:1–44, 2019.

[22] Chris Dannen. *Introducing Ethereum and solidity*, volume 1. Springer, 2017.

[23] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. *ACM SIGPLAN Notices*, 26(4):245–257, 1991.

[24] Seth Gilbert and Nancy Lynch. Perspectives on the cap theorem. *Computer*, 45(2):30–36, 2012. doi:10.1109/MC.2011.389.

[25] John L Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[26] Seungyeop Han, Haichen Shen, Taesoo Kim, Arvind Krishnamurthy, Thomas Anderson, and David Wetherall. MetaSync: File synchronization across multiple untrusted storage services. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 83–95, Santa Clara, CA, July 2015. USENIX Association. ISBN: 978-1-931971-225.

[27] Mark D Hill and Michael R Marty. Amdahl's law in the multicore era. *Computer*, 41(7): 33–38, 2008.

[28] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, page 11, USA, 2010. USENIX Association.

[29] Flavio Junqueira and Benjamin Reed. *ZooKeeper: distributed process coordination.* " O'Reilly Media, Inc.", 2013.

[30] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. Cheapbft: Resource-efficient byzantine fault tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, page 295–308, New York, NY, USA, 2012. Association for Computing Machinery. ISBN: 9781450312233. doi:10.1145/2168836.2168866.

[31] Kadir Korkmaz, Joachim Bruneau-Queyreix, Sonia Ben Mokthar, and Laurent Réveillère. Alder: Unlocking blockchain performance by multiplexing consensus protocols. *arXiv preprint arXiv:2202.03186*, 2022.

[32] Jae Kwon. Tendermint: Consensus without mining. *Draft v. 0.6, fall*, 1(11), 2014.

[33] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, December 2001.

[34] Rust Language. Rust's fearless concurrency primitives. https://doc.rust-lang.org/book/ch16-00-concurrency.html.

[35] Ricardo Mendes, Tiago Oliveira, Vinicius Cogo, Nuno Neves, and Alysson Bessani. Charon: A secure cloud-of-clouds system for storing and sharing big data. *IEEE Transactions on Cloud Computing*, 9:1349–1361, 12 2021. doi:10.1109/TCC.2019.2916856.

[36] Satoshi Nakamoto. Bitcoin whitepaper. *URL: https://bitcoin. org/bitcoin. pdf-(: 17.07. 2019)*, 2008.

[37] Diego Ongaro and John Ousterhout. The raft consensus algorithm. *Lecture Notes CS*, 190, 2015.

[38] Roberto De Prisco, Butler Lampson, and Nancy Lynch. Revisiting the paxos algorithm. *Theoretical Computer Science*, 243(1):35–91, 2000. ISSN: 0304-3975. doi:https://doi.org/10.1016/S0304-3975(00)00042-6.

[39] Signe Rüsch, Kai Bleeke, and Rüdiger Kapitza. Themis: An efficient and memory-safe bft framework in rust: Research statement. In *Proceedings of the 3rd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, SERIAL '19, page 9–10, New York, NY, USA, 2019. Association for Computing Machinery. ISBN: 9781450370295. doi:10.1145/3366611.3368144.

[40] João Sousa and Alysson Bessani. From byzantine consensus to bft state machine replication: A latency-optimal transformation. In *2012 Ninth European Dependable Computing Conference*, pages 37–48, 2012. doi:10.1109/EDCC.2012.32.

[41] Chrysoula Stathakopoulou, Tudor David, Matej Pavlovic, and Marko Vukolić. Mir-bft: High-throughput robust bft for decentralized networks. *arXiv preprint arXiv:1906.05552*, 2019.

[42] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolic. Mir-bft: High-throughput bft for blockchains. *arXiv preprint arXiv:1906.05552*, 2019.

[43] Robbert van Renesse, Chi Ho, and Nicolas Schiper. Byzantine chain replication. In Roberto Baldoni, Paola Flocchini, and Ravindran Binoy, editors, *Principles of Distributed Systems*, pages 345–359, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN: 978-3-642-35476-2.

[44] Giuliana Veronese, Miguel Correia, Alysson Bessani, Lau Lung, and Paulo Veríssimo. Efficient byzantine fault-tolerance. *Computers, IEEE Transactions on*, 62:16–30, 01 2013. doi:10.1109/TC.2011.221.

[45] Paulo Veríssimo and Luís Rodrigues. *Distributed Systems for System Architects*, volume 1. 01 2001. ISBN: 0792372662. doi:10.1007/978-1-4615-1663-7.

[46] Dmitry Vyukov. Bounded multi consumer multi producer implementation. http://www.1024cores.net/home/lock-free-algorithms/queues/bounded-mpmc-queue.

[47] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 307–320, USA, 2006. USENIX Association. ISBN: 1931971471.

[48] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. Crush: Controlled, scalable, decentralized placement of replicated data. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, pages 31–31, 2006. doi:10.1109/SC.2006.19.

[49] Sage A. Weil, Andrew W. Leung, Scott A. Brandt, and Carlos Maltzahn. Rados: A scalable, reliable storage service for petabyte-scale storage clusters. In *Proceedings of the 2nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing '07*, PDSW '07, page 35–44, New York, NY, USA, 2007. Association for Computing Machinery. ISBN: 9781595938992. doi:10.1145/1374596.1374606.

[50] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus in the lens of blockchain. *arXiv preprint arXiv:1803.05069*, 2018.

[51] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.