Implementation Of A Robust Cryptographic Broker For Chop Chop

Ilan Nissim

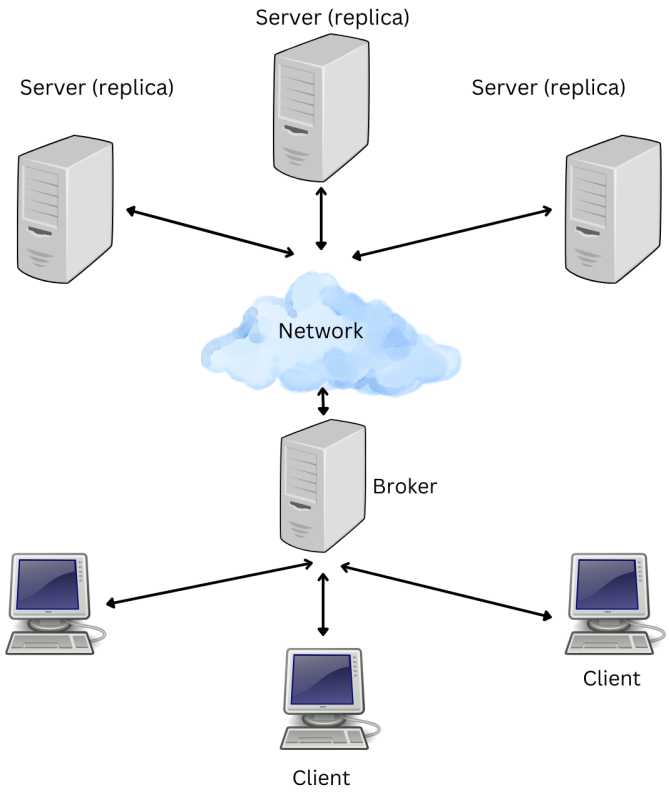

Figure 1: Distributed system architecture showcasing Chop Chop's innovative interactive protocol involving brokers

This page is intentionally left blank.

1 Introduction

The Internet's original conception embodied three guiding principles: decentralisation, universal accessibility and security. It would be resilient and immune to points of failure and malicious actors, free from centralized control and highly-available to all. Although *state machine replication* (SMR) has paved the way towards building reliable and resilient online services, the road ahead is still long and challenging. State machine replication is a concept in distributed computing which enables fault tolerance services and consistency across multiple servers in a network. Fault tolerance refers to the ability of a system to operate without disruption even if some of its components fail. Basically, in SMR, each server maintains a copy of the same state machine, so that if one server fails, the impact is limited to that server alone, allowing the network to continue without disruption. The states machines start with the same initial state, and when a client request is received, each server processes the request and updates its copy of the state machine accordingly. State machine replication relies on a consensus algorithm,namely *Atomic Broadcast*, that orders, authenticates, and deduplicates messages. Since every server has the same initial state, processes requests in the same order, and follows a deterministic logic –where state transitions from one state to the other are consistent across all servers–each server's state will evolve in a identical manner. Unfortunately, despite decades of research, modern implementations of *Atomic Broadcast* fall short of matching current industry standards. These systems are far from being able to process the sheer volume of requests–millions per seconds–that centralized Internet services process. This performance gap is often ascribed to the intrinsic cost of *Atomic Broadcast*, namely ordering, authenticating and deduplicating. But is it really the case? The paper *Chop Chop: Byzantine Atomic Broadcast to the Network Limit*, written by the Distributed Computing Laboratory (DCL) at EPFL argues that no and proposes an innovative solution involving *brokers* and a new form of batching called *distillation*. *Brokers* act as middlemen between clients and servers. They produce *distilled batches*, containing digest information, that allows servers to authenticate and deduplicate messages in bulk faster. The brokers essentially amortize the cost of authentication and deduplication. By spoonfeeding the pre-processed information to the servers, they significantly reduce the servers' workload.

2 Objectives and Requirements

I was tasked to implement a production-ready prototype broker designed for deployment on resource-constrained machines. Specifically, the broker was designed to run on machines with just 4 CPU cores and 8 GB of RAM. For the development of the code, I followed *NASA's Power of Ten - Rules for Developing Safety-Critical Code*. This set of guidelines is intended to guarantee reliability and safety of software in high-stakes environments. However, they also impose strict limitations on certain coding practices, including dynamic memory allocation, recursion, and the use of explicitly-sized types. These constraints reduce the flexibility and freedom we would otherwise have. In addition of designing the code, my work also involved benchmarking and testing the implementation on AWS EC2 instances.

3 Background information

In the following section, I have outlined and detailed the key technical concepts that I consider to be the backbone of this project. My approach has been to present and explain each concept in a digestible and illustrated manner, specifically tailored for a non-target audience. Starting from the basics and gradually building up, I've structured the explanations to help you understand how they fit in the greater picture.

3.1 Cryptographic Hash functions

To put it simply, it's a mathematical function that maps an input of arbitrary size to a fixed-size output. Whether you feed the function a single word or the content of the book *A la recherche du temps perdu* of Marcel Proust, the world's longest book, the output will have the same length. Good cryptographic hash functions satisfy these properties :

1. Determinim. Given the same input, the hash function will always produce the same output hash.
2. Pre-image resistance. Given an output hash value h, it's computationally infeasible to find a input m such that $hash(m) = h$. This stems from the fact that hash functions are one-way functions. In simpler terms, we can't invert the function, give it the input $f(x)$ and get x. Basically, they can't be reverse engineered.

3. Collision resistance. It is computationally infeasible to find two different inputs m_1 and m_2 such that *hash(m_1) = hash(m_2)*. A pair such that $m_1 \neq m_2$ but *hash(m_1) = hash(m_2)* is a called a hash collision.

4. Avalanche effect. The hash function is highly sensitive to even slights alterations in the input. A minor change "snowballs" into something much larger, hence the "avalanche effect".

There exists several families of hashing algorithms, including Message Direct (MD) and Secure Hash Algorithm (SHA) among others. This project utilizes the BLAKE3 cryptographic hash function.

3.2 Merkle Trees

Merkle trees are a popular data structure in computer science. They play a crucial role in distributed systems like Git (distributed version control system), Bitcoin (the world's most famous blockchain), and BitTorrent (a peer-to-peer communication protocol), to name just a few. To understand them better, let's begin with some key terminology. Each member of the tree is called a node, and in this structure, every node can have up to two child nodes, making it a binary tree. The lines connecting the nodes are called edges. The topmost node is called the root, while the bottom nodes of tree are called leaves. A Merkle tree is constructed from the bottom up. We start by hashing the inputs. These hash values will be the leaf nodes of our Merkle tree. We then group them into pairs starting from left to right. For each pair, we concatenate their hash values and then calculate a new hash of this combined value. We repeat this process, continually pairing and hashing until one hash value remains: the Merkle root. This last hash serves as a fingerprint for the entire dataset. Due to the properties of hash functions, even a tiny change in any piece of data in the tree will result in a completely different root hash.
The depth of the tree, defined as the number of edges from the root node to a leaf node, is given by $\log_2$(number of leaves). This will come in handy later, so keep it in mind. You might be curious about what happens when the number of inputs is odd. There are two approaches to handle this: a common method is just to duplicate the last node, but for this project, we chose to move the leftover node to the next level.

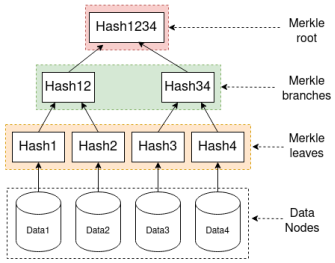

Figure 2: Merkle tree topology and how a Merkle tree is constructed

One of the key usecase of Merkle trees is to prove **efficiently** that a piece of data, in our case a message, is present in the original set, in our case, the batch. This is what we call **Proofs of inclusion**. A naive approach to verifying whether a piece of data is part of a dataset would require scanning the entire dataset, resulting in a linear time complexity of *O(n)*, where n is the size of the dataset. Simply put, this means that the verification time increases proportionally with the size of the input, making this method inefficient. In addition to that, a naive proof requires sending the entire dataset, which is inefficient in time and space, and also divulging potential information. A more clever and elegant solution involves using **Merkle Paths**. The idea here is to leverage the uniqueness of the Merkle root. The verifier is provided only the bare minimum information to reconstruct the path from the leaf node to the root. This path consists of the data itself, a sequence of siblings hashes and directions to concatenate. To verify, we start with the data, compute its hash, and then iteratively combine it with the provided sibling hashes until we have only one hash left. If the recomputed hash value matches the known Merkle root, we can conclude that the data is indeed part of the original dataset. This method is considerably more efficient, requiring only a number of operations equal to the depth of the tree.

3.3 Cryptography primitives

3.3.1 Asymmetric Cryptography

Let's say you want to send a birthday wish to your friend on Whatsapp and ensure that only your friend can read its content. How would you prevent hackers, Whatsapp employees or even government agencies from reading the

content? Well you would need a encryption algorithm to convert your message (plaintext) from plain english into a sequence of random-looking gibberish (ciphertext). The encryption algorithm is basically a mathematical formula designed to scramble data. It's generic, often known by all. The encryption algorithm always produces the same output for any given input, so how do you add any uniqueness? With a key. The key will be used to encrypt the data in a predictable and unique way so that even though it appears random, it can be accurately decoded back it its original form using the corresponding decryption key. For instance, the key might determine how to shuffle letters around or which operations to perform. There are many types of encryption algorithms but there are two types of keys - symmetric and asymmetric keys. We'll focus on the latter. It involves a mathematically related pair of keys: a public key and a private key. As the name suggests, the public key is known by everyone and used to encrypt data, while the private key, known only to the key holder, is used to decrypt data. We have seen how asymmetric cryptography maintains **confidentiality** by ensuring that even if a attacker intercepts the message, they're unable to decipher it.

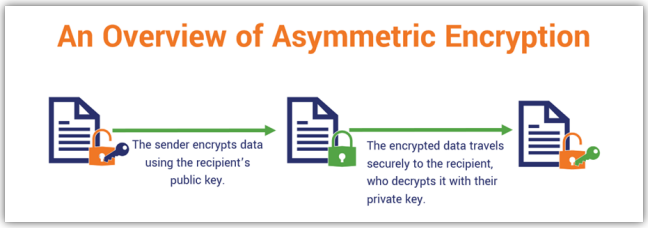

Figure 3: Merkle tree topology and how a merkle tree is constructed

3.3.2 Digital Signatures

But there is still a crucial aspect missing: **authenticity**. How can your friend verify the sender's identity and ensure that the message hasn't be tampered with by a malicious actor. That's where digital signatures come in. Digital signatures serve as the digital counterpart to handwritten personal signatures. They blend both of the concepts we've seen beforehand: asymmetric cryptography and cryptographic hash functions. Here's how they work: the sender first hashes the message they want to send, then encrypts this hash with their private key. This forms the digital signature. He then transmits both the digital signature and the original message. Upon receipt, the recipient decrypts the digital signature using the sender's public key, revealing the original hash. They then hash the received message and compare it to the decrypted hash. If these match, it confirms that the message has not been altered and was signed by the possessor of the private key, as only the corresponding public key could successfully decrypt the signature, thereby verifying both the message's integrity and the sender's identity. Now we arrive to where I wanted to come to. Once again, there exists different digital signatures schemes, but in this project we're using BLS signatures. It's a signature scheme with aggregation properties. A collection of n signatures $\{\sigma_1, \ldots, \sigma_n\}$ corresponding to signed messages $\{m_1, \ldots, m_n\}$ can be aggregated into a single signature σ_{agg}. Aggregation can also be done on public keys and private keys. Now, instead of verifying n signatures, it can become as easy as verifying a single signature. The use of aggregation significantly reduces the storage requirements and the bandwidth.

4 Methodology

The project was broken down into several key phases, each building upon the previous one. The initial phase involved designing and implementing the cryptographic component and the batch processing logic of the broker. Following this, I tackled the networking aspect, establishing communication between the broker, clients, and the server. Finally, the last stage involved integrating all these components– blending the cryptography and network functionalities into a cohesive and fully functional system.

Given that my research centered around implementing a piece of software, it was inherently hands-on and practical. Each step of the project entailed several iterative cycles: developing a first version (V1), running tests and simulations in realistic conditions, analysing the results and refining the software based on findings. Therefore, the analysis and findings were not singular but rather emerged progressively, shaping the project's evolution over time.

4.1 Batch Processing Logic and Cryptographic Components

This is the technical section, so I'll aim to explain it without going too deep in the details. Should you want to delve into the technical side, I encourage you to visit the first link in the Reference section.

One of the goal of Chop Chop is to amortize the cost of **authentication** and **deduplication** of batching. As seen in the background section of this paper, digital signatures help us verify both the message's integrity and the sender identity at the same time. With classic authentication, each client authenticates their message with a digital signature. The batch that is sent to the server contains payloads where each payload contains a public key, a sequence number, a signature and a message. When the server receives the batch, it must verify each payload individually and ignores incorrectly authenticated messages. This method is inefficient. Distillation introduces a new batching approach that leverages the power of multi-signature schemes to reduce verification complexity. Instead of verifying multiple individual signatures, the server only needs to verify one aggregate signature in constant time. The process works as follow: when the broker receives enough packets from clients, it constructs a Merkle tree of the batch. The broker then sends each client a proof of inclusion and the root of the tree, allowing them to verify that the their message is correctly included in the tree. Once clients verify their inclusion, they sign the root of the tree and return their signatures to the broker. The broker then aggregates these individuals signatures into a single aggregate signature and sends it the server. On the server side, knowing which clients participated in the batch it can construct an aggregate public key to verify the aggregate signature.

At the heart of the distillation phase lies the batch processing logic, which manages the interactions between the client and the brokers. The logic relies on several foundational elements, such as Merkle trees. To facilitate this, I organized the project into distinct, modular components, each designed to handle a specific functionality. The batch processing logic is in the file *src/batch.rs*. The overall structure of the project is illustrated below:

```
├── bin/
│   ├── server.rs
│   └── client.rs
├── src/
│   ├── lib.rs
│   ├── batch.rs
│   ├── merkle.rs
│   ├── signaturetree.rs
│   └── recvmessage.rs
└── benchmark/
    └── mybenchmark.rs
```

For the batch processing logic, we decided to represent it as a Finite State Machine (FSM) with three distinct states:

1. **Batch in Construction**: This represents the initial state, where a placeholder batch temporarily stores incoming payloads. The batch transitions to the next state when one of two conditions is satisfied: either the batch is filled to its limit of 65,536 payloads or the timeout period has expired. It this batch, we essentially store the payloads, client addresses and clients IDs.

2. **Batch Proposal**: This represents the intermediary state, where a Merkle tree is constructed from the payloads of all clients. After constructing the tree, inclusion proofs are sent to each client, awaiting their signatures. The batch transitions to the next state when all signatures are collected or when the timeout (initiated after sending all the proofs) expires. During this state, a bitmap is stored to track client responses (sequence of 0s and 1s where '0' indicates no response and '1' indicates a received signature).

3. **Distilled Batch**: This represents the final stage before submission to the servers. Using the collected signatures, we build an aggregate signature, which serves as the final output.

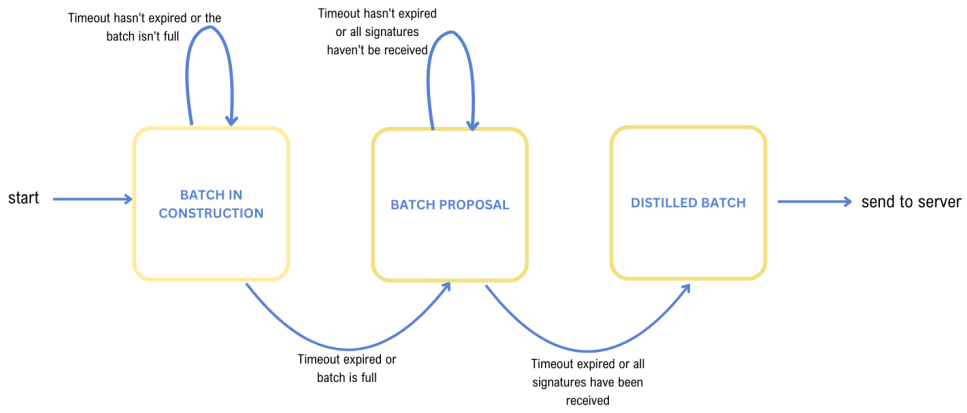

Figure 4: Finite State Machine representing the batch logic

We decided to have a dedicated thread handle this logic because of its complexity.

4.2 Networking Component

Now the question is how do we connect the clients, broker and servers together. We had two transport layer protocol options and ultimately chose TCP (Transmission Control Protocol) over UDP (User Datagram Protocol). For the reasons behind this choice, I recommend checking the Challenges and Findings section. One major advantage of UDP over TCP is its compatibility with the *sendmmsg* (send many messages) and *recvmmsg* (receive many messages) system calls, which enable batch sending and receiving of network packets. To understand why it's beneficial, let's break down what normally happens: each time a program wants to send a UDP packet, it has to do a context switch (transition from user space to kernel space). But each of this operations is expensive. By batching multiple packets into one single system call, *sendmmsg* reduces the non-trivial overhead associated to multiple user-to-kernel space transitions.

However, there's a caveat, in Rust's *std::net*, the module in the standard library that provides network functionalities, *sendmmsg* and *recvmmsg* are not directly available. We had to implement it ourselves, which required using the *libc* crate in Rust. This crate provides a way to access low-level system calls from the C standard library. The function signature of *sendmmsg* is the following:

```
int sendmmsg(int sockfd, struct mmsghdr *msgvec, unsigned int vlen, int flags);
```

It takes a *struct mmsghdr *msgvec* as an argument, which is also not directly available in Rust's standard library. To address these issues, we created a file named *src/recvmessage.rs* which contains all the definitions, data structures and implementations needed to use *sendmmsg* and *recvmmsg* efficiently.

Finally, since it's an interactive protocol, both clients and brokers send and receive messages. We decided to separate the sending and receiving operations in two different threads for several reasons. Firstly, this separation simplifies the overall design and makes the code more maintainable. Each thread can be updated, tested and developed separately. Secondly, from a performance standpoint, this approach allows us to fully leverage multiple CPU cores and optimize resource usage. While a thread may be blocked waiting for incoming messages, the other can continue sending messages, preventing unnecessary idling. Finally, this design is built with scalability in mind. If the application needs to handle a higher volume of messages, mores instances of the sender and receiver threads can be spawned.

4.3 Putting the pieces together

Let's walk through the architecture of our multi-threaded system. Our system is composed of three distinct threads, each responsible for a specific task:

1. The Receiver Thread. This thread is dedicated to incoming data.
2. The Worker Thread. This is our processing powerhouse, where the behind-the-scenes of the distillation phase happen.

3. The Sender Thread. Once the data has been processed, this thread takes over. It's responsible for transmitting the processed data either to the clients or the servers.

   But how do we connect all of these components together? We decided to use MPSC (Multi Producer, Single Consumer) channels. Channels offer a simpler but thread-safe communication while avoiding the hassle of shared mutable states and all the problems which come along (race conditions, contentions...) . You can visualize a channel as a pipe. It has two ends: a producer and a consumer. The producer sends data into the channel, and the consumer receives it.

Figure 5: Channel programming pattern used for the broker

5 Challenges and Findings

In the initial draft of the prototype, we intended to use TCP (Transmission control protocol) for communication between the broker and clients. However, simulations revealed that this setup did not meet our performance goals.

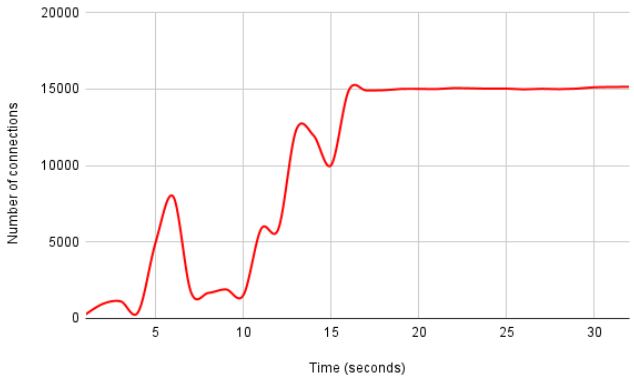

Figure 6: This graph displays the average number of TCP connections established per second, based on the mean from three test runs.

As illustrated in the graph above, the number of TCP connections established per second is limited to approximately 15,000. This means that our broker could only handle at most 15,000 packets per seconds, with each connection corresponding to a client sending data. This is significantly below our target of the broker processing a batch of 65,536 per second. This lower performance was somewhat expected and can be explained with the overhead associated to TCP. Indeed, TCP is a connection-oriented protocol, which means it requires establishing a dedicated connection between the two hosts before any data can begin. This connection is established through a process known as the three-way handshake. It involves three steps: the sender sends a request (SYN), the receiver acknowledges and responds (SYN-ACK), and the sender confirms (ACK). This initial setup introduces a delay of at least one round-trip time (RTT) - the time it takes for a message to go from sender to receiver and back. In addition, TCP offers reliable data delivery through mechanisms like acknowledgements, timeouts, transmissions, and congestion control, which add overhead. We decided going forward to switch to a connectionless protocol like UDP (User Datagram Protocol). However, while UDP eliminates the connection setup delay and reduces some overhead, it lacks TCP's delivery guarantees and reliability features.

Packet loss –when transmitted data fail to reach their destination– was our next most significant challenge. During simulations in realistic conditions, we consistently lost 5-10%, which is well over the 1% threshold we fixed ourselves. Our investigation into this issue revealed two main contributing factors, for which we subsequently developed effective solutions, as outlined below:

- Rate Limiter

  During the simulations, the client was transmitting data at a rate of 84000 packets per second (pkts/s), placing the broker under constant strain. Large, uncontrolled bursts of packets can quickly overwhelm both network and system resources, leading to congestion, packet loss and eventually system crash. To solve this issue we implemented a rate limiter (mechanism to control and limit the packet flow) for the client. One key component of the rate limiter is the burst control, which prevents excessively large bursts. Instead of allowing sporadic and high-volume transmissions, it encourages a smoother and more predictable traffic pattern. It helps maintain a consistent throughput and reduces the risk of having congestion-related issues. We found that we obtained the best results with a small burst size, at around 100 packets.

- CPU Pinning

  We discovered that the c6a.xlarge EC2 instances we were utilizing had hyper-threading enabled. Hyper-threading is a technology developed by Intel that allows a single physical core to behave like two separate logical cores, allowing the CPU to process two tasks (threads) simultaneously. In our situation, the kernel assigned both the receiving and sending tasks on the same physical core, leading to contention. This meant that these two resource-intensive threads competed for the same resources (CPU time, execution units and cache memory), preventing them from running efficiently as they continuously interrupted each other. We solved this problem by pinning the threads to separate physical cores. CPU pinning ensures that the scheduler of the operating system always executes a process/thread on the designated core. This is particularly useful for isolating workloads in high-performance applications. For this we used the rust external crate *core_affinity*.

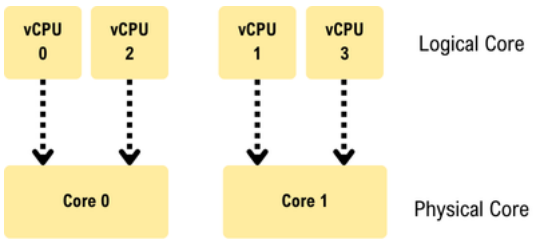

Figure 7: AWS c6a.xlarge Instance CPU Topology description

6 Conclusion

In the end, we successfully developed a broker for Chop Chop. We have managed to reduce packet loss to around 1% and satisfy the performance specifications. This code will later be integrated with the implementation of the servers and the clients to complete the broader system architecture. However, the broker's code still requires some further work to comply to the guidelines outlined in *NASA's Power of Ten*. The top priority being the creation of a safe abstraction layer for unsafe functions and handling memory allocation with a custom allocator.

7 Acknowledgements

First of all, I would like to express my sincere gratitude to my supervisor, Gauthier Voron, for his invaluable support, guidance and mentorship throughout my summer research internship. His exceptional pedagogy, dedication and constructive feedback were instrumental in the success of the project. I'm thrilled to announce that I will be continuing this project under his supervision for my bachelor's thesis. I would also like to express my gratitude to the Distributed Computing Lab (DCL) for accepting to host me this summer. Finally, I want to acknowledge the Laidlaw Foundation and the EPFL Laidlaw program coordinators for offering this opportunity.

References

[1] Camaioni, M., Guerraoui, R., Monti, M., Roman, P.-L., Vidigueira, M., Voron, G. (2023). Chop Chop: Byzantine Atomic Broadcast to the Network Limit. https://arxiv.org/pdf/2304.07081