What if Atomic Broadcast Could be as Fast as a Simple Broadcast?

Making It a Reality with Chop Chop: A Robust Cryptographic Broker Implementation

Ilan Nissim, École Polytechnique Fédérale de Lausanne (EPFL) - Distributed Computing Laboratory (DCL)

Introduction

- **State Machine Replication** (SMR) enables **fault-tolerant** services by duplicating servers and keeping them in sync.
- Fault tolerance is the ability of a system to operate without disruption when some of its components fail.
- SMR uses Atomic Broadcast, a consensus mechanism, to ensure all server copies agree on their state, by ordering, verifying and deduplicating messages from clients.
- However, current state-of-the-art Atomic Broadcast implementations still fall short in real-world situations, limiting their practical utility.
- **Chop Chop**, a system developed by the DCL, accelerates Atomic Broadcast by 100x with an interactive protocol involving brokers and a new form of batching: **distillation**.
- Distilled batches contain pre-processed information, making it much faster to authenticate and deduplicate messages in bulk, thereby amortizing the cost of Atomic Broadcast.

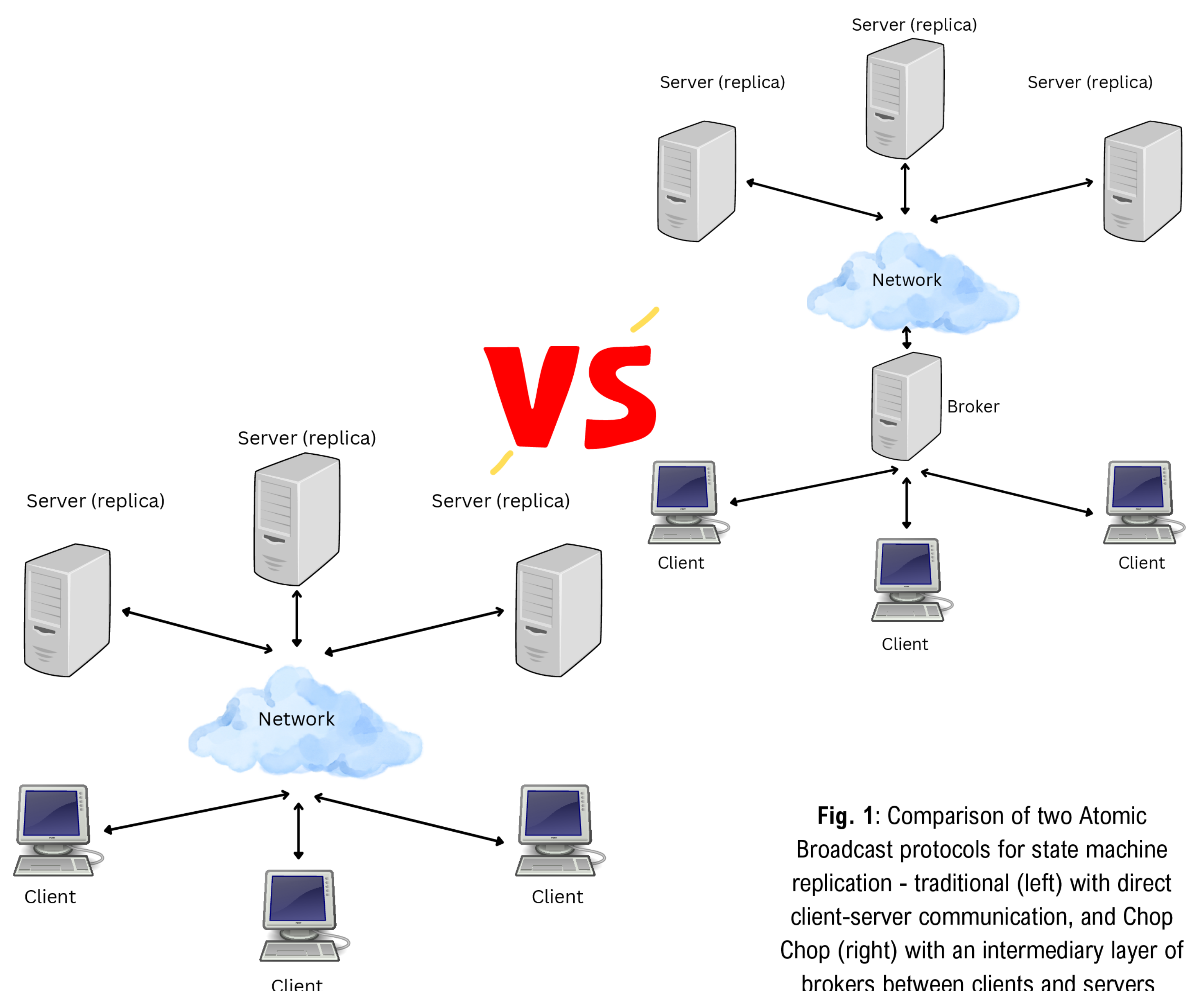


Fig. 1: Comparison of two Atomic Broadcast protocols for state machine replication - traditional (left) with direct client-server communication, and Chop Chop (right) with an intermediary layer of brokers between clients and servers

Objectives & Requirements

- I was tasked to implement a prototype broker designed for deployment on resource-constrained machines (4 vCPU and 8 GB of RAM)
- Follow NASA's Power of Ten -- Rules for Developing Safety Critical Code
- Key target specifications for broker : handle 84000 packets/sec, assemble batches of 65536 payloads/sec, with packet sizes of 124 bytes for incoming data and 514 bytes for outgoing data

Approach

1 Batch Logic / Cryptographic component

The first phase involved designing and implementing the cryptographic components and the batch processing logic for the broker. This is the backbone of the distillation protocol.

Link to GitHub repository

Link to original paper paper for technical details

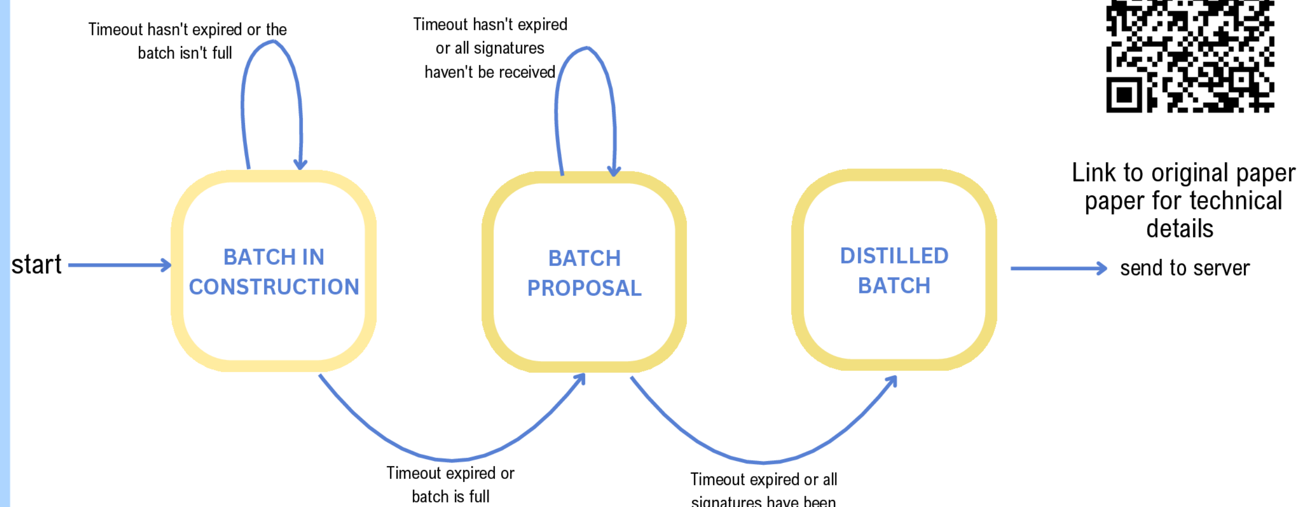


Fig. 2: Finite State Machine representing the batch logic

2 Networking Component

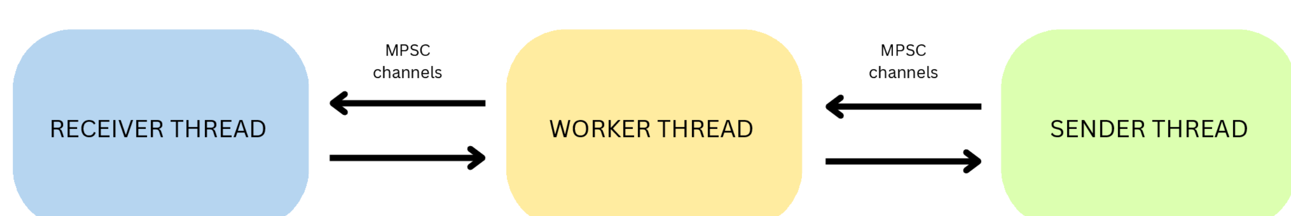
The second phase involved the networking aspect, establishing communication between the broker, clients, and server. It focused on efficiently managing large volumes of incoming and outgoing data.

3 Putting the pieces together

The last stage involved integrating all of the components. We decided to go with a multi-threaded architecture with 3 distinct threads:

1. Receiver Thread: Manages incoming data from either brokers or clients.
2. Worker Thread: The processing powerhouse, where the behind-the-scenes of the distillation phase happen.
3. Sender Thread: Responsible for transmitting data to clients and server.

To link everything, we used MPSC (Multi-Producer, Single-Consumer) channels, ensuring thread-safe communication without dealing with shared mutable state.



Challenges & Findings

Each step of the project was rigorously tested and benchmarked under realistic conditions using AWS EC2 c6a.xlarge instances. The networking component consumed the majority of our time, with packet loss --when transmitted data packets fail to reach their destination-- being our most significant challenge. We consistently lost ~ 5-10% of packets, which is well over the <1% threshold required for reliable operation.

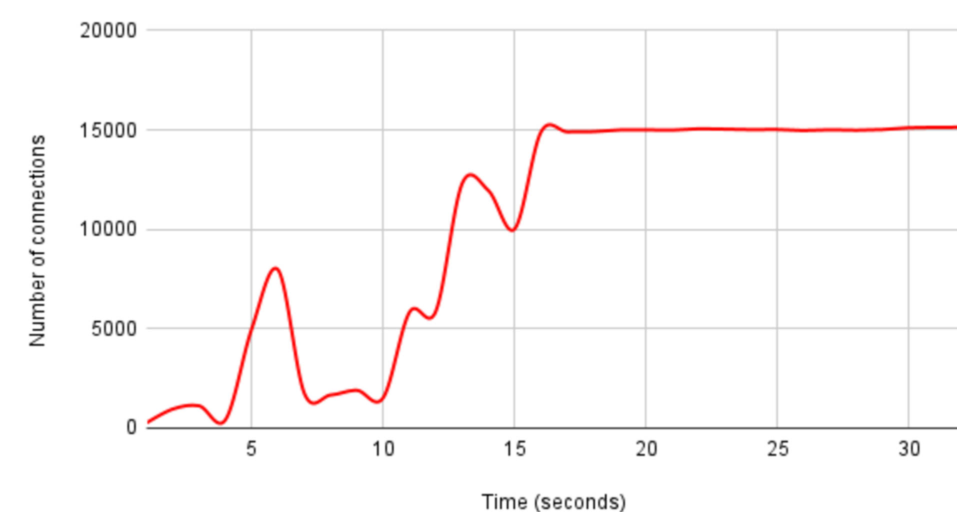


Fig. 3: Average number of TCP connections established per second, based on the mean of three test runs

Early simulations revealed that TCP for communication between clients and broker would not meet our performance goals, leading us to switch to UDP instead.

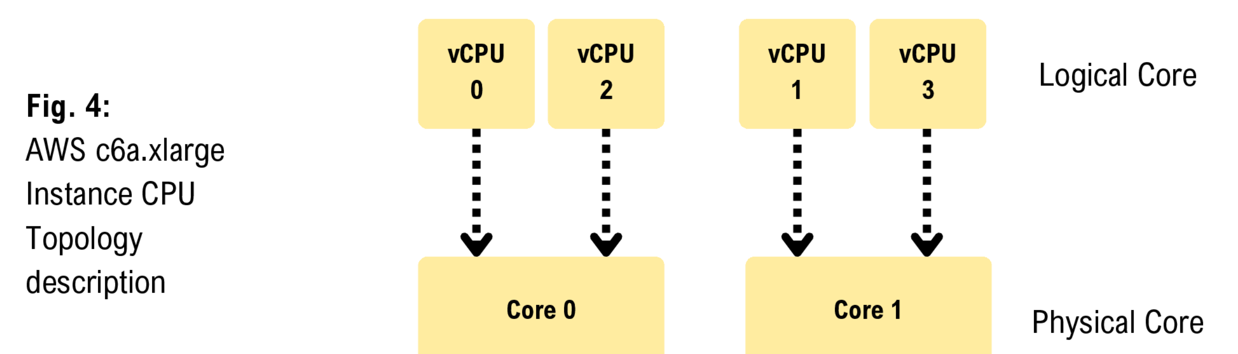


Fig. 4: AWS c6a.xlarge Instance CPU Topology description

We saw a major reduction in packet loss after identifying that the c6a.xlarge instance was using hyper-threading, which caused the kernel to assign both sending and receiving tasks both to the same physical core, leading to contention. We fixed this problem by manually pinning the threads to separate physical cores.

Conclusion & Next Steps

In the end, we successfully developed a functional implementation of a broker for Chop Chop. This code will later be integrated with the implementation of the servers to complete the broader system architecture.

The broker's code still requires further work, with the top priority being the creation of a safe abstraction layer for unsafe functions and handling memory allocations using a custom allocator