Design Exercise 3
Engineering Notebook

Adarsh Hiremath and Andrew Palacci

April 10, 2023

Introduction

In this design exercise, we re-implemented the chat application from design exercise 1 so that our system is both persistent and two-fault tolerant. Our implementation takes a modified client multicasting approach. Instead of the client establishing a single socket connection to the primary server, it establishes connections to all three servers in the architecture. Before sending requests to the server, we pulse all of the socket connections and pick a live socket connection to act as our "primary" server for that iteration. We only receive and display responses from this "primary" server, but we broadcast our requests to all live replicas in order to maintain a consistent snapshot across replicas. Finally, we implemented persistence by using CSVs to more permanently store the information in our data structures from design exercises 1. Our code can be found here.

Replication & 2-Fault Tolerance

Instead of implementing a more complicated approach involving Paxos or leader election, we opted to handle the two-fault tolerance from the client side. Running the client code establishes web socket connections to all three servers in our architecture. However, communication is only done through a single web socket.

Before a client makes a request to the server, we call a function that iterates through each of the web socket connections between the client and the server, only returning the web socket connections that are alive. We do this by attempting to send a byte through each of the web socket connections using the socket library's native send method. The send method returns an error if the socket connection is dead, so we can detect this using a simple try-catch block.

In the event that a server dies and the client makes a request, our method will iterate through each of the server connections, attempt to send a byte through each of them, and detect that a server has died. Since our implementation has three total servers but only needs one functioning server for the client chat application, it is two-fault tolerant. All but one of the three servers can be dead and the app will still work.

Since we are given a guarantee that the client will remain alive, we can simply pick the first alive server to communicate with via a socket connection. In the event that all the servers are being rebooted, our persistence implementation will ensure that the client chat application functions properly.

We simulated server disk writes using CSV files for our persistence implementation. For each server, we maintain two output CSVs: {port}.csv and {port}Users.csv. In port {port}.csv, we store the pending messages queue so that each line has a recipient and message to be delivered. We iterate through these lines when pickling and unpickling our data. Finally, for {port}Users.csv, each line is simply an account name, and we iterate through these to form a list of accounts. This implementation has persistence of both accounts and messages being sent to each account; even if a server dies, the pending messages queue (with data about the accounts the messages are being sent to) is not lost.

Our approach has benefits and disadvantages. Here are some of the benefits:

- Scalability: alternative approaches of client multicasting would involve the client sending requests to all of the servers simultaneously. Then, all of the servers would forward the message to the appropriate recipient client. The recipient client would then preserve only one of these forwarded messages from the servers. However, this is highly inefficient because there are three times as many operations being done compared to our approach. While it may be fine for smaller scale distributed systems, this system is not scalable because it is very plausible that a client would be overburdened with inbound from a singular server and all of its backups.
- Code modifications: as detailed in the section about code modifications from design project 1, there was very little handling to be done on the server side other than abstracting the server implementation into a class. Most of the modifications can be done on the client side, which makes our implementation far less complex than doing something like leader elections.
- Compute usage: our approach requires just three

servers to maintain two-fault tolerance. Generally, to achieve k fault tolerance, you need $2k + 1$ servers. All the methods below require a minimum of $2k + 1$ servers to function and so do other methods like erasure coding. Here, to achieve k fault tolerance, you just need 2 servers. In a scenario where you are paying for compute usage by the hour, unnecessary server costs can add up to make a very expensive distributed system.

However, our system does also have several crucial disadvantages:

- Practicality: in a real-world scenario, it's unlikely that clients would have information about any of the replica servers to ping them. Instead, clients would likely only have information about the primary server. Given that we are implementing this design project using TCP sockets, we chose to ignore this constraint since web socket connections would have to be established between the client and the backup servers regardless.
- Data loss: even with our two-fault tolerance and presistence implementations, there is the potential for data loss. If a message is sent to a server and the server crashes before it can forward the message to the recipient, the message will not be delivered. This can be solved with a simple acknowledgement system. While we could have implemented this, we learned it was okay for messages to be lost in this way for this design project with input from Varun in the walkthrough.
- Performance bottlenecks: only one server handles all correspondence with clients, failing to leverage the extra power of a distributed system.

Persistence

Alongside making our system replicated as detailed in the previous section, we also made our system persistent — when messages are queued to be sent to a given user, they are not lost even if the system goes down and comes back up. We implemented this by writing to memory on the disk, instead of our server processes' memory — put plainly, we stored information in a CSV file as opposed to only a Python dictionary or list. We saved both the message queue (in *PORT*.csv files) and the list of available accounts (in *PORT*users.csv files), using two methods in *server.py*: save() and unpack(). The save() method writes to these CSV files and is called at a specified interval (currently set to once a second), such that we can be guaranteed to have a recent snapshot of the message queue and available accounts. On the other hand, the unpack() method is called only when initializing a server, since this is the only time that a server might be locally unaware of what is in its message queue. This method takes the CSV files and loads them back into Python data structures, allowing the servers to "remember" which messages were queued even if they crashed/shut down.

In terms of design choices here, we firstly chose CSVs because of ease of use and access — almost everyone understands how CSVs work and how to parse them, and they are easy to store and manage. Second, we chose to save() at a specified interval as opposed to for every action in order to prevent a client from spamming messages, and then causing extreme delays as a result of the time it takes to save to a CSV. Additionally, the more intensive process of unpacking a CSV file's data into either a list or dictionary is performed infrequently — only when a system goes down and comes back up does it need to fetch CSV data, since otherwise the CSVs will match their respective data structures.

Additionally, we benefit from the broadcasting framework detailed earlier because this does not necessitate inter-replica communication. This is because all active replicas (eventually) will have the same snapshot of the message queue and accounts, since they are all receiving the same information from each client. This hinges on the assumption that crashed servers will not recover/reboot, as detailed in several Ed posts and the walkthrough for this design exercise. Of course, when forming a replicated system where each client only forms a connection with one other node, this would no longer be feasible since that node would need to update the others about

Comparisons With Other Approaches

In this section, we intend to compare and contrast two popular methods for implementing fault tolerance with our approach: Paxos and the primary-secondary model.

With Paxos, fault tolerance is achieved through consensus. In a Paxos-based system, the replicas play one or more of the following roles:

- Proposers: the entities that initiate requests to change the system's state. They propose a value to be agreed upon by the replicas.
- Acceptors: the core of the Paxos consensus algorithm. They vote on the proposed values and ultimately decide the agreed-upon value. The acceptors maintain a majority quorum, meaning a majority dictates the agreed upon value.

At an implementation level, the algorithm is both hard to grasp and requires substantial modifications to the design project 1 code since the consensus portion is brand new. Paxos is also far less efficient than our approach, since it requires multiple rounds of communication between the nodes (proposers, acceptors, and learners) to reach consensus. This can lead to increased message overhead and latency, especially in large-scale distributed systems or systems with high network latency.

Additionally, requires a majority of nodes to participate in the consensus process, which can limit its scalability. As the number of nodes increases, the required quorum size also increases, leading to higher message overhead and potential bottlenecks. This can make Paxos less suitable for very large-scale distributed systems. For our implementation, however, we just need to scale with one additional server and change nothing else about our code since the client just has to use a single functional web socket connection for correspondence.

With the primary-secondary model, fault tolerance is achieved through passive replication. The primary server processes client requests and updates the secondary servers with any changes to the system state. If the primary server fails, one of the secondary servers takes over as the new primary server, ensuring that the system remains operational.

Our implementation can be viewed as a slightly modified version of the primary-secondary model because we functionally designate the server that the client is currently communicating with as the "primary" server. Additionally, we do passive updates to ensure that data from the server currently being communicated with The issues with our implementation are also present in the pure implementation of the primary secondary model but to a greater extent:

- If the primary server crashes before it can relay an update to any of the secondary servers, data will be lost. This is magnified because the primary server is a single point of failure since the client cannot communicate with any of the replicas directly.
- There are performance bottlenecks since one server handles all communication, just like with our approach.
- The primary-secondary model relies on passive replication, resulting in a lag between the primary and secondary servers' states, leading to weaker consistency guarantees than consensus-based approaches like Paxos, which actively involve multiple servers in processing client requests.

Testing

One other important aspect of our codebase was testing — both in terms of unit testing and also verifying that our system was in fact two-fault tolerant. We split up tests on the basis of the 2 main goals of our specification — persistence and two-fault tolerance. In order to test that our persistence mechanisms were in fact working as we expected them to, we created a robust unit testing file, *tests.py*, that verifies the validity of both the save() and unpack() methods in saving and then reloading to/from CSV files. We made sure to test separately for accounts and the pending message queue.

Our fault tolerance testing came through running three servers, and then exiting two of the three servers while running a client. Each exit constitutes a "fault," and since our client was still able to send and receive communication from the server after these two faults, we determined that our system did indeed fit the specification. We implemented this test at the bottom of *server.py*, and performed it by running three simultaneous processes to each simulate one server, and then forcing one closed at a given interval.