Chamber Of Secrets: A Distributed Diary

Final Report

UBC CPSC 416 Spring 2018

Aleksandra Budkina (f1l0b, 21573150)
Graham Brown (c6y8, 33719121)
Harryson Hu (n5w8, 30552137)
Larissa Feng (l0j8, 29223112)
Sharon Yang (l5w8, 36268134)

1 INTRODUCTION

A distributed system contains multiple nodes that wish to do work on the same system. To maintain consistency of information, a distributed system must be able to share state. Should the state never change, sharing is trivial. The problem becomes more complex when the state can change. What if two nodes wish to write at the same time? What if a node disconnects from the network, rejoins, and attempts to write on top of an outdated value?

To resolve these problems for a logging system, a network must come to consensus over what values to read and write. To ensure no conflicts between nodes, the network can order independent writes and resolve conflicting writes. A read will relay the current value store of all previously agreed upon values at each node. For a new node joining the network, it will get the majority's current value store.

We built such a conflict resolution library system for Project 2. Under the hood, the library uses the Paxos Consensus Algorithm (Paxos Made Simple, L. Lamport, 2001) to resolve conflicts and ensure a consistent state across the network.

To demo our Paxos library, we built a diary application where users can read and write strings to a persistent, consistent log across application nodes. The library can support different applications with similar functionality, such as a distributed game issuing commands or distributed weather recordings.

1.1 Algorithm

The Paxos Consensus Algorithm (Paxos Made Simple, L. Lamport, 2001) is a consensus algorithm for a single value. It is suited to use in unreliable communication channels in applications which require fault tolerance and linearizability. The algorithm defines a peer-to-peer consensus protocol that is based on majority rule and ensures that one and only one result can be achieved. Each peer consists of a proposer, acceptor and learner.

The algorithm operates in the following **phases**:

Proposer Perspective

Phase One – A proposer selects a proposal number *n* and sends a *prepare* request with number *n* to all acceptors.

Phase Two – If the proposer receives a positive response to its *prepare* requests from a majority of acceptors, then it sends an *accept* request with a value *v* that it wants to propose to the entire network. If less than a majority accepts the *accept* request, then the proposal is invalid.

Phase Three – If the proposal is valid, the learners are notified and the value *v* is stored in the value store. Given this, the learner perspective is trivial.

Acceptor Perspective

Phase One – If an acceptor receives a *prepare* request with number *n*

- If *n* is greater than that of any *prepare* request to which it has already responded, then it responds to the request with a promise not to accept any more proposals numbered less than *n* and with the highest-numbered proposal (if any) that it has accepted.
- If the received *prepare* request has a number *n* less than another proposal value the acceptor has already promised to, then it will respond with the existing value for the *n* that the network has already come to consensus on.

Phase Two – If an acceptor receives an *accept* request for a proposal numbered *n,* it accepts the proposal, unless it has already responded to a *prepare* request having a number greater than *n.*

1.2 Assumptions

Our system makes the following assumptions:

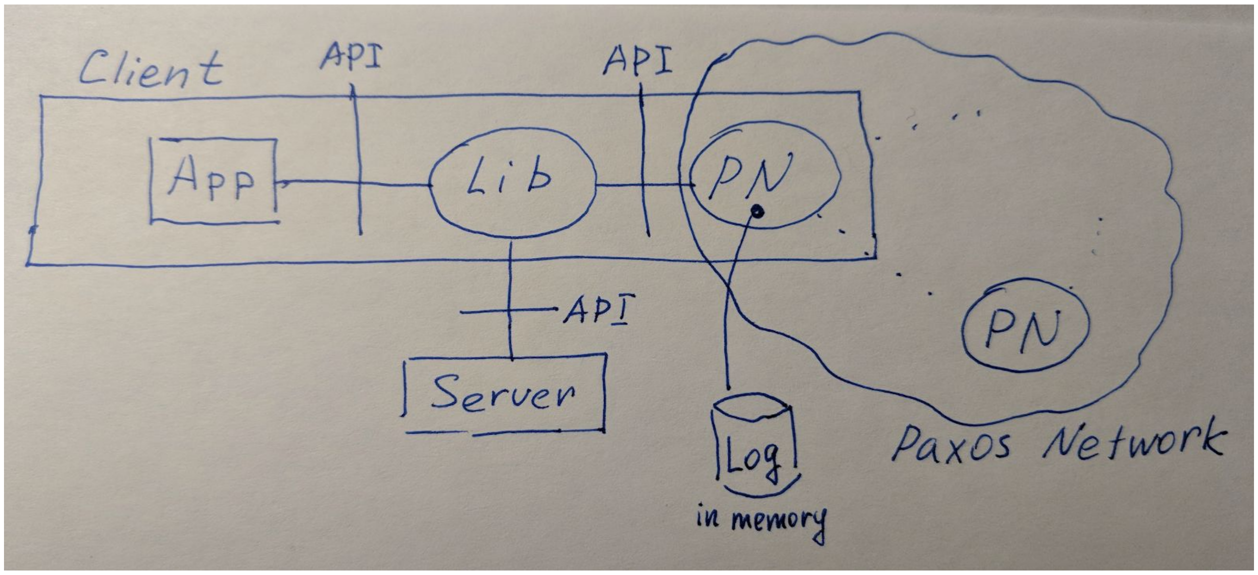
- There exists a single infallible server to handle user registration.
- Each Paxos node has a unique IP-port pair
- Messages do not get corrupted in transit between entities
- Messages are always delivered successfully between entities once sent.
- No nodes in the system are malicious.
- The frequency of writes is fairly low in comparison to the reads.
- Nodes can fail and rejoin after failing

Our system does not assume that:

- All nodes are synchronized
- Nodes cannot fail in the middle of the process
- Latency is zero

2 Our System

2.1 High Level Overview



The physical representation of our system consists of two components: server and clients. A client can be decomposed into: **the application and the Paxos Network (Paxos NW), consisting of Paxos Nodes (PN). These are joined by the Library.** The system starts with the initialization of the server. The users start a writing application which creates a Paxos Node via the Library. The application will read and write through the Paxos Node into and out of the Log, which is kept consistent across all users.

2.2 Component Details

Server

The server is a unique infallible entity. The server registers users and provides the addresses of other connected users. The server is initialized with its IP Address:Port pair. For each newly connected user, they get the list of all other connected users. The server receives heartbeats from every member of the system to keep the list of valid connections up-to-date.

Library

The Library joins the app and a PN. Upon initialization, the library creates a Paxos Node. Then it connects to the server and joins the Paxos Network, connecting to each peer. As requests come from the application, they are resolved via the Paxos Network.

Writing Application

The writing application is a command line application through which the user interacts with the system. It provides a basic interface where a user may read the current shared text state and write to the shared text state. It is initialized with the server's address and a port number. The application then makes requests to the library to satisfy the users' commands.

<u>Paxos Network</u>
Paxos Network is a collection of the Paxos Nodes each acting as peers. State is only persistent while at least one Paxos node is connected and online, since state is stored on each Paxos node.

<u>Paxos Node</u>
A Paxos Node executes the Paxos consensus algorithm. Each PN has three roles: Proposer, Acceptor, and Learner. All the PNs communicate with each other via RPC calls to create a Paxos Network.

While acting as a Proposer, the proposal number will be determined depending on the state on its local Acceptor side. If the Acceptor does not have any value it has accepted/promised, the Proposer will start with the proposal number: n = 1. Otherwise, the Proposer will generate a new number: n = last number at its Acceptor + 1.

2.3 System APIs

There are four types of interactions in our system.

1. Paxos Node to Paxos Node
    - This API includes setup of RPC connections, retrieving the current state of the diary, failure detection between PNs, communicating (sending and receiving) prepare requests, accept requests, and new values to be written that has been reached via consensus.
2. Application to Library
    - This API connects an app to a client instance to write messages to the network and retrieve the new updates.
3. Server to Library
    - This API registers the client to a PN and the PN to the network such that the PN can connect to other PNs and form a Paxos NW.
4. Library to Paxos Node
    - This API translates the requests from the application to the Paxos NW and retrieves the latest state of the log.

2.4 Handling Failures

<u>Paxos Node Failures</u>
PNs may fail at any time. Upon a fail, only its associated app is affected. We detect the failure of PNs through failed RPCs. For example, one PN will send its proposal out to all the other PNs using RPCs. For RPCs that fail to resolve, the caller will mark those associated PNs as failed. At the end of each Paxos round, each PN will remove failed PNs from its list of neighbours.

Each PN also sends heartbeats to the server. This ensures that the server always sends a list of connected PNs for a new PN to connect to upon registration.

<u>Common Failure Cases And Solutions</u>
1. PN fails while it is idle. It has not sent out any prepare request or accepted any other prepare requests
Solution: Other PNs remain at "waiting" state. No other PNs are affected by the failure of a particular PN in this state. Other PNs may continue to send out proposals.

2. PN fails after it has sent out a prepare request
Solution: For PNs that sent back a promise for this prepare request, it will time out waiting for an accept request. Eventually, these PNs will promise another prepare request with a higher proposal number.

3. PN fails after it has sent out a prepare request, received a majority of positive responses back, and sent out an accept request.
Solution: Our system is still able to learn about the consensus value because each PN's acceptor will send the accepted value to every PN's learner. Each learner will count the number of acceptances it has received and it will write the value to the log once it has reached a majority

4. The last PN leaves the network
Solution: The distributed log in our system is only maintained if there is at least one PN in the network. When the last PN leaves the network, any state of the log is lost, and the log will restart upon a new PN connects.

5. A majority of PNs fail during a Paxos round
Solution: Each live PN updates its neighbours by the failed neighbours protocol. The current round ends and the next begins. Majority threshold is updated given the live PNs.

6. A Node fails and reconnects when there are 3 or more peers present
Solution: If the node re-connects with the same port number, it restores the last proposed and last accepted values from its backup file and pulls a new log from the Paxos NW.

2.5 Design Rationale

We built a centralized server for our system to handle node registration, heartbeats, and sending out neighbour information. This reduces networking complexity and allows us to focus on building the Paxos algorithm.

We built a flexible library which combines the application, server, and the PN network. Applications may import the library to leverage Paxos. Examples include weather recordings and playing a distributed game.. The only change would be in the topology of the PN network. The level of abstraction introduced on the consensus library allows the Paxos NW to be independent from the type of data provided at the application level.

We opted to not have a single proposer serializing the proposals. This way, there is no single point of failure. Also, the main application of Paxos in our library is to agree on a value across nodes, rather than leader election.

We introduced a proposal TTL to handle conflicting proposals. Since every node can propose, two proposals may simultaneously circulate, leading to a race condition. This TTL is the number of times a proposer can generate a new proposal with a higher PSN before sleeping for a random amount of time. This spaces out proposals, allowing one round to complete in the case of a race condition.

We implemented a distributed diary application to demo our library because the read and write operations of a diary fit well with Paxos. The simplicity of the application allowed us to focus on the underlying Paxos. We initially considered Battleship (a game where players have to guess the location of their opponents' ships on the board) as our demo application, but instead opted for distributed diary as it did not have the unnecessary overhead associated with a game.

In our system, multiple rounds to agree on multiple values is necessary. The canonical way to solve this would be multi-Paxos, but this adds much more complexity than feasible. Instead, we introduced timeouts to handle progression of rounds. Whenever a node is waits for a majority of peers to respond, we have a timeout. Since we assume messages are always delivered, we allocate a generous timeout, and assume we will get a response in that time. As a result, we can support multiple rounds, as the timeout allows functionality under almost all situations.

Working Parts Of Our System

- Server. The server will remove the Client from its list of active Clients when a Client fails. The server tracks heartbeats from clients to ensure that it does not send "stale" (failed) neighbours to newly joining Clients.
- Client. The client runs at the application level. It connects to the server and gets the addresses of all other clients. It opens a RPC connection with each of the clients. Each Client sends heartbeats to the server to signal that it is still alive.
- Proposer, Acceptor, Learner roles. The proposer creates prepare/accept requests. The acceptor responds to prepare/accept requests, notifies learners of accepted values, and restores its state after disconnections. The learner writes the consensus value to the log.
- The PN is the main workhorse of our system. The PN can establish RPC connections with neighbours, learn the initial state of the log, detect and delete failed neighbours, read, write, and disseminate prepare and accept requests to all neighbours. Also, it serves as an aggregator for Proposer, Acceptor, and Learner roles.
- Each PN receives a list of neighbours from its client upon connecting. A PN will open a RPC connection to every other PN in the network and will learn about the current state of the log through the learners of other PNs.
- Recalibrating the majority and failed neighbours protocol. Our Paxos network is resilient towards majority failure (the state of the log is kept as long as one node is still alive). If the proposer detects that a majority has failed, either through a timeout or a failed RPC call, it recalibrates its list of active neighbours by sending a notification to all PNs that are alive. We use a timeout to indicate a PN failure because we assume that a PN cannot be alive and never respond to messages. Other nodes, upon receiving this notification, contacts every neighbour from their list to determine which PNs failed. After recalibration, the Paxos network increments the round number across all PNs and continues with the new round.
- Multiple rounds of Paxos. One value is written in one round, and a round is over when a value is written to the learner's log. All PNs increment rounds independently, and the round number across the network is consistent.
- The state of the log is maintained as long as one PN remains connected to the network. If all PNs disconnect during a session, the state of the log during that session is lost.

- Each PN's acceptor saves the value that it last promised (last promised prepare request) and last accepted in a local file so that the PN will remember these values if it fails. If its failure did not result in a majority failure, it may continue with the acceptor's last available backup and a newly obtained log from other PNs upon reconnection.
- Front-end command-line UI that allows a user to join the network, read and write to the diary, and check if it's client is still alive. Further it allows the user to activate breakpoints and kill points in the paxos process, and produce a round report. Finally this interface also allows us to demo our underlying Paxos system.

Parts That Are Not Fully Working

- We didn't address transitive connections, where one PN may appear "failed" towards some other PNs, but appear alive for others. This is a complex task to handle in a short period of time, and we did chose not to do it.

2.6 Testing And Performance

The server is deployed on an Azure VM. We have applications and PNs on other Azure VMs in different regions. As we run the diary application, we read the contents stored at each PN to verify that the state of the diary is correct. We also have a test suite of different cases we walk through manually. These cover cases such as one PN read/write/alive/exit, multiple PNs read and write, and multiple PNs read/write with PNs disconnecting and new PNs connecting during a session.

By having our application VMs situated all around the world, our infrastructure implicitly tests dealing with (sometimes significant) network latency. This allows us to have extra confidence in our system fulfilling Paxos' promise of eventual consistency.

In our testing, we ran the server and 9 nodes locally, and had good performance with minimal latency when calling reads and writes. We also tested with 5 nodes on VMs in various regions, which slightly more latency, but performance was still good overall.

3 Challenges

A challenge we encountered was determining how to demo our underlying Paxos system. As mentioned previously, we considered building a Battleship game, using our Paxos system to determine the next position in the game to fire upon. However, we decided to focus more on building the library and decided on a simple diary application.

We had some difficulty designing and implementing our Paxos Node and the Paxos algorithm. We had differing views on how the different components interacted with each other, and how to make sure values were actually learned. We decided on acceptors contacting every learner and having learner count until it reached the majority threshold, as this reduced reliance on a single proposer. Furthermore, we did not fully grasp the concept of 'rounds' and what was necessary to implement them until later in development.

A big challenge was to address the main issue caused by the nature of the Paxos algorithm: liveness. Since Paxos can run forever if two proposers are competing, we introduced a TTL counter for the message to bounce around the network; when the counter reaches 0, the node sleeps for some time before proposing again. To test it, we set up several “sleeping” points in our PNs to emulate the race condition.

We also initially recalibrated the majority threshold of each round every time a PN failed. However, after discussing with Matthew, he recommended that we keep the majority threshold consistent throughout the round and only recalibrate at the beginning of a new round.

After meeting our initial MVP timeline, our velocity decreased as we were a bit stuck on figuring out the best ways to test the correctness of our system, implement corner cases, and fix bugs. We ran into a few bugs that were challenging to reproduce consistently and pinpoint the exact issues (we relied on print statements for those).

Towards the end of our project, another challenge was to cover most of the common failure cases for a distributed system. We addressed the failures that occurs in between Paxos stages, but it was more difficult to simulate all possible failure scenarios and to come up with the proper implementation for every case. One example is when a node fails at the time of re-calibrating the majority. We omitted this case due to time constraints and its low likelihood of occurring.

4 Conclusion

We discovered Paxos. Paxos was proposed. We came to consensus to implement Paxos. We implemented Paxos in a flexible library with debugging features. It is functionally multi-Paxos. We used Paxos in a distributed diary application to better focus on Paxos. We now know some more Paxos.