Project 2 Proposal
Chamber of Secrets: A Distributed Diary

UBC CPSC 416 Spring 2017

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

Introduction **2**
Challenge 2
Algorithm 2
Assumptions 3
The System **4**
High Level Overview 4
Component Details 4
Server 4
Library 4
Writing Application 5
Paxos Node 5
Paxos Network 5
System APIs 5
Handling Failure 5
Paxos Node Failures 5
Joins 8
Testing + Deployment to Azure 9
SWOT **9**
Timeline **10**
Future Extensions **11**

Introduction

A distributed system is composed of multiple nodes that wish to do work on the same system. In order to work from the same basis, a distributed system must be able to share state. Should the state never change, the sharing is trivial; you could use a flooding protocol to ensure every node has the same state.

The problem becomes more complex when the state is able to 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 What if a node disconnects from the network, rejoins, and attempts to write on top of an outdated value?

In order to resolve these problems, a network must come to consensus over what values to read and write. To ensure that there are no conflicts between nodes, the network nodes can combine the requests to read and write with a value store of all previously agreed upon values at each node.

We will build such a conflict resolution system for a distributed writing application for Project 2. Under the hood, we will use the Paxos Consensus Algorithm (Paxos Made Simple, L. Lamport, 2001) to resolve conflicts to ensure there is a consistent piece of prose across the network.

Our team will build a Paxos library and pair it with a user-friendly application in order to maintain a persistent, consistent log. The library will support different applications with similar functionality, such as a distributed diary or distributed weather recordings. For demonstration purposes, we will be building a distributed diary.

Challenge

The primary problems associated with a distributed writing application are handling:

- Node disconnections and rejoins
- Globally consistent state
- Multiple writers at the same time
- Simultaneous reads and writes

The main difficulties of the implementation are handling the disconnection of the members of the Paxos network at the different stages of the Paxos algorithm, and obtaining a consensus on the proposed entry without falling into a deadlock.

Algorithm

The Paxos Consensus Algorithm (Paxos Made Simple, L. Lamport, 2001) is a well-studied consensus algorithm to achieve consensus on 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 can ensure that one and only one resulting value can be achieved. Each peer consists of a proposer, acceptor and learner. The proposer posits a new value to be added to the value store.

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.*

Assumptions

We assume that

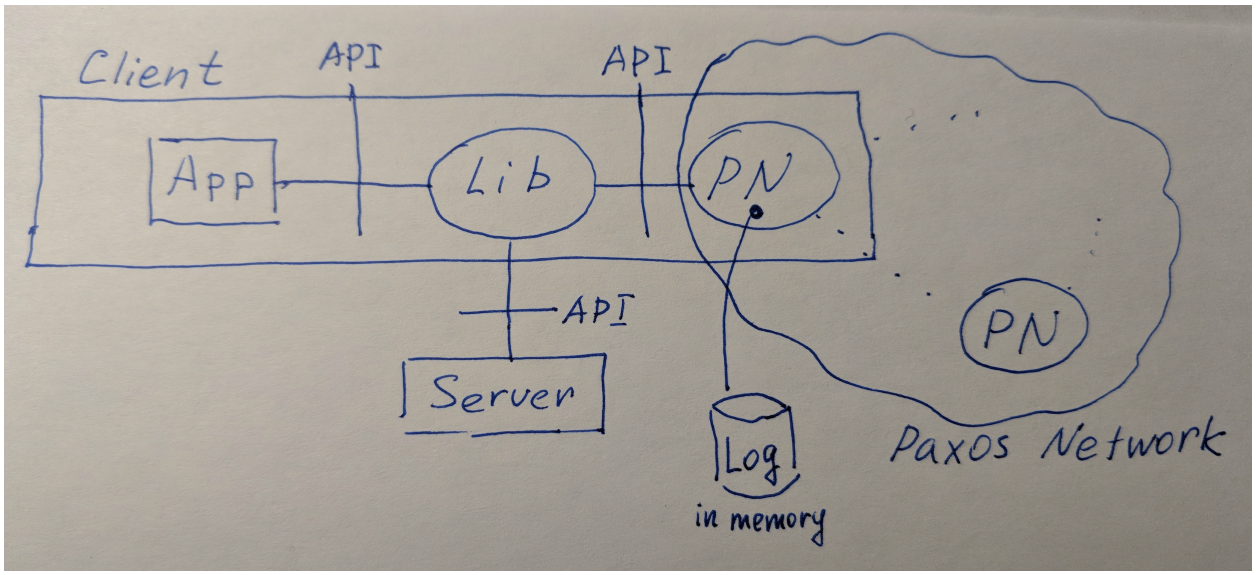
- There exists a single infallible server to handle user registration.
- Each Paxos node has a unique IP-port pair
- Messages do not get corrupted in transit between entities
- Messages are always delivered successfully between entities once sent.
- No nodes in the system are malicious.
- The frequency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

We do not assume that:

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

The System

High Level Overview



The physical representation of our system consists of two major components: server and clients. However, there are slightly different logical components to the system: **a server, the writing application, and the Paxos Network (Paxos NW), consisting of Paxos Nodes (PN). These are joined by the Library on the client side.** The system starts with the initialization of the server. The users start a writing application which is paired with a Paxos Node via Library. The application will write through the Paxos Node into the Log, which is kept consistent across all users.

Component Details

Server

The server is a unique infallible entity. The purpose of the server is to register users and provide the addresses of other components. 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 the heartbeats from every member of the system to keep the list of valid connections up-to-date.

Library

Joins together the app, PN, and server. Upon initialization, connects to the server and receives the list of other clients (and, therefore, PNs) addresses to connect with. Then it initializes a Paxos Node and transfers the list of the addresses to the PN. Library translates the requests from the app into the format which is accepted by the PN and vice versa, decodes the PN's output into the format suitable for the writing app.

Writing Application
The 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. The application will be implemented through a command line interface.

The writing application is initialized with an IP:Port pair, which is transferred to the library to connect to the server. The application then interacts with the Paxos NW through its Library's PN to read the shared text and write to the shared text.

Paxos Node
An entity which implements methods of the Paxos library 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 NW, a forum which comes to consensus on the next state of the shared text.

While acting as a Proposer, the proposal number will be detected depending on the state on the local Acceptor side. If the Acceptor hasn't any value it accepted/promised, the Proposer will start with the ID: n = 1. Otherwise, the Proposer will generate a new ID: n = last ID at Acceptor +1.

Paxos Network
Paxos Network is a cumulative notation of the Paxos Nodes acting together. State is only persistent while at least one Paxos node is connected and online, since state is stored on each Paxos node.

System APIs

There are four APIs in our system.

1. Paxos Node to Paxos Node
    - This includes setup of RPC connections, failure detection between nodes, communicating (sending and receiving) prepare requests, accept requests, and new state that has been reached via consensus.
2. Application to Client
    - This connects an app to a client instance to write messages to the network and retrieve the new updates.
3. Server to Client
    - This translates the requests from the application to the Paxos NW and retrieves the latest state of the log.
4. Client to Paxos Node
    - 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.

Handling Failure

Paxos Node Failures

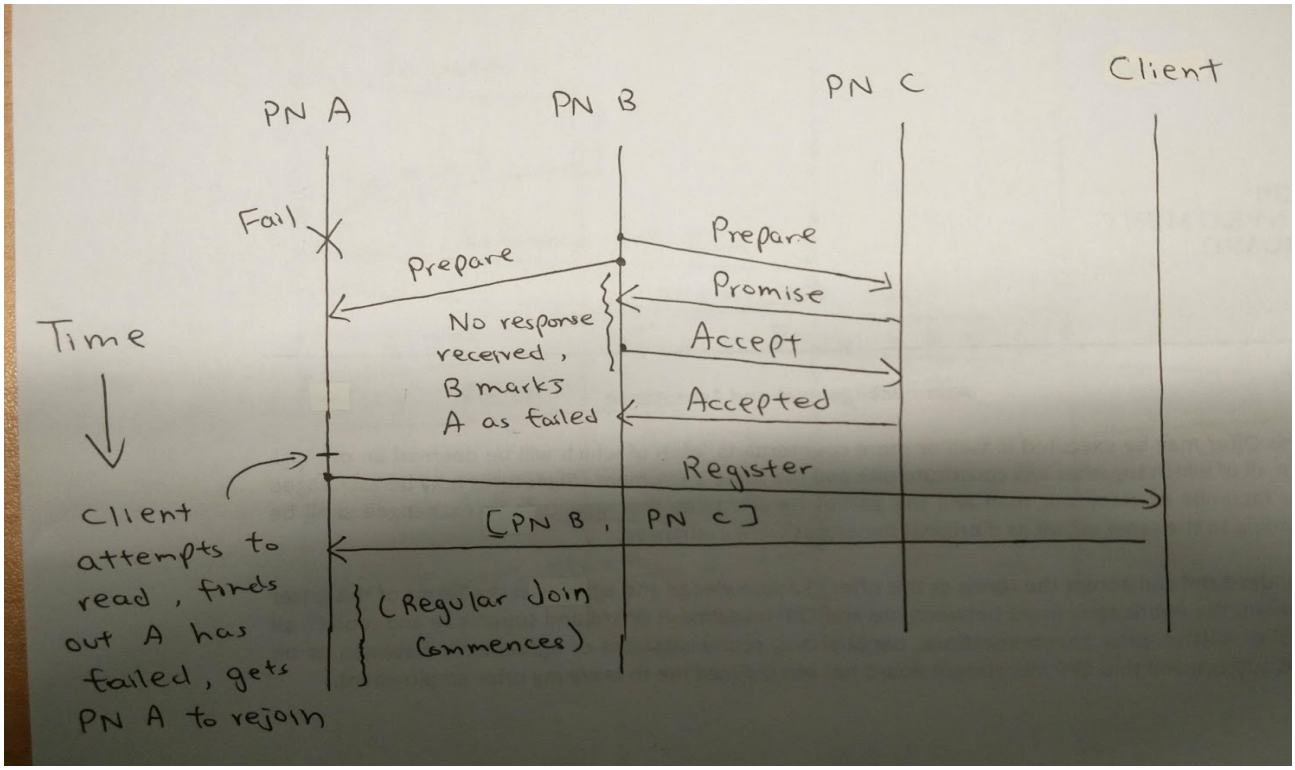
One main problem for our system will be handling failures of PNs (and therefore clients) because PNs may fail from the network at any time. We detect the failure of PNs through RPCs to other PNs that fail to return. For example, one PN will send its proposal out to all the other PNs using RPCs. For RPCs that fail

to return any value, the calling PN will mark those associated PNs as failed. However, a failing PN only should affect the app connected to it. It will not impact any other applications connected to other PNs.

Each PN will also be responsible for sending heartbeats to the server. This is to ensure that the server always sends a list of connected PNs for a new PN to connect to upon registration. The server should not send stale PNs that may have failed.

Failure Cases and State of Other PNs upon failure

1. PN fails while it is waiting for the next state, and it has not sent out any prepare request or accepted any other prepare requests
    a. Proposers: PNs remain at "waiting" state. No other PNs are affected by the failure of a particular PN in this state. PNs may continue to send out proposals.
    b. Acceptor: PNs may still continue to accept incoming proposals
    c. Learner: No change
    d. Client: The client connected to this failed PN will receive an error on the next attempt to send a "read/write" request. The client may then choose to have its PN rejoin the Paxos Network through the standard join workflow

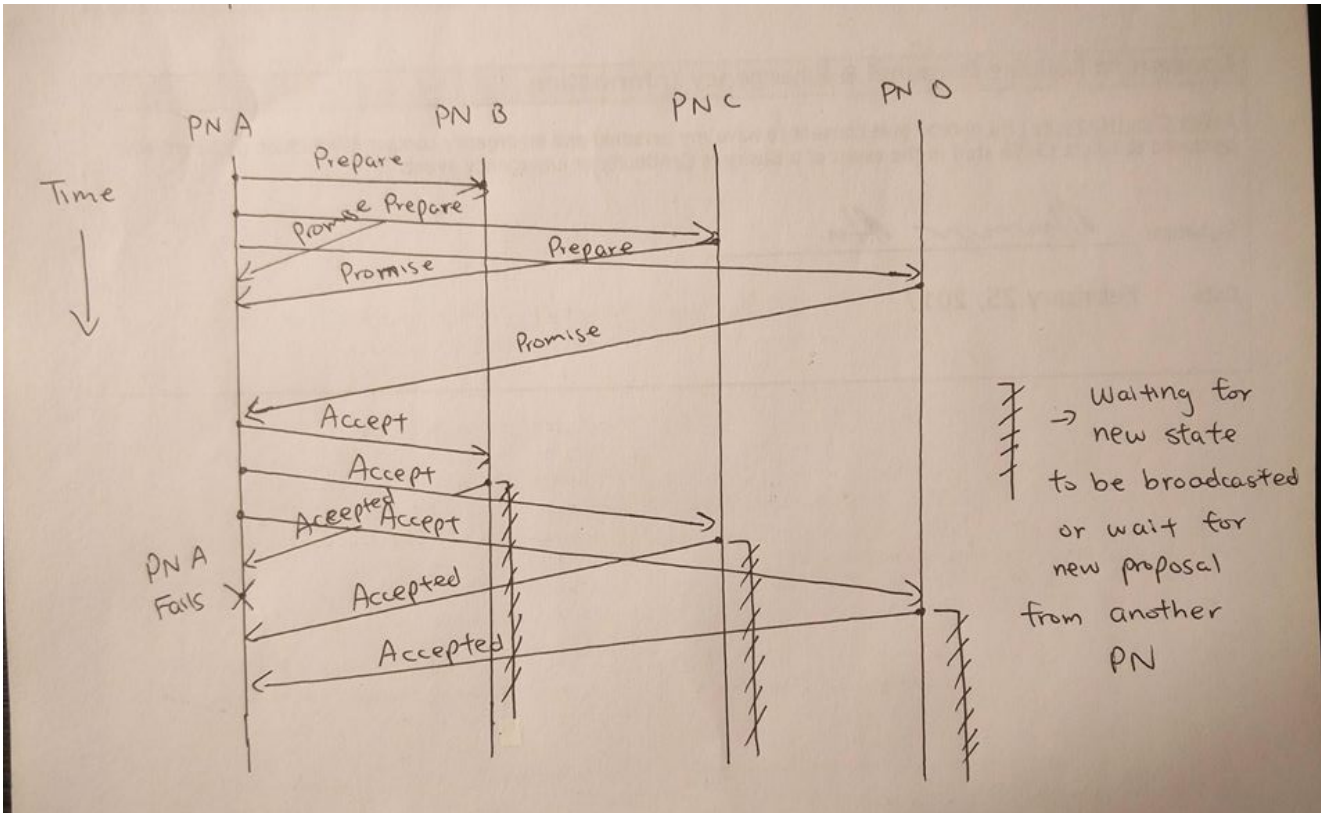


2. PN fails after it has sent out a prepare request
    a. Proposer: PNs are still able to send out a prepare request with a higher *n*
    b. Acceptor: PNs that have rejected this prepare request no longer care about this failed PN. For PNs that did accept this prepare request, the acceptors will wait around for an accept request, which will never come. Eventually, these PNs will accept another prepare request with a higher proposal number.
    c. Learner: No change

d. Client: The client connected to the failed PN will receive an error on the next attempt to send a "read/write" request. The client may then choose to have its PN rejoin the Paxos Network through the standard join workflow

3. PN fails after it has sent out a prepare request, received a majority of positive responses back, and sent out an accept request.

a. Proposer: PNs are still able to send out a prepare request with a higher *n*
b. Acceptor: PNs that have accepted the prior prepare request will accept the accept request proposal
c. Learner: The learners of these PNs will not learn about the majority value chosen because the PN that sent out the accept request failed. There will be no change in the distributed log in this case
d. Client: The client connected to the failed PN will receive an error on the next attempt to send a "read/write" request. The client may then choose to have its PN rejoin the Paxos Network through the standard join workflow



4. PN fails after broadcasting the new state reached via consensus from the rest of the Paxos network

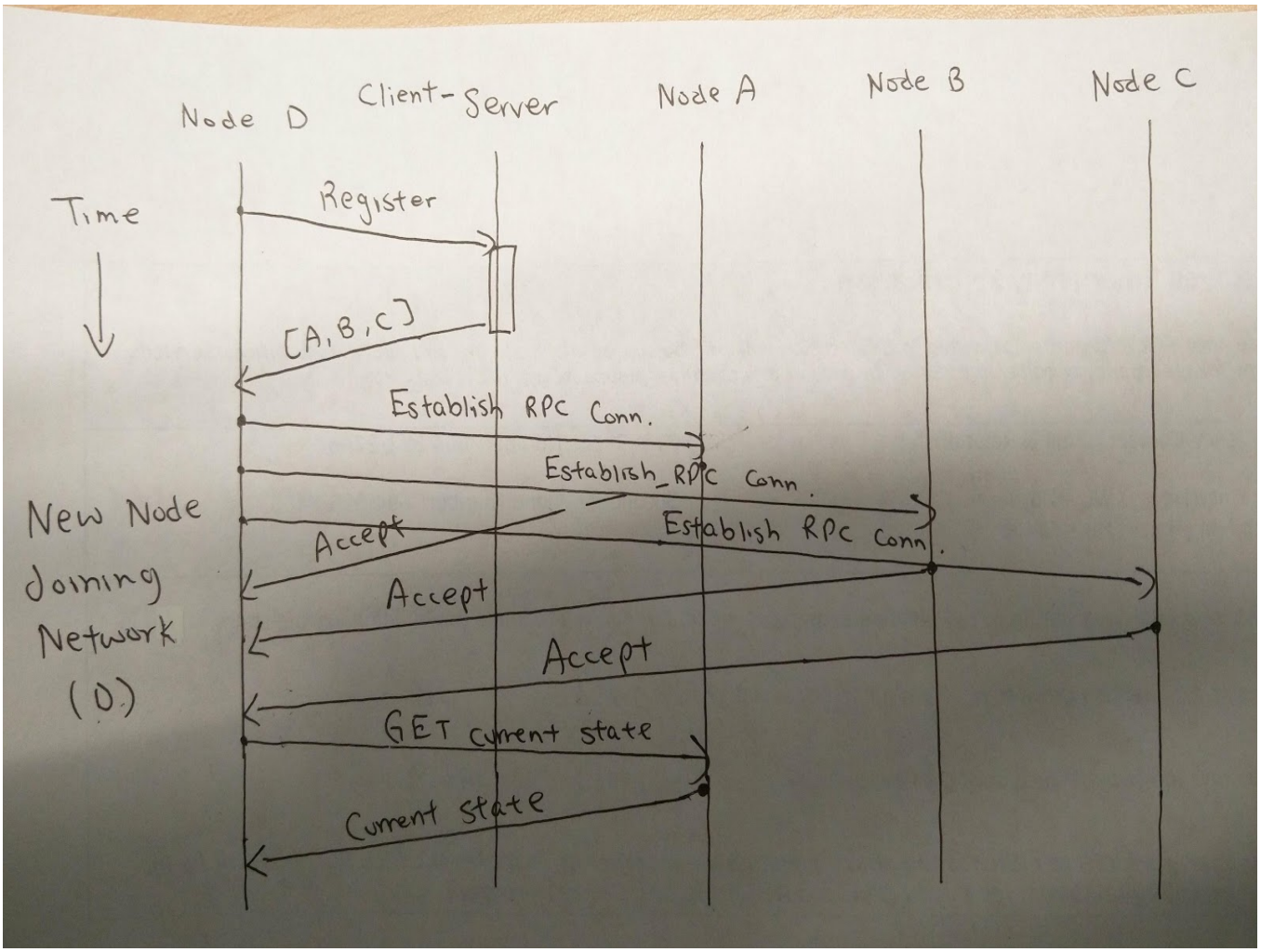
a. Proposer: PNs are still able to send out a prepare request with a higher *n*
b. Acceptor: No change. Still waiting for new proposals.
c. Learner: PNs will update its copy of the distributed log. They are not affected in any other way. When the failed PN eventually re-joins the network, it will get the most updated copy of the distributed log via the learners of other PNs.
d. Client: The client connected to this PN will receive an error on the next attempt to send a "read/write" request. The client may then choose to have its PN rejoin the Paxos Network through the standard join workflow

5. The last PN leaves the network
    a. Our assumption is that the distributed log 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 connecting.

Joins

1. New Client and PN joins the Paxos network
    a. Server: This new client will register itself with the server. The server will send this new client a list of the IPs and ports of the other clients-PNs in the network to connect to.
    b. Client-PN: The client will pass the list of IPs and ports to its PN, and this new PN connects to all the other PNs and obtains the latest copy of the distributed log from the the learners of each PN. It will set its copy of the distributed log based on the majority.

*(Minor Correction** This diagram should have Node D sending a get current state to all the other nodes, not just to Node A)*



Testing + Deployment to Azure

We will deploy the server on an Azure VM. We will also start an arbitrary number of additional Azure VMs, with each VM representing a Paxos node (minimum 4-5), that will form the Paxos network.

We will write sample diary applications that will connect to the Paxos network and send read/write requests. After running, we will check the contents of the diary or distributed log stored at each PN to verify that the state of the diary is correct.

More specific breakdown of parts to tests:

- A user is able to register itself with the server and receive a list of the other PNs that it needs to connect to
- A newly joined PN is able to set up RPC connections with all the other PNs in the Paxos network
- A PN is able to use RPC calls to send out its prepare requests, accept requests, and any state that has been reached by consensus
- A PN is able to receive from other PNs via RPC prepare requests, accept requests and any new state that has been reached by consensus

SWOT

Strengths	**Weaknesses**
- 4/5 group members have worked together previously on Project 1 - Larissa has devops experience - Alex manages our project from a higher-level perspective, identifies issues, and seeks out TAs and professor for help - Sharon is great at math and algorithms - Harryson writes clean code for other members to understand - Graham is great with structure, diagrams, and naming - Because we are focussing more on building a library that can be used by multiple applications, building a sample application to run on top of it should not be too difficult.	- Difficult for everyone to meet at the same time in-person - All group members are relatively new to Go - None of us have worked with Paxos before - We often have differing approaches to building the system (which could be a strength as well)

Opportunities	Threats
- We are learning Paxos in lecture soon - Create a simple to use application, which implies a simple demo - Azure provides lots of resources to set up game players + servers to test with - We are able to use 3rd party libraries during project implementation - Paxos is a well-researched topic, and there are lots of papers + resources online to learn from	- Paxos is a complex algorithm to implement - Less time spent on the project due to other commitment/courses - Because Paxos is an algorithm that runs "under-the-hood" of other applications, it may be difficult for us to sufficiently demonstrate our system.

Timeline

MVPv0: Single user which can join the network and write by itself
MVPv1: Two users which can join the network and be aware of each other. One reads, one writes.
MVPv2: Two users which can join the network, be aware of each other, and write to a shared log via Paxos.
MVPv3: Multiple users which can join the network, be aware of each other, write to a shared log, and handle failures and disconnects.

Date	Description	Preliminary Assignee
Mar 9th	Project Proposal due	All
Mar 12th	API Interfaces & Specifications	All
Mar 12th	Initial Server Implementation & Client-Server API/connection	Alex
Mar 17th	Implement Client - PN mounting	Sharon
Mar 17th	Implement PN-PN connection	Graham
Mar 17th	Handle PN-PN failures/disconnections	Harryson
Mar 20th	Write testing script	Larissa
Mar 20th	Implement the different parts of the paxos library*:*	All
-	- Proposer	Alex
-	- Acceptor	Sharon

-	- Learner	Larissa
Mar 20th	Implement entire Paxos Node behaviour (connect parts of the paxos library together)	Harryson
Mar 21st	Implement app & Implement App - Client mounting	Graham
Mar 22nd	Connect the entire system together, bug fixing	All
Mar 22nd	MVPv2 Due	All
Mar 23rd	Send email to assigned TA to schedule a meeting to discuss project status	All
Mar 29th	MVPv3 Due	All
Apr 4th	Demo Done and Automated	All
Apr 6th	Project Code Due	All
Apr 6th	Final Reports Due	All
Apr 9-20th?	Demo	All

Future Extensions

- Extract the paxos library into a separate library which will not depend on the top application
- Create clusters of logs, when clients are connected to the logs by the log names
- Handle the infinite proposal loop
- Implementation of disk storage for Client to go offline, come back, and to be able reconcile its log with the rest of the network
- EC1 [2% of final mark]: Add support for GoVector and ShiViz to your system. Generate comprehensible ShiViz diagrams that explain your distributed system data/control flow and protocol design. These diagrams/explanations must be in your final report and you must show a live demo (loading logs into ShiViz and generating and explaining the result). Store the logs for your diagrams in the report/demo in the report repository.
- EC2 [2% of final mark]: Demonstrate the likely correctness of your system by using the Dinv dynamic program analysis tool. You must generate at least 3 types of invariants that illustrate 3 different kinds of correctness conditions of your system. These must be listed and explained in the final report. The logs that lead to the properties you describe must be part of the report repository. You do not have to demo Dinv.