# Distributed System COMP90015 Project

Ishaan Patel (pateli@student.unimelb.edu.au)
Shajid Mohammad (shajidm@student.unimelb.edu.au)

## Abstract

This project proposes a custom server protocol which aims at providing maximum availability and consistency among the servers in presence of possible server failure and network partitioning.
We build upon the existing server protocol in order to promote flexibility in server integration, maintenance and thereby also ensure quality client-to-client communication.

## 1. Aims

We implement a server protocol that:

1. Allows servers to join the network at any time: even after some clients have joined.
2. Allows clients to join (register/login) and leave (log out) the network at any time.
3. Maintains that a given username can only be registered once over the server network.
4. Guarantees that an activity message sent by a client reaches all clients that are connected to the network at the time that the message was sent.
5. Guarantees that all activity messages sent by a client are delivered in the same order at each receiving client.
6. Ensures that clients are evenly distributed over the servers as much as possible.

We aim to construct a system that encompasses the above-discussed capabilities with the assumptions stated below:

1. Servers can crash at any time, whereby the server process abruptly terminates without a chance to inform other servers before it does so.
2. Clients can crash at any time, whereby the client process abruptly terminates without sending a logout to the server before it does so.
3. Network connections between servers and between servers and clients can be broken at any time.
4. Broken network connections will eventually be fixed, such failures are transient.

### 1.1. CAP & PACELC

No distributed system is safe from network failures, thus network partitioning generally has to be tolerated. In the presence of a partition, one is then left with two options: consistency or availability. When choosing consistency over availability, the system will return an error or a time-out if particular information cannot be guaranteed to be up to date due to network partitioning. When choosing availability over consistency, the system will always process the query and try to return the most recent available version of the information, even if it cannot guarantee it is up to date due to network partitioning.

In the absence of network failure that is, when the distributed system is running normally both availability and consistency can be satisfied.

CAP is frequently misunderstood as if one has to choose to abandon one of the three guarantees at all times. In fact, the choice is really between consistency and availability only when a network partition or failure happens; at all other times, no trade-off has to be made.

Database systems designed with traditional ACID guarantees in mind such as RDBMS choose consistency over availability, whereas systems designed around the BASE philosophy, common in the NoSQL movement, for example, choose availability over consistency.

The PACELC theorem builds on CAP by stating that even in the absence of partitioning, another trade-off between latency and consistency occurs.

PACELC builds on the CAP theorem. Both theorems describe how distributed databases have limitations and tradeoffs regarding consistency, availability, and

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

partition tolerance. PACELC, however, goes further and states that a trade-off also exists, this time between latency and consistency, even in absence of partitions, thus providing a more complete portrayal of the potential consistency tradeoffs for distributed systems.

A high availability requirement implies that the system must replicate data. As soon as a distributed system replicates data, a tradeoff between consistency and latency arises.

The PACELC theorem was first described by Daniel J. Abadi from Yale University in 2010 in a blog post, which he later formalized in a paper in 2012. The purpose of PACELC is to address his thesis that "Ignoring the consistency/latency tradeoff of replicated systems is a major oversight [in CAP], as it is present at all times during system operation, whereas CAP is only relevant in the arguably rare case of a network partition."

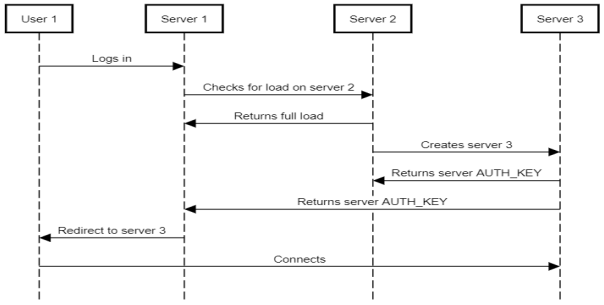
## 2. The New Server Protocol

We build upon the protocols configured in Project 1 (removing some protocols and adding more) to facilitate and achieve the aims mentioned.
Also each server now has more inherent automated services (like **SERVER_ANNOUNCE**) built into thereby reducing the need for inter-server communication that was necessary for the previous build.

### 2.1. Re/Creation

**Addresses Aims: #1, #2, #6.**

Every server aims to evenly distribute the load over the network by making sure that if there's another server available that has minimum load difference from the server that is at present being accessed, it'll redirect the client connection to that server instead.




Also, a **MAX-CAP** of 5 was implemented over the servers such that every server can have a maximum load of 5 clients at a time.
If every server on the network exceeds the **MAX-CAP** limit, the server that is being attempted for connection will start a new server and redirect the client to that instead.

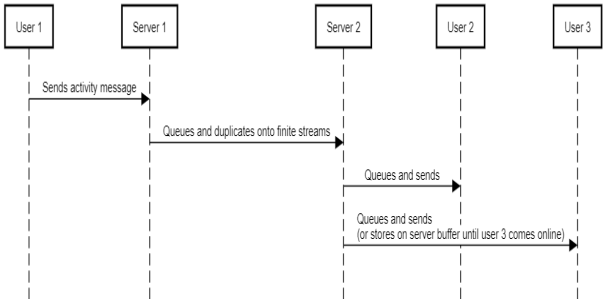
This makes sure that every server on the network has load evenly distributed over them and makes server creation a breeze as more clients connect to them. Scalability is handled efficiently.

### 2.2. Queuing

**Addresses Aims: #4, #5.**

A catch-and-send mechanism would work best in a scenario where a server needs to guarantee definite on-time deliveries and manage order at the same time.

For such a system, every client message that needs to be sent ahead should be treated as an entity of a finite stream that stores the message until they are broadcasted.




We implement a single outbound finite stream that manages all outgoing messages that fail to send to the clients in the following way:

1. The server notes the set of connected clients upon receiving a message **M** and associates that message with target recipients.
2. Outbound flush tries to deliver the message **M** to all targeted recipients.
3. A list of undelivered messages is maintained if any associated connections are closed before the message is delievered.
4. This list is constantly checked upon every activity broadcast's target recipients and re-broadcasted if those target recipients are online.

Note that client-connections and finite streams for these connections are not the ports of remote addresses (clients) but rather the usernames of the clients.

This helps in two ways:

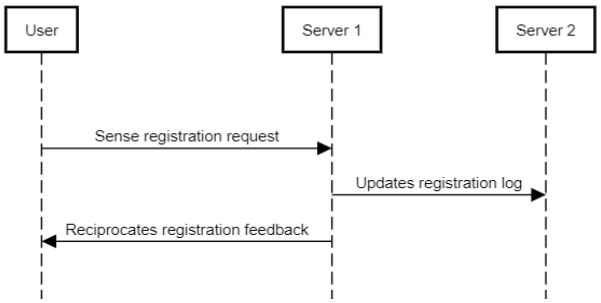
1. The server can track which messages were sent to the user and which messages weren't so that when the user logs in the next time, they can be delivered at that time.

2. The finite streams are ordered with receive-times so that they are sent in the same order as received from the client.

### 2.3. One-Time Registration Only

**Addresses Aims: #3.**

In the previous build, if there are 100 users per server and there are 100 servers, that would mean a total of 10,000 pairs of username-secrets.
For a new user to be registered in this system, the server responsible for validating availability for the desired username will be waiting for an indefinite amount of time which depends on how well the other servers are individually performing.
It also means that there have to be 10,000 check-iterations before a server can successfully result in locking; otherwise, denying availability. There could also be scenarios where lock conflicts may happen if the system grows larger than its potential ability.




We propose an approach that can significantly reduce these check-iterations as well as server-communications by removing the lock-request/lock-granted/lock-denied approach and increase server-performance while reducing response-times in the following manner:

1. Every server maintains a log which stores a list of usernames.

2. Upon Registration:
   (a) Server registering the new user sends out a "NEW_REGISTER_ANNOUNCEMENT" message to all connected servers.
   (b) This message includes: "username" & "secret".
   (c) The receiving server adds the registration info to the associated Registration Log.

3. Upon login, the targetted server simply iterates over th eRegistration Log and responds accordingly.

### 2.4. Security

The servers need to be authenticated so that each message transaction between servers can be made sure that they originated from an actual server on the network and that it's not just some dummy client sending messages to servers by posing as a server.

This was achieved by enabling each server to generate an authentication-key when the server was created. This authentication-key is then sent out to all other connected servers via "SERVER_AUTH_KEY_ANNOUNCEMENT", upon whose arrival each connected server would store the credentials on an Authentication Log.

Basically, for all transactions between servers, each server sends out it's AUTH_KEY which is juxtaposed on the receiving end with the AUTH_KEY received upon "SERVER_AUTH_KEY_ANNOUNCEMENT".

## 3. Assumptions & Possible Failues

The system proposed assumes the following on top of already discussed assumptions in section 1:

1. Registration Log is successfully updated every time a "NEW_REGISTER_ANNOUNCEMENT" is made for all servers.

2. Authentication Log is successfully updated every time a "SERVER_AUTH_KEY_ANNOUNCEMENT" is made for all servers.

3. Queued finite-streams are updated correctly on each server.

## 4. Conculsions

In section 2 we see multiple solutions to address the aims discussed in section 1 considering the assumptions provided. We also discuss possible system failures based on the assumptions made while constructing such a system in section 3.

The system could have been more fail-safe if affirmation-flags were sent over the server for each new protocol introduced but that would result in higher message complexity for the network architecture.

## 5. Instructions for usage

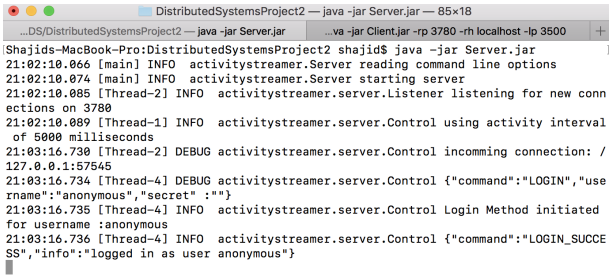



**Figure 1.**

**java -jar ActivityStreamerServer.jar -rp "anyPortNumber" -rh "anyServerName" -lp "localPortNumber"**

1. Run first: java -jar ActivityStreamerServer.jar
2. Run next: java -jar ActivityStreamerServer.jar -rp 3780 -rh localhost -lp 3781
   (a) When no param, server starts as the first server and does not connect to the others.
   (b) When has -rp and -rh, server starts as the server connecting to the remote server.
   (c) When has all params, server starts as the server connecting to the remote server with specific port number.


**Figure 2.**

**java -jar ActivityStreamerClient.jar -rp "serverPortNumber" -rh "anyServerName" -lp "localPortNumber" -u "userName" -s "secret"**

1. java -jar ActivityStreamerClient.jar -rp 3780 -rh localhost -lp 3500 -u "test" -s "asdfasdn"
   (a) When no -u or -s params, login as anonymous.
   (b) When has -u but no -s params, register the user with user name and the client will generate the secret for user.
   (c) When has -u and -s params, login with userName and secret.

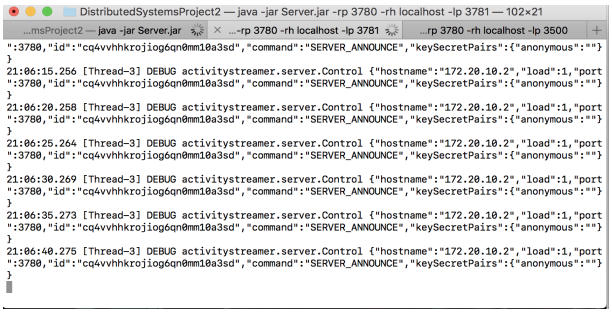



**Figure 3.** Creating Server 2 and connecting to Server 1 automatically.

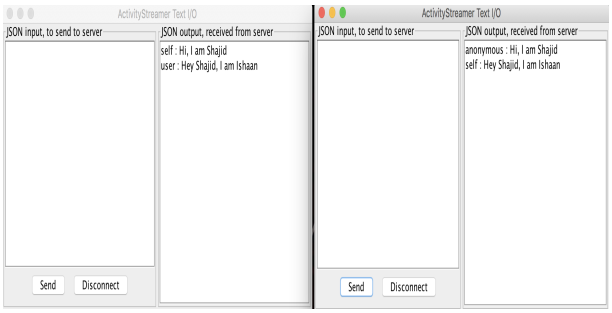



**Figure 4.** The chat interface after an over-loaded server creates a new server and redirects this client to the new server.

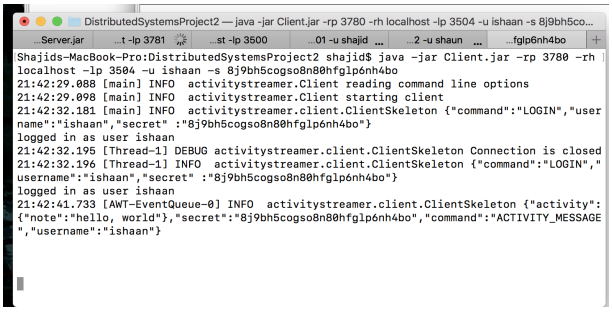



**Figure 5.** The console for the newly instantiated server.