



CS4262  
Distributed Systems

Group Project

## **INTERIM REPORT**

**Group\_NKC**

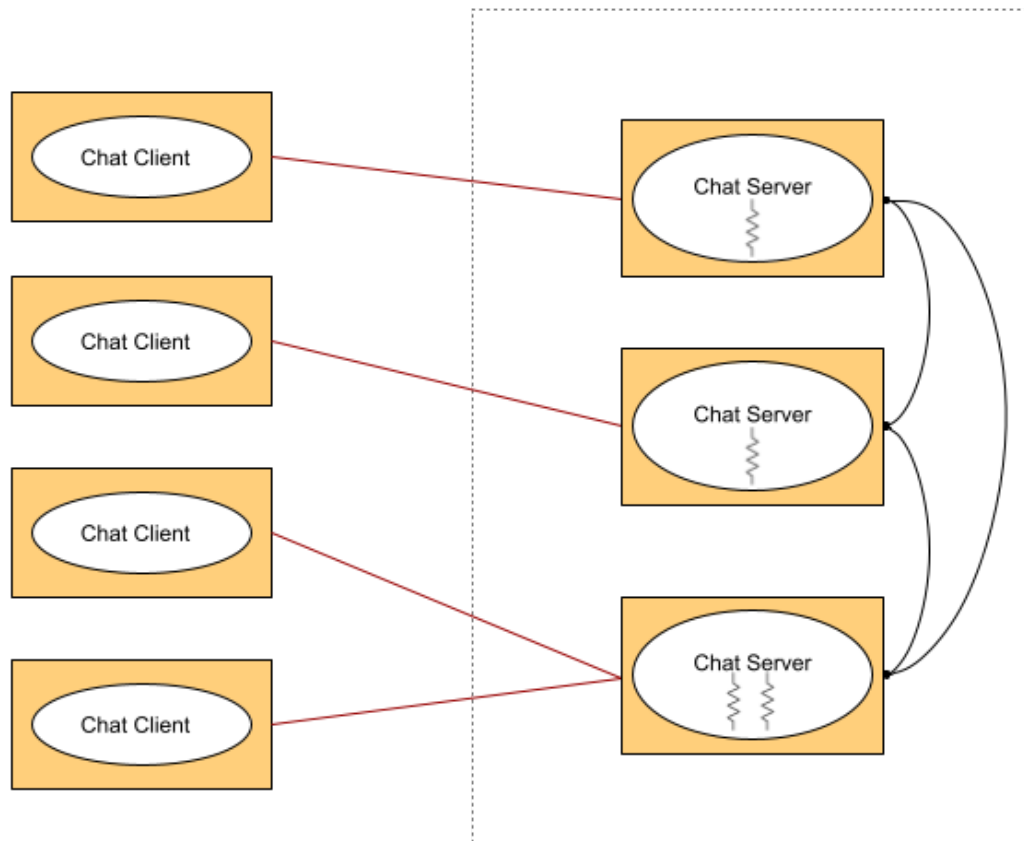
170387F : M. W. G. V. Melaka  
170524B: R. M. T. S. Ratnayake  
170535J: M. S. S. Saleem  
170723J: N. M. Zameel

16 February 2022

2017 Intake

Department of Computer Science and Engineering  
Faculty of Engineering  
**University of Moratuwa**

## 1. Software Architecture Diagram and brief explanation.



This project is about implementing a distributed client-server chat system consisting of two main distributed components: chat servers and chat clients, which can run on different hosts.

Chat clients can connect to at most one available server where they can be used to send requests to create, join, delete, list and quit chat rooms. They can also communicate with other chat clients that are linked to the same chat room.

Chat servers are programs accepting multiple incoming TCP connections from chat clients. There are multiple servers working together to serve chat clients. Once the system is active, the number of servers is fixed. Each server is responsible only for a subset of the system's chat rooms. In order to join a particular chat room, clients must be connected to the server managing that chat room. As a result, clients are redirected between servers when a client wants to join a chat room managed by a different server. Chat servers are also responsible for broadcasting messages received from clients to all other clients connected to the same chat room.

## 1. Planned improvements and why do you plan to implement them.

### 2.1 Having a Leader/ Coordinator

When implementing this application we decided that we will try to achieve consistency and the partition tolerance, but we will not be focusing on achieving the availability factor in this application according to the CAP theory.

We figured out the following pros and cons when using a coordinating server.

Pros	Cons
Having unique room identities and client identities can be achieved.	More message protocols and therefore additional delay.
Deletion of rooms can be done in a consistent manner.	One server gets a high load compared to the other two servers.
Quitting of clients can be achieved in a consistent manner.	

*By considering the above pros and cons and since we are trying to achieve consistency and partition tolerance application coordination between the servers is crucial, because we are not trying to achieve availability additional delay can be justified.*

### 2.2 Gossiping protocol

In a large-scale distributed system determining the entire system becomes harder with the number of nodes. Hence it is difficult to rely on a hardware-based solution like multicast to properly share data among nodes. Hence in our application, we have decided to use the gossiping protocol to share information among nodes. Since the gossiping protocol allows decentralised scheme data can be shared timely and in an efficient manner.

We selected gossiping protocol since,

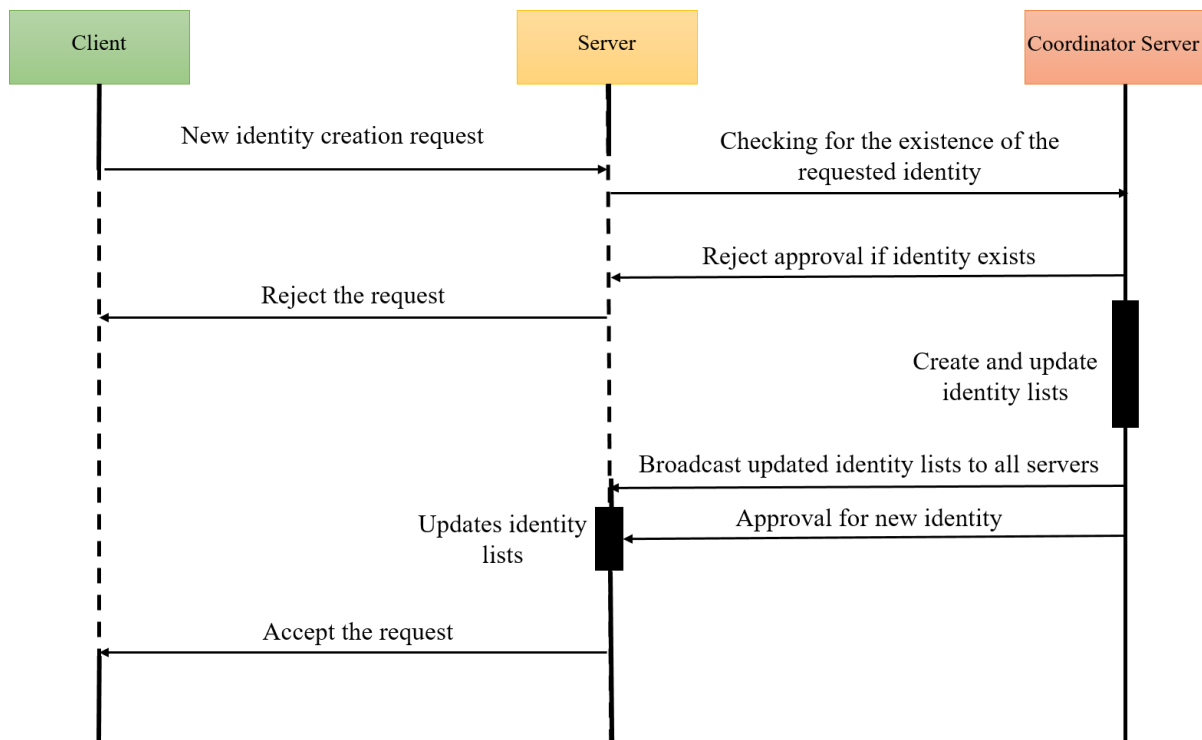
1. Fault tolerance - since data is shared by multiple nodes, nodes with connectivity issues are compensated by working nodes,
2. Scalability - protocol is highly scalable since multiple nodes which received data will share data with other nodes.
3. Robustness - since no node has a specific role. Hence a failed node will not prevent other nodes from sharing data

The gossiping protocol has the disadvantage of high complexity in implementation.

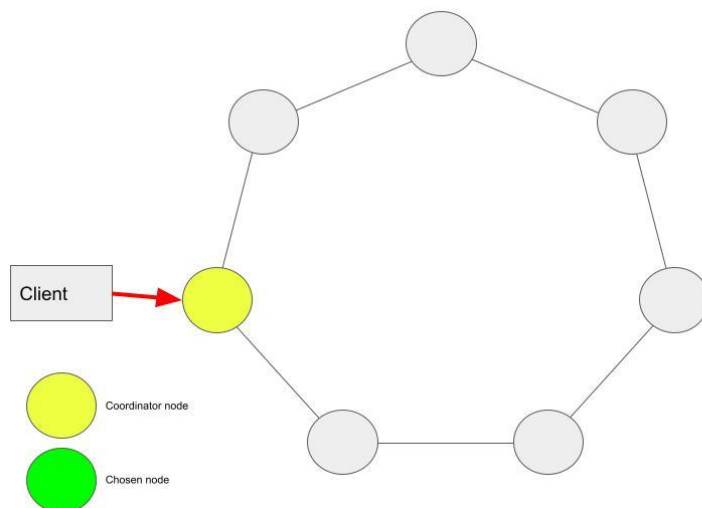
*Since the advantage of the protocol outweighs the disadvantages we have chosen to implement the gossiping protocol to share data among nodes.*

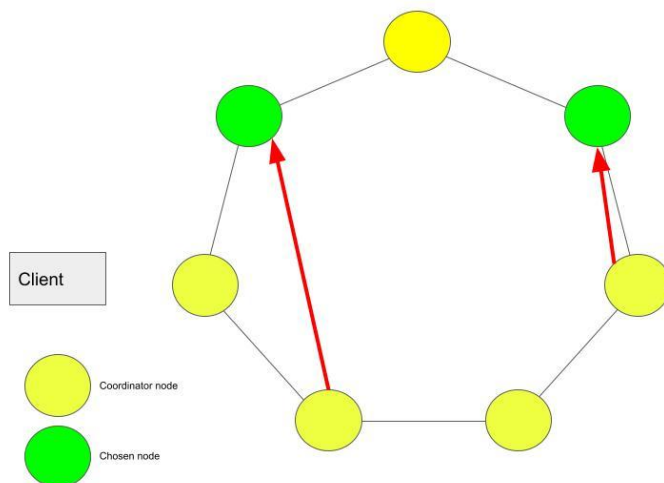
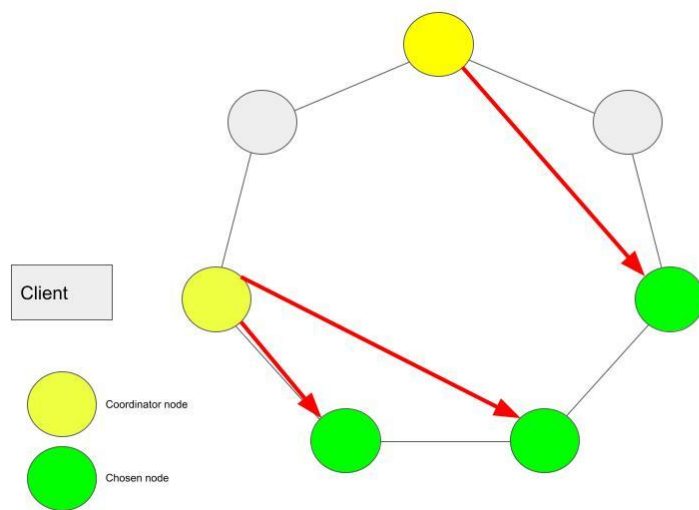
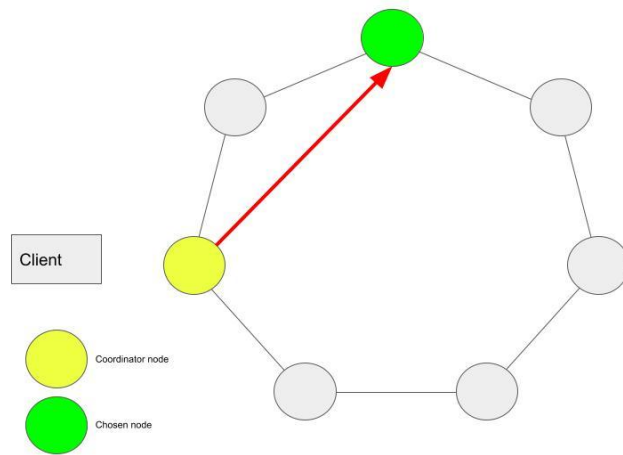
## 2. Interaction diagrams for planned improvements.

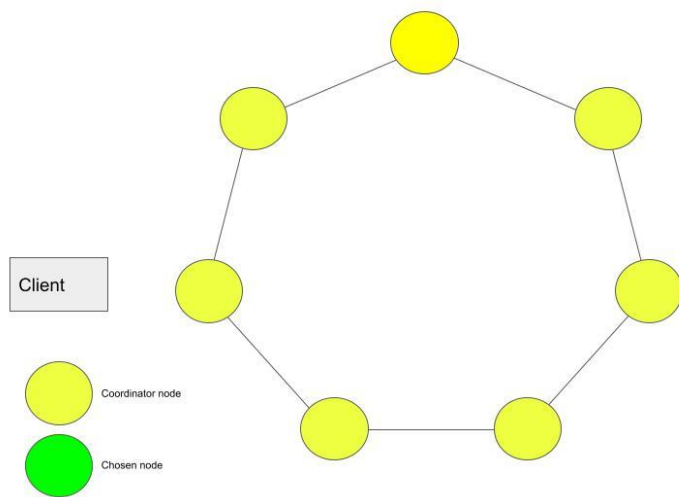
### 3.1 Having a Leader/ Coordinator



### 3.2 Gossiping protocol







### 3. Amendments to the server to server communication protocol to facilitate the improvements.

In the proposed solution each server is expected to maintain a list of all the existing client IDs and room IDs of the system to have a robust operation if the existing coordinator fails.

In the proposed improvements, there exists a coordinator server which is elected through the Fast Bully Algorithm. This coordinator server is responsible for,

- Provide information about existing clients and rooms to the other servers.
- Broadcast update messages to other servers upon creation or deletion of existing client or room IDs.

#### Verifying that the same identity is not being used in other chat servers

When a server receives a new identity request,

If the current server is the coordinator server:

1. Check for the requested ID in the client ID list
2. If requested client ID not exists in the list,
  - 2.1. add the client ID to the clients list and send other servers a broadcast update message to update their lists.  
`{"type": "update", "id": "client", "value": "$updated_list"}`
  - 2.2. add the client ID to the list and send the client a newidentity message with approved status set to true.
3. If requested client ID exists in the client ID list
  - 3.1. Reply to the client with a newidentity message with approved status set to false.

If the current server is not the coordinator server:

1. Send a register message to the coordinate server  
`{"type": "register", "id": "client", "identity": "$requested_identity"}`
2. In the coordinate server upon receiving a register message,
  - 2.1. Check if the requested identity is in the client id list
  - 2.2. If requested identity already exists,
    - 2.2.1. Send a register message approved set to false to the requested server.  
`{"type": "register", "approved": "false"}`
  - 2.3. If the requested identity does not exist in the current list,
    - 2.3.1. register the new client identity and send a broadcast update message to all the servers.  
`{"type": "update", "id": "client", "value": "$updated_list"}`
    - 2.3.2. Finally send a register message approved set to true to the requested server.  
`{"type": "register", "approved": "true"}`

## Verifying that the same room\_ID is not being used in other chat servers

When a server receives a createroom request,

If the current server is the coordinator server:

1. Check for the requested ID in the room ID list
2. If requested room ID not exists in the list,
  - 2.1. add the room ID to the rooms list and send other servers a broadcast update message to update their lists.  
`{"type": "update", "id": "room", "value": "$updated_list"}`
  - 2.2. add the room ID to the list and send the client a createroom message with approved status set to true.
3. If requested room ID exists in the room ID list
  - 3.1. Reply to the client with a createroom message with approved status set to false.

If the current server is not the coordinator server:

1. Send a register message to the coordinate server  
`{"type": "register", "id": "room", "identity": "$requested_identity"}`
2. In the coordinate server upon receiving a register message,
  - 2.1. Check if the requested ID is in the room id list
  - 2.2. If requested ID already exists,
    - 2.2.1. Send a register message approved set to false to the requested server.  
`{"type": "register", "approved": "false"}`
  - 2.3. If the requested ID does not exist in the current list,
    - 2.3.1. register the new room ID and send a broadcast update message to all the servers.  
`{"type": "update", "id": "room", "value": "$updated_list"}`
    - 2.3.2. Finally send a register message approved set to true to the requested server.  
`{"type": "register", "approved": "true"}`

## Upon receiving a quit message

If the current server is the coordinator server:

1. Remove the client identity from the client ID list and send a broadcast message to the other servers  
`{"type": "update", "id": "client", "value": "$updated_list"}`

If the current server is not the coordinator server:

1. Send a delete message to the coordinate server  
`{"type": "delete", "id": "client", "value": "$client_id"}`
2. In the coordinate server upon receiving a register message



- 2.1. Remove the client identity from the client ID list and send a broadcast message to the other servers.  
`{"type": "update", "id": "client", "value": "$updated_list"}`

### Upon receiving a deleteroom message

If the current server is the coordinator server:

1. Remove the room ID from the room ID list and send a broadcast message to the other servers  
`{"type": "update", "id": "room", "value": "$updated_list"}`

If the current server is not the coordinator server:

1. Send a delete message to the coordinate server  
`{"type": "delete", "id": "room", "value": "$room_id"}`
2. In the coordinate server upon receiving a register message
  - 2.1. Remove the room ID from the room ID list and send a broadcast message to the other servers.  
`{"type": "update", "id": "client", "value": "$updated_list"}`

**Note that in each of the above scenarios if the coordinator server fails following steps will be taken,**

- If the coordinate server does not respond in a T1 time interval resend the message.
- If the coordinate server fails to respond again in a T1 time interval, initiate an election to appoint a new coordinate server.

### A server(Si) does not receive a response within T1 from the coordinator in two attempts

1. Si sends an election message to every process with a higher priority number.  
`{"type": "election"}`
2. Si waits for answer messages for the interval T2.
  - 2.1. If no answer within T2,
    - 2.1.1. Si sends a coordinator message to other processes with lower priority numbers.
    - 2.1.2. Si stops its election procedure.
  - 2.2. If the answer messages are received within T2,
    - 2.2.1. Si determines the highest priority number of the answering processes.
    - 2.2.2. Si sends a nomination message to this process.  
`{"type": "nomination"}`
    - 2.2.3. Si waits for a coordinator message for the interval T3.  
`{"type": "coordinator", "host": "ip.ip.ip.ip", "port": "port_number"}`
  - 2.3. If the coordinator message is received,
    - 2.3.1. Admit the sender as the new coordinator.
    - 2.3.2. Si stops its election procedure.
  - 2.4. 2.6 If no coordinator message within T3,
    - 2.4.1. 2.6.1 Repeat step 2.4.2 for the next highest priority numbered server.
    - 2.4.2. 2.6.2 If no process is left to choose, Si restarts the election procedure.

#### If a server $S_j(i < j)$ receives a coordinator message from $S_i$

1.  $S_j$  sends an answer message to  $P_i$ .  
{ "type": "cord\_response", "priority": "\$p", "host": "\$ip.ip.ip", "port": "\$port\_number" }
2.  $S_j$  waits for either a nomination or a coordinator message for the interval  $T_4$ .
3. If no *coordinator* message or *nomination* message within  $T_4$ .
  - 3.1.  $S_j$  restarts the procedure.
4. If a server  $S_j(i < j)$  receives the *nomination* message from  $S_i$ ,
  - 4.1.  $P_j$  sends a *coordinator* message to all the processes with lower priority numbers.  
{ "type": "coordinator", "host": "\$ip.ip.ip", "port": "\$port\_number" }
  - 4.2.  $P_j$  stops its election procedure.

#### If a process $S_j(i > j)$ receives a coordinator message from $S_i$

1. Admit  $P_i$  as the new coordinator.
2. Stops its election procedure.

#### A Server $S_i$ recovers from failure

1.  $S_i$  sends an lamUp message to every process.  
{ "type": "lamUP", "host": "\$ip.ip.ip", "port": "\$port\_number" }
2.  $S_i$  waits for view messages for the interval  $T_2$ .  
{ "type": "view", "coordinator\_ip": "\$ip.ip.ip", "coordinator\_port": "\$port\_number", "client\_IDs": "\$client\_id\_list", "room\_IDs": "\$room\_id\_list" }
3. If no view messages within  $T_2$ ,  $S_i$  stops the procedure. //  $S_i$  is the coordinator
4. If the view messages are received within  $T_2$  :
  - 4.1.  $S_i$  compares its view with the received views.
  - 4.2. If the received view is different from the  $S_i$ 's view,  $S_i$  updates its view.
  - 4.3. If  $S_i$  is the highest priority numbered process,
    - 4.3.1.  $S_i$  sends a coordinator message to other processes with lower priority numbers.
    - 4.3.2.  $S_i$  stops the procedure.
  - 4.4. Otherwise,
    - 4.4.1. Admit the highest priority numbered process as the coordinator.
    - 4.4.2.  $S_i$  stops the election procedure.

#### A server $S_i$ may receive an lamUp message from $S_j$




1.  $S_i$  sends a view message to  $S_j$ .  
{ "type": "view", "coordinator\_ip": "\$ip.ip.ip", "coordinator\_port": "\$port\_number", "client\_IDs": "\$client\_id\_list", "room\_IDs": "\$room\_id\_list" }

## 5. Backlog of the project and Individual Contribution

For the chat server implementation project we identified necessary data models, algorithm implementation, and classes based on the MVC pattern. Since the server does not have a ui component we have only listed work related to model and controller components.

### Legend

	Utility classes		Algorithm implementation
	Model classes		Controller classes

Mohamed Zameel	Shafeek Saleem
JSON parser to marshal and unmarshal JSON code	Socket connection class for TCP communication
User session class	Message class
Remote user session class	Consensus Algorithm implementation
Consensus Algorithm implementation	Delete room controller
Consensus controller class	Consensus controller class
List controller class	WHO controller class
Create room controller class	Join room controller class
Server change controller class	Message controller class
	
Tharaka Ratnayake	Vihan Melaka
Thread class for thread pool	File reader class for configuration reading
Configuration class	Local chat room class
Chat room information class	Remote chat room class
Fast bully algorithm implementation	Fast bully algorithm implementation
New identity controller class	Room change controller class
Fast bully controller class	Fast bully controller class
Route controller class	Movejoin controller class
Quit controller class	Abrupt exit controller class
	

## 6. Project timeline

	February																				March																			
Date	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15		
Planning																																								
Class diagrams																																								
Interim report																																								
Utility classes																																								
Model classes																																								
Algorithm implementation																																								
Controller classes																																								
Deployment																																								
Final report																																								