Robbie Fishel
Birju Patel
Dr. Amy Babay

Project 2: Final Design

Introduction:

Our language of choice for this project is Python. We will use RPCs to implement the client-server interface of the application, specifically the RPyC library (documentation here). RPyC is symmetrical, meaning the client can run code on the server, and the server can run code on the client. We will use this feature to asynchronously send group updates from server to client.

For server-server interactions, we implemented a group communications protocol over UDP. This protocol utilized Lamport timestamps to establish a total order on messages. When a server wants to make an update to the system, it uses this protocol to broadcast it to all other servers. All servers will run a separate thread that listens for incoming messages and handles them.

To deal with crash faults, servers will store their state in main memory, which can be used for recovery. When a server crashes and attempts to recover, it will first load the data from the backup file. Then it will request missed messages from other servers. If a server is partitioned away from other servers, once the partition heals, it uses nacks to check for updates it may have missed.

Client:

The client logic remains mostly unchanged. The client stores a data structure called group_obj, which contains the most updated state of the group the client is currently in. Clients will interact with the application through a command line interface built as specified in the project description. The client will initially specify which server in the cluster it wants to connect to by supplying its IP address and port. Whenever the user issues a command, the client will then try to make the corresponding RPC call to that server. If a client's connection to the server terminates, the client will be shown a message that the connection has failed, and will be given a prompt to allow it to reconnect to another server in the cluster.

When the client joins a new group, it must monitor that group to forward any updates to that group from other clients to itself. We send a callback function to the server which the server calls every time there is a modification to the state of that group. The callback function updates the state of group_obj at the client to contain the latest data. We spawn a thread to monitor the state of group_obj and print the latest messages to the console every time there is an update.

Server:

The client-server interface is similar to the previous design. The server will contain a ChatServer class, which is an RPyC service. When a new client connects to the server, a new ChatServer instance will be created to serve that client. All instances share access to the

group_list and active_users data structures. Access to these data structures is protected by a mutex lock. The server exposes the following services to the client.

- __init__(): Allows a new client to connect to the server. Assigns them a unique user_id.
- set_username(String): Resets the client's username. If they are in a group, they are removed from that group.
- join_group(String, callback_function): Adds the client to an existing group. If no such group exists, it creates a new group. Spawns a monitor thread to check when new messages are added to this group. When new messages are detected, the server calls the callback function provided by the client. This allows us to asynchronously send new messages to clients. This thread will also monitor when users are added and removed from the group, and will send clients those updates as well.
- write_message(String): Adds a new message to the group that the client is currently in.
- add_like(int, int): Given a message server id and timestamp, adds like to that message.
- remove_like(int, int): Given a message server id and timestamp, if the user is listed as having liked that message, removes that like.
- message_history(): Returns all messages written in the group the user is currently in.
- leave_group(): Exit group.
- view(): Gets a list of the servers that this server can talk to. Initially we assume all servers are in this server's view. If we have not received a message from a server in 10 seconds, we assume there is a partition and drop that server from the view.

The client must first connect to the server, then set their username, then join a group, and then may write messages, like and dislike, and request a message history. If they attempt to do this in an incorrect order, it will fail.

We uniquely identify messages in a group by using the server id where the message originated from and the message's lamport timestamp. We order the messages on the screen based on these variables, to ensure the ordering obeys a causal order. When a user wants to add or remove a like from a message, they supply these values to specify which message they want to change.

When a client joins a group, we start a new thread that runs the function update_client_status. This contains an infinite loop that sleeps on a condition variable. Whenever an update to the group occurs on the server, we signal this condition variable, which copies the updated state of the group and sends it back to the client via the callback function.

When a function is called that modifies the state of the server, this change is immediately applied to the server and the update is then packaged and broadcast to every other server in the cluster.

To startup a server, run the command *python3 chat_server.py -id {id} -file {config_file}*. The id is a number between 1 to 5 that is unique for each server in the cluster. The config_file is

the location of a text file with a list of the IP addresses and ports of each of the servers in the cluster.

To support crash recovery, each server routinely backs up its current state to a file titled *log{id-1}*. This file contains the current state of the server's application, as well as a log of messages being passed between servers that have not yet been garbage collected, and a buffer of messages this server has sent and have not yet been confirmed to have been received by all other servers. This file is written in the JSON format, and contains all state information for the reliable broadcast protocol and the server state. The file is updated every time there is a change to the state of either components, and is loaded from and read into memory every time the server starts up.

Reliable Broadcast:

We created a protocol to provide a totally ordered broadcast service over UDP. We assume that the membership in the protocol is fixed, so each server has an assigned IP address and port number and knows the IP address and port number of every other service. We also assume that each server may crash and restart, and the network connecting these servers may partition.

The user interacts with the protocol by utilizing the functions *broadcast* and *listen*. The *broadcast* function takes a particular message and packages it into a packet, updates the lamport timestamp, and sends the packet to all other servers in the system via UDP. This function is directly called by the user every time they want to broadcast a new message. The *listen* function runs in the background. On startup, the user spawns a new thread that runs this function, which contains a select loop that waits for incoming messages. This function places messages in an ordered log, and cleans the log periodically.

Each server initially learns the IP addresses and port numbers of all other servers by reading the *config_file*. We have included a file titled *clusterconfig* with this information. Line i in the file contains the information for the server with the id of i. On startup, a server loads its own information and opens a socket at the specified port to listen for incoming messages.

To ensure reliability, each server stamps a sequence number on each of its packets and buffers those packets to serve retransmission requests. The server also maintains a vector that contains the highest sequence number it received from every other server in the system. If a server receives a new message, and its sequence number does not match the expected sequence number (highest sequence number received + 1), the server rejects the packet and sends a nack to the message's sender requesting a retransmission of the missing packets.

A causal order is maintained by utilizing lamport timestamps. Every time a novel message is broadcast (not an ack, nack, or retransmission), the node increments its timestamp. When the next message is accepted (sequence number = highest sequence number received + 1), the timestamp is updated to max(message timestamp, timestamp) + 1. When a packet is sent or accepted it is placed in order in a log. To determine the order, two messages, msg1 and msg2, are compared on the following condition, msg1 > msg2 if msg1.timestamp > msg2.timestamp or

(msg1.timestamp == msg2.timestamp and msg1.sender_id > msg2.sender_id). The server maintains a vector of the highest timestamp it has received from every other server.

The message log and retransmission buffer are routinely garbage collected to remove unnecessary packets. This is necessary to ensure that the backup file does not become unreasonably large. After a particular server receives a total of ACK_TIME messages, currently set at 10, it sends an ack containing its sequence number vector, timestamp vector, its current sequence number, and its current timestamp. Each server maintains two matrices to store copies of the most recent timestamp vectors and most recent sequence number vectors it has received from every other server. On receiving an ack, if the vectors contained within it are more updated than what the server currently has, they are placed in their respective matrices. Also, if the sending server's current timestamp is larger than what the server has stored in its vector for that server, and the server has received every packet from that server up to the current sequence number, the server updates its own timestamp vector with this value and places it in the timestamp matrix.

This ensures that eventually, each server has a view of the state of every other server. A message can be removed from the retransmission buffer if every server has confirmed it has accepted a sequence number higher than that message's sequence number. Formally, we can remove any message whose sequence number is <= min(sequence_matrix[i][server_id]) for all i = 1,…,5 and the current server's server_id. We can remove a message from the message log if every server has received a message from every other server with a higher timestamp than the timestamp of the message. Formally, we can purge any message whose timestamp is < min(timestamp_matrix[i][j]) over all i,j = 1,…,5.

To allow for proactive recovery of potentially lost packets, we periodically send nacks to all other servers requesting the message with the latest sequence number + 1. When a server receives this packet, if they have no new packet, they disregard it, or else they retransmit their latest packets. Nacks are sent every NACK_TIME seconds, currently set at 3.

The state of this protocol on a given server is periodically backed up on the *log{id-1}* file. This protocol allows servers to crash and restart and still recover from the other servers. When a process crashes, the other servers may still make progress, but they cannot delete messages from their retransmission buffer or message log because they have not received the updated vectors from the crashed server, so packets the crashed server have missed are available. When the crashed server restarts, it will send nacks to all other servers requesting the latest packets. When the other servers send those packets, the crashed server will eventually get up to speed, and the messages will then be garbage collected by everybody when the server sends its next batch of acks. The same mechanism also ensures partition tolerance. When a partition occurs, two sides may accept new messages and place them in an order. When the partition heals, the nack cycle ensures that both sides eventually receive every message from the other side. When this occurs, the total ordering condition ensures that these messages are placed in the same order.

State Replication:

The above protocol provides us with a service that can reliably broadcast messages and establish a total causal order within the group. On top of this service, we built a replicated chat application. The chat application involves applying operations to a dictionary group_dict. This dictionary contains Group objects, which has a name, a set of active users, and a list of Message objects. The server also maintains an active_user list, a list of users that are currently connected to this server.

The group_dict can be modified in the following ways. A user can create a group, join a group, leave a group, write a message, add a like to a message, or remove a like from a message. We define the following functions.

- create_group(group_name, username)
- join_group(group_name, username)
- leave_group(group_name, username)
- write_message(group_name, username, content, ts, server_id)
- add_like(group_name, username, ts, server_id)
- remove_like(group_name, username, ts, server_id)

When a user enters a command, it makes an rpc call to the server, which determines if the user can legally make this modification, and if it can, it calls one of these functions. It then encodes the function call as a message and sends it to all other servers using the broadcast protocol. Below is the message format.

```
"{
  operation_name: [Value]
  arg_1: [Value]
  arg_2: [Value]
  arg_3: [Value]
  arg_4: [Value]
  arg_5: [Value]
}"
```

We also modified the broadcast packet structure to include a new flag titled executed. When the executed flag is true, it implies that this function call has been applied at this server. A server sends these commands to other servers only after it has executed it itself, so in the packets it broadcasts, it sets the executed flag to false, but when it puts the packet in its own log, it sets the flag to true.

When a server receives a new message, it places that message in the log in its correct order. It then sweeps the log from left to right, and when it sees a message with the executed flag set to false, it attempts to run the command encoded inside the message. Each command contains internal logic to determine whether or not it is legal to run, so for instance if we receive a

leave_group command but no group of that name exists, we do nothing and return that the function has failed. In this case, we leave the packet unchanged. If, however, we successfully execute the change, we flip the executed flag to true and put it back in the log. This pattern allows us to keep a consistent state even when packets are received out of order. For instance in the previous case, when we eventually receive the missing commands that occurred before the *leave_group* command, they will be placed before that command in the log, and when all commands are run in order, they will all be correctly executed. This logic is implemented inside of the *listen* function, which calls a helper function titled *handle_packet*.

While implementing this component, we noticed a few edge cases. If two servers are in a partition, it is possible that on both sides of the partition, two groups can be legally created with the same name. When the partition heals, each side will receive duplicate *create_group* commands for a group that already exists. To resolve this, we ignore the duplicate *create_group* functions and instead treat them as a call to the *join_group* function with the same arguments. So when the partition heals, the two new groups are merged into one group. We also found that when a server crashes, every user that is currently connected to that server should be removed from the groups those users are in. However, because the server crashes, it never has a chance to make those updates. To handle this, we use the active_users list, which contains a list of the users that are connected to this server and the groups that they are a member of. When a server restarts, it reads the active_users list, evicts each of those users from their current group, and sends *leave_group* messages to all other servers in the system. We assume that servers that crash are eventually brought back up, so the system will become eventually consistent.

Data Types:

- Message
    - id: List[int, int]
    - content: String
    - sender: String
    - likes: [int]
- Group
    - name: String
    - users: [int]
    - messages: [Message]
- IPV4
    - addr: String
    - port: int
- BroadcastMsg
    - sender_id: int
    - ack: boolean
    - nack: boolean
    - executed: boolean

    - ts: String
    - seq: String
    - content: String

Data Structures:

- Client
    - group_obj: An object containing the current state of the group that the client is currently in. This includes a list of all messages received by this client. The user interface displays information from this group object. When the monitor thread on the server calls the callback function, it modifies this object.
- Server
    - group_dict: A dictionary of Group objects, where group name is mapped to the Group object itself. Used to track the state of the entire application
    - timestamp: An integer representing this server's current Lamport timestamp
    - log: An ordered list of unconfirmed and outstanding messages
    - buf: A buffer of sent messages that may be retransmitted
    - highest_seq_rcvd: A vector of the highest sequence number received from each other server in the system, where the index represents the node ID
    - highest_ts_rcvd: A vector of the highest timestamp received from each other server in the system, where the index represents the node ID
    - seq_matrix: Contains highest_seq_rcvd vectors of every server
    - ts_matrix: Contains highest_ts_rcvd vectors of every server
    - confirmed_order: A list of messages whose order is definitely consistent across the system

Disk:

- clusterconfig
    - File with a list of IP : Port mappings for our network partition
    - Makes it so that all other IPs know where to send messages and/or check to see whether other servers are up and running
- log{server_id - 1}
    - Stores the state of the broadcast protocol and the chat application
    - Each server has its own file

Testing:

To test our system, we used a modified version of the provided p2_test.py file. The only way it was modified was by changing the way that the program was called from './chat_server.py' to 'python3 chat_server.py -file clusterconfig'. Our Makefile target 'make run' establishes the 5 chat servers and the bridge network. The chat servers automatically run when this make target is executed. Once in the bash terminal created for a client, the client program

can be started with ‘python3 client.py’. You also can make various calls to ‘python3 test_p2.py’ in this bash script in order to add constraints to the system (partitions, loss, etc.).