Andrew King (aking65)
Yu-Chung Peng (ypeng22)

Final Project Design Document

Overview:

Our client design is a simple loop that ingests user commands from the command line, packages them into a message format, sends them to the server it's connected to, then waits for a response from the server. To connect with a server, the client establishes a private two-way communication channel using spread. The details of how this is done are outlined in the next section.

Five servers are responsible for maintaining the state of this application. The servers will quickly reach consensus about the state of the application once they are connected to each other, although two servers that are partitioned may recognize different states. These states will be reconciled once the servers' information can reach each other.

This consensus algorithm in this design is based on an eventual path propagation algorithm. When there's a change in the membership of a group, all servers in the group exchange their view of the network, expressed through a summary 2D matrix of messages they believe have been received by the other servers. The server that has the most up to date messages from a given server is responsible for distributing those messages to the other servers.

The messages that change the state of the application are: new messages, read updates, and delete updates. For each server, an append only log is maintained to keep track of these updates. This is described in further detail below.

Group Membership:

Each server starts as a member of two groups. Server i will be a member of group server_i_in and group all_servers. server_i_in is used to establish a connection and send messages to server i, and only has server i as a member. All_servers is the group used for communication among servers.

Each client starts as a member of two groups, client_<name>_connect and client_<name>_in where <name> is a uniquely identifying integer generated at runtime.

Communicating between client and server:

When a client wants to connect to server i, it will send a CONNECT message to group server_i_in with the name of the connecting client. Upon reading this message, server i will join the group client_<name>_connect. The client will receive a membership message when this connection occurs, which is how it can confirm that a successful connection has been established. If server i crashes or the client leaves, the other party will get a membership message in client_<name>_connect notifying it of the absence.

As long as both parties are members of client_<name>_connect, the client is allowed to send messages to the server. To send a request, the client will send a message to group server_i_in with a command. The server will respond by sending a message to client_<name>_in. This is how a private two way communication channel can be created between server and client.

Logging

The primary method of persistence in our application is an append-only log for each server. For server i the file server_i.log on server j contains a list of all message, read, and delete updates known by server j to have originated from server i. The messages can be applied in any order to achieve the same state because they have a precedence of delete > read > message. That is, if a server tries to apply a read or delete update for a message it has not yet received, it must apply that update to the message upon receiving it. Similarly, if a read update and delete update are both applied to the same message, no matter the order the message will be deleted after both updates.

To serve user requests more quickly, the server stores the state in memory as a data structure which maps usernames to inboxes. This way the server does not need to apply all updates in a log in order to show a user her mailbox. We would also like to avoid the unscalable practice of applying all log updates to the state every time the server starts. To do this, we periodically serialize the local in-memory state to a file as a JSON object so that on startup, only the updates that were written after serializing the state need to be reapplied.

The final wrinkle in this logging procedure is that each file named server_i.log is actually a series of small files (named server_i_X.log where X is a monotonically increasing integer) so that the server can discard messages once they are no longer necessary. Once a server has knowledge that all other servers have received message n from a particular server, it can delete all log files for which the entries contained within the file have indices less than n.

Client protocol:

State:

Bool connected
Bool logged_in
string connected_server_group
String username

Int session id

list<MailMessage> inbox

Repeat:
    wait for input from user or message from spread

```
On receive user input:
    If command == ‘c <new server>’:
        If connected:
            Call Disconnect
        Set connected_server_group = server_<new server>_in
        Send ConnectMessage to connected_server_group with client id as data
        Set connection timeout
    If command == ‘u <username>’
        If logged_in:
            Call Log out
        Set username = username
    If command == ‘m’
        Require client is connected
        Require user is logged in
        Send MailMessage with user input as data to
            Connected_server_group
        Add message stamp as a key to requests
    If command == ‘r <message_index>’
        Require client is connected
        Require user is logged in
        Require message index exists
        Send ReadMessage with inbox[message_index].id as data
        Wait to read server response
    If command == ‘d <message_index>’
        Require client is connected
        Require user is logged in
        Require message index exists
        Send DeleteMessage with inbox[message_index].id as data
        Wait to read server response
    If command == ‘l’
        Require client is connected
        Require user is logged in
        Send ShowInboxMessage
        Clear inbox
        Wait for server response

            If membership message is received from client_<name>_connect:
                Log out
            If ServerInboxResponse is received
                Append mail headers to inbox list
                Print mail headers
    If command == ‘v’
        Require client is connected
```

```
        Require user is logged in
        Send ShowComponentMessage
        Wait for server response

While waiting for server response:
    On receive INBOX message:
        Append contents to inbox
        Print out contents
    On receive COMPONENT message:
        Print out contents
    On receive membership message from client_<name>_connect:
        Disconnect
        Stop waiting
    On receive ACK message
        Stop waiting

On receive membership message from client_<session_id>_connect
    If number of members == 2:
        Set connected = true
    Else
        Call Disconnect
        Alert user that they must connect to a different server
On connection timeout:
    Call Disconnect
    Alert user that they must connect to a different server

Subroutine Disconnect:
    If connected:
        Set connected = false
        Leave group server_<session_id>_connect
        Generate new session_id
        Join group server_<session_id>_connect

Subroutine Log out:
    Set logged_in to false
```

Server protocol:

State:

int knowledge[5][5] where knowledge[i][j] is the last message the ith machine that it has received form machine j, determined by knowledge messages we have received.

```
UserCommand * log[5][variable]
UserCommand * queue[MAX_QUEUE_LENGTH]
Set<string> client_connections

State state

Algorithm:

Read file state.json (if exists) into boost::property_tree object, populate state with object's data

For each server n = 1 to 5:
    Read log file starting at line state.messages_safe_delivered[n]
    While more lines to read:
        If command index > state.applied_to_state[n]
            Call apply_update(log update)
        Append log update to log[this machine][n]


On receive CONNECT message in server_i_in:
    Add client name in message to client_connections
    Join group client_<name>_connect

On receive membership message from client_<name>_connect
    If number of members < 2:
        Leave group
        Remove client_<name>_connect from client_connections

On receive M,R,D message in group server_i_in data:
    Increment knowledge[this machine][this machine]

    Extract message info into UserCommand command
    Write command to log file
    Call apply_update(command)

    Increment state.applied_to_state[this machine]
    Send command to group

On receive user command in group all_servers:
    If message index == knowledge[this machine][message origin] + 1
        Write command to log file
        Increment knowledge[this machine][message origin]
        Call apply_update(command)
        Increment state.applied_to_state[message origin]
```

```
        If state.applied_to_state % knowledge_timeout == 0
            Send knowledge message to all_servers

On receive knowledge message from server i in group all_servers:
    Replace row i in knowledge with message content
    Call collect_garbage()

On receive membership message from all_servers
    Go to state synchronize

Synchronize:
    Send knowledge message
    Wait for knowledge message from all others in group
        If receive M,R,D message in group server_i_in data:
            Write message to log file
            Push message into queue
    Set S to be server indices in this group
    For i = 0 to 4
        If this server has the largest value in column i of knowledge
            Send messages starting from min(column i of knowledge among servers
            in S) to max(column i of knowledge among servers in S)
    If message are safe to discard, discard and increment state.message_safe_delivered
    Pop queued messages one by one and process them as if a user just sent them.

Subroutine collect_garbage(i):
    State.safe_delivered[i] = min element in column i of knowledge
    While first entry in log[i] is valid message and minimum entry in column i of knowledge >
    log[i][0]:
        Pop front of log[i]
    Delete all log files from machine i that are full and only contain commands with index \
    less than state.safe_delivered[i]

Subroutine serialize_state
    Convert state to boost::property_tree object
    Write object to file state.json

Subroutine apply_update(UserCommand command)
    If type == DELETE
        If message in inboxes[uid]:
            Remove message from inboxes[uid]
            append command.id to state.deleted
        Else:
            If command.id in state.pending_read:
                Remove command.id from state.pending_read
```

```
                Append command.id to state.pending_delete
    If type == READ
        If message in inboxes[uid]:
            Mark message in inboxes[uid] as read
        Else if command.id in state.pending_delete or state.deleted:
            Do nothing
        Else:
            Append command.id to state.pending_read

    If type == MAIL:
        If command.id in state.pending_delete:
            remove command.id from state.pending_delete
            Put command.id in state.deleted
            return
        Else if command.id in state.pending_read:
            Remove command.id from state.pending_read
            Extract command info into MailMessage m
            Mark m as read
        Else:
            Extract command info into MailMessage m
        Insert m into inboxes[uid]
    Periodically serialize state and broadcast knowledge message to network
```

Data Structures

```
enum MessageType
{
    // Client to server messages
    CONNECT,
    MAIL,
    READ,
    DELETE,
    SHOW_INBOX,
    SHOW_COMPONENT,

    // Server to client message
    ACK,
    INBOX,
    RESPONSE,
    COMPONENT,

    // Server to server messages
```

```
    COMMAND,
    KNOWLEDGE
};

/* Unique identifier for any command */
struct MessageIdentifier
{
    int index;
    int origin;
};

/* Message signaling incoming client connection */
struct ConnectMessage
{
    MessageType type = MessageType::CONNECT;
    uint32_t session_id;
};

/* Message containing new mail message from client */
struct MailMessage
{
    MessageType type = MessageType::MAIL;
    uint32_t session_id;
    int seq_num;
    char username[MAX_USERNAME];
    char to[MAX_USERNAME];
    char subject[MAX_SUBJECT];
    char message[EMAIL_LEN];
};

/* Message requesting mail to be marked as read by client */
struct ReadMessage
{
    MessageType type = MessageType::READ;
    uint32_t session_id;
    int seq_num;
    char username[MAX_USERNAME];
    MessageIdentifier id;
};
```

```cpp
/* Message requesting mail be deleted by client */
struct DeleteMessage
{
    MessageType type = MessageType::DELETE;
    uint32_t session_id;
    int seq_num;
    char username[MAX_USERNAME];
    MessageIdentifier id;
};

/* Message sent by client to request the current user's inbox */
struct GetInboxMessage
{
    MessageType type = MessageType::SHOW_INBOX;
    uint32_t session_id;
    int seq_num;
    char username[MAX_USERNAME];
};

/* Message sent by client to request all connected server names */
struct GetComponentMessage
{
    MessageType type = MessageType::SHOW_COMPONENT;
    uint32_t session_id;
};

/* Wrapper for Read, Mail, and Delete messages with metadata used by
Server */
struct UserCommand
{
    MessageIdentifier id;
    time_t timestamp;
    std::variant<
        MailMessage,
        ReadMessage,
        DeleteMessage
    > data;
};

/* Message used for sending a server's network view to other servers */
```

```
struct KnowledgeMessage
{
    MessageType type = MessageType::KNOWLEDGE;
    int sender;
    int summary[N_MACHINES][N_MACHINES];
};

/* Message to end response to a client */
struct AckMessage
{
    MessageType type = MessageType::ACK;
    int seq_num;
    char body[300];
};

/* Struct containing all critical user data about a mail message */
struct InboxEntry
{
    bool read;
    time_t date_sent;
    char to[MAX_USERNAME];
    char from[MAX_USERNAME];
    char subject[MAX_SUBJECT];
    char message[EMAIL_LEN];
};

/* Struct that stores user and server data for a mail message */
struct InboxMessage
{
    MessageType type = MessageType::INBOX;
    MessageIdentifier id;
    InboxEntry msg;
};

/* Message used to share the names of current connected group with client
*/
struct ComponentMessage
{
    int num_servers;
    char names [5][MAX_GROUP_NAME];
```

```
};

/* Wrapper for any server message to client */
struct ServerResponse
{
    MessageType type = MessageType::RESPONSE;
    int seq_num;
    std::variant<
        AckMessage,
        InboxMessage,
        ComponentMessage
    > data;
};

/* Struct containing data displayed by client after an inbox request */
struct InboxHeader
{
    time_t timestamp;
    char subject [MAX_SUBJECT];
    char sender [MAX_USERNAME];
    bool read;
    MessageIdentifier id;
};

/* Message containing up to 20 inbox headers */
struct ServerInboxResponse
{
    MessageType type = MessageType::RESPONSE;
    int mail_count;
    InboxHeader inbox[20];
};









/*
     In memory state. This is what gets serialized to disk periodically.
```

```
*/
struct State
{
    int knowledge[N_MACHINES][N_MACHINES];
    int safe_delivered[N_MACHINES];
    int applied_to_state[N_MACHINES];
    std::unordered_map<std::string, std::multiset<InboxMessage>> inboxes;
    std::set<MessageIdentifier> pending_delete;
    std::set<MessageIdentifier> pending_read;
};
```

Scenarios that have been tested:

If no additional note, means it behaved as expected during testing.

List, mail, read, delete, component with multiple servers connected, changes synced

RECOVERY/CRASHING
State recovery with multiple servers (1-5 synced, mail from s1, kill all, then only bring back 5, mail still there)

Start all servers, crash one, action, recover

PARTITIONING
partition, read, merge
Partition, mail, merge
Partition, read & delete same message, merge

Partition, mail + read + delete, partition, mail + read + delete, merge (does not work, read and inbox is all wrong)

*Occasional bug with listing components, number is right but server not printed after partitioning - STASHED
*Cant connect to a partitioned server - FIXED
*mail not displayed in time sorted order - FIXED
*Occasionally connecting with another client will disconnect some other clients/itself

Garbage collection - IN PROGRESS
Periodically send knowledge messages - IN PROGRESS

Crash before completely sending synch

- No mail being sent
- Pending delete may not work