Tyler Sorrels
Ankur Brahmbhatt

Distributed Mail System Design Document

General System Description:

This system runs with up to five servers and unlimited clients. Clients are used to create users (which must be a unique ASCII string), create emails, delete emails, read emails, list emails, and print the current membership of the server group of the server to which the client is connected.

Clients must be connected to a server to operate. A client program will indicate whether it has a connection with a server.

A server can tolerate a system fault and recover from saved state written to disk.

Servers cache updates to the system. In this way, they can forward to other servers updates generated and received from a different server. In this way, knowledge will propagate between servers if an eventual path exists between them.

Servers reconcile after network partitions heal by exchanging matrices that indicate servers' most current knowledge. Only servers with the most current knowledge (tie breaker by highest server ID) will send updates to the new membership. Updates are sent in order of Lamport Time Stamp.

The system uses weak consistency, and therefore allows duplicate operations on state. An example would be creating a user 'A' on two partitioned servers. When they reconcile, they will exchange updates for creating user 'A,' but neither will modify its state.

The read email and delete email operations are always applied in every server after the update to create the email. We accomplish this by sending all updates in order of LTS.

In the event of cascading partition merges, servers will stop reconciliation processes and begin again.

Updates

This system revolves around generating, sending, and applying updates. All changes to state in the servers happen when a server operates on an update to modify the state.

The most critical component of state in the server is a linked list of user data structures. A user consists of a unique string identifier and a list of mails that servers as an inbox. These mail lists are linked lists of email data structures, sorted by the Lamport Time Stamp in each of the email data structures.

The system uses weak consistency to maintain causal order among emails. Mails generated at logically the same time (on two partitioned servers) are sorted by LTS during reconciliation.

When a server receives a new update, it updates it's update vector to reflect it received this update.

Servers send updates in order, so it is not possible for servers to apply updates generated in a logical order other than the order in which they were generated. Functionality between the client and server further ensure a server only performs a read or delete operation only after applying the update that generated that email in the server state.

During reconciliation, servers process their update matrices to identify updates which all servers report as having received, and then purge any of these updates which the server still has stored in state.

Reconciliation

When a server receives a membership change message for the server group, it will trigger the reconciliation process (providing the number of members in the group is greater than 1). This process begins by the servers sending their matrix.

The matrix is a data structure representing the most current knowledge a server has about all other servers. This knowledge is the latest updates received, sorted by the server that generated the update. Visually, this is depicted as:

Updates, Generated on Server

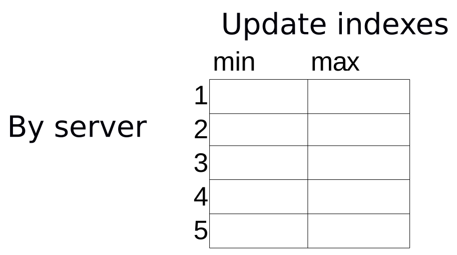
From Server

	1	2	3	4	5
1					
2					
3					
4					
5					

Servers initialize this matrix to -1, indicating servers have not yet generated any updates. A row in the matrix that corresponds to a server's ID is the server's vector. The server updates this vector when receiving and processing updates.

Servers will wait to continue with reconciliation until they have received the number of matrices equal to the number of members in this group. For every matrix received, a server will compare it's local matrix with the received matrix, and update the local matrix with any data more current. After receiving the proper number of matrices, the server will examine it's matrix and compare it's vector with the vectors of all other servers in the group. Servers will mark updates to send if a) it has a more recent update from a server than a server in the current membership is reporting, b) the more recent update is the highest update for that server reported by any server in the current membership, and c) this server's ID is the highest of any other servers in the group that also report having the same update for the server. Then, this server will mark the lower bound of the range of updates to send by finding in the matrix the lowest update reported for the target server. Finally, the server will send the updates marked in order of LTS.

The server marks these updates in state in a structure called FC (short for Flow Control), which holds the minimum and maximum indexes for updates to send, sorted by the server that generated the update. Visually, this is depicted as:



Entries of -1 indicate no updates are marked as to-be-sent.

Cascading network partitions/mergers: when a server receives a membership message, it initiates reconciliation, regardless of it's status (whether it is currently reconciling, or not).

Flow Control

The server implements flow control when sending updates during reconciliation. This protects against the system failing in a case in which a server sends in immediate succession a number of messages (updates) that is greater than the size of the spread mailboxes of the servers. A server may need to send such a large number of updates/messages in a case where a network heals after being partitioned for a long time.

The servers make use of Spread guarantees to ensure they do not overwhelm a mailbox. Servers do this by reading the messages they send. Safe delivery service ensures that any message delivered from spread to the application has been received at the other spread nodes. Therefore, a server can throttle it's own messages by ensuring spread delivers back to it any message it sends. The general implementation is a server initially sends up to 25 updates out to spread during reconciliation, and then has an opportunity to send another message after it receives one of it's own messages.

Tyler Sorrels
Ankur Brahmbhatt

State Recovery Design

Servers write their state to disk in multiple files: a file for the current update matrix, a file holding the list of all users, and file for every user, and a file for each server that stores the updates generated from this server. Servers store their files in a folder "./recovery_files*" where * is the server ID.

When a server starts, it checks for the existence of recovery files. If they exist, the server loads state from these files. The recovery continues iteratively in users, emails, and updates, so that if the server crashed while writing to state, only the final entry will be incomplete, and all other state can be recovered.

When updating state on disk, the server first makes a copy of a file it will update. It then writes the new state to this temporary file. After successfully writing, it will replace the old state file with the new file. This protects against losing all state when overwriting recovery files.

With every update, we always first overwrite the new update buffer and new update matrix to their respective files, before applying the update. We then apply the update and overwrite the specific user recovery file with the new snapshot of user's mailbox.

There is a chance the server can crash in a precise place in the update processing pipeline in which a server which generates an update has written the update to disk but not yet applied it to local state. For this reason, during server initialization, server always replays the last update in the update buffer. This protects against the edge case described, and has no adverse effect on the state, as the server will reject duplicate updates (for example, emails with identical LTS).

Client Design

The client communicates with the server using a query – response model. To "connect" to a server, the client loads the server's hard coded public group from which it receives messages from clients in the 'To' fields for spread messages. It also joins the server's membership group, which is used only to test whether the server is currently in the group.

When a client sends a command to the server, it sends the message to the public server group to which the client does not belong, and encodes in the message the client's own private group as a return address. The server processes the command and sends a message to the private address of the client.

When a client connects, or is connected to a server, is monitors all membership messages and checks to see whether the server to which it is connected is among the members in the group. If not, the client "disconnects," and requires the client to call connect again with a server with which it shares a partition.
Emails are presented for a user in order of LTS.

Tyler Sorrels
Ankur Brahmbhatt

Critical Structures and Types:

```
/* System wide message types */
typedef enum {
    RECONCILE,
    SENDINGUPDATES,
    PARTITIONED,
    NORMAL,
} server_status_type;

/* System wide message types */
typedef enum {
    COMMAND,
    UPDATE,
    MATRIX,
    REPLY,
} message_type;

/* Update message types */
typedef enum {
    NEWMAILMSG,
    NEWUSERMSG,
    DELETEMAILMSG,
    READMAILMSG,
} update_type;

/* Command message types */
typedef enum {
    NEWMAILCMD,
    NEWUSERCMD,
    DELETEMAILCMD,
    READMAILCMD,
    LISTMAILCMD,    // does not modify state
    CONNECTCMD,     // does not modify state
    SHOWMEMBERSHIPCMD,  // does not modify state
} command_type;


/* Header for all system wide messages, sent through spread */
typedef struct message_header_type {
    message_type type;
    int proc_num;
} message_header;
```

Tyler Sorrels
Ankur Brahmbhatt

```
/* Message type, send to spread API for group multicast */
typedef struct message_type{
    message_header header;
    char payload[MESSAGE_PAYLOAD_SIZE];
} message;



/* Lamport Time Stamp  placed on all updates and memails
typedef struct mail_id_type{
    int procID;
    int index; // global counter, can count emails or udpates
} mail_id;

/* Structure for update */
typedef struct update_type{
    update_type type;
    char user_name[MAX_USER_LENGTH];
    mail_id mailID;
    char payload[UPDATE_PAYLOAD_SIZE];
    struct update_type *next;
} update;


typedef struct update_vector_type{
    int size;
    int capacity;
    update * updates;
} update_vector;


/* Structure to cache updates from all processes */
typedef struct update_buffer_type{
    update_vector procVectors [NUM_SERVERS];
} update_buffer;

/* Structure to maintain latest update index from all processes */
typedef struct update_matrix_type{
    /* first dimmension is server the vector is from, second is vector */
    int latest_update[NUM_SERVERS][NUM_SERVERS];
    /* reset to 0 during every change to membership */
    int num_matrix_recvd;
} update_matrix;
```

Tyler Sorrels
Ankur Brahmbhatt

```
/* Email data type */
typedef struct email_type{
    int read;
    mail_id mailID;
    char from[MAX_USER_LENGTH];
    char to[MAX_USER_LENGTH];
    char subject[MAX_SUBJECT_LENGTH];
    char message[MAX_MESSAGE_SIZE];
    struct email_type * next;
} email;

/* Structure for command */
typedef struct command_type{
    command_type type;
    int ret;
    char user_name[MAX_USER_LENGTH];
    char payload[UPDATE_PAYLOAD_SIZE];
    char private_group[SIZE_PRIVATE_GROUP];
} command;

/* For vector library */
typedef struct email_vector_type{
    int size;
    int capacity;
    email * emails;
} email_vector;


/* user data type*/
typedef struct user_type{
    char name [MAX_USER_LENGTH];
    email_vector emails;
    struct user_type * next;
} user;


/* State local to each server */
typedef struct state_type{
    int proc_ID; /* this server's ID */
    server_status_type status;
    char server_group[MAX_GROUP_NAME]; /* spread group name for server */
    char private_group[MAX_GROUP_NAME]; /* server's private group */
    int updateIndex; /* counter of updates received, for LTS */
    user_vector users;
    update_matrix local_update_matrix;
    update_buffer local_update_buffer;
    int recoveryFD;
    char current_membership [NUM_SERVERS][MAX_GROUP_NAME];
    flow_control FC;
} state;
```

Tyler Sorrels
Ankur Brahmbhatt

Server Pseudo Code

```
init()
    join server group
    join unique server group
    join unique server membership group



processMembershipMessage()
    ensure membership message is for the server group
    copy to state the current membership of the server group
    set # or matrices received to 0
    checkReconcile()
        return 1 if number of group members is greater than 1
    if (reconcile)
        set state.status to RECONCILE
        sendMatrix()


processRegularMessage()
    if (message is a command)
        if (command generates an update)
            generateUpdate()
                repackage critical pieces in command into an update

            applyUpdate()
            sendUpdate()

        generateResponse()

    else if (message is an update)
        if this is an update from this server && status == SENDINGUPDATES
            sendMissingUpdates(num_to_send = 1)
                finds next update to send in reconcilation, and sends it
            if none left, status = NORMAL
        applyUpdate()


    else if (message is a matrix)
        updateMatrix()
        writeMatrix()
        increment num_matrix_recvd
        if(num_matrix_recv == size_of_current_group)
            state.stutus = SENDINGUPDATES
            continueReconcile()
```

Tyler Sorrels
Ankur Brahmbhatt

```
continueReconcile()
    purge update buffer of updates already reported as recvd by all procs
    for every server (column) in the update matrix:
        check if local vector reports a more recent update than reported by all
            other processes in the current membership, store in max
        if True:
            search matrix for lowest update for this server in current
                membership, store in min
            If min < max:
                load into state.FC this min and max for this server

        if False:
            load into state.FC -1 and -1 for min, max to indicate no updates
                to send for this server

    sendMissingUpdates(MAX_MESS):
        while (num_sent<MAX_MESS && there are messages to send)
            traverse state.FC to find update marked to send with lowest LTS
            if that update exists, send that update
            incrememnt num_sent
            if (min == max)
                mark min and max for that server as -1, marking that
                    there are no updates to send for that server
            increment min for that server in state.FC



applyUpdate()
    search state for update with the recv'd update's LTS
    if (exists)
        ignore this update

    store update in state by procID
    updateVector()
    write update buffer to disk
    write update matrix to disk
    switch(update type)
        addUser()
            append to vector of users, initialize new user's email vector
        addMail()
            search state for user
            insert mail into user mail vector, sorted by LTS
        deleteMail()
            search state for user
            search for email, remove if found
        readMail()
            search state for user
            search for email, mark as  read if found

    set local update index to max(local_update_index + 1, index on update)
```

Tyler Sorrels
Ankur Brahmbhatt

Client Pseudo Code

```
userCommand()
      wait for input;
      parseCommand();

parseCommand(command)
      switch(command){
       case 'c': connectToServer();
       case 'u': loginUser();
       case 'l': listHeaders();
       case 'm': mailSetup();
       case 'd': deleteMail();
       case 'r': readMail();
       case 'v': printMembership();

readSpreadMessage(
      if REGULAR Spread message:
            if(commandType == LISTMAILCMD)
                  if(firstPacket)
                        record how # emails the to receive in maxNumMails

                  else
                        append the mail to the list;
                        if received maxNumMails
                              displayList()


            else if(commandType == NEWUSERCMD ||
                  commandType == NEWMAILCMD ||
                  commandType == DELETEMAILCMD)
                        check whether operation was successful;

            else if(commandType == READMAILCMD)
                  display the content of the mail;


            else if(commandType == SHOWMEMBERSHIPCMD)
                  display the current membership.


      If MEMBERSHIP Spread message:
            check whether this is for the connected server membership group
            check whether the server is in the group
            if not:
                  set connected to 0
                  print “Disconnected from Server*”
```

Tyler Sorrels
Ankur Brahmbhatt

Implementation Details on Select Methods

userCommand()

-Waits for user input.
-calls parseCommand()

parseCommand()

-Parses User command.
-Checks whether the user is connected to a server
before running any other commands.
-Checks whether the user is logged in with a username
before running any other commands, besides connect command.
-Checks whether the user has called 'l' command before
executing 'd' or 'r' commands.
-Validates the command and its arguments.
-Calls the relevant function to prepare and send a query to the server.

loginUser()

-Creates a struct command with type NEWUSERCMD.
-Copies the username inside that command struct.
-Copies the private group of the client inside the command struct.
-Wraps the command struct in a message struct.
-Sends it to the server-client group of the server that it is connected to.

connectToServer()

-Joins the server-membership group in order to keep track of
server's state.

listHeaders()

-Creates a struct command with type LISTMAILCMD.
-Copies the username inside that command struct.
-Copies the private group of the client inside the command struct.
-Wraps the command struct in a message struct.
-Sends it to the server-client group of the server that it is connected to.

Tyler Sorrels
Ankur Brahmbhatt

mailSetup()

-Waits for message related input from user.
-Creates a struct command with type NEWMAILCMD.
-Copies the username inside that command struct.
-Copies the sender name inside that command struct.
-Copies the receiver name inside that command struct.
-Copies the subject inside that command struct.
-Copies the message content inside that command struct.
-Copies the private group of the client inside the command struct.
-Wraps the command struct in a message struct.
-Sends it to the server-client group of the server that it is connected to.

deleteMail()

-Creates a struct command with type DELETEMAILCMD.
-Copies the username inside that command struct
-Copies the private group of the client inside the command struct.
-Copies the relevant email from its local buffer into the command struct.
-Wraps the command struct in a message struct.
-Sends it to the server-client group of the server that it is connected to.

readMail()

-Creates a struct command with type READMAILCMD.
-Copies the username inside that command struct.
-Copies the private group of the client inside the command struct.
-Copies the relevant email from its local buffer into the command struct.
-Wraps the command struct in a message struct.
-Sends it to the server-client group of the server that it is connected to.

printMembership()

-Creates a struct command with type SHOWMEMBERSHIPCMD.
-Copies the username inside that command struct.
-Copies the private group of the client inside the command struct.
-Wraps the command struct in a message struct.
-Sends it to the server-client group of the server that it is connected to.

Tyler Sorrels
Ankur Brahmbhatt

readSpreadMessage()

- calls processRegularMessage() on receiving a regular message.
- calls processRegMembershipMessage() on receiving a membership message.

processRegularMessage()

- Check If message type is LISTMAILCMD
- Deletes the previous email list from local buffer.
- First packet of this type would include the number of emails that follow.
- Append each email at the end of the list.
- call displayList() once all the emails are received.
- Check If message type is NEWUSERCMD or NEWMAILCMD or DELETEMAILCMD
- Check the "ret" field in the packet to see if the operation was successful.
- Check If message type is READMAILCMD
- Print the content of the mail if the operation was successful.
- Check If message type is SHOWMEMBERSHIPCMD
- Print the membership if the operation was successful.

displayList()

- Display the username and server index.
- Display all the emails in the list, mark unread mails with an asterisk.

processRegMembershipMessage()

- Check to see if the membership message is for the server-membership group that the client is connected to.
- Check to see if the server name, that the client is connected to, is present in the groups.
- If server name is not found, then notify the client about the disconnection

loadState()

- calls recoverUserList() if "userlist" file exists.
- calls recoverUser() for every user in "userlist" file.
- calls recoverUpdateMatrix()
- calls recoverUpdateBuffer()

recoverUserList()

- Create the list of users, copies the name of the user to its node.
- Initializes the rest of the fields of the user node.

Tyler Sorrels
Ankur Brahmbhatt

recoverUser()

-Opens the user email file.
-Loads the emails of the user into the memory one by one.
-Loads the size field of the user inbox.

recoverUpdateMatrix()

-Opens the "updatematrix" file.
-Loads the update matrix in the memory.

recoverUpdateBuffer()
-Opens update buffer file for each server.
-Loads update buffer for each server in the memory.

writeUserList()

-Creates "userlisttemp" file.
-Writes all the user names to it.
-Removes "userlist" file if it exists.
-Renames "userlisttemp" to "userlist".

writeUser()

-Create a temp file for the user.
-Writes all the emails of that user in that file.
-Removes the existing user emails file.
-Renames temp file to username.

writeUpdateMatrix()

-Creates "updatematrixtemp" file.
-Writes the update matrix to the file.
-Removes the "updatematrix" file.
-Renames the "updatematrixtemp" file to "updatematrix".

writeUpdateBuffer()

-Creates temp update buffer file for the given server.
-Writes the update buffer of that server to that file.
-Removes the old update buffer file.
-Renames the temp update buffer file to update buffer file.