FinalDesign

Jiayao Wu, Suyi Liu

November 2016

Note: We are assuming the username doesn't contain any space in this project. For example: u Suyi Liu will be considered as u Suyi.

1 Basic Idea

We want to build a fault tolerant mail service which consists of a mail server program and a mail client program using anti-entropy protocol.
There are 5 mail servers (daemons) that run forever and may partition and re-merge.
There are countless number of clients possible.

2 Methods

mail client's methods:
1. Login/Login with a new name (eg: u Tom)
2. Connect/Switch connect to a server (eg: c 3)
3. List the headers of received mail
4. Mail a message to user
5. Delete a mail (eg: d 10)
6. Read a mail (eg: r 5)
7. Print membership.

mail server:
1.receives messages(from sp_monitor, or from client, or from server)
2.Do the appropriate action on the data structures for: Read/Delete/Mail
3.multicast updates into partition
4.multicasts knowledge (mails) array into group
5.Give client back all the needed information
6.All the methods needed to construct our data structure (the lists, like add a node, delete a node, get a node etc.)

3 Data Structures

Note: the size of many data structures we chose to be 6 just so that 1 to 5 can be stored at 1 to 5.
**Each Mail(message) has:
int stamp
int server
int read(0 as unread, 1 as read)
char[] sender
char[] receiver
char[] subject
char[1000] msg

**Each Update struct has:
int type (1 means normal update, 2 means update_add, 3 or 4 means update_r_d, 5 or 6 means update_l_p)
char[] client(if the update is 'add', the client should be the receiver)
int operation(2 for add, 3 for read, 4 for delete)
int server_index
int server_stamp
int server
int index(the index of specific mail in the list)
int array[6]

**Each Update_add struct(Which is used when client tells server sending a mail) has:
int type (must be 2)
char client[], int server_index, Mail mail

**Each Update_r_d(Which is used when client tells server to delete or read, so it doesn't need space for mail) struct has:
int type (must be 3 or 4 for read or delete)
char client[], int server_index, int server_stamp

**Each Update_l_p(For listing and printing, (It turned out only be used for listing actually)we only need client's name) struct has:
int type (must be 5 or 6 for listing and printing membership)
char client[], server_index

**Each Update_node struct has:
Update *update, Update_node *next

**Each Update_list struct has:
Update_node *head, Uodate_node *tail, int count

**Each Mail_node has:

Mail *mail, Mail_node *next

**Each Mail_list has:
char client_name[], Mail_node dummyhead, Mail_node *tail, int count, Mail_list *next

**Each Mail_lists has:
Mail_list *head, Mail_list *tail, int count

**Each Update_k has (used in anti-entropy protocol, as well as printing membership info to user, since the infomation they need is mainly a vector):
int type (must be 9 for anti-entropy protocol and 8 for printing infomation sending back to client, no conflict), int array[6]

Summary: each Update_list is consisted of Update_node that have Update pointers, each Mail_list is consisted of Mail_node that have Mail pointers, each Mail_lists is consisted of Mail_list.

**mail client has the following data structures:
char[] username (the client currently connected to this server)
int server (the current server for this client)
Mail_list list: needed for read and delete option because the client will first be given a list of most updated mails, then need the list to get the specific mail's stamp and index numbers for the server to find the correct mail.

**mail server has the following data structures:
int server_id(its server index)
an array of Update_list(size 6).
int matrix of size 6*6 called matrix, where each entry(i,j) represents length of server i's knowledge of updates happened on server j(the length of j's update linkedlist stored on server i).
a Mail_lists called mls that stores each client's mails
int timestamp: gets incremented each time adding a new Mail_node to the mls.
int timeindex: get incremented each time before multicasting the update to the group
other_members[6] array stores: those who are in the same partition with this server will be marked as 1
who_is_in_charge_of_i[6] array stores: which server has the best knowledge for updates for each column in the matrix
max_knowledge[6] array stores: for each column in the matrix, what is the largest number of updates (only look at servers in the same partition)
min_knowledge [6] array stores: for each column in the matrix, what is the smallest number of updates (only look at servers in the same partition)
min_knowledge_of_five[6] array stores: for each column in the matrix, what is the smallest number of updates (look at all 5 servers)

4 Algorithms

Client and server are connected through a spread.
Each client is a private group that sends unicast message to the server it connects to. Each server is a public group that any client can connect to. Servers are in the partitions managed by spmonitor.

We are trying to maintain consistency of all the data among all the servers when they are in the same partition.
**For the client's perspective:

1. Initially, it connects to a Private_group via SP.
2. The menu is repeated prompted until the user chooses to quit.
3. For login function: client copies the user name to its variable for later use (for other functions, first need to check if client already logs in)
4. For Connect with a server function: check if it already connects to the same server and if not the same one, then make the group to be server_id where id is 1 to 5, so that later when server needs to communicate with the server, can just be done via this group. If it already connects with the same server, just prompt the menu again and print to the user that "already connects". (for other functions, need to check if the client already connects with a server)
5. For mail a message to user function: Read all the needed input from user and malloc a Mail object. Construct a Update_add (type is 2) object with mail in it. client then sends the Update_add to the server.
6. For read a mail function: first, gives the client a list of most updated mails (same procedure as listing all headers function), then let him choose which one to read, and client needs to find that mail's stamp and index and put them in the update_r_d object (type is 3), which is sent to the to the server. Then the server just go through the mls to find the corresponding message. Client waits to receive the mail back and display to the user.
7. For delete a mail message: similar to read a mail, instead, here, Update_r_d's type is 4, and the server just removes the message's node in mls.
8. For listing all the mails: Make a Update_l_p object (type is 5) and send it to server. Wait for mails one by one sent from the server (with a end flag in the last one) and prints to the screen.

9. For printing the membership: Make a Update_l_p object (type is 6) and send it to server. Wait for a Update_k object from the server and prints out the array information in it.

***For the server's perspective:

The server runs forever waiting for messages. The server joins a group whose name is server_id where id is 1 to 5, so that client and server can communication via this group.

1. In an infinite for loop, receive the message and perform actions based on the message types in the following ways:
2. If the message is of membership type (might result from servers joining the group and the partitions by spmonitor):
   (a) First, reinitialize who_is_in_charge_of_i and other_members arrays. If it's regular membership message, then we can get currentMember-Num (number of current members in this group.)
   (b) multicast its own array from the 6x6 matrix into the group in the form of Update_k object. So, for server 3, just multicast the 4th row in the 6x6 matrix.
   (c) Wait to receive all the other arrays sent from other members in the partition, and update its matrix based on them, ie: copy the array from server 3 to the 4th row in matrix.
   (d) Reinitialize max_knowledge array to have -1 values and min_knowledge and min_knowledge_of_five arrays to have MAX values. Then, vertically compare knowledges of each specific server j, and fill the max_knowledge with the max number from each column(only consider servers in this partition), and fill the min_knowledge with the min number from each column (only consider servers in this partition), and fill the min_knowledge_of_five with the min number from each column (consider all servers). Meanwhile fill the who_is_in_charge_of_i array with its server number if it is the server that knows most about server j (ie: has the max value)
   (e) During testing, we encountered a case where: If server 5 joined 12(1, 2 both have knowledge <1 0 0 0 0>), while server 2 is in charge of sending server's 1st update into the partition, if 1 immediately adds 2nd update, and server 5 gets 1's 2nd before getting 1's 1st update from server 2, so it ignores 2. Then server 5 will never get 1's further updates even though they are in the same partition. It will only get 1's further updates when they are separated and merged again. So, to prevent this, we make servers to be in charge of themselves' updates as much as possible, by traversing through who is in charge of what, if the server is in the partition, set the server who is in charge of sending to be server itself.

(f) If the server is in charge of certain server i, it multicasts updates from k (corresponding number from min_knowlegde) to the end of its updates for server i.

3. If the message is not of membership type:

   (a) First, cast the mess into Update and make decisions based on the type field of the update

   (b) If type is 1 (from other server, regular update mess). Make decisions based on the operation type

      i. If operation if 2 (add operation)
      The server gets the corresponding mail list, then add a Mail node to it, and add Update node to the update list and update matrix accordingly (very similar to the case when type is 2).

      ii. If operation is 3 (read operation)
      The server gets the corresponding mail list, then read the mail node, add Update node to the update list and update matrix accordingly (very simular to the case when type is 3)

      iii. If operation is 4 (delete operation)
      Similar to the above case, except that it deletes the mail node.

   (c) If type is 2 (from client, it's Update_add)
   Retrieve the mail from the mess, then wrap it into a Mail_node and insert it into mls. Then, make a new Update object to contain the operation, add it to the update list and multicast it to the group.

   (d) If type is 3 (from client, it's Update_r_d for read)
   Since we used timestamp and server_index to uniquely label each mail, then traverse through the mls to find the requested mail. If cannot find such mail, multicat a special message back to client. If can find such mail, multicast the requested specific mail back to the client
   Then also, make an Update object based on the operation and add it to the update list and multicast it to the group.

   (e) If type is 4 (from client, it's Update_r_d for delete)
   Similar to the above option, the only difference is that the mail needs to be deleted.

   (f) If type is 5 (from client, it's Update_l_p for listing)
   Server sends the corresponding mails from the mls one by one with the last one that has an ending mark on it.

   (g) If type is 6 (from client, it's Update_l_p for printing membership)
   The server multicasts a Update_k object that contains same array as other_member array.

5 Termination

Server's program: server never terminates, it can only be killed manually after testing.
Client has a choice to exit the program. No need to logout since switching user has the same functionality.

6 Flow Control

The flow control between server and client are established in the flowing ways:

1. whenever the server or client sends messages to each other, it just uses SP_receive to receive the feedback(s), so it always gets what it wants fast
2. Specifically, in the listing headers function, the last mail sent by the server is marked with a finishing mark, so that the client knows when to stop receiving
3. when servers receive updates from other servers, it first checks if the update's index is the next one it's waiting for (so should be consecutive). If not, ignore the update.
4. Other flow control used: for example, when exchanging matrix we only send ours and wait for other members'. For getting membership info, we used prestored other_members array again(This array is used for multiple purpose in the project).

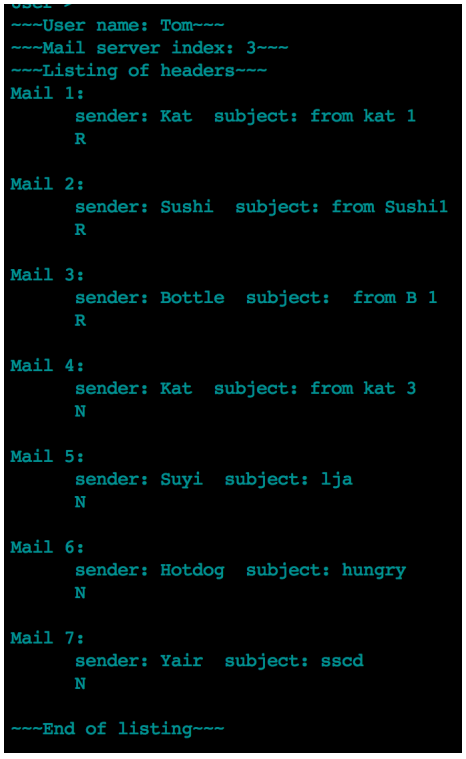
7 Performance Analysis

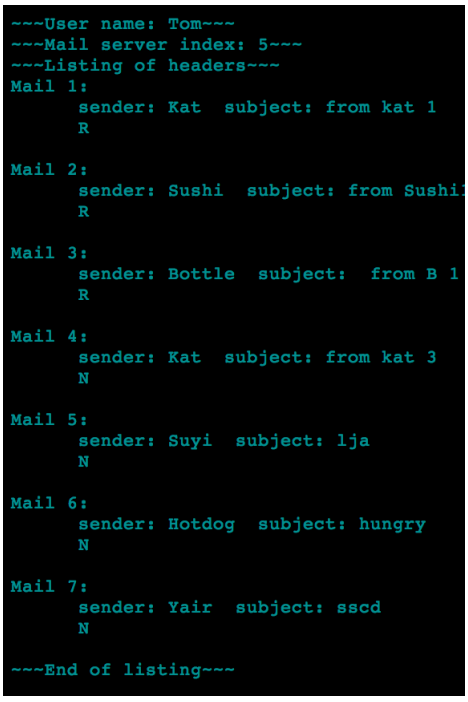
We ran the following scenarios:

1. Partition 5 servers into 5 groups (each one is in its own group). Then connect to each server and send identical messages to the same receiver. Then merge. The result is that it can distinguish the 5 mails. Then, partition again. Connect to each server, and delete the first mail. Then merge, the results are 4 mails in the same sequence.
2. Partition into 2 groups (left and right). 1 client connects to left group and 1 client connects to right group. Read on the left one and delete on the right one the same mail. When merge, it successfully deletes one mail.
3. Similar to above case, but just perform delete on left and right. When merge, it successfully deletes only one mail.
4. Similar to above case, perform add on left, delete on right. Then merge.

5. before the former merging is completed, a new merging happens, our system handles this elegantly

6. the servers are bombarded with multiple partitions in a row and during each partition, we manipulated some random operation. In the end, when the servers merge, the listing results are consistent (but it may take a little bit time to catch up) [following are some screenshots of the results]





8 Design Improvements

1. Before, we wanted to use vectors for data structures, but we thought building our own linkedlist might be better, so we built the data structures for mail list and update list. So we changed a lot on the data structures design.
2. Our anti-entropy method did not work quite correctly before, but now, since we used new data structures, like max_knowledge and min_knowledge, it becomes more clear about what to do.
3. During our testing, we were so perplexed why after the partition, our client can communicate with some server but not the others. But afterwards, we realized that only clients that login to the ugrad that's in that partition can communicate those within the partition.
4. Initially, we were confused with how the private and public group work between server and client, but after discussed with TA, it's more clear what to do.
5. We got very confused about how upon receiving a mess, the server can distinguish what to do(such as whether it is sent from a server or directly from a client, or it needs adding or deleting or reading or simply listing). Then we went to the office hour and learned about casting.
6. We got very confused about struct that contains pointers. How to send them over network? After going to the office hours we realized that we should make struct for sending to contain as little pointers as possible. So we changed the Update struct to contain Mail object instead of pointer.
7. We initially didn't realize that every receive actually overwrites our fields of the update from last round. Improvement: we allocate memory and copy fields every time we receive an update.
8. We got stuck with how to send back a list because it is a nested struct. We came up with the idea in the end of sending mails in order to the client because spread does AGREED ordering for us.
9. Initially we implemented unicast during merging, so we don't check update index since there's no need to ignore previous updates that we've already known and performed on. However, we realized that it is more efficient to multicast to the entire group so that the members themselves can figure out based on the indices. So we removed receiver field in each update and added indices accordingly.
10. We were confused about duplicative mails and how to handle deletion of same mail from two different partitions and merge them together. Actually, the creator server's index and timestamp is necessary to make those mails unique. So we made that improvement accordingly.
11. We improved the efficiency of our system by setting the mail's serverindex and mail's server stamp fields directly into the update if we only want deletion and reading. Because these two are enough to distinguish a mail, no need to contain the mail's other content.
12. We thought over the case where a server accidentally get an update that has rather high index, because it joined a group where there are messages in the air. Should we accept the new update or ignore it? We decide to ignore it because if it is a deletion, it will disturb the order if an addition is not received

yet, so we have to stash it. It will be better to just wait for a short moment till other members get the partition message.
13. During implementation, we missed some of deletion at the beginning of testing, this is because when server b receives server a's update about a deletion, if b has already deleted the corresponding mail, it immediately breaks the statement instead of adding it into its update lists. However, deletion should be done differently from what is done in client-server interaction, because the server should be consistent with other servers about updates and may have the responsibility to further send out the updates to other servers. So we changed the algorithm accordingly.

==========Below is old design==========

9 Basic Idea

We want to build a fault tolerant mail service which consists of a mail server and a mail client.

10 Methods

mail clinet's menu:
1. Login/Login with a new name (eg: u Tom)
2. Connect/Switch connect to a server (eg: c 3)
3. List the headers of received mail
4. Mail a message to user
5. Delete a mail (eg: d 10)
6. Read a mail (eg: r 5)
7. Print membership.

mail server:
There are 5 mail servers (daemons) that run forever and may partition and re-merge.

11 Data Structures

***Each mail(message) has:
int sender
int receiver
char[] subject
char[] content
int read(1 as unread, 0 as read)

**Each update struct has:

int client
int operation(for example, 1 means delete, 2 means read, 3 means receive)
char[] content(only used when some client receives some message)

**mail client:
int sender(its sender id)

**mail server:
int server(its server index) The server keeps a linkedlist of messages based on timestamp.
The server also keeps a linkedlist of updates(each update in the linkedlist also stores the respective update id, eg. the 5th update in this linkedlist stores 5).
The server has a counter called view, which increases when it experiences an update. Maybe this is not necessary
The server keeps an array of size 5 called knowledge that records each other server's knowledge of its update array.
The server has a counter called minimum knowledge, which records minimum knowledge of all other servers, so that the server can delete the updates that all the other servers know to save some memory.

12 Algorithms

For the server's perspective:

1.

After the interaction with the client, the server has added some new instructions into the updates linkedlist. For the group's perspective:
When a group is partitioned(old group will disassemble). The server in the new group that has the least server index is the "leader" of the new group. All the other servers merge their current updates with the leader, and then the leader broadcasts its newly updated updates with all other servers. then after merging with the leader, ther other servers updates all the other server's(all the other servers in the group) knowledge as the latest knowledge itself has. And also do the respective operations on their linkedlist of messages. Then everytime there is an update from a client, the connected server p tells leader and leader broadcasts this update to all servers in the group, then all group adds 1 to every other server's knowledge. The goal is to make sure all the servers in one group have synchronized knowledge of each other before they get partitioned again.
Specifically, When server x is experiencing a merging with server y, server x sends all the updates with id larger than knowledge[y] to y, then set knowledge[y] to the latest y it has. And server y does the same thing to server x.