Design for Distributed System Final Project

Junjie Lei / jlei10
Chanha Kim / ckim135

Requirements

- The number of clients is unlimited.
- Maximised and eventual consistency
- While keeping a consistent view, there should be maximized availability.
- Efficient use of messages (no redundant messages)

General Ideas

Replication

For this project, the replication of emails is a state for each server, and updates are sent through the network. Updates are initialized by the client and sent between the servers.

Availability

When updates are received, the server applies these incremental updates to the state, but still keeps the updates as a log file

Consistency

When there are partitions between the servers, we could only achieve consistency within each connected group. The servers will eventually have the same state when connected in one group.

Knowledge

We use a knowledge matrix for each server to keep track of what other servers know about other servers. This knowledge matrix is used to do garbage collection, i.e., to calculate which updates are not needed and can be freed of memory (allows to check consistency, too).

Recovery

We prepare to crash at any time with the help of log and state. The log is where we store the updates and the state is used to respond to user requests. Both are written to files when any changes happen. The server will have the data restored whenever it reruns, and the reconciliation steps will be the same as that of other network partitions.

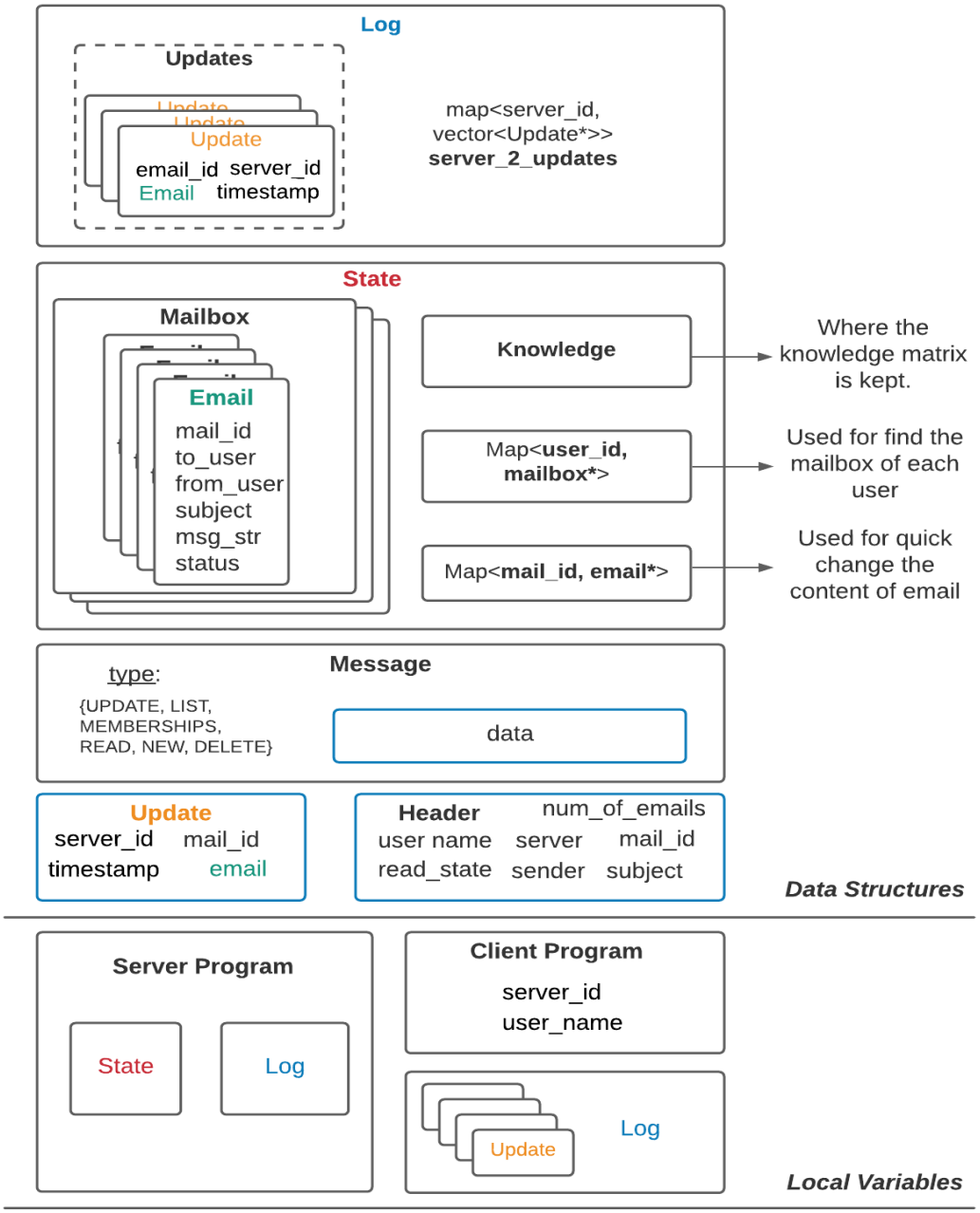
Timestamp

Using a strategy similar to Lamport Timestamps Protocol, when each server sends an update, it will increment its own logical timestamp and put this timestamp on the update message. And the receiving server uses this timestamp to renew its own section in the knowledge matrix(the row on which indicates its own knowledge).

Connections

All connections are done with the help of the spread toolkit and all messages in concern are either membership messages or regular messages.

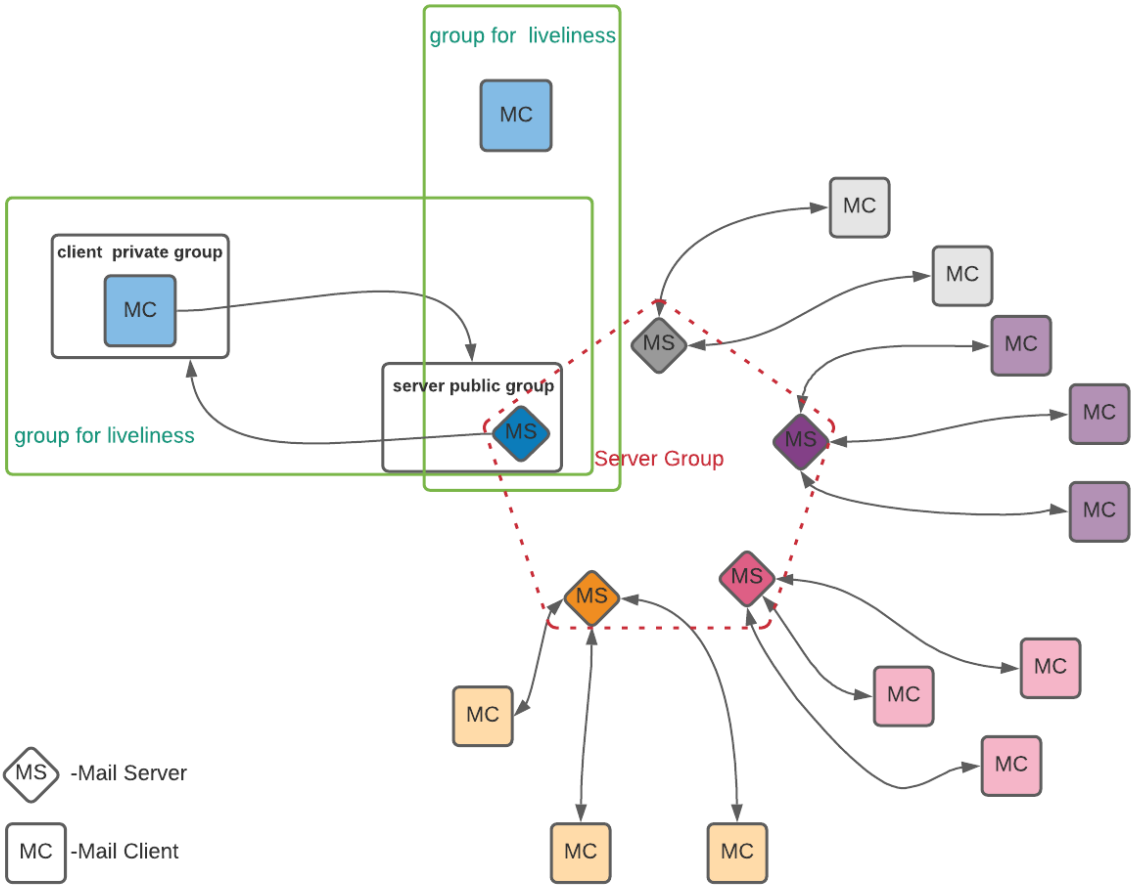
Data Structure and Local Variable



Graph 1 - Local Variables & Data Structures

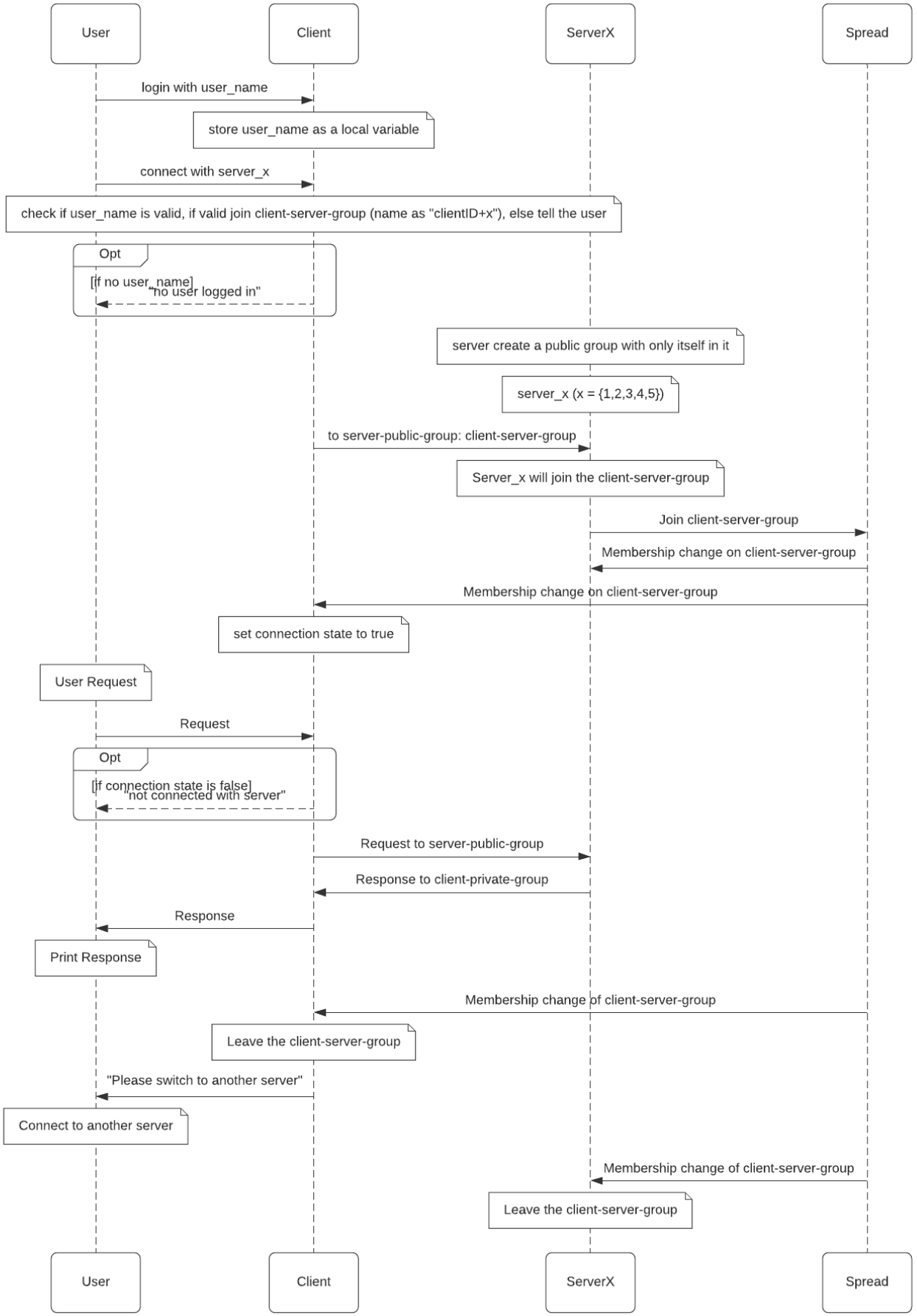
Local variables for a server program and client program are in the lower part of Graph 1 and the data structures are in the upper level of Graph 1.

Client-Server Communication



Graph 2-1 Client-Server Communication

Client/Server communication should be done using Spread via the client's private group and a public group with only that mail server in it (used by connected client programs to send messages to that mail server). And the overall communication is done as graph 2-1 indicates and more details are in graphs 2-2 as a sequence diagram.



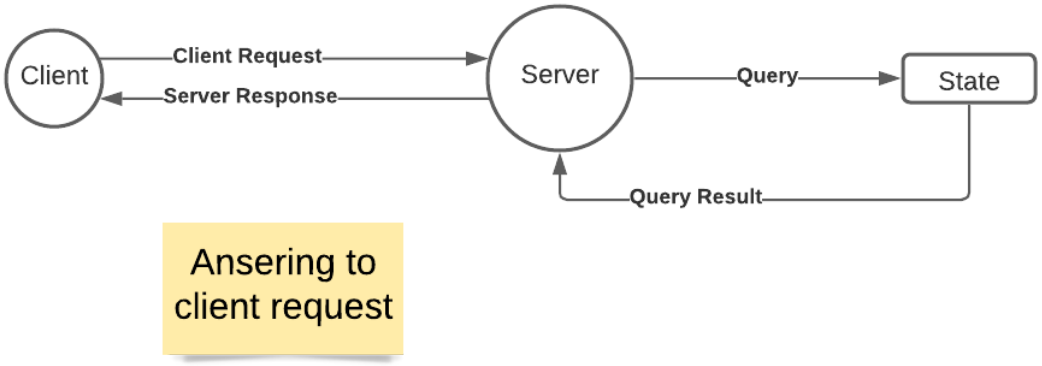
Graph 2-2 Sequence Diagram for Client-Server Communication

Client Functions

- **User Login**
    1. The client will take in “u” as the indication for a wanted login action and another string as the username for this login action. After this, the client program will keep the current username. And whenever it receives another action input beginning with “u” and a new username string, the user state on this client program will be updated.

- **Client Connections with Servers (Graph 2-1 and 2-2)**
    1. First, check whether the user_name is set. Indicates login action if not set.
    2. The client program first connects to the Spread network.
    3. Update the local variable *server* with the user input server number.
    4. Join the client-server-group and send this group name to the server by server-public-group.
    5. When the user tries to connect with a new server:
        a. The client will first leave the joined group.
        b. Join the new client-server-group and send {client-server-group, client-private-group} to the server.
        c. After sending {client-server-group, client-private-group} to the server, a timeout is set, if the server has not joined this client-server-group before the timeout,i.e. if no new member joins the client-server-group before timeout the client will tell the user that the connection has not been established.
    6. When the connection is established, all requests of client functions(list email, read email, new email, delete email, list membership) will be sent to the server-public-group. And all responses will be sent to the client-private-group.
    7. When a server has crashed or partitioned by the network, the client connected to this server a member has exited. Then each client program will exit is the client-server-group. And the client program will indicate to the user that the server has crashed and “please switch to another server.”

- **List E-mail Info**
    1. The client will first check the input to see if it is a valid action input.
    2. Send a list request to the server. When a reply is received, show(in this case, print) the info requested on the client program.
    3. The email data includes the user name, the responding mail server index, the sender, and the subject of each message.
    4. Sort the email on the user interface by sent time.



Graph 3 Answering to client list request

- **Send e-mail**
  1. The client will first check the input to see if it is a valid action input.
  2. Put the corresponding information into the message and send it to the server.
  3. The size for the message is bound to 1000 bytes.

- **Delete e-mail**
  1. The client will first check the input to see if it is a valid action input.
  2. Send a delete request message to the server.

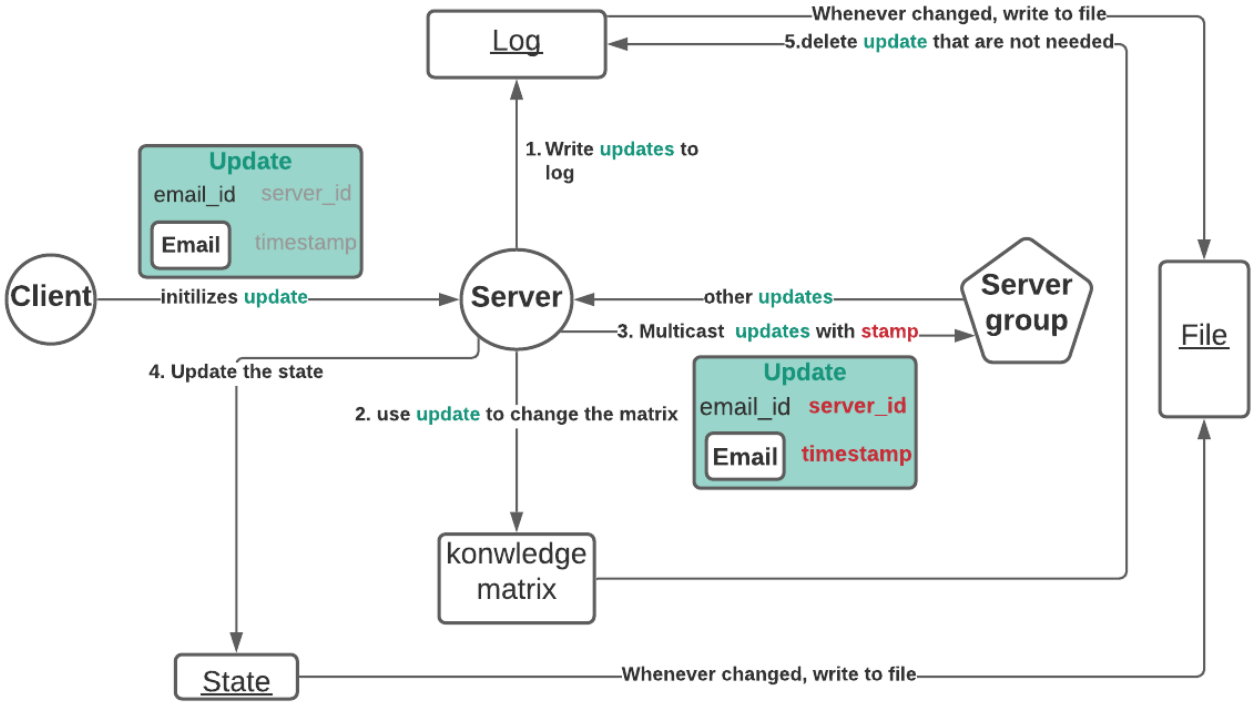
- **Read e-mail**
  1. The client will first check the input to see if it is a valid action input.
  2. Send a reading request to the server, and show the body of the email.

- **View email server memberships**
  1. The client will first check the input to see if it is a valid action input.
  2. Get the information about membership using Spread.
  3. Print out the information on the client program.

Server Functions

- **Keep a consistent view of all the emails**
  1. **The updates**
     As we can see from the client functions, there is a lot of updating of the email inbox for each of the users. There is deletion, marking as read, and receiving new messages. When the server has received such updating messages for a certain user, it will send an incremental changing multicast message to all other servers in the same spread group, so all the servers in this group will update their state with this multicast message.
     - **Update the knowledge matrix**
       When receiving an update, it can change its own row in the knowledge matrix accordingly.
  2. **The consistent feedback to the client**
     Because the same user may be logged in on different servers, this means that whenever a server has received an updates message indicated above, it will update its state. So as long as for the connected group, all feedback will be as consistent as it can get.

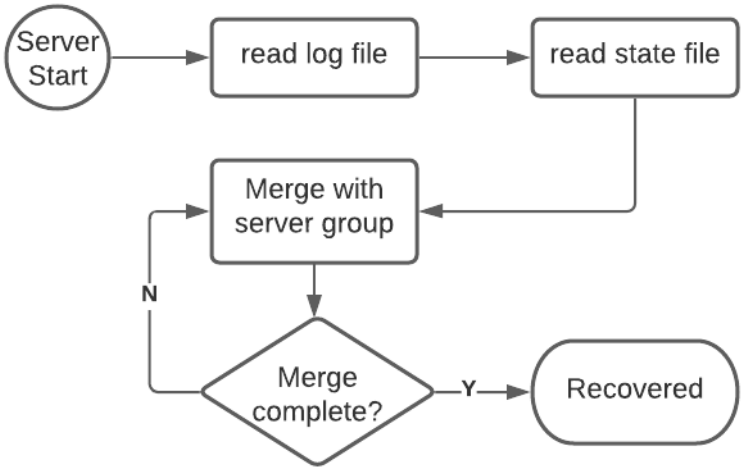
Replication



Graph 4 - Replication

We try to send by network the updates from the servers, not the states, and each server will update their state accordingly. As shown in graph 4, the update is initialized from the client program and taken by the server. The first server that gets this update will stamp it with its logical timestamp. Then the server will send the update to the connected server group. And when receiving an update message from the server group, it will store the update to the log file right away (see section *recovery from disk*). Then it will update the knowledge matrix accordingly. And it will update the state for future client requests.

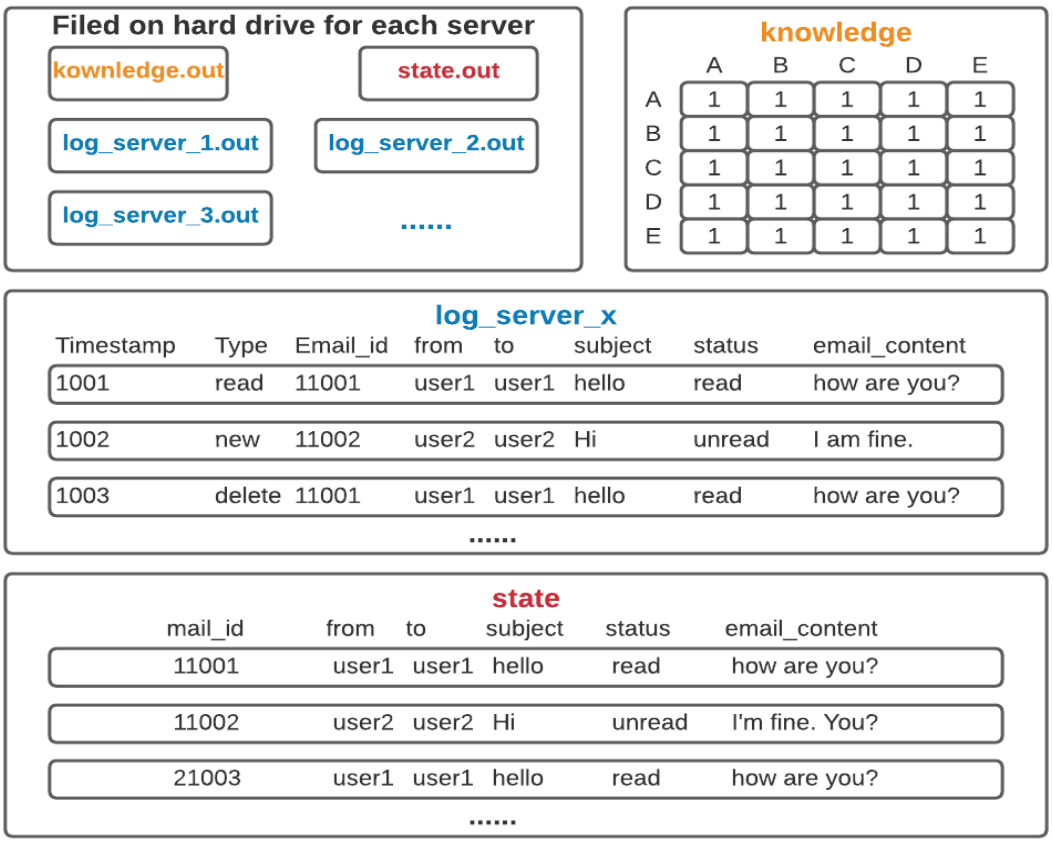
Recovery from disk



Graph 5 - Recovery from disk

When a server starts, it will first read from state and log file to see if a previous state to recover from. It will read the updates and the state into the memory and try to join

the public server group. The server program will be recovered when joined with the server group and done with reconciliation. And the process is the same as a reconciliation after partition. And all need files for each server are indicated below in graph 6.



Graph 6 - Local files on each server

Problems to handle

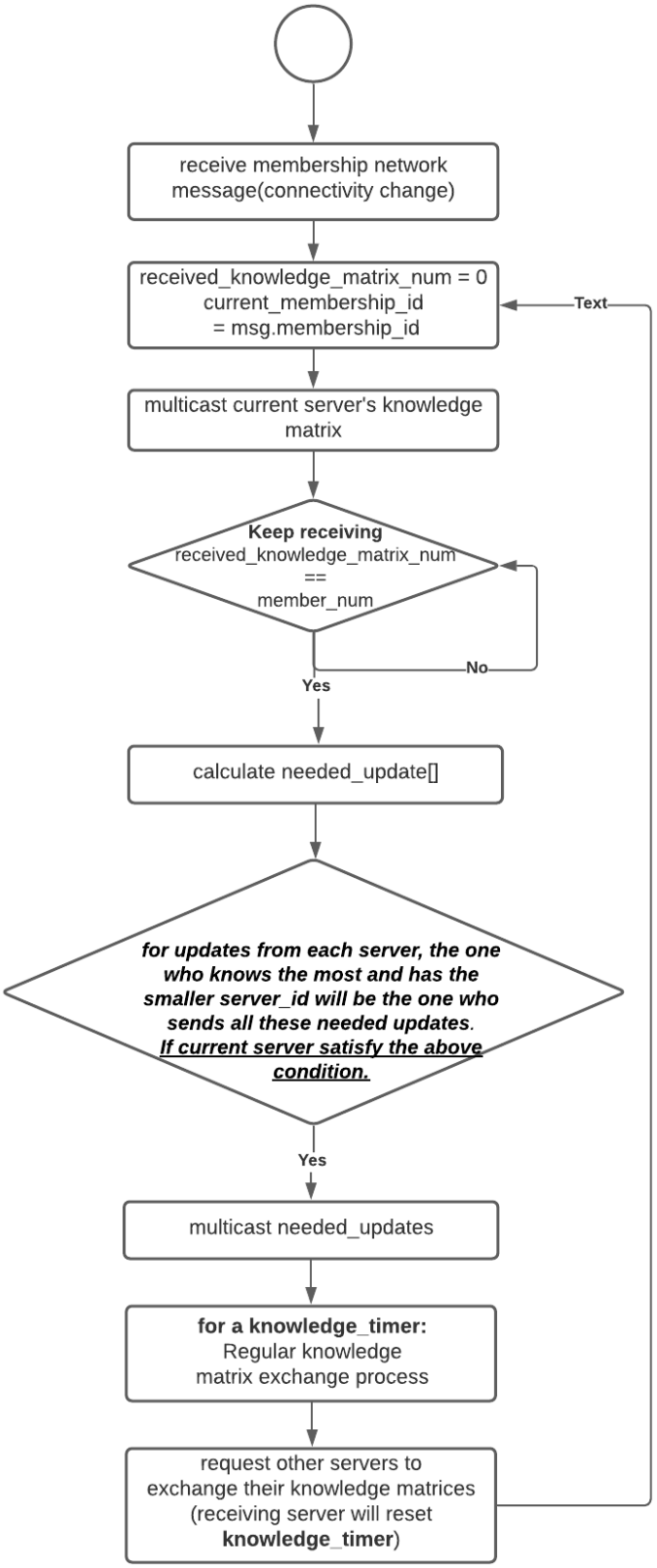
- **Partition**

  One group of servers may sometimes be split because of network partitions, i.e, one group of the server may be split into n groups, so they will not be able to communicate with each other, and when partition happens, we do nothing on the software side. But we are prepared for remedy when the partition is over, the partition is why we keep updates in the log.

- **Reconcile**
  - **When**

    We reconcile when there is a new member in the group. (connectivity change) The groups might have different updates when merging, so how should we merge all this data and not use redundant messages? We will show our design using an example.
  - **How to reconcile**

    In our design, we reconcile whenever there is a connectivity change indicating new members, and every server in the same group will be multicasting their knowledge matrix, and when each server has got all other servers' matrices, it is going to calculate the needed updates and send them by multicast. A more detailed process could be found in graph 7.



Graph 7 Flow chart - Reconcile

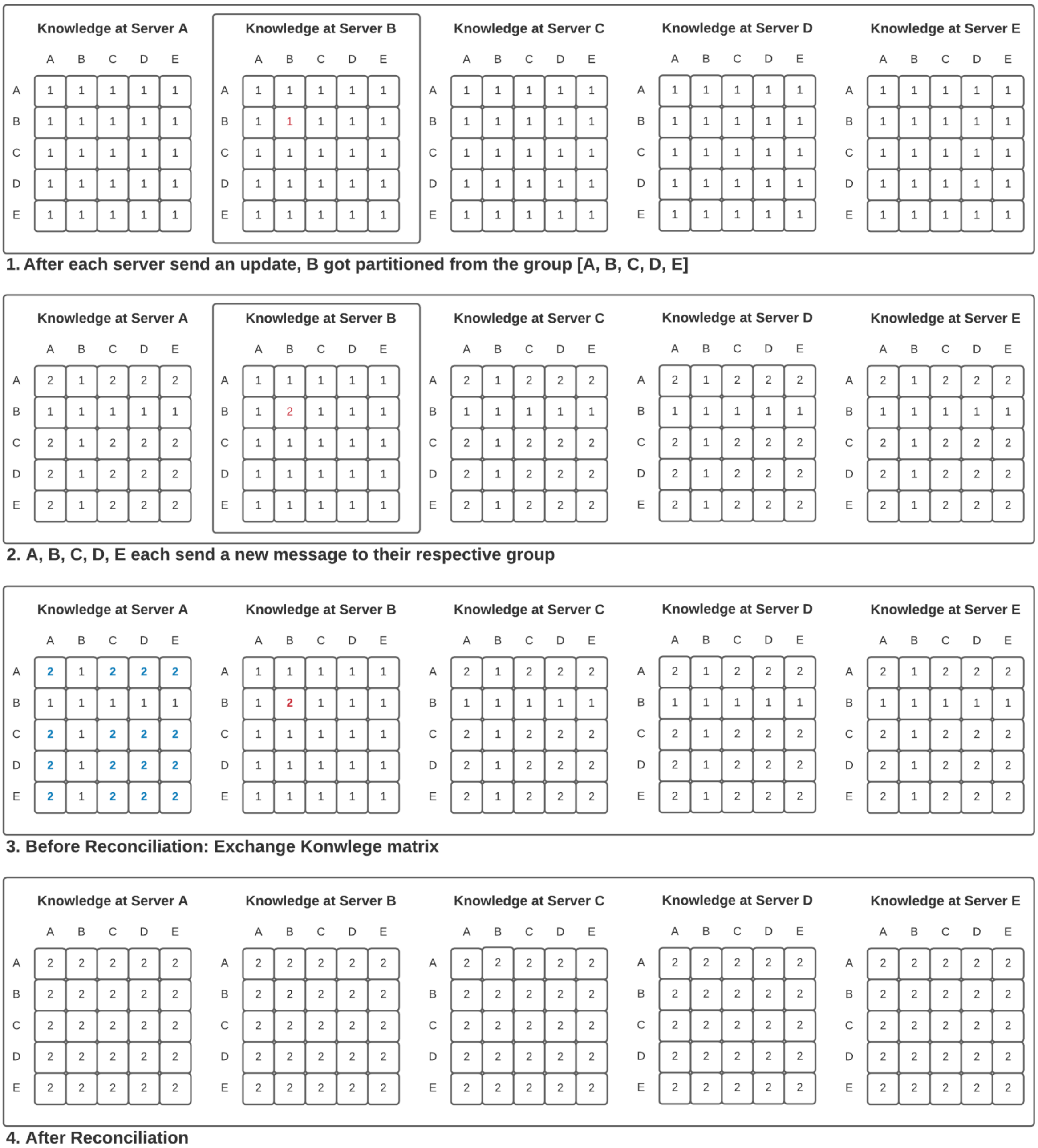
- **Preparation**

We are using a method similar to Lamport Timestamps Protocol for each server when it sends an update, each update contains two fields:

– timestamp

– server_id

When sending an update, the server will increment its timestamp. And the combination of {server_id, timestamp} servers as a composite primary key and the combination of {server_id, timestamp} will be unique for every update.



Graph 8 - Merging Reconciliation (4 Steps)

In this example, we can view the whole process in 4 steps, and reconciliation happens between step 3 and step 4. All steps are indicated in graph 8.

1. First, we have five servers in the group, and each one has sent an update to others, and because they all have the membership information, their knowledge matrixes will indicate that all other servers have got the others updates. So at step 1, they will all have the same knowledge matrixes, And the updates they are preserving for [A,1],[B,1],[C,1],[D,1],[E,1] can be deleted from the file. At the end of step 1, a partition happens, and B exits the group.

2. At step 2, each of the group members in [A, C, D, E] has sent a new update. But their membership info indicates that B is not in their group. So they will update their knowledge matrixes accordingly. If we take the minimum and the maximum of each column, we will see that there is one server that has not got the update between [col_min, col_max], but they will be preserved at least one server's log(updates). In this case, it indicates that updates [A,2], [B,2], [C,2], [D,2], [E,2] are needed for preservation.
3. Now B has rejoined the group. And in our design, we will do reconciliation when new members join. So they will multicast those updates and change the knowledge matrix accordingly. And we send needed updates by multicast in the hope that it is received and processed. And we will be checking for knowledge of all servers from time to time using a knowledge timer. Step 4 is the anticipated state of knowledge. And at step 4 is when [A,2], [B,2], [C,2], [D,2], [E,2] could be deleted.
4. Another question is who will send the needed updates? What we have here is a deterministic way, since every server in the group after knowledge exchange will know what others know, they could make a common decision: ***for updates from each server, the one who knows the most and has the smaller server_id will be the one who sends all these needed updates***. In our example, it will be: server A, B, A, A, A to send the updates from A, B, C, D, E respectively.

- **State, Log & Knowledge**

  Here we elaborate on the function of these classes. They in combination are for preparation to crash at any time, recover from disk and deal with the reconciliation.

  - **State**

    The state is used to keep the newest information of all emails and knowledge.

    - *Functions*
      - *State*()
        Read the state from the drive to RAM.
      - *get_email_list*(user_name)
        When the user requires the headers of all received emails, we use this function to get the list concerning one user.
      - *update_email*(type, email_id)
        When the user reads or deletes an email, we use this function to commit the change to the state.
      - *new_email*(email)
        When the user sends a new email to another user, add the email to the receiving mailbox of the receiver.
    - *Write to file*

      When update_email or new_email, the state instance will write to the file.

  - **Log**

    The log is used to keep the updates in case of possible reconciliations.

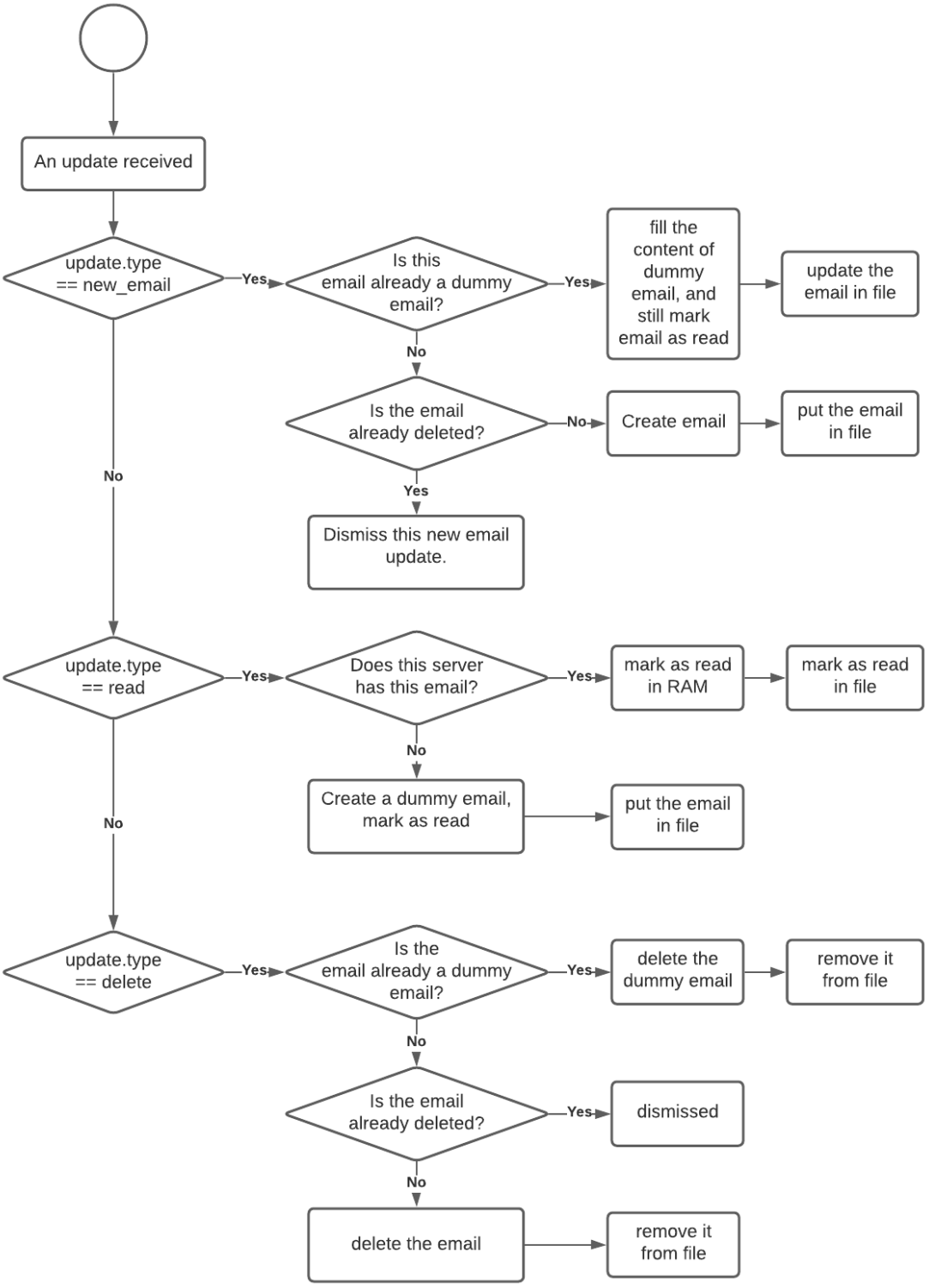
    - *Functions*
      - *Log*()
        Read the log from the drive to RAM.
      - *add_to_log*(update)
        Whenever an update is received, write to file and keep the update in RAM.

        - *delete_update*(server_id, timestamp)
          During reconciliation, we might know by the knowledge that an update is not needed anymore, so the log will have a function to delete it.
    - **Knowledge**
      A class where we keep the information up to which update other servers have. And it is stored in state.
        - *Functions*
            - *Knowledge*()
              Read from file and initialize a 2d array as a knowledge matrix.
            - *get_knowledge*()
              return the current server's knowledge matrix. Used by server program to send to other servers.
            - *update_knowledge*(server_id, Knowledge)
              Update on the current server's knowledge matrix using the received knowledge matrix from other servers.
            - *get_sending_updates*()
              Return a vector of update pointers, used to multicast to other servers.

- **The solution to operations on the same email during one reconcile process**
    - *Objective*
      No matter what situations happen, what we want is consistency in the end.

    - *Notation*
      [early update, late update] => Our solution

    - *Auxiliary data structures*
      set<update> deleted_emails; // mark update as delete, whenever updated, aux_file will update
      set<update> read_email; // mark update as delete, whenever updated, aux_file will update
        - **Why** do we need them?
          We keep this information because of possible future messages from a reconcile process, they could be used to avoid inconsistency during reconciling.
        - When to do garbage collection on them?
          When a server knows from its knowledge that he knows no less than the other servers. This is when we clean its reconciled data. Because it knows that currently no other server's update will be used to update its state, so no future updates are not depending on this information in this reconcile process. Therefore clear this information will not cause inconsistency simply for that the state will not change during this reconcile process.



Graph 9 - Dealing with edge cases

- **Normal Cases**
  [new, delete] => normal, deletion happens on the email that is already there.
  [new, read] => normal, read happens on the email that is already there.
  [read, delete] => normal, deletion happens after the read.

- *Dismissed Cases*

[read, read] => whenever receive read update, mark the email as read. If mark twice, it is ok for consistency.

[delete, delelte]=> Call delete_update_from_file, this function will only delete if the email exists. So it is still ok for consistency.

- *Not Possible Cases*
  [new, new] => not possible in our design. Mail_id is {update_init_server_id, timestamp}, which is unique for each email, it is only sent once during reconcile process. Even if a reconcile process happens inside a reconcile process, whoever has this update will mark the knowledge matrix as already has it so next time it will dismiss it since the knowledge says this update is not needed. (This is guaranteed if only the knowledge matrix is consistent with what is actually carried out on the server's state, there are still concerns on the atomic level of operations).

- *Specially Dealt Cases (Graph 9)*
  [delete, new] => Whenever deletion happens on an email that does not exist, add it to deleted_emails, when new comes, dismiss the new operation. Clean delete_emails at the end of the reconcile process so it would not grow indefinitely.

  [read, new] => When read on a nonexistent email, create a dummy email, add it to read_email to keep track, new email rewrite on the dummy email, but keep the read_state as read. new email operation will also remove this email from read_email. read_email is cleaned when reconcile process is done.

  [delete, read] => When this read happens and the email is not here, the server does not know if it is deleted or the new operation is late. So it checks the previous information in delete_emails, if this email is in deleted_emails, dismiss this read(possibly [delete, read, new]). If not in deleted_emails, we will create the dummy email and keep it in the read_email like in the above solution. If a new operation comes later, it will write the dummy and remove the email from read_emails. If a new operation never comes when reconcile process is over(a reconcile process is over for a server when its update_timestamp from each server is the maximum in its knowledge, i.e, it knows as much as it thinks it could know), we delete every dummy email on the read_email and clear read_email.

Some Questions

- How to accomplish replication between servers?
  In typical situations, we use reliable multicast messages to inform all servers in the group of the new updates. And each server will preserve the updates until they are not needed anymore(elaborated below). When new members join the group, they will reconcile(elaborated below) to get all needed updates delivered. We could achieve a consistent view by a consistent updates application on the state.

- How to do reconciliation when merging groups?
  Elaborated on the *problems to handle*.

- How to be ready to crash at any time?

  When a server receives an update from a client, it will store that update to the log file before sending it to the other mail servers. The reason is if the mail server sends the update to other mail servers and then crashes, it will not have these updates when recovering, and what if all other servers are partitioned, then the client will not see a consistent view of this mail update for it is lost forever. We need at least one copy of the update.

- When to delete the updates?

  Since there is a memory limit, we cannot keep the updates in RAM forever, and also, some old updates are not needed, so we can remove them. The question is when to delete these updates. Our design is to have a *regular checking on the knowledge of all servers when they are all connected for a certain amount of time*. When we know that all other servers have the updates to a certain point of {server_id, timestamp}, we know we can discard all the updates before that(with the same server_id and smaller timestamp).

- Is there an order preference when merging updates on the same email?

  During reconciliation, for all the updates to one email, they usually are ordered by the timestamp, that is, which update comes earlier is done earlier. But this order is only guaranteed for updates initialized by the same server. In our design, there will be other preferences for which updates to be applied. For example, if server A has a deletion to mail 1, and server B has a read operation to mail 1. The only update that needed to be delivered was the deletion of mail 1. In this case, we can say that the read update from server B is not important anymore when there is a deletion. So, if they are all read updates, we could just apply one of the read operations, because it does not affect if they are both read operations on the same email.; If any deletion operation, no more updates will be applied.

- Why a matrix for knowledge?

  This is used to store the second level of knowledge, i.e. a server could tell a lot of information from this matrix and use it for decision makings:

  1. What other server does not know?
     For example, Maybe server A has communicated with server B, and server B has communicated with server C, but A and C never communicate, it could happen for a partition that goes from [[A, B], [C]] to [[A,], [B, C]]. When merging together as [[A, B],[C]], A will know what C knows, so it could discard some of the updates that it is holding for C. This is just an example over the top of our minds.
  2. When deciding who is the one to send the needed updates?
     When reconciling, we do not want every server that has a needed update to send it. That would be too saturated rescue, since we know what other servers know, we could make a common decision here for who should send it, for example in our design, for updates from each server, the one who knows the most and has the smaller server_id will be the one who sends all these needed updates initialized from that server.
  3. What I do not know?

And of course, from this matrix, a server could know what updates it does not have. And that is also needed for some of our designed operations.

----------END OF DESIGN----------

TODOs

- **Implementation**
  - **Keeping Updates**
    - Time-based garbage collection

lucid chart links(*delete when done*): client-server commu replication Data Struct Recovery disk