

UMEÅ UNIVERSITY
Institution for computer science
Report of obligatori project

Distributed Systems(HT16)

5DV147

Project

Name	Timmy Olsson (c12ton) Simon Lundmark (c13slk)
E-Mail	c12ton@cs.umu.se c13slk@cs.umu.se
Date	23/10-16
Teacher	
execution	NameService: project/java -jar RMIServer.jar Chat-client: project/java -jar Client.jar Debug-client: project/java -jar Debug.jar

Introduction:

This project was about implementing a distributed group communication, that is tested as a chat application. The minimum requirements was to implement it as a non-reliable multicast, and sort messages with causal order algorithm.

System description:

GCOM:

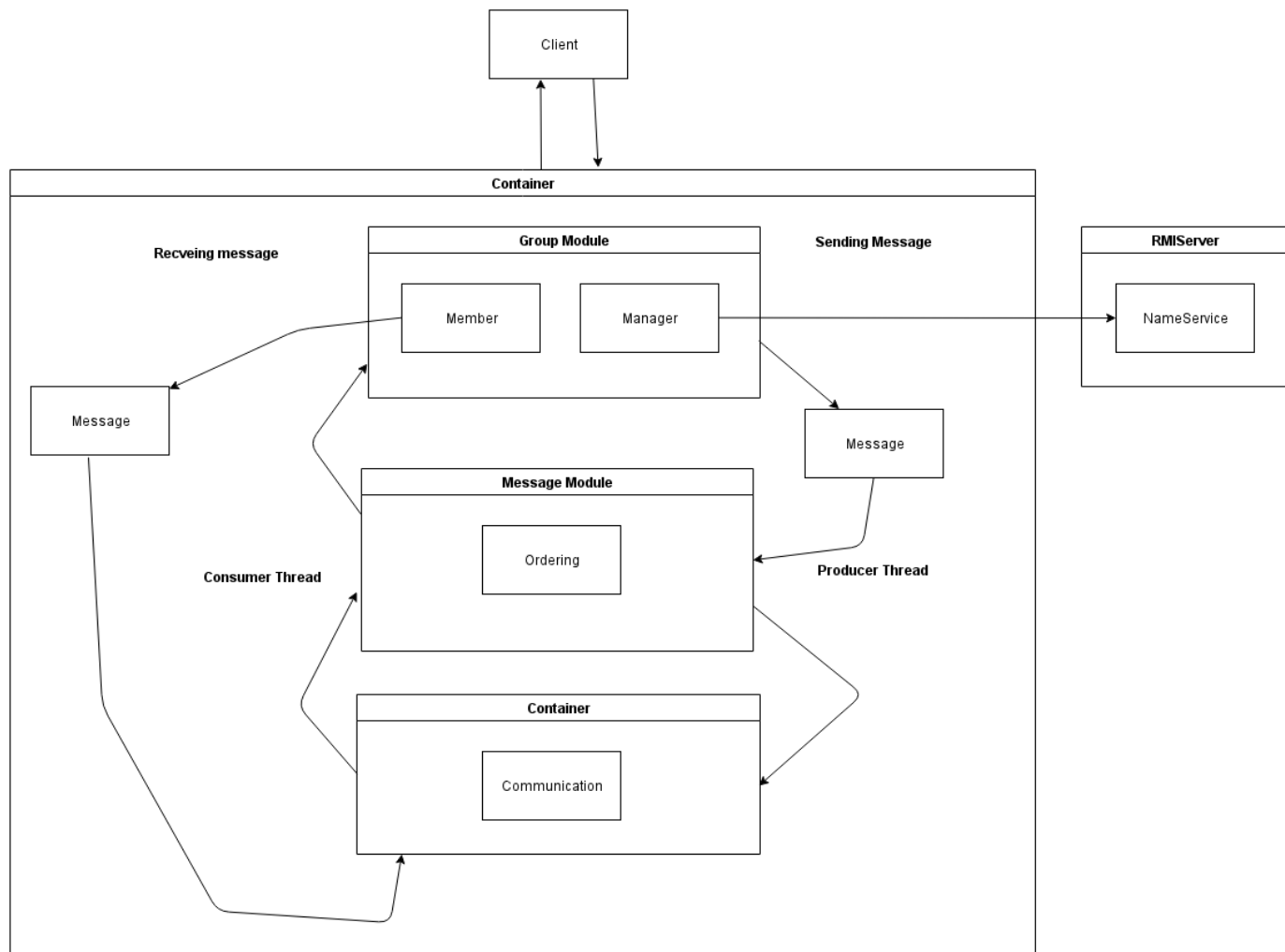


Figure 1: GCOM architect and the flow of messages

The GCOM class combines the modules in a certain order so that they don't need to know of each other. The Client composes the message to be sent to the group, it will be stored in an outgoing message queue in the GCOM class. A producer thread will fetch one message at a time from the outgoing queue, and then collect all working members from the group module. And if any member is detected not to be working it will be removed from the member list and a message specifying the event will be added to the outgoing queue.

Every message that is being sent will be timestamped, with a vector clock. This is done in the message module. Then later it will be sent to the communication module together with a list of members that should intercept the message.

Received messages will be stored in an ingoing message queue in the communication module, a consumer thread will be notified of this and fetch the message. Then go through the message module to order the message. And if message is related to the group relation it will be reflected in the group manager. And then finally it will be sent to the client.

Group Module

As described in the project description, the group module only handles the group relation and keeps a list of references to the other members. Group module is represented by the class GroupManager and GroupMember. GroupManager maintains the list of members and GroupMember act as reference, for other members to gain access to that member properties and be able to send messages to that particular member.

Active members

When trying to retrieve members, the GroupManager will check each of its current members that they are still active, this is done by trying to access a members name. And if an exception occurs it's considered dead and message of notification is sent to the group. And then removed from the list.

Election

If a leader crashes during the session then the next member to send a message will discover that the connection with the leader is dead and tell the nameservice directly that it is the new leader. In the case that two members discover this at almost the same time then they check with the name service again if the leader is dead, if it still is then it tells the nameservice that it is the new leader, then when the second member checks, then the leader is alive so it does nothing.

Join requests

GroupManager will handle join request from outside clients, through the leader instance. It will check that no other member has the same name. If it did an exception will be thrown to notify the requesting client. If successful, the joining member will receive a list of all member instances in the group, including it's own. Then the leader will send a "join" message to the group, that contains the instance of the new member.

Connect to group

GroupManager retrieves the groups available in the NameService. And connects to a specified group. If successful it returns all current members names else an exception will be thrown.

The leader will give the joining member all the current members in group.

Create group

GroupManager can create groups if given the properties for the group.

These properties should determine the type of communication, message ordering the group should have and the name of the group.

Observers

In order to intercept messages being sent through the member or notifications messages from the GroupManager, it requires observers to receive messages.

Message ordering module:

We have two types of ordering, unordered and causal ordering. Each type of order has it's own class that implements a common interface. That implements a message order method and a method for "time stamping" outgoing messages. The unordered implementation won't do anything with the messages, just pass them as they are.

Causal algorithm

Because the algorithm as presented in the course material can't handle received messages from it self. We had to adjust it. The algorithm as presented in the course:

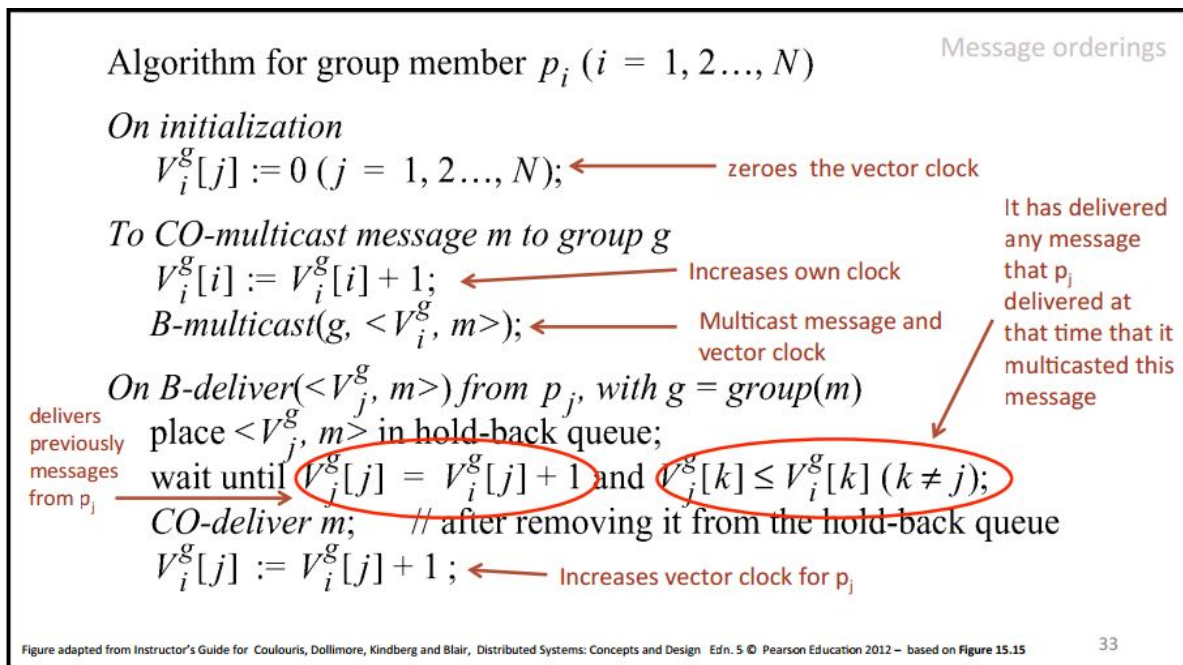


Fig 2: Algorithm of message ordering

The problem occurs when comparing the clock of it's own process with the condition that the received message is the next message sent from that process.

This is solved by keeping track of how many messages have one received from it self that is a response from the message earlier. And use the same condition as in the algorithm. That the next message should be a response to the last message received. If not, then add it to the queue.

And we didn't want the thread to wait till there's a match, so in our solution we looped through the queue and did the comparison, as many times as the size of the queue.

Since a new member can join at any time we need to have a dynamic vector clock that becomes bigger when someone joins and smaller when one leaves. Since we are sending hash tables as vector clocks there is no problem when comparing them.

Communication module

The communication module consists of the non-reliable communication that just sends the messages to all members that are connected, it does not care if messages arrives, it just sends the message. When the messages is sent to each member, the message is cloned because of reference errors that can occur. This was a big problem a number of times when implementing the system, the references was changed between the members and errors occurred, so the clone was the answer.

RMI Server:

The RMI server creates a registry instance at an available port. And then register an instance of NameService and DebugService .

Debug service holds messages from each member in selected group. And will be released by the debug client.

Name service:

Name service contains the hashmap of leaders for each group. Where groupname is the key. Name service also manages deletion and registration of groups.

Message:

We use multiple types of messages to communicate in the group. These messages implements the same interface for a basic message.

The basic implementation of a message, is the name of the sender, receiver and the vectorclock.

The GCOM makes no distinction between the messages during the sending process. It only checks if received message is related to the relation of the group. And all messages is sent to the observing client.

This make it possible for other uses than just chat messages, for this middleware.

GCOMDebug:

GCOMDebug extends the GCOM and overrides methods that creates the communication and message ordering. And instead use those methods to create instances that extends those modules but they communicate directly to the DebugService instance on the RMI server.

GCOM

--GCOM--

- GCOM combine all modules
- Modules are loosely coupled to each other
- Through member messages are sent. Observed by the communication module
- Consumer thread awaits message from communication module
- Go through all the modules, then back to gui
- Producer thread awaits message from gui
- Send it down the modules
- General interfaces for each of modules, so if change is needed it won't affect the other components.

--RMI Server

- RMI Server Creates instances of NameService and DebugService
- NameService

System limitations:

- Election, node or nodes goes directly to nameservice to decide new leader.
- Unreliable communication. Can't handle dropped packets.
- Clone every message
- Only detects dead members when collecting all members to send to, not when actually multicasting
- Can't debug clients that are both in debug mode and in non-debug mode. Only debug mode
- Must have the debug-client running if some clients with debug mode on is alive.

User manual:

There are three separate files that can be executed in our project, these are:

Name service: must run before the client can connect to something.

Debug-client: This client will hold messages of a selected group and then can select in what order the messages will be delivered and see the current vector clocks if the group has causal ordering.

Chat-client: The chat client has two modes, one just for chatting just with other clients and one for debugging the selected group. If no group is selected in the debug gui then the chat will continue as usual.

Name service:

The only requirement is that the nameservice is running for the chat-clients to work.

Run Command: `project/java -jar RMIServer.jar`

Chat-client:

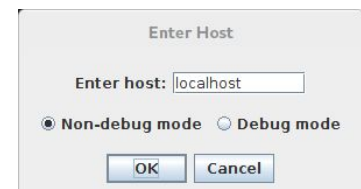
Run command: `project/java -jar Client.jar`

The window to the left is the first thing that is shown. The host location that is needed to connect to the name service is default on localhost for convenience. After the host location is specified that you can choose if you want to use the chat-client for communication or debugging the chat. Note that the debug-client must be started before creating the group in debug mode.

If the connection is not made then the previous window will be shown again. Else the window to the left will be shown, before anything can be done, specify the username that you want to join a group with or create a group with. When this is done then the button create group will be accessible to click.

When creating a group the following window will be displayed. Our implementation only support the basic non-reliable communication so the other is not clickable.

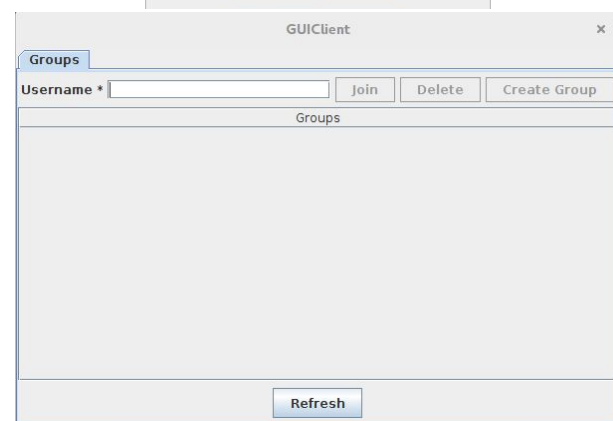
Then there are a choice in how the messages will be ordered to the client. Then it is necessary to choose a groupname, if it is not specified then a warning popup will tell you to specify one.



Enter Host

Enter host:

☒ Non-debug mode ☐ Debug mode

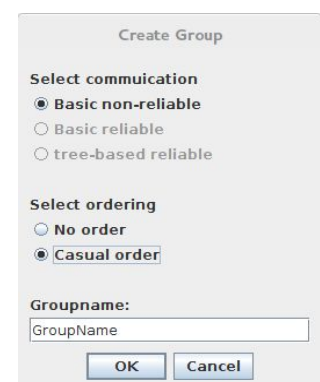


GUIClient

Groups

Username *

Groups



Create Group

Select communication

☒ Basic non-reliable
☐ Basic reliable
☐ tree-based reliable

Select ordering

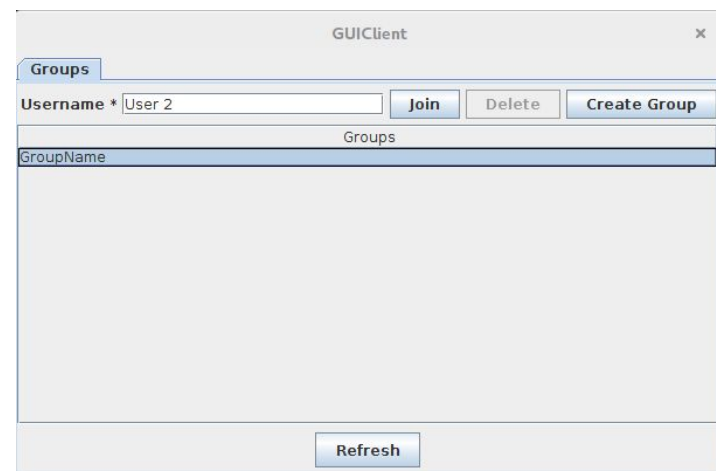
☐ No order
☒ Casual order

Groupname:

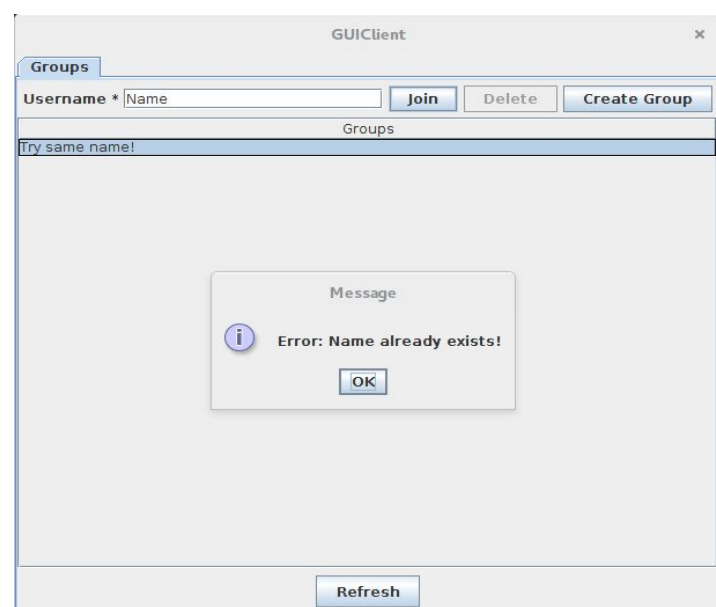
When the group is made a new tab will be shown and be in focus, then others can join the chat. The biggest area is for the messages to be displayed, the right area for the usernames and the bottom for typing the message and of course the send button to send the message (pressing enter will also send the message).



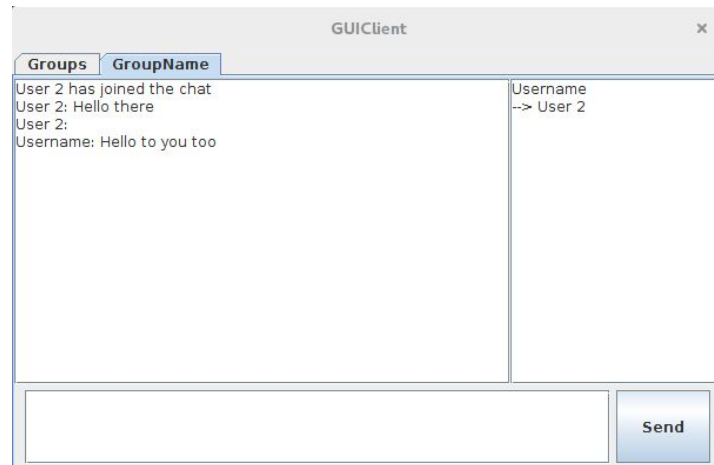
Now if a new client is started in non-debug mode and the same host is specified then there should show the newly created group. To join the group, select it and press join after specifying the user name.



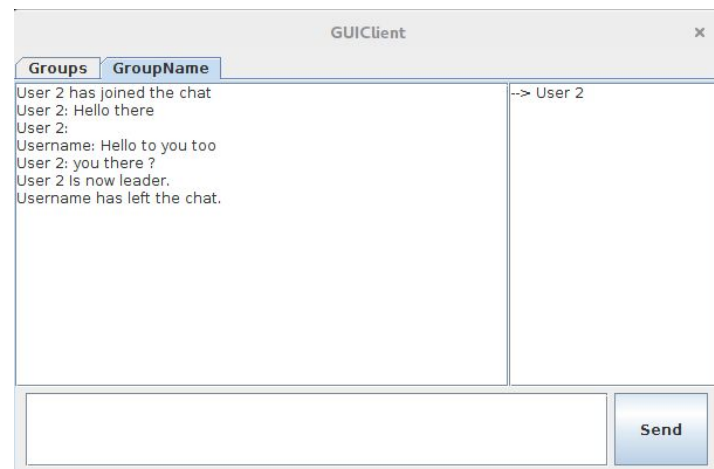
If you try to join the group with the same username as another user, then there will be a error message displayed and the you can change the username and try again.



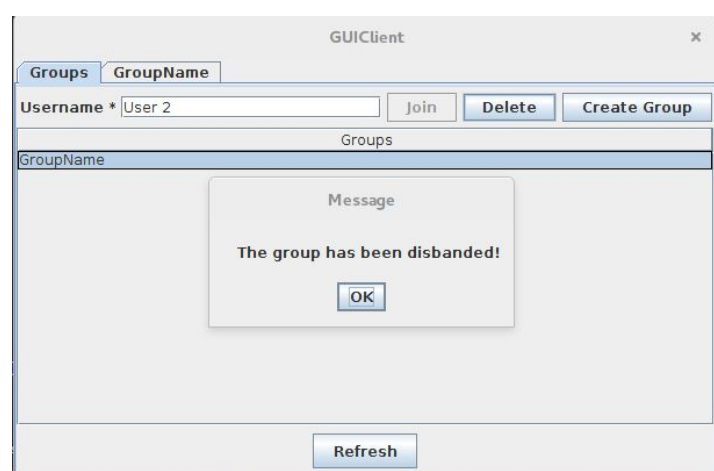
If the new user connected successfully then there will be a join message shown that the user has joined and the username will be displayed to the left. Then the communication can begin.



If a user closes the client then it will not be displayed at first but when another client writes a message then then it will see that the message to the dropped user will not be able to be sent. Then that user sends a message to all other users telling them that the user has disconnected. If the disconnected member was the leader then the member that discovered the disconnect will be the leader and that will be notified to all other members too.



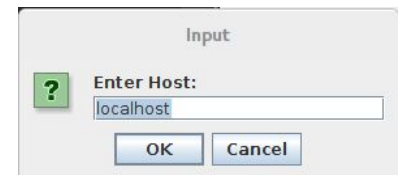
If you are the leader of a certain group then you can also delete the group. When you select the group and press delete then a message will be sent to the other members telling them that the group has been disbanded. A error message will be shown then the tab will close.



Debug-client:

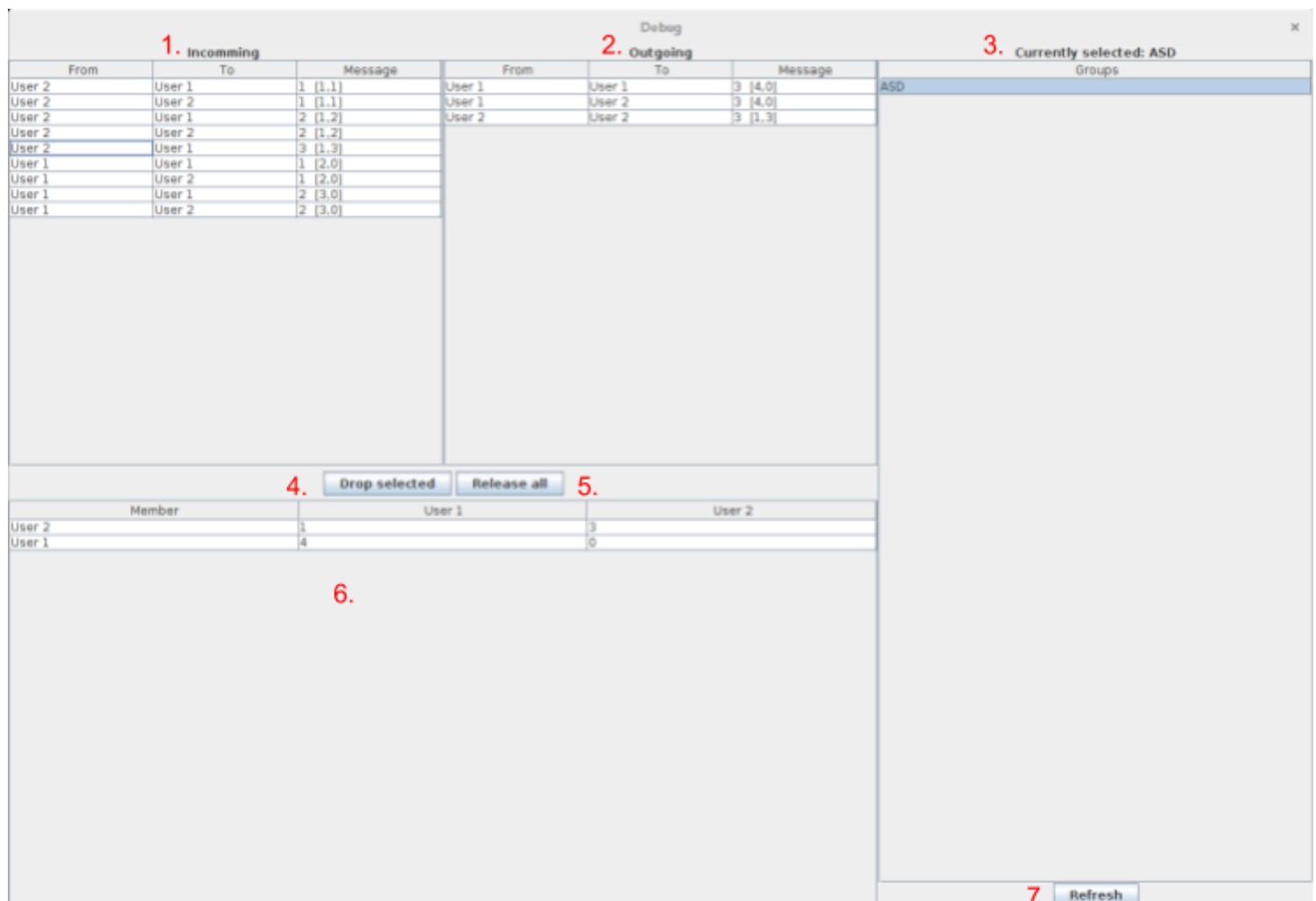
Run command: `project/java -jar Debug.jar`

When the debug-client is started the following window will be displayed, here you must specify the host location to continue, if it cannot connect then the window will be shown again.



A small dialog box titled "Input" with a green question mark icon. It contains a text field labeled "Enter Host:" with "localhost" entered. Below the text field are two buttons: "OK" and "Cancel".

If the connection was successful then the main window will be shown:



The main window is titled "Debug" and contains several sections:

- 1. Incoming:** A table showing incoming messages.
- 2. Outgoing:** A table showing outgoing messages.
- 3. Currently selected: ASD:** A section for the currently selected group, showing "ASD" under "Groups".
- 4. Drop selected:** A button to drop the selected message.
- 5. Release all:** A button to release all messages.
- 6. Member:** A table showing members of the selected group.
- 7. Refresh:** A button to refresh the data.

From	To	Message
User 2	User 1	1 [1,1]
User 2	User 2	1 [1,1]
User 2	User 1	2 [1,2]
User 2	User 2	2 [1,2]
User 2	User 1	3 [1,3]
User 1	User 1	1 [2,0]
User 1	User 2	1 [2,0]
User 1	User 1	2 [3,0]
User 1	User 2	2 [3,0]

From	To	Message
User 1	User 1	3 [4,0]
User 1	User 2	3 [4,0]
User 2	User 2	3 [1,3]

Member	User 1	User 2
User 2	1	3
User 1	4	0

The following window will be explained below.

1. The incoming window where all the messages is stored before sending them to the client in the order you want. It displays where the message is from, where it is going and the message with its vector clock behind it.
If you want to send a message from the incoming window then just double click on the row that you want to send and it will be sent.
2. The outgoing window is the holding queue if causal order is selected for the group, if a message is sent in the wrong order then the message will be delayed until the previous messages has been sent, else if the messages is sent in the right order then the outgoing queue will be empty.
3. This is the window where all the groups is displayed, to hold the messages from a group, double click the group and the group name will be displayed in the label over the area. If no window is selected then the groups will work and sent as usual.
4. If a row is selected and you press this button then the message will be dropped, because causal order does not handle dropped packages then the holding queue will just continue to grow for the user that the package was dropped. If it is no order then the messages will just flow on as if nothing has happened.
5. Release all button will release all the messages from the incoming queue.
6. This window will display the current vector clock for all users, it will be shown when the first message is sent to the debug-client and updated whenever the clocks is changed.
7. Refresh the groups to be accessed.

Discussion:

- *Understanding of different communication types*, their characteristics and scenario suitability.
 - *Understanding of causal message ordering algorithm*, its characteristics and effects.
 - *Understanding of basic fault tolerant techniques*, with focus on maintaining robustness and availability of a system under network partitions and unexpected events.
 - *Understanding of system architecture design impact*, with special focus on differences between centralized and distributed solutions.
 - *Development of software engineering skills*. Practice the ability to convert a loosely specified problem into a concrete design and implementation.
 - *Development of written communication skills*.
-
1. We only implemented the non-reliable multicasting but reliable was discussed many times, how we could implement it and how much better it would be, but the time and some problems was a factor why we didn't implement it.
 2. We understands how the causal ordering algorithm works since we needed to modify it to work with messages that were sent to our selfs, that needed understanding and thought to make it work. We even did our own implementation of the algorithm to understand how it worked and then used one of them.
 3. Since our program handles when members and the leader crashes and still have the same availability, this is achieved.
 4. Since we have already done a chat-client/ server solution in the data communications course and now in this course we can see how much difference it is between centralized and distributed solutions.
 5. This was indeed the situation, after some clarifications the design and implementation was a success.

Analysis

This project was not handled as a project but as an assignment.

By reading the project description, we made the assumption that the leader was the one responsible to inform members if a new member joined. This could have been done differently, the member that joined could have informed the rest of the group.

When implementing the causal ordering we thought that the algorithm from the course slides should work directly with our program, but the algorithm did not handle messages that were sent to the member that sent the message, so this was something that needed some thought. An alternative solution to our problem was that the member should not send messages to itself.

When designing the program we thought that the program should handle multiple groups at a time, when reading the specification again we discovered that it didn't have to have this feature.

So an alternative solution would have been that you could either join or create a group for each client.

When reading the specification it said that the debugger should be logically separated and we interpreted it so the client and the debugger should be separate programs, which was not the case. Our interpretation of how the debugger should work was that the messages should pass it to reach members. When the debugger has selected the group, all the messages for that group are on hold and the debugger chooses what message to be sent, this is a more advanced way because of thread synchronization between clients and the debugger, when the debugger then sends the message to the member and it comes in the holding queue then the debugger shows all messages that are in the queue for all clients.

An alternative solution to this is that each client has its own debugger and then the debugger only controls that client instead of all messages that are sent and received, so the debugger is a part of the client.

Test Protocol for non-reliable communication minimum requirements:

Client non debug:

1. Start name service
2. Start client, try to connect to wrong location
3. Connect to the correct location
4. Enter user name and try to create a new group
5. Try to create group without specifying the group name.
6. Specify group name and create the group.
7. Create a group with same name.
8. Create group with new name.
9. Try to write in the group and see that the message arrives.
10. Start another client and connect to the host.
11. Try to join with the same name.
12. Join with different name.
13. Both clients write something.
14. The leader disconnects.
15. The client try to write to the other client.(is now leader)
16. Another client joins and writes new messages.
17. The leader of the group deletes the group.

Client debug:

1. Start name service
2. Start the debug-client
3. Enter the incorrect host
4. Enter the correct host
5. Start new chat client
6. Create a group (with causal ordering)
7. Debug-client doubleclicks the group to select it so the messages is delayed
8. Client tries to send messages
9. Debug client sends one message
10. Start new client and join the group
11. Debug sends the joins to all
12. Each chat client types 1, 2, 3 separated by enter
13. The debug sends the messages in the wrong order
14. Another client joins but join message is only sent to the 2 first
15. Try to sent messages from the two first, should not come to the third, send join to third.
16. All chats types 1, 2, 3 separated by enter
17. Messages is sent in the wrong order
18. One client leaves
19. Clients sends new messages
20. A new client joins the group with the same name as the one that left.
21. The new client leaves again.
22. Disband the group.