UMEÅ UNIVERSITY
Department of Computing Science
Report

Distributed Systems
7.5 hp, 5DV186, VT17

Project GCOM

Name	John-John Markstedt, Jonathan Westin
CAS	joma0245, jowe0017@umu.se
CS	c14jmt, jwestin@cs.umu.se
Date	2018-06-09
Source	https://github.com/Johnstedt/GCom

Compile and run instructions:

```
ant
java -jar client.jar
```

Teachers and tutors
Jonny Pettersson (jonny@cs.umu.se)
P-O Östberg (p-o@cs.umu.se)
Jakob Lindqvist (jakobl@cs.umu.se)

CONTENTS

I. Overview

In this report we will start with giving an introduction of theory behind the GCom components in section II, III and IV. The report continues with showing an Java Implementation of the components in Section V. The report will end with an discussion and reflection in Section VI.

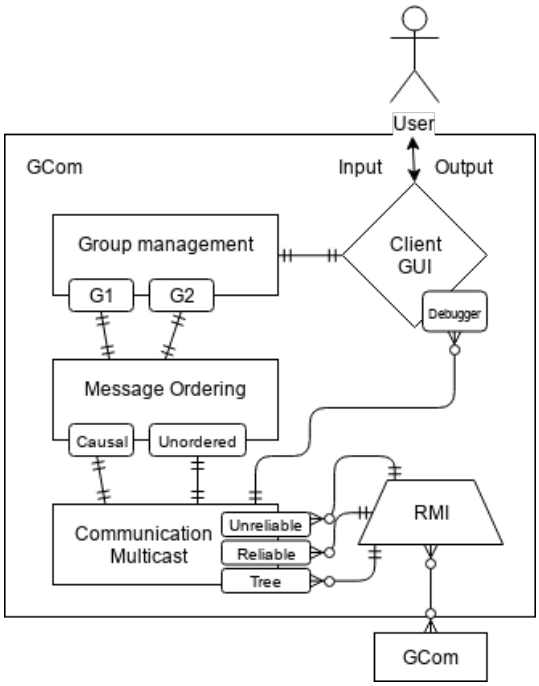

Figure 1: *View over GCom's components.*

As seen in Figure 1, the GCom components have three distinct layers: Group management, message ordering, and communication. To give a better interpretation the Figure also components towards the GUI and usage of *Java RMI*[1]. The communication between the adjecent layers is directly connected and works both ways. This will, as seen in Section V, create some coding obstacles.

[1] https://docs.oracle.com/javase/tutorial/rmi/

II. Group Management

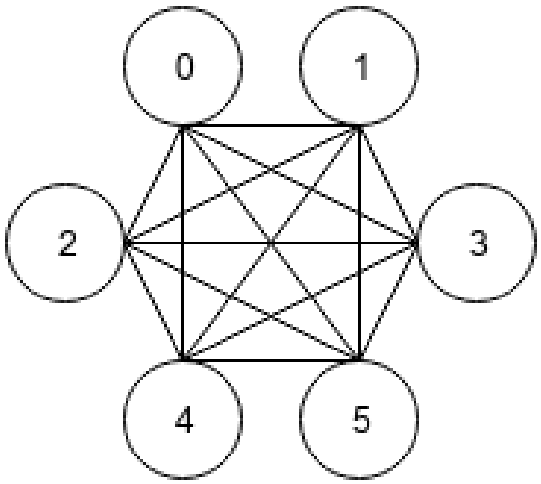

Figure 2: *Showing the mapping between client id's and it's position in the sender specific tree.*

The group manager module is the outermost layer of the model component. It is responsible for creating groups, keeping track of groups, all group messages and, if a nameserver, sending groups to other asking users.

It should be noted that every member of the group establishes a connection to every other member as seen in figure 2

A. initiation

Messages that are sent to other GCom clients to retrieve knowledge about other groups. An invisible group is used to let users establish a connection and making initiation possible by sending groups.

1) ASK GROUPS: Prompts another GCom client to send its groups.

2) SEND GROUPS: Upon receiving an 'ASK GROUP' and its state isn't set as a 'nameserver' it will send it's list of groups to the user who asked.

B. in group

1) JOIN: When a received list of groups and the user choose to join a group, a 'JOIN' is sent to the group notifying each member of the new member.

2) MESSAGE: A text message

3) LEAVE: If a user decides to leave the group a 'LEAVE' is sent to the other members to notify and remove the user.

III. Message Ordering

In the message ordering layer two different types of orderings are implemented, namely **Unordered** and **Causal**. The order is selected at the group creation and is group specific.

A. Unordered

Unordered does nothing with the messages and delivers the messages when it retrieves them.

B. Causal

Causal ordering takes the happen before relation into. If a message doesn't fulfill the happen before relation it will be held until it does. Following is the used algorithm to realize causal ordering[1].

$$for\ every\ process\ p_i \in group\ g$$
$$init:$$
$$v_i^g[j] := 0\ \ for\ \ (j \in p)$$
$$send:$$
$$v_i^g[j] := v_i^g[j] + 1\ \ for\ \ (j \in p)$$
$$multicast(g, < V_i^g, m >$$
$$receive :< v_j^g, m >\ \ from\ p_j (j \neq i)\ with\ group\ g)$$
$$hold\ \ < v_j^g, m >$$
$$until\ v_j^g[j] = v_i^g[j] + 1\ and\ v_j^g[k] \leq v_i^g[k]\ \ (k \neq j)$$

$$v_i^g[j] := v_i^g[j] + 1$$

IV. Multicast Communication Layer

Three different types of multicasts are implemented and can be chosen at the creation of a group. The multicast type is group specific and not user specific.

A. Unreliable Multicast

The unreliable multicast sends a single message to each of the group members.

B. Reliable Multicast

The reliable multicast is implemented according to the algorithm for reliable multicast given in Distributed Systems[1]. The algorithm basically checks if it received the message already and if it hasn't it forwards the message to the known members of the group. This is reliable in the sense that even if members don't know each other they will receive the message. Since each member in the group has established a connection to each other member(see figure 2) there is really no point of flooding the group with messages in this system, however, if the group manager is exchanged it might become useful. The Unreliable multicast is just as reliable with less overhead than the reliable in this system with this group dynamic.

C. Tree Based Multicast

The tree-based multicast works by sending the list of receivers with the message and which user who sent it.

Dependent on initiating sender a binary tree can be deterministically drawn at each node with the sender as the root. This implementation uses a **source based tree**, where *each* message source constructs a unique tree [2]. This is different from a group-shared tree.
In 3 the mapping of a tree with node id three as sender/root can be seen. Equations to deterministically map a node's position in the tree, determine which node to send to and determine if to send follow in figures, 4, 5, 6.

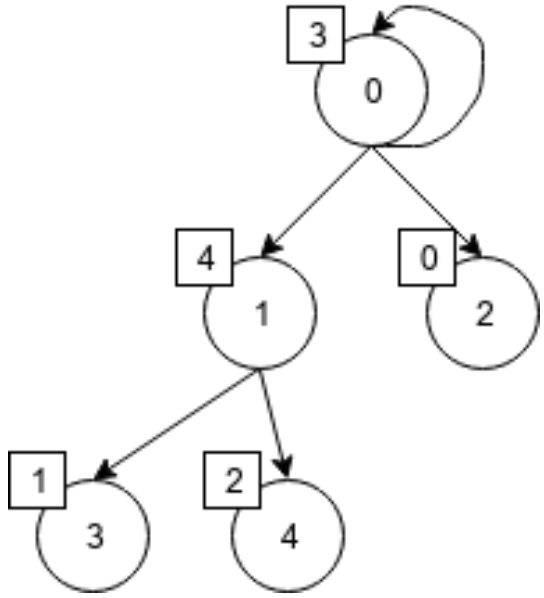

Figure 3: *Showing the mapping between client id's and it's position in the sender specific tree.*

c : client id
s : sender id
l : length

c, s, l are all known in multicast level.

$$p = (c - s) \bmod l$$

Figure 4: *Retriving position in tree.*

$$c_1 = ((c - s) \bmod l) * 2 + 1 + s) \bmod l$$
$$c_2 = ((c - s) \bmod l) * 2 + 2 + s) \bmod l$$

Figure 5: *Retriving client id which to send to.*

$$c \quad should \quad send \quad to \quad c_x(c_1, c_2) \quad iff$$
$$(c_x - s) \bmod l > p$$
$$and$$
$$log_2(l)^2 <= p$$

Figure 6: *Two equations determining if client have 0,1 or 2 leafs to send to.*

V. Implementation

As seen in Figure 1, the GCom modules offers a good implementation structure as Figure 7 follows. In the implementation, we can follow that an observer-observable design pattern has been used. This decision is made due to the coupling generated having knowledge of both classes used to receive and send messages from different layers. The implementation will have the structure of using the method *void send(Message)* to send a message outbound from the client to other clients while the observer-observable chain is used for an incoming message (from other clients).

The abstract class *Multicast* in the *communication*-package/layer has three implementations: *Unreliable-Mutlicast*, *ReliableMultiCast* and *TreeMulticast*. To implement the classes three methods needs to be implemented: *send*, *sendToSender* and *receiveFromReceiver*. The method is added as an extra layer to simplify the underlying abstract implementations to diminish any confusing factors. The Multicast needs to be able to receive message both from the client and other clients and be able to send to other messages. This differs from other parts of the GCom whereas they only need to receive or send messages.

Next layer, consisting of the *messageordering*-package/layer, will get message from a group (through the method send) and receive messages from the underlaying layer of communication-package. The abstract class has the methods of *send(Message)* and *queueAdd(Message)* to implement. The class is extended by *Unordered*, *Fifo* and *Causal*.

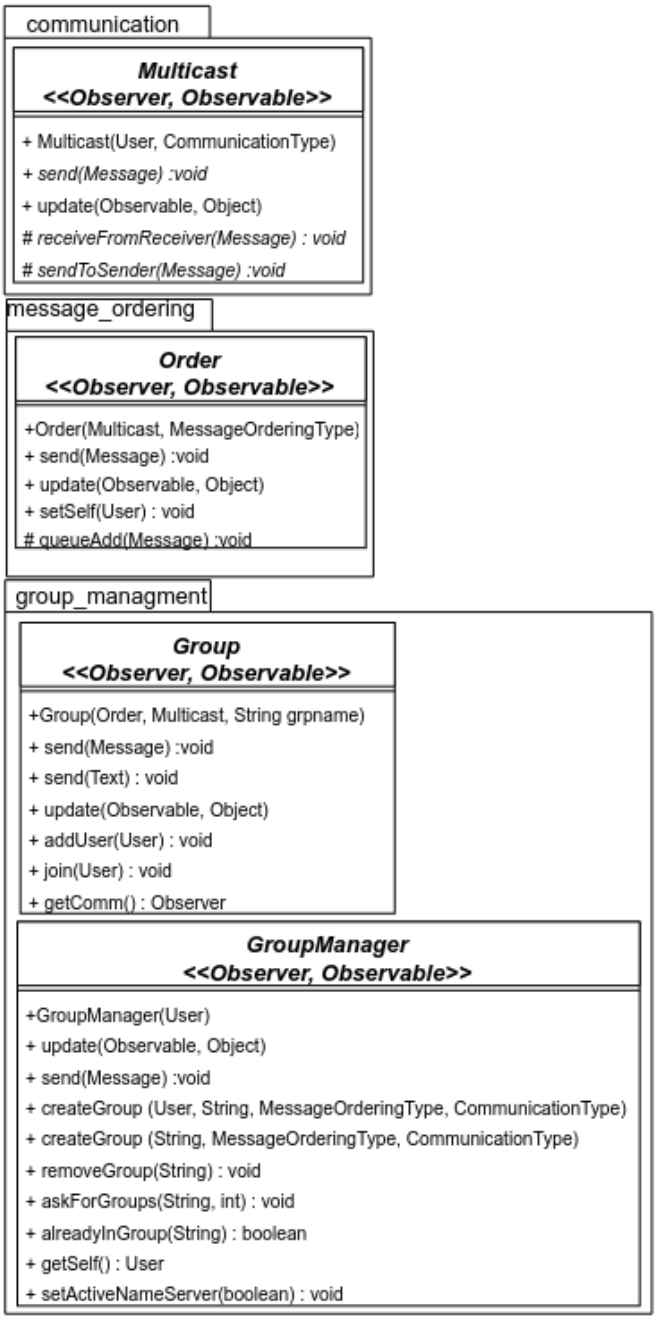

Figure 7: *Simplified UML-diagram of the implementation, excluding the message, rmi and client packages, this image focus on the GCOM-related pakages.*

GroupManager has the possibility to create, ask and remove groups as shown in Figure 7. The *group*-class is seen to have a pretty simple interface where the next layer (in this case this will be the user-interaction-layer).

In this implementation, the NameServer, the service that resolves group names and makes it possible for the user to find groups to join, is embedded within the GroupManager. A special group is created in the initialization that will override the observer-observable-design pattern, this makes it possible for the GroupManager to take care of any request from other users to request a list of groups. This also implicates that all users can be NameServers for all the groups that the client has joined or created, this service is, however, fully possible for the GroupManager to disable on initialization.

A. Debugger

To test the GCom modules, a debugger was created hidden within the communication-package. Since the Multicast has the situation of needing to resolve the problem of handling three possible inputs and two outputs (to the Sending-module and to the message ordering-layer). This in combination to be able to debug all parts of the program it would need to be able to create a queue-debug-system for the given inputs and outputs. To resolve this, two debugging points exist: message going to the Sender and messages coming from the Receiver (as seen in Figure 1) into the communication-layer. These two points make it possible to see how the communication-layer, as well as the message ordering-layer, solves any message permutation when using the Gcom Module.

The debugger will give the user permission to hold any messages, delete or change the order that they come to the multicast. This makes it possible to create any given permutation of messages occurring in the GCom module.

VI. Discussion

The NameServer solution, of letting each client be a NameServer is in some sense quite brilliant, code-wise - not so much. Even though that the GroupManager will take over the special group created in initialization and will solve the NameServer logic it can from time to time get a bit messy. This could easily have been solved with extending a group that would take care of this to minimize the coercion that the GroupManager got.

Another implementation-wise reflection is the debugger. Even though the valid points of needing to wrap the multicast of a debugger it has some problems. Since a message into the communication-layer needed to be stopped before (but within the Multicast-class), the instance would add the Message to a queue. This BlockingQueue would then be received in the debugger (by a created thread) that would check if the debugging-setting was turned on if so it would put it in a queue making

it possible for the user to debug. When the user lets a Message leave the debugger. The debugger would add it to a new BlockingQueue. The last mentioned queue is watched in the Multicast (another thread) that would allow the message into the ”real” communication-part of the layer. This is done for both the Receiver and Sender. Creating four extra threads in this layer. This creates a messy Multicast-class. This could possibly be cleaned up with another class. In addition, we needed to handle the thread creation in a correct way, that seems a bit off-putting, because of RMI characteristics.

In a positive note of the implementation, the code is very abstract. It is easy to add another subclass of Order or Multicast and the Observer-Observable design-pattern works quite well: The communication-layer has no idea what a message ordering class is. And message ordering has no idea what a group or a message ordering class is.

Also, the Message part works really well, for each layer just add the needed information for the message (for instance, the message ordering layer can add any type of Clock). The class relies on an enum saying what type of message it is (TEXT, ASKGROUP, JOIN, LEAVE et c.) and an Object that reconciles the message related to the type of message (String for TEXT, User for Join)

A. *Requirement fulfillment*

The implementation makes it fully possible to add, join and leave groups. In some manner, the user cannot remove a group even if the user created the group. However, when the last user leaves the group, the group will be automatically removed/unavailable.

The multicast implements the non-reliable multicast, reliable multicast, and tree-based multicast. The non-reliable multicast is rather reliable, working as a broadcast since the connection points to each client as seen in Figure 2. However, if a message disappears on the way from the sending user to a receiving user. It will not get the message, while if the group uses reliable multicast the user would.

The message ordering implementation shows the problems of getting a completely reliable ordering without using a total ordering. This was very helpful understanding using the debugger to understand both the application as well as the theory.

References

[1] Tim Kindberg Gordon Blair Jean Dollimore, George F. Coulouris. Distributed systems, 2012.
fifth edition.

[2] James F. Kurose and Keith W. Ross. Computer networking - a top-down approach featuring the internet., 2013.
sixth edition.