

Distributed Systems. 5DV020, HT09  
Ludvig Widman(dit06lwn), Emil Eriksson(c07een)

Tutors: Lars Larsson, Daniel Henriksson  
Date: October 26, 2009  
Bonus level: 2

Running the program:

```
$ cd ~dit06lwn/edu/DS/Distsys-Project/  
Start registry and sequencer: $ ant seq  
Start client: $ ant gui
```

## Assignment part 2

### GCom

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Tools and language for implementation</b>	<b>1</b>
<b>3</b>	<b>Realization of the project plan</b>	<b>1</b>
<b>4</b>	<b>Using the demo application</b>	<b>1</b>
<b>5</b>	<b>System Overview</b>	<b>1</b>
5.1	The gcom package . . . . .	1
5.1.1	GCom . . . . .	3
5.1.2	BasicCommunicationModule . . . . .	3
5.1.3	ReliableCommunicationModule . . . . .	3
5.1.4	GroupManagementModule . . . . .	4
5.1.5	RMIModule . . . . .	4
5.1.6	momNonOrdered . . . . .	4
5.1.7	momFIFO . . . . .	4
5.1.8	momCausal . . . . .	4
5.1.9	momTotal . . . . .	4
5.1.10	ReferenceKeeper . . . . .	4
5.1.11	HashVectorClock . . . . .	5
5.1.12	GroupDefinition . . . . .	5
5.1.13	RemoteObject . . . . .	5
5.1.14	Group . . . . .	5
5.1.15	Member . . . . .	5
5.1.16	Message . . . . .	5
5.1.17	Debug . . . . .	5
5.2	The rmi package . . . . .	5
5.2.1	RMIServer . . . . .	5
5.2.2	BackdoorOpener . . . . .	6
5.2.3	Sequencer . . . . .	6
5.2.4	SequencerCommunicationModule . . . . .	6
5.2.5	GroupSequencer . . . . .	6
5.3	The gui package . . . . .	6
5.3.1	GroupPanel . . . . .	7
5.3.2	GUIViewOther . . . . .	7
<b>6</b>	<b>Message ordering and message ordering modules</b>	<b>7</b>
6.1	Non ordered / momNonOrdered . . . . .	7
6.2	FIFO ordering / momFIFO . . . . .	8
6.3	Causal ordering / momCausal . . . . .	9
6.4	Total ordering / momTotal . . . . .	9
6.5	Causal total ordering . . . . .	10
<b>7</b>	<b>Design decisions</b>	<b>10</b>
7.1	Selection of bonus level . . . . .	10
7.2	Why an chat Application? . . . . .	12
7.3	Why using a sequencer for total ordering? . . . . .	12
7.4	Why the sequencer lives in the rmi-server . . . . .	12

<b>8</b>	<b>Fault tolerance</b>	<b>12</b>
8.1	Detecting crashed members . . . . .	12
8.2	If the registry restarts or is unresponsive . . . . .	12
8.3	If the group leader crashes . . . . .	13
8.4	Election of a new leader . . . . .	13
8.5	If the last member of a group crashes . . . . .	13
8.6	Netsplit . . . . .	13
<b>9</b>	<b>Implementation of requirement specification</b>	<b>13</b>
<b>10</b>	<b>Limitations</b>	<b>13</b>
<b>A</b>	<b>Project plan</b>	<b>14</b>
<b>A</b>	<b>Requirement specification from project plan</b>	<b>15</b>
A.1	Group management . . . . .	15
A.2	Communication . . . . .	15
<b>B</b>	<b>Bonus Level</b>	<b>15</b>
B.1	Message Ordering . . . . .	16

## **1 Introduction**

- Vad det är och vad det gör - Grupper, medlemmar, statiska, dynamiska, andra begrepp

## **2 Tools and language for implementation**

Since the clear recommendation was to use Java and Java RMI we have decided to follow the recommendation. While there are languages that would be more interesting and easier to design GUIs in (cocoa/objective c), Java is more easily ported and the available help from lecturers also helped tip the balance in Java's favor.

JUnit has been used for unit testing and logging has been done through the log4j-package.

## **3 Realization of the project plan**

The project plan from deliverable 1 is included in appendix A on page 14. We have been slightly behind the plan during most of the project part due to the new influenza that struck Emil during one of the first weeks.

We also moved forward the design of the GUI to milestone 4 since it was easier to do while implementing the actual application. We also moved total and causal total ordering forward one week since the other message orderings took most of the assigned week.

Generally things took slightly longer than anticipated, part due to tricky bugs and part due to tricky algorithms.

## **4 Using the demo application**

## **5 System Overview**

This project consists of a middle-ware for group communication and a demo application to show that it works. The middle-ware is located in the gcom and rmi packages, the demo application is in the gui package. An overview over all the packages and their classes can be seen in figure 1 on the next page.

### **5.1 The gcom package**

The gcom package is the main part of this project. This is where communication, message ordering and group management is handled.

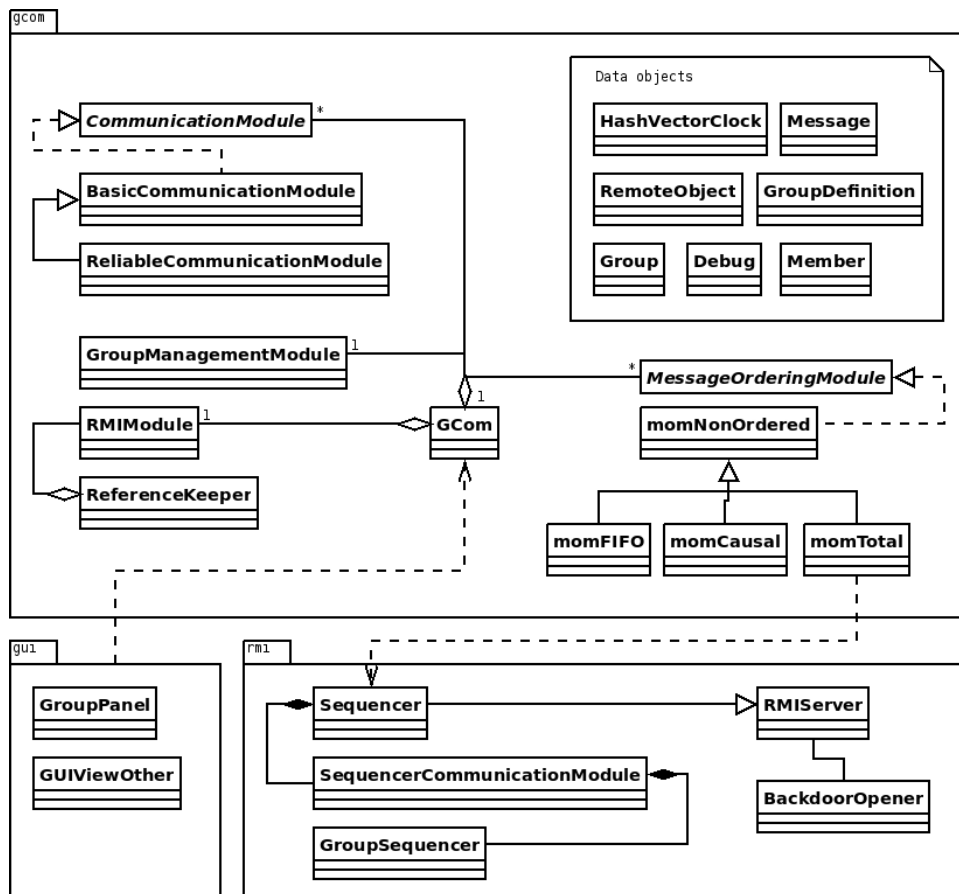


Figure 1: All the classes in the program and an overview of their relations.

### 5.1.1 GCom

GCom is the main component in this package and is where the demo application is connected. The application gives commands to GCom, for example create a group or send a message, these commands are translated to messages that GCom sends through the appropriate communication module and to commands to the GroupManagementModule so that the current group state can be maintained.

Information also flows from the communication modules via the message ordering modules back to gcom when messages from other clients is received. These messages either effects the group state or is application data that GCom forwards to the application.

GCom has one GroupManagementModule, one RMIModule and the same amount of communication and message ordering modules as the number of groups it handles.

### 5.1.2 BasicCommunicationModule

The BasicCommunicationModule (BCM) is the simplest implementation of the interface CommunicationModule. It implements basic multicast and has the ability to send and receive messages. Each communication module corresponds to a specific group.

When a message is sent the BCM sends it to every member of the group, including the sender. It also detects if the connection is refused to any of the senders and removes these members from the group via a message to GCom.

When the BasicCommunicationModule receives a message the message is sent directly to the message ordering module for the group.

### 5.1.3 ReliableCommunicationModule

The ReliableCommunicationModule (RCM) extends the BCM and implements reliable multicast.

When the RCM receives a message it checks if it has received it before, if not the message is added to a cache of received messages, sent to all group members via BCM and then delivered to message ordering.

Sending messages in the RCM also stores the message in the cache. The message is then sent to all members of the group and the message is delivered to the sender.

When a message is added to the cache, the size of the cache is also trimmed by removing the oldest message if the cache is larger than a specified max size.

#### **5.1.4 GroupManagementModule**

The GroupManagementModule (GMM) stores information about groups and can notify listeners when something changes.

The main responsibilities for the GMM includes storing group state, resolving group names to members and updating group state when changes are received.

#### **5.1.5 RMIModule**

The RMIModule connects to a rmi registry at a given host and port. It then supplies an interface for binding, finding, unbinding and rebinding objects.

#### **5.1.6 momNonOrdered**

The momNonOrdered is the simplest implementation of the MessageOrderingModule (MOM) interface. Every group has a corresponding message ordering module. The type depends on what the creator of the group chose.

Message ordering is described in detail in section 6 on page 7.

#### **5.1.7 momFIFO**

This message ordering sorts messages in a First In First Out order, in this case this means that messages is delivered in the same order the sender sent them. See section 6.2 on page 8 for details.

#### **5.1.8 momCausal**

Causal ordering orders messages according to a happened before relationship. See section 6.3 on page 9 for details.

#### **5.1.9 momTotal**

Total ordering means that all clients deliver the messages in the exact same order. See section 6.4 on page 9 for details.

#### **5.1.10 ReferenceKeeper**

The ReferenceKeeper is used by GCom only for the group leader. It checks that the rmi registry has the reference to the leaders RemoteObject and tries to rebind it if not. The ReferenceKeeper is run in it's own thread and rechecks the reference in the registry every minute.

#### **5.1.11 HashVectorClock**

The HashVectorClock implements VectorClock. It stores a clock state with integer values for different process IDs. The clock can be compared to another clock, merged with another clock and ticked, which increases the clock value for the owner by one.

#### **5.1.12 GroupDefinition**

A GroupDefinition stores information about a group like message ordering, communication type and if the group is static or dynamic.

#### **5.1.13 RemoteObject**

The RemoteObject is used for communication between processes via java RMI.

#### **5.1.14 Group**

Group stores members and GroupDefinition for a specific group. This class is only used by the GroupManagementModule for storing state.

#### **5.1.15 Member**

Member stores id, name and RemoteObject for a member in a group.

#### **5.1.16 Message**

Message is a serializable class used for sending information between processes and modules. A message contains a vector clock, group name, source, message type and message data. Messages used for requesting serial numbers from the sequencer also has a return address.

#### **5.1.17 Debug**

Debug holds a log4j logger and is used for printing debug and error messages.

### **5.2 The rmi package**

The rmi package contains classes related to the rmi registry.

#### **5.2.1 RMIServer**

The RMI server creates a registry and opens a backdoor for binding objects remotely.



### **5.2.2 BackdoorOpener**

BackdoorOpener is used by RMIServer to create a backdoor. The backdoor is needed since the used implementation of the registry does not allow remote processes to bind objects.

### **5.2.3 Sequencer**

The Sequencer is used for getting a serial number on messages that all clients can agree to. There is therefor only one sequencer per group, and in our implementation only one in total.

It extends the RMI-server and therefor first starts a registry. It then binds in a SequencerCommunicationModule SCM as "sequencer" in the registry which the clients that uses any of the total orderings will know that they can connect to.

Clients can request serial numbers by sending a message to the sequencer and supplying a return address in the message. The Sequencer will order the messages it receives in either a momNonOrdered or a momCausal depending on if total or causal total ordering is requested. Messages that has already been sequenced are cached and the same serial number will be returned if a new request for the same message arrives.

### **5.2.4 SequencerCommunicationModule**

The SequencerCommunicationModule (SCM) implements CommunicationModule since it makes communication with it convenient, but the module itself is only used by the sequencer.

It receives messages and returns them to the return address with a serial number. The serial numbers is supplied by a GroupSequencer. Each group has it's own GroupSequencer.

### **5.2.5 GroupSequencer**

GroupSequencer (GS) is used by SCM for setting serial numbers on messages. The GS uses a MOM to order the messages before a serial number is given to them. Messages are cached with their serial numbers so requests for the same message gets the same number.

## **5.3 The gui package**

The gui package implements our demo application: a chat application with tabs for groups.

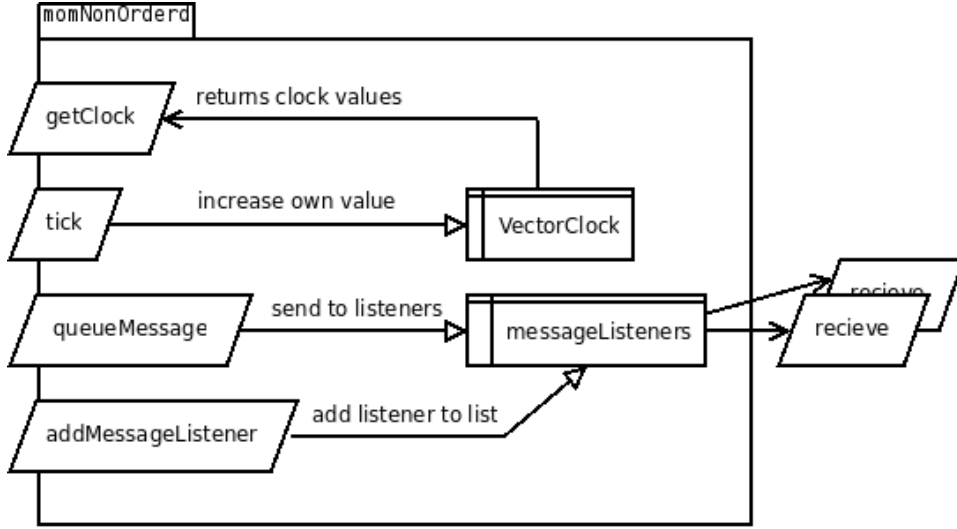


Figure 2: The non ordered message ordering module just forwards messages to it's listeners. The state of a vector clock is also kept in the module.

### 5.3.1 GroupPanel

### 5.3.2 GUIViewOther

## 6 Message ordering and message ordering modules

GCom features five kinds of message orderings: NonOrdered, FIFO, Causal, Total and CausalTotal. The first four of these has their own message ordering module. Each type of ordering and it's corresponding class is described in detail below.

The message ordering modules have each been thoroughly tested with unit tests to make sure that the algorithms works like intended.

Additional information about algorithms and definitions of the ordering types above can also be found in [1].

### 6.1 Non ordered / momNonOrdered

The non ordered ordering module (MOM) does not order messages at all. This is the base class for the ordering modules. Every message ordering module has a queuing method, a vector clock and message listeners. All the NonOrdered does is to receive messages in the queue method and send them through to it's messages listeners directly. This is also shown in figure 2.

The module also has two functions related to the vector clock. It increases the vector clock when told so and returns the clock values if asked. These functions is used in several of the child modules that inherits from momNonOrdered.

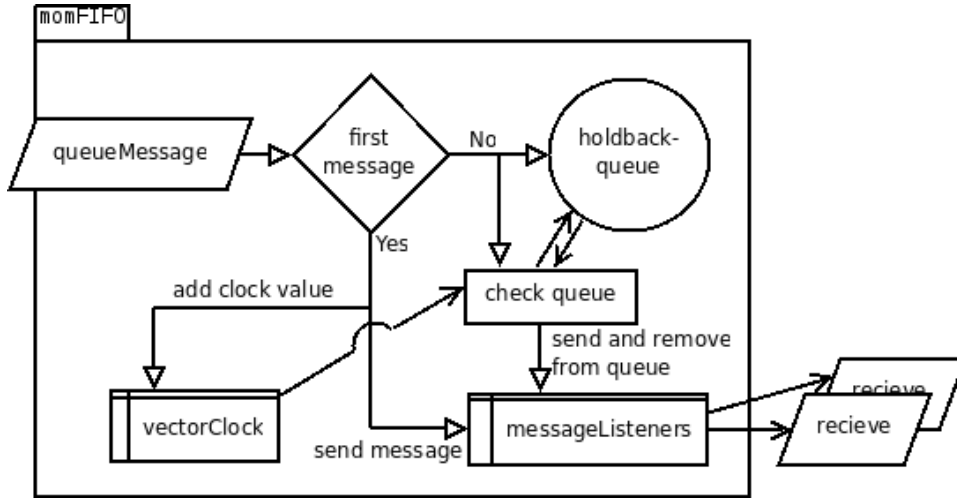


Figure 3: The FIFO ordered MOM places messages in a hold back queue until they can be delivered in the order they where sent. Messages from new senders are delivered directly as no previous clock value is known.

## 6.2 FIFO ordering / momFIFO

The FIFO ordered MOM delivers messages in the order they where sent. This works by numbering every sent message with a vector clock and storing received messages in a hold back queue until they can be delivered in order. Messages in the queue is compared to the previous last delivered message from the same sender. Messages from new senders is delivered directly as no clock value is known. This is also described in figure 3.

In detail the sorting algorithm works as follows:

1. Check if no previous messages has been received from this sender, if so:
  - (a) Send message to message listeners.
  - (b) Increase local clock value for sender by one.
2. If not, add message to the hold back queue.
3. Loop through the hold back queue and with any message that has a clock value that equals the local clock value plus one:
  - (a) Send message to message listeners.
  - (b) Increase local clock value for sender by one.
  - (c) Remove message from hold back queue.

### 6.3 Causal ordering / momCausal

Causal ordering is an ordering where a message can not be delivered before a message that happened before it. Just like in the FIFO ordering the message is first placed in a hold back queue. If a new sender is detected their clock value is added to the local clock.

Messages in the queue is checked when a message is received. If all previous messages from the same sender has been delivered, and all messages delivered by the sender when the message was timestamped has been delivered, then the message can be delivered. See figure 4 on the following page.

In detail the sorting algorithm works as follows:

1. Check if no previous messages has been received from this sender, if so:  
Add senders clock value minus one, from message to local clock
2. If not, add message to the hold back queue.
3. Loop through the hold back queue and with any message that:
  - (a) Has a clock value for the sender that equals the local clock value for the sender plus one
  - (b) Has clock values before or equal to corresponding local clock values, excluding the value for the sender

Do the following:

- (a) Send message to message listeners.
- (b) Increase local clock value for sender by one.
- (c) Remove message from hold back queue.

In step 1 the clock value minus one is added to the local clock, this is since the message can not be delivered directly like in FIFO ordering (the message might not conform to criteria b).

### 6.4 Total ordering / momTotal

Total ordering is an ordering where every client delivers the messages in the same order. In our implementation this is solved by the use of a sequencer, which is an application that is global for the group.

When a message is received it is sent to the sequencer. The sequencer gives the message a serial number and sends it back. When the MOM gets a message with a serial number it puts it in hold back queue until it can be delivered in sequence (that is until all messages with a lower number is delivered). This is also described in figure 5 on page 11.

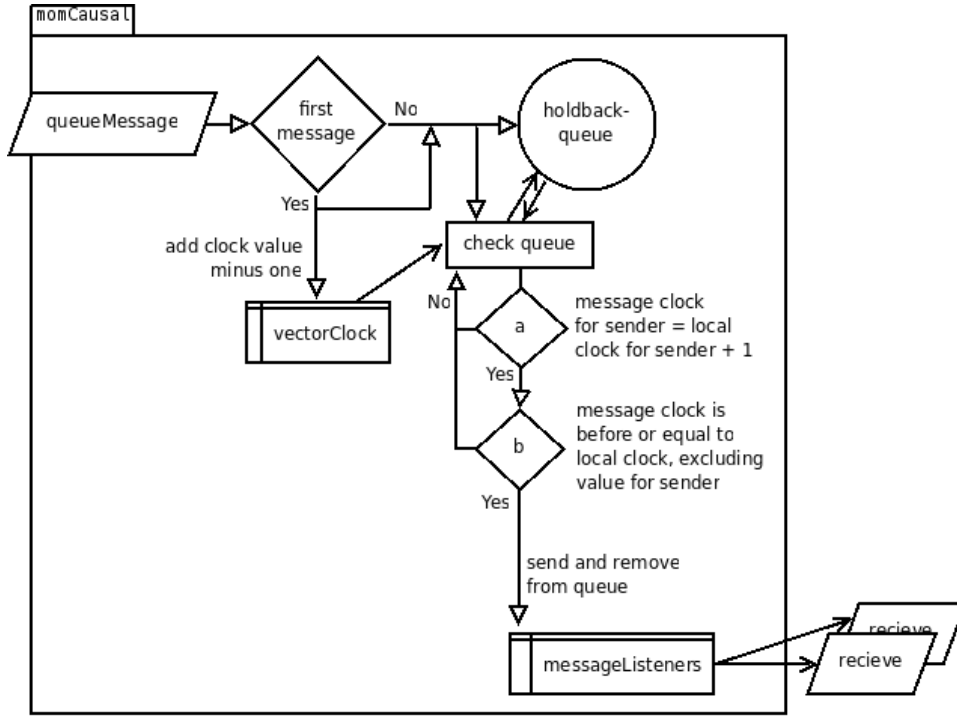


Figure 4: The Causal ordered MOM places a message in a hold back queue until all messages that happened before it has been delivered.

The first message with a serial number the MOM receives is delivered directly and any message with a lower number than that is discarded.

## 6.5 Causal total ordering

Causal total ordering uses the same MOM as total ordering. The only difference is in the sequencer.

With causal total ordering the sequencer uses a momCausal to order the messages in causal order and gives them serial numbers in the order they are delivered. This interaction is shown in figure 6 on the following page.

## 7 Design decisions

### 7.1 Selection of bonus level

We chose to solve bonus level two with dynamic groups since it seemed like a reasonable trade off between extra credits and extra workload.

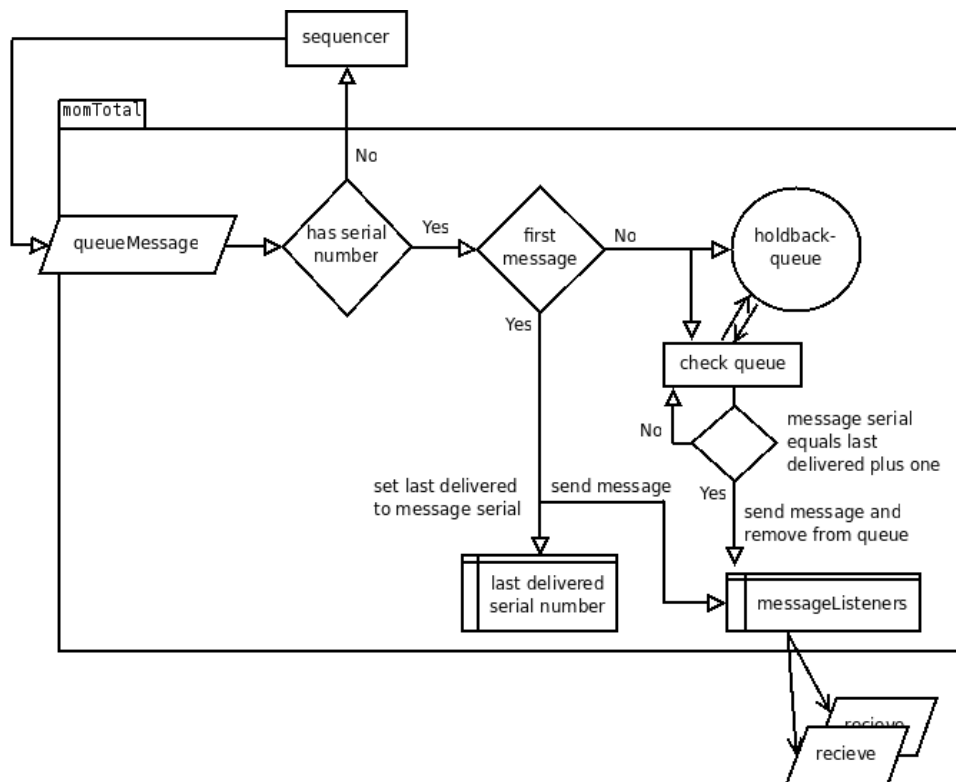


Figure 5: Total ordering uses a sequencer to deliver messages in the same order in every client.

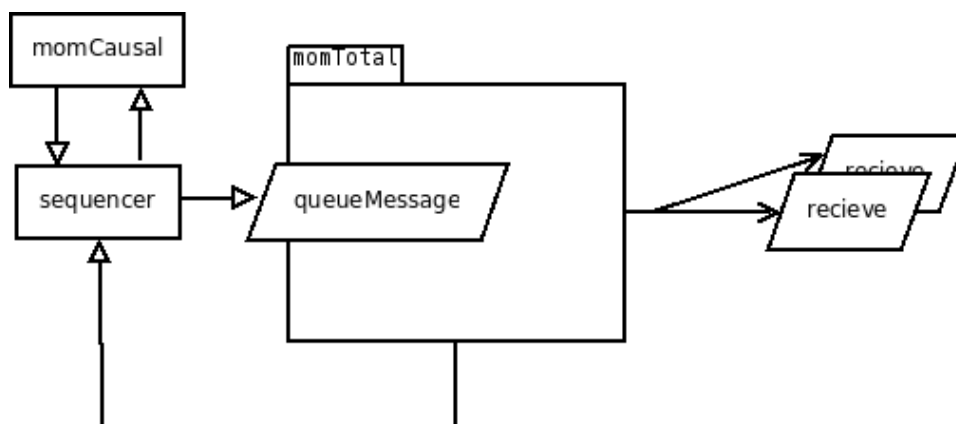


Figure 6: Causal total ordering also uses a sequencer, but here the sequencer has a MOM of it's own to give the messages serial numbers in causal order.

## **7.2 Why an chat Application?**

As our demo application on top of gcom we built a chat application with tabs for different groups. This type of application maps well to the underlying structure of members that belongs to groups. It also gave an easy way to see that messages arrive unharmed and in the correct amount.

## **7.3 Why using a sequencer for total ordering?**

When implementing total ordering one can either use a sequencer or a distributed algorithm. The sequencer method has the downside of the sequencer being the single point of failure. However we still choose that method since it seemed more straightforward to implement and we already had a single point of failure.

## **7.4 Why the sequencer lives in the rmi-server**

The RMI-server has to be started before the sequencer and we only wanted a single instance of both so it made sense to put them in the same module (or rather, make the sequencer extend the RMI-server). The RMI-server is also a hopefully quite stable host which makes it a good location for the sequencer.

It would have scaled to more users to put the sequencer in the group leader, but this would make the sequencer less stable (since the group leader might choose to leave the group).

An obvious drawback of having a single sequencer for all groups is that it becomes a bottleneck. This could be solved by moving the sequencer to several dedicated hosts and assigning groups to one of many sequencers.

# **8 Fault tolerance**

## **8.1 Detecting crashed members**

The communication module detects if a client refuses connections or if connections timeout. If this happens the member is removed from the group management module and a lost member message is sent.

If the crashed client is the group leader a new leader is elected.

## **8.2 If the registry restarts or is unresponsive**

The leader of each group occasionally checks that the group is registered in the RMI-registry. If the registry has restarted for some reason the leader will re-register the group.

### **8.3 If the group leader crashes**

If the group leader crashes the group will detect that no messages arrive from the leader and then elect a new leader.

### **8.4 Election of a new leader**

### **8.5 If the last member of a group crashes**

If the last member of a group crashes no one will unregister the group. This will potentially lead to a lot of "dead" references in the registry if there is a bug causing nodes to crash when they are the last to leave a group. In order to counter this extra testing has been done to make sure that nodes behave when leaving and unregistering with the registry.

There is currently no way to clear out old references in the registry (other than restarting it periodically).

### **8.6 Netsplit**

If the group is split in two partitions, due to a cable failure or similar, the partition with the leader will continue to work as before. The other partition will elect a new leader when they discover that they no longer receive messages from the leader.

The new leader will try to reregister the group in the registry. The leader for the other part will do the same and therefore compete over the reference in the registry. Because of this newcomers will join the group having the reference in the registry at the time the node connects.

## **9 Implementation of requirement specification**

Below our requirement specification from our project plan is included in appendix A on page 15. We have succeeded with implementing all required specifications.

## **10 Limitations**

- The RMI-registry is a single point of failure. If the registry crashes no processes can find or join groups until the registry has been restarted.
- If a group is split in two parts due to network failure and both parts still have contact with the registry bad things will happen. This is assumed to never happen.



- All processes are assumed to be good. If a process starts to lie or send malicious messages bad things will happen.
- If the sequencer crashes all groups using the sequencer must be restarted since the message cache is lost and messages will get new serial numbers starting at 1. The sequencer is placed at the same host as the registry which is assumed to be relatively stable for this to not be a big problem.
- If a group contains too many users java will run out of memory and the client will crash. If reliable multicast is used the network will most likely also be very congested.

## References

- [1] Distributed systems : concepts and design / GeorgeCoulouris, Jean Dollimore, Tim Kondberg.—4th ed. p. 490-498

## A Project plan

Date	Milestone	Content
11 Sept	Part 1 due	Written project plan
18 Sept	Milestone 1	Create all interfaces Build dataobjects Design GUI mockups for test app
25 Sept	Milestone 2	Implement basic methods like non-ordered ordering and non-reliable multicast.
2 Oct	Milestone 3	Implement advanced methods like total ordering and reliable multicast.
9 Oct	Milestone 4	Implement GUI-application and debug mode.
16 Oct	Milestone 5	Resolve distributed problems. Write on the report. Make test protocol.
23 Oct	Finalization	Everything should be finished. Minor adjustments.
27 Oct	Hand in report	
28-30 Oct	Demonstration	

## A Requirement specification from project plan

### A.1 Group management

**REQUIRED** External processes must be able to find and connect to group leaders via a RMI registry.

**REQUIRED** The group leader must regularly make sure that the group and leader is registered in the RMI registry.

**REQUIRED** Any process must be able to create a new group and register itself as leader in the registry.

**REQUIRED** Any process must be able to join a group by connecting to the group leader.

**REQUIRED** Every process in a group must notify the other processes of any known changes to the group composition.

**REQUIRED** Every process must keep track of the members of all the groups it's a member of.

**REQUIRED** Every process must monitor if another process in the group stops responding and report a corresponding change in group composition.

**REQUIRED** If the leader of the group leaves or crashes a new leader must be elected in the group.

### A.2 Communication

**REQUIRED** It must be possible to send messages to

## B Bonus Level

We have solved bonus level two with dynamic groups.  
o the members of joined group.

**REQUIRED** Both non-reliable and reliable multicast must be supported for sending messages.

**REQUIRED** The communication method chosen by the group creator (reliable or non-reliable multicast) must be used when communicating in a group.

**REQUIRED** Every message must contain a updated vector clock.

## **B.1 Message Ordering**

**REQUIRED** Any message must arrive in proper order (according to the selected sorting requirements).

**REQUIRED** A message must not be delivered until all previous messages are delivered if the message ordering demands so.