UMEÅ UNIVERSITY
Department of Computer Science
Report

October 27, 2009

Deliverable 2

Distributed systems, HT-09

GCom

Anton Johansson, dit06ajn@cs.umu.se
Jonny Strömberg, dit06jsg@cs.umu.se

Start-server: `~dit06ajn/edu/dist/GCom> ./start-server.sh`

Start-gui: `~dit06ajn/edu/dist/GCom> ./start-gui.sh`

Supervisors
Lars Larsson, larsson+ds@cs.umu.se
Daniel Henriksson, danielh+ds@cs.umu.se

Contents

1 Introduction

This report explains a solution for implementing a distributed group communications middleware.

A distributed system is composed of separated processes that coordinate activities by passing messages. A middleware is a software layers that enables rapid development of other software by supplying simple method-calls that hides the underlying implementation details off the middleware and provides some desired functionality.

The middleware described in this report is called *GCom* and provides an API[1] for group communication with different message sending/delivery rules. Two communication methods are implemented: *Reliable multicast*, *Basic multicast*, described in greater detail in section 4.6.

Four message-ordering types are implemented: *Non-ordered*, *First in first out*, *Casual*, *Total* and *Casual-Total*, described in greater detail in section 4.5.

The system is implemented in the programming language *Java*[2] and uses *Java RMI*[3] for network communication.

The original specification of this practical assignment can be found at (*October 27, 2009*):
`http://www.cs.umu.se/kurser/5DV020/HT09/assignment.html`

2 Problem analysis

The group communication for *GCom* is specified to be a distributed system, which means there can be no central server that coordinates all activities for individual group members. Four guidance on how to implement such a system the book *Distributed Systems: Concepts and Design*[**?**] list three important consequences of a distributed system:

- **Concurrency:** Program execution are concurrent. In the case of *GCom*, message receiving and handling are concurrent itself and with other parts of the middleware such as message sending and ordering.
- **No global clock:** There is no global clock to coordinate activities by. That is clock timestamps can not be used to order messages received by *GCom*.
- **Independent failures:** All individual parts of the distributed system can fail at any time and place in execution. This must be considered when implementing algorithms for coordinated actions of *GCom*.

The environment in which *GCom* will execute will defined by a model for distributed system called *asynchronous*-system defined by three assumptions [**?**]:

[1] Application programming interface
[2] `http://java.sun.com/`
[3] `http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp`

- There is no guarantee of **execution speed**, a process may respond to a request immediately or after several years.
- In a similar manner there can be **transmission delays** in the network were messages are passed. A message can take arbitrary long time to arrive at its destination.
- As stated before, there is **no global clock**. One process can make no assumptions about the clock in another process.

2.1 Group partitioning

When considering the previous characteristics of the environment for *GCom*, a group of processes can at any time be divided in two groups without any means for communication between them. It would be impossible for the groups to determine whether the group members of the other group still executes and behaves normally. Therefore a partition of a group is treated as a crash of all the members cut off. This means that merging such a group when communication can be achieved again is done by a new join for all the members in one off the groups.

2.2 Member failures

A member of a group is considered to have failed only when it throws a *RemoteExceptioin*[4] as defined by *Java RMI*. This means that *GCom* makes no guarantee about the time it takes to send a message to a group. This guarantee could be achieved simply by changing the definition of a member failure to include a time-limit for message delivery.

2.3 Group discovery

When a process wants to communicate with other processes using *GCom* there must be a way to find groups and group members already existing. That starting point is defined by a global address known by all *GCom* members. This starting point will contain a service for group discovery, described in more detail in section 4.3.

3 Usage

All files needed to use this middleware are located at:
`~dit06ajn/edu/dist/GCom`

This catalog contains the following sub directories:

- The directory `src` contains the source code.
- The directory `src/main/resources/` contains configuration files for standard behaviour of the compiled system, see section 3.2

[4] `http://java.sun.com/javase/6/docs/api/java/rmi/RemoteException.html`

- The directory `src` contains the source code.
- The directory `bin` will, after a successful compilation, contain all the compiled sources as well as configuration files used by this middleware.
- The directory `lib` contains all requires third-party libraries needed by *GCom*, se section 3.3.
- The directory `doc` contains the Javadoc API for *GCom*.

3.1 Compilation

The following commands will require the software tool *Apache Ant*[5]. More details about what happens using *ant* in this project is found in the file *build.xml*[6].

To compile *GCom* issue the following command:

```
salt:./GCom> ant
```

This will create a directory `bin` if it does not already exists and compile/move source-code and configuration files to that directory.

The root-directory for class-files when using *GCom* is compiled to *bin/main/java*, while the root-directory for test-code is compile to *bin/test/java*.

To create *jar*-file of the compiled sources issue the following command:

```
salt:./GCom> ant jar
```

This will create *GCom.jar* which can be used when developing in third party software or directly as a *GNS*-server (see section 4.3) by running:

```
salt:./GCom> java -jar GCom.jar
```

3.2 Configuration

The compiled system uses two configuration-files to define its standard behaviour, these files are located in the directory *src/main/resources/*.

3.2.1 application.properties

The file *application.properties* defines the standard multicast and ordering types to use when communication with a group. Notice though that these settings are only used for the creator of a group that did not exist from before. When connecting to an existing group, the settings from that group will suppress the settings in *application.properties*. CodeSnippet 1 shows the content of an example configuration that uses *FIFO* ordering with *Reliable multicast*. These settings can be overridden by setting a system property. For example starting a *GCom* member with the command-line parameter *-Dgcom.ordering=CASUAL_ORDERING* will make the member use *Casual* ordering instead of the order stated in the *application.properties* file.

[5]http://ant.apache.org/

[6]http://ant.apache.org/manual/using.html

CodeSnippet 1 applications.properties

```
# Used by GNS
gcom.gns.port=1078

# FIFO, TOTAL_ORDER, NO_ORDERING,
# CASUAL_ORDERING, CASUALTOTAL_ORDERING
gcom.ordering=FIFO

# BASIC_MULTICAST, RELIABLE_MULTICAST
gcom.multicast=RELIABLE_MULTICAST
```

3.2.2 logback.xml

The file *logback.xml* defines the behaviours of logging when using *GCom* and *Logback*, see section 3.3.1. The settings in this file is ignored by default if a system property *logback.root.level* is set to a specified logging level, e.g. *logback.root.level=OFF* turns all logging off.

Every class has a separate logger-name consisting of its fully qualified class-name. This means that logging can be configured per class or package. For example to only print debug information from the *se.umu.cs.jsgajn.gcom.management* package you could use the configuration shown in CodeSnippet 2. For more information configuring logback check their manual[7].

CodeSnippet 2 logback.xml

```
<configuration>
   <!-- ... -->
   <logger name="se.umu.cs.jsgajn.gcom.management">
      <level value="${logback.root.level:-DEBUG}" />
   </logger>

   <root>
      <level value="${logback.root.level:-OFF}"/>
      <appender-ref ref="STDOUT" />
   </root>
</configuration>
```

3.3 Required libraries

Basic functionality of *GCom* requires no extra libraries other than the standard edition *Java 6* platform. However for some extra functionality *GCom* internally uses some third party software located in the *lib* directory and described in the following sections.

3.3.1 SLF4J and Logback

For logging capability *GCom* uses *Simple Logging Facade for Java (SLF4J)*[8] which provides a facade for different implementations of logging frameworks. The logging back-end used by default is *Logger*[9]. This combination provides logging capabilities to get information about *GCom*s status at runtime.

[7]`http://logback.qos.ch/manual/configuration.html`

[8]`http://www.slf4j.org/`

[9]`http://logback.qos.ch/`

3.3.2 JUnit

For testing the individual parts of *GCom*, tests are written using the *JUnit testing framework*[10]. The tests cover some special parts parts of the system where unexpected results would otherwise be very hard to debug. All tests are located in the folder *src/test/java/* and are compiled to *bin/test/java/*.

To run all tests issue the following command:

```
salt:./GCom> ant test
```

Note hoverer that some tests require that some ports are note bound on *localhost* by other processes, and therefore the tests can fail because of bind exceptions.

3.3.3 Implementing a GCom application

It is easy to implement *GCom* into an application. After importing the classes necessary it is just to create a ManagementModuleImpl with requisite parameters. Two methods is included in the Client interface, *send(Object)* and *deliver(Object)*, these are the only thing that is used when communicating with the GCom middleware, for minimal sample see CodeSnippet 3.

CodeSnippet 3 GCom application

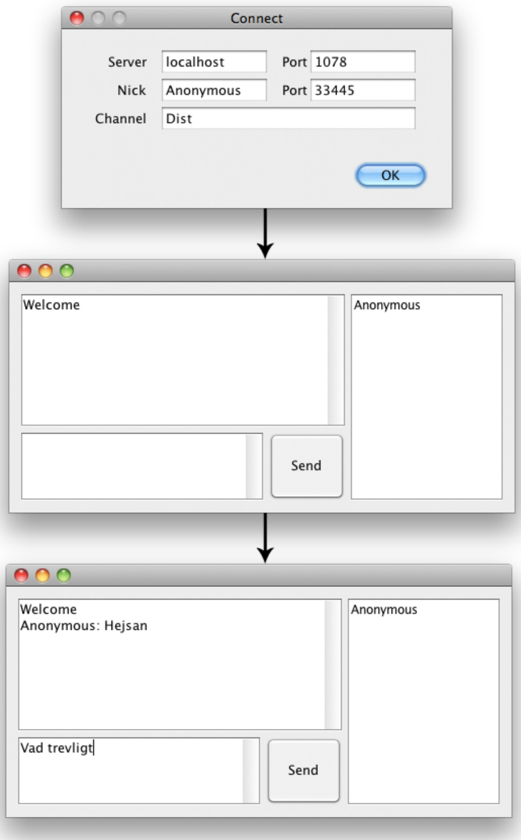
```
import se.umu.cs.jsgajn.gcom.Client;
import se.umu.cs.jsgajn.gcom.management. /
                             ManagementModule;
import se.umu.cs.jsgajn.gcom.management. /
                             ManagementModuleImpl;

public class App implements Client {
    public App() {
        String host = "hostname";
        int hostPort = "1098";
        int localPort = "33445"
        String group = "YourGroupName";
        try {
            managementModule =
                new ManagementModuleImpl(
                    this, host, hostPort,
                    group, localPort);
        } catch (RemoteException e) {
        } catch (IllegalArgumentException e) {
        } catch (AlreadyBoundException e) {
        } catch (NotBoundException e) {
        }
    }

    public void send(){
        managementModule.send(Object message);
    }

    public void deliver(Object message) {
        // Do whatever you want with message
    }
}
```

[10]http://www.junit.org/

3.3.4 Example application

A GUI-application using GCOM is created for demostration purpos. It can be viewed in figure 1. To test the applications use this line of code:

```
salt:./GCom> ./start-gui
```



Server localhost Port 1078
Nick Anonymous Port 33445
Channel Dist

Welcome
Anonymous

Welcome
Anonymous: Hejsan
Anonymous
Vad trevligt

Figure 1: GUI-application example

4 System description

The *GCom* middlewares distributed network consist of one *Group Name System* (GNS), *group views* and *group members*. Each *group views* contains one or more *group members* and one of these *group members* servers as *group leader*. The *GNS* has no other knowledge of the *group views* except for a referens to the *group leader*.

4.1 Remote Method Invocation

To keep track of the groups, the *GNS* has a referens to a remote object that represent each groups leader.

Each group member has a referens to a remote object representing the GNS and one for each other group member. This means all group members are connected to each other.

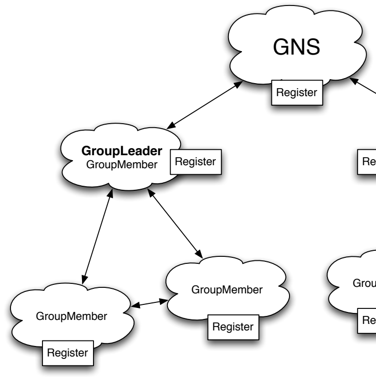


Figure 2: GCOM overview

4.2 The modules

The *GCom* middleware is separated in three different modules to separate different behaviours, see figure 3 on the next page. The first module which will have the closest connection to implementing software is the *management module*. This module handles group membership changes and actions. The second module *ordering module*, handles message ordering. The third module *communications module* is the one that actually sends and receives messages to and from the group. These modules will be discussed in more detail in the following sections.

4.3 Group Name System

To act as an entry point for group members in a *GCom* system there must be an instance of *se.umu.cs.jsgajn.gcom.GNS* running on a machine with a known address. The *GNS* acts as a Group Name System service which means that it resolves a group name consisting of string, to an instance of the remote interface of the group leader.

When a process wants to connect it provides a groupname and if the group exists, the *GNS* returns the instance of the group leaders remote interface. If the group doesn't exist a new group is created and the connecting process is assigned as group leader.

The *GNS* is *GCom* only critical point of failure. If the *GNS* crashes no new group members can join groups without an instance of any group members remote interface.

If the *GNS* where to crash it can be started once again with a serialized object of the groups it contained before the crash. This is done with the following command:

```
salt:./GCom> java -jar GCom.jar GroupSettingMap.ser
```

The previous command provides the *GNS* with a file *GroupSettingMap.ser* which is saved by the last instance of a running *GNS*.

4.4 Group management module

The *group management module* is the top one in the application stack, see figure 3. Software using the *GCom* middleware will should handle all communication through this module by creating an instance of it and passing an instance of *se.umu.cs.jsgajn.gcom.Client* to the constructor of the *Group management module*.

A newly created *group management module* will initialize all other modules needed for a fully functional *GCom* application stack.

The default implementation of the *group management module* is implemented in the class (package truncated):
s.u.c.j.g.management.ManagementModuleImpl.
This implementation will contain one thread to send messages and one thread to receive messages. All messages to be sent are first paced in a priority queue where all messages are ordered according to the *first in first out* principle, except some system messages that are prioritized before client messages. The system are those sent when a process wants to join a group, the message sent when a group constellation has changed and when a member has detected that another member has crashed.

4.4.1 Group leaders

Since groups may contain large amounts of group members the *GNS* only contains a reference to one of the group members. This member is called the group leader. A group leader is responsible for receiving and managing joining members. In the case of the total ordering of messages the group leader will act as a sequencer for the group, see section 4.5.4.

4.4.2 Error handling

When sending messages a group member may detect that one of the receiving members has crashed. If the detecting member is a group leader it will directly send a groupchange message, otherwise it will send a membercrash message which when received by the group leader will result in a groupchange message multicasted to the group. A groupchange message contains information about the complete new group composition whereas a membercrash message only contains information about the members that have crashed.

When the member that has crashed is the group leader the exact same procedure is used except that when processes receive membercrash messages they will check if they are the new group leader, and if they are they will send a groupchange message to the group. Group members test if they are the new group leader by checking if they have the maximum value of *UUID*s in the current group. This is a variant of the *Bully alghoritm*, see page 482 [**?**].

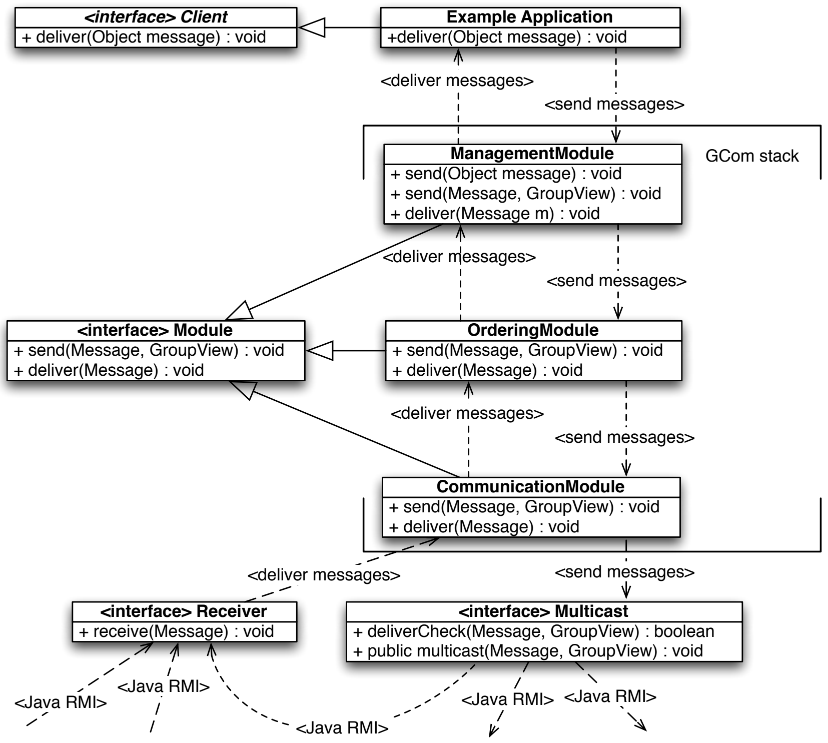

<interface> Client
+ deliver(Object message) : void
Example Application
+deliver(Object message) : void
<deliver messages>
<send messages>
GCom stack
ManagementModule
+ send(Object message) : void
+ send(Message, GroupView) : void
+ deliver(Message m) : void
<interface> Module
+ send(Message, GroupView) : void
+ deliver(Message) : void
OrderingModule
+ send(Message, GroupView) : void
+ deliver(Message) : void
CommunicationModule
+ send(Message, GroupView) : void
+ deliver(Message) : void
<interface> Receiver
+ receive(Message) : void
<interface> Multicast
+ deliverCheck(Message, GroupView) : boolean
+ public multicast(Message, GroupView) : void
<Java RMI>

Figure 3: GCom stack

4.5 Message ordering module

The *Message ordering module* is responsible for message ordering. That is it will for different orderings guarante that all messages delivered from this module up to the group *management module* is order according to the current ordering. The *management module* will send all outgoing messages to this module, which will prepare the message with necessary information such as vector-clocks and pass the message along to the *communications module*. The *communications module* will in turn deliver received messages from other group members through the *ordering module*. When messages are delivered here they are immediately put in the current ordering implementation. All ordering implementations describe in the following sections act as *BlockingQueue*[11]. When taking messages from them they block until they can release a message which confirms to their message ordering guarantee.

Because *GCom* is implemented to allow members to join already existing groups the different type of orderings handle their first ever message from a group member as a starting point for ordering subsequent messages. This means that if a message is received after the first message that should have been received before the first message, it is discarded.

[11]http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/BlockingQueue.html

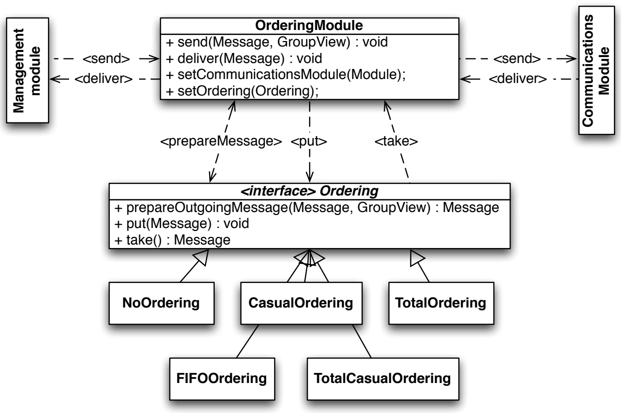


Figure 4: Ordering module

4.5.1 Non-ordered

The *non-ordered* ordering does not make any guarantee about the ordering of messages. It do however release messages it receives in the exact same order as it gets them.

4.5.2 FIFO ordering

The *FIFO* ordering will release messages received from a correct process in the same order as they are sent from that process. This is done by keeping a counter of how many messages it has delivered for each group member. When preparing an outgoing message *FIFO* will piggyback onto that message the total number of messages it has prepared and sent. When receiving a message *FIFO* compares the counter in the message with its own receive counter for the sending message. If the message is not the next in order for that sender, the message will be put in a hold-back queue until it is.

Table 1 shows a scenario where one process *p3* receives all messages in the reverse order they are sent, this results in an *FIFO* order for message from the same sender but messages from different senders are not guaranteed to be delivered in the same order for all processes.

	p1	p2	p3 (hold all messages)
Send order	one		
		2	
	three		
		4	
			(Release messages in reverse order)
Receive order	one	one	2
	2	2	4
	three	three	one
	4	4	three

Table 1: Fifo ordering tests

4.5.3 Causal ordering

The *Casual* ordering guarantees that messages will be ordered by a cause and event ordering. That is if one message is sent as a reply to another message all members will deliver those messages in the correct order. More formally from [?]: *"If multicast(g, m) → multicast(g, m'), where → is the happened-before relation induced only by messages sent between the members of g, then any correct process that delivers m' will deliver m before m' "*.

This is done by letting each group member keep track of the number of messages it has delivered from each other group member as well as keeping a counter for the number of messages it has sent to the group. This record of message counters is called a *vector-clock*, see page 447 [?].

When preparing a message for sending every process piggy-backs its vector clock on the outgoing message.

When receiving a message *Casual* ordering places the message in a hold back queue until it the next to be delivered from the sending member, and it has delivered all messages that the sending member had delivered when sending that message.

Table 2 shows a scenario where process *p3* receives all messages in the reverese order from which they are sent. This results in a *casual* order where all messages are ordered according to the messages delivered by the sending processes at send time. All messages are in order for process *p3* except message *V* since no process had delivered that message when sending the other messages.

	p1	p2	p3 (hold all messages)
Send order	one		
		2	
	three		
		4	
			V
			(Release messages in reverse order)
Receive order	one	one	V
	2	2	one
	three	three	2
	4	4	three
	V	V	4

Table 2: Casual ordering tests

4.5.4 Total ordering

Total order makes all the processes in the system to deliver all messages in the same order. This doesn't mean it's the right order, but it is the same. This is possible by letting the group leader become a sequencer. This means that all messages in the system have to ask the leader for a sequence number before they can multicast their messages. Then all processes have to order incoming messages after this sequence numbers.

Table 3 show a scenario where process *p3* receives all messages in the reverse order from which they are sent. This does not effect the outcome at all, messages are delivered at all processes in the same order.

Table 4 show a scenario where the sequencer deliveres sequence numbers in the reverese order for which they are received. This results in a reverse order for all processes.

	p1	p2	p3 (hold all messages)
Send order	one		
		2	
	three		
		4	
			V
			(Release messages in reverse order)
Receive order	one	one	one
	2	2	2
	three	three	three
	4	4	4
	V	V	V

Table 3: Total ordering tests

	p1	p2	p3 - Sequencer (hold all messages)
Send order	one		
	two		
			(Release messages in reverse order)
Receive order	two	two	two
	one	one	one

Table 4: Total ordering tests

4.5.5 Causal-Total ordering

The *Casual-total* ordering is almost identical to *Total* ordering except that the sequencer will use an instance of the *FIFO* ordering implementation to make sure that before sending out sequences number for requests, the requests should first be casualy ordered. This means that all messages delivered in the group are both casualy and totaly ordered. For a discussion about the reasoning behind using a *FIFO* instead of a *Casual* ordering in the sequencer, see section 5.1.

Table 5 shows a scenario where the sequencer receives messages in reverse order. This does not affect the sequence numberse released from the sequencer since they are first *FIFO* ordered.

	p1	p2	p3 - Sequencer (hold all messages)
Send order	one two		
			(Release messages in reverse order)
Receive order	one two	one two	one two

Table 5: Casual-Total ordering tests

4.6 Communications module

The *communications module* is responsible for the actual sending and receiving of messages for a group member. All communication is done through *Java RMI* by calling methods on remote objects.

A *communications module* will use an instance of a *Multicast* interface to multicast messages to a group. Two multicast methods are implemented, *Basic multicast* and *Reliable multicast*, see figure 5.

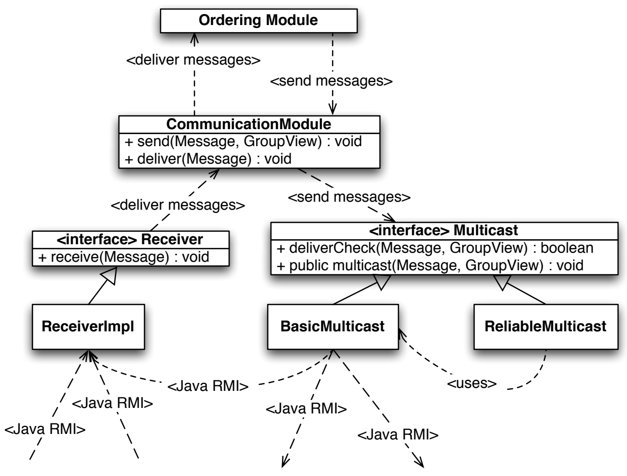


Figure 5: Communications module

4.6.1 Basic multicast

The multicast method *Basic multicast* makes no guarantee that all members of a group will receive a message sent. Sending messages is done with *one-to-one* communication with every member of the group. If for example the process where to fail when halfway through sending messages to the group, then the reminding half of the group will never receive that message. This is handled by *Reliable multicast*.

4.6.2 Reliable multicast

The multicast method *Reliable multicast* guarantee that a message delivered at one correct process is delivered at all correct processes.

4.7 Debugger

The debugger is a singelton and works as an entirely independent application. It can be activated in all classes of GCOM and it has a lot of useful functions. The debugger is uniq for the group member that starts it, that means it montors only the things which that group member.

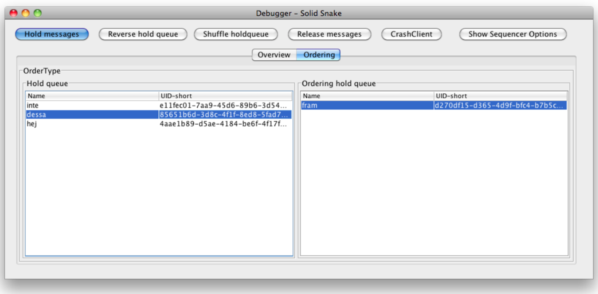
The debugger can be used to reproduce the ordering examples in the tables 1-5. The interface is shown in figure 6 on page 9 and figure 7 on the next page.

4.7.1 Debugger functions

Here is a list of all the functions in the debugger.

- Table with all received messages including this info
  - Number of times the message been received
  - Short id (all message receives a debugger-id 1,2,3++)
  - UUID
  - Message content
  - Original sender UUID
  - The message-object
  - Doubleclick : Path message has taken, is it a systemmessage, its sequencenumber
- List all delivered messages including this info
  - Short id (all message receives a debugger-id 1,2,3++)
  - UUID
  - Message content
  - Original sender UUID
  - The message-object
  - Doubleclick : Path message has taken, is it a systemmessage, its sequencenumber
- Information internal vector clock
- List of the members in the current group view
- Hold/unhold messages from being delivered to the client
- Shuffle or revers the hold queue
- Release one or all messages from the holdqueue
- Table with all de helded messages

  - Message content
  - UUID
- Table with the messages that is currently held in the ordering-class (except for the CasualTotal-ordering)
  - Message content
  - UUID
- Possibility to let two messages when using the CasualTotal-ordering change order before goring to the sequencer.



Hold queue

Name	UID-short
inte	e11fec01-7aa9-45d6-89b6-3d54...
dessa	85651b6d-3d8c-4f1f-8ed8-5fad7...
hej	4aae1b89-d5ae-4184-be6f-4f17f...

Ordering hold queue

Name	UID-short
fram	d270df15-d365-4d9f-bfc4-b7b5c...

Figure 7: Debugger hold queues

5 Discussion

The following sections will discuss some of the problems and solutions encountered while implementing *GCom*.

5.1 Casual-Total ordering

The *Total* ordering in our implementation of *GCom* doesn't allow group members to multicast messages without first having piggybacked a correct sequence number to the message. This implies that because our first implementation only had one sending thread there would not exist any case where a message got an a sequence number which was not both total and casual.

To make a different between the total order and the total-casual ordering as defined by the assignment specification we made it possible for the process to use multiple processes to simultaneously send messages. This way there can exist a case where a message on its way to the sequencer will be surpassed by a later message from the same process. That would in a totaly orderd system mean that the messages would be received in the reverse order, but our total-casual sequencer will FIFO order the messages back to their original order before giving them sequence numbers.

5.2 Passing remote objects

Our implementation pass remote objects directly between different processes to enable group members to get a connection to other group members. This could become a problem on systems that are connected to internet behind a firewall. Since the lookup of the remote object only is done once different hosts might not have an open route to a process behind a firewall.

5.3 No automatic failure detection

We have no automatic ”ping-function” in our system. That means that if for example a group member crashed, no one will realize this before they tries to send a message. This also implies that if the leader crashes and no messages is sent, there will be no way to connect to the group.

5.4 Security

Here there are great opportunities for development. For instance, teoreticly anyone could send I GroupChange-message and noone will know it the sender is the leader or not. It is the same way with all other messages types.

6 Scenarios

This section shows different scenarios that explains how the system works.

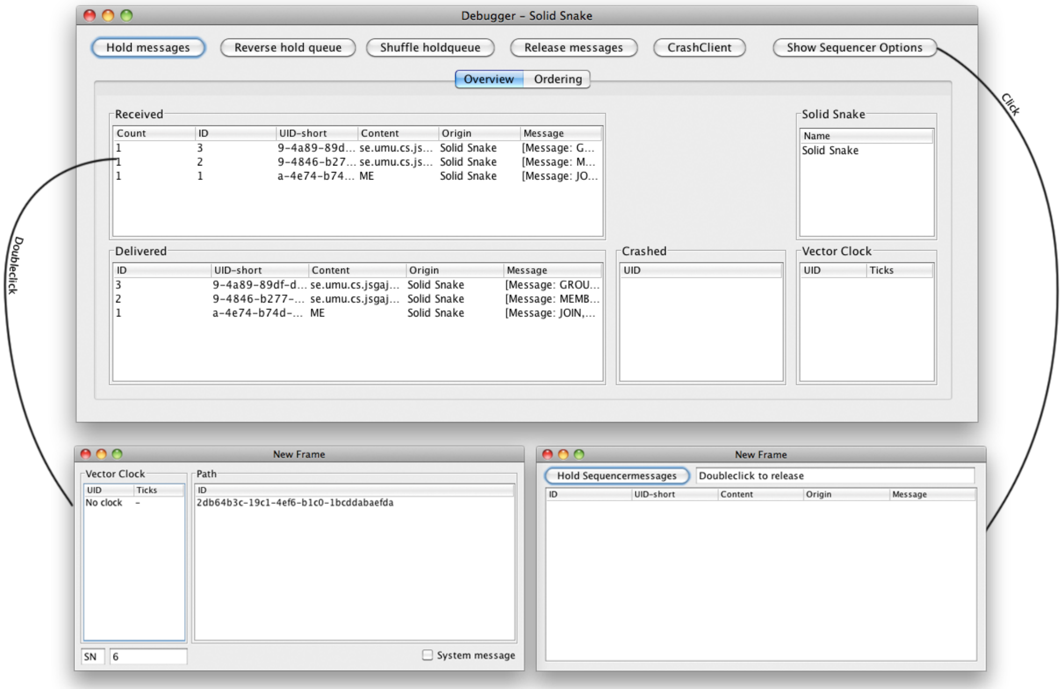
6.1 Create new group

P = The process that wants to create new group.

1. P: Create GroupSettings-object with information about name, leader (the process itself) orderingtype and multicasttype.
2. P: Create a remote object of the GNS.
3. P: Connect to the GNS with the GroupSettings-object.
4. GNS: Saves the GroupSetting-object and then returns it to the process again (with a notice if its new or if the group already existed, in this scenario we presume the group didn't existed).
5. P: Creates a GroupLeaderImpl()-instance.
6. P: Starts the receive- and senderthreads.

6.2 Join existing group

P = The process that wants to join new group. GL = GroupLeader



Received

Count	ID	UID-short	Content	Origin	Message
1	3	9-4a89-89d...	se.umu.cs.js...	Solid Snake	[Message: G...
1	2	9-4846-b27...	se.umu.cs.js...	Solid Snake	[Message: M...
1	1	a-4e74-b74...	ME	Solid Snake	[Message: JO...

Solid Snake

Name
Solid Snake

Delivered

ID	UID-short	Content	Origin	Message
3	9-4a89-89df-d...	se.umu.cs.jsgaj...	Solid Snake	[Message: GROU...
2	9-4846-b277-...	se.umu.cs.jsgaj...	Solid Snake	[Message: MEMB...
1	a-4e74-b74d-...	ME	Solid Snake	[Message: JOIN,...

Crashed

UID

Vector Clock

UID	Ticks

Vector Clock

UID	Ticks
No clock	-

Path

ID
2db64b3c-19c1-4ef6-b1c0-1bcddabaefda

SN 6

Doubleclick to release

ID	UID-short	Content	Origin	Message

Figure 6: Debugger start screen with popups

1. P: Create GroupSettings-object with information about name, leader (the process itself) orderingtype and multicasttype.
2. P: Create a remote object of the GNS.
3. P: Connect to the GNS with the GroupSettings-object.
4. GNS: Discovers that I group with the given name already exist and then returns the already saved group.
5. P: Create a join-message and inserting a remote object referring to itself as message content.
6. P: Creates new GroupView with the leader from the GroupSettings-object.
7. P: Sends the join-message to the OrderingModule where it skips the ordering because it counts as an system message.
8. P: CommunicationModel receives the message from ordering module and then multicasts it.
9. All members: Receives the message.
10. GL: Adds new member to current GroupView-object.
11. GL: Create a GroupChange with the updated GroupView-object as content.
12. GL: Multicasts the message to all members (this message is also a system messages which makes it skip ordering)
13. All members: Receives GroupChange-message and updates thier groups

6.3 Member crash (not leader)

P = The process that discovers crashed member. GL = GroupLeader

1. P: Tries to multicast message to process C and discovers I remote exception because C has crashed.
2. P: Throws MemberCrashException which includes C.
3. P: ManagementModule catches the Exception.
4. P: Creates MemberCrash-messages with C as its content and multicasts it to the group view (this is I system message which means it skips ordering)
5. All members: Receives message.
6. GL: Removes C from it's GroupView.
7. GL: Create a GroupChange with the updated GroupView-object as content.
8. GL: Multicasts the message to all members (this message is also a system messages which makes it skip ordering)
9. All members: Receives GroupChange-message and updates their groups.

Note: Some other member in the group will maybe also discovers that C has crashed so this reaction chain may be executed more times than one.

6.4 Leader crash

P = The process that discovers the crashed leader. GL = GroupLeader

1. P: Tries to multicast message to process C and discovers I remote exception because C has crashed.
2. P: Throws MemberCrashException which includes C.
3. P: ManagementModule catches the Exception.
4. P: Creates MemberCrash-messages with C as its content and multicasts it to the group view (this is I system message which means it skips ordering)
5. All members: Receives message.
6. All members: Discovers that is the leader that is dead, then removes the leader from their group view.
7. All members: Checks if their own MangagementModule has the highest UUID in the GroupView.
8. Member with highest UUID: Notifys the GNS that the group has a new leader.
9. GNS: Updates it's leaderreferens for the group
10. GL (new): Creates a GroupLeaderImpl-instance.
11. GL: Create a GroupChange with the updated GroupView-object as content.
12. GL: Multicasts the message to all members (this message is also a system messages which makes it skip ordering)
13. All members: Receives GroupChange-message and updates their groups

Note: Some other member in the group will maybe also discovers that C has crashed so this reaction chain may be executed more times than one.

7 Tests