UMEÅ UNIVERSITY
Department of Computer Science
Report

October 26, 2009

Deliverable 2

Distributed systems, HT-09

GCom

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

`~/dit06ajn/edu/dist/GCom`

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 and 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.

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 4.4.

Four message-ordering types are implemented: *Non-ordered*, *First in first out*, *Casual*, *Total* and *Casual-Total*, described in greater detail in section 4.3.

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 26, 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*[DKC05] list three important consequences of a distributed system:

- **Concurrency:** Program execution are concurrent. In the case of *GCom*, message receiving and handling are concurrent 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*.

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

The environment in which *GCom* will execute will defined by a model for distributed system called *asynchronous*-system defined by three assumptions [DKC05]:

- 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 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 4.1.

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

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
- 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 4.1) 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/*.

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

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

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 connection 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

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.

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

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 run-time.

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
    }
}
```

4 System description

The *GCom* middleware is separated in three different modules to separate different behaviours, see figure 1 on the following 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.

4.1 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.

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*.

4.2 Group management module

The *group management module* is the top one in the application stack, see figure 1. 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

[8] http://www.slf4j.org/
[9] http://logback.qos.ch/
[10] http://www.junit.org/

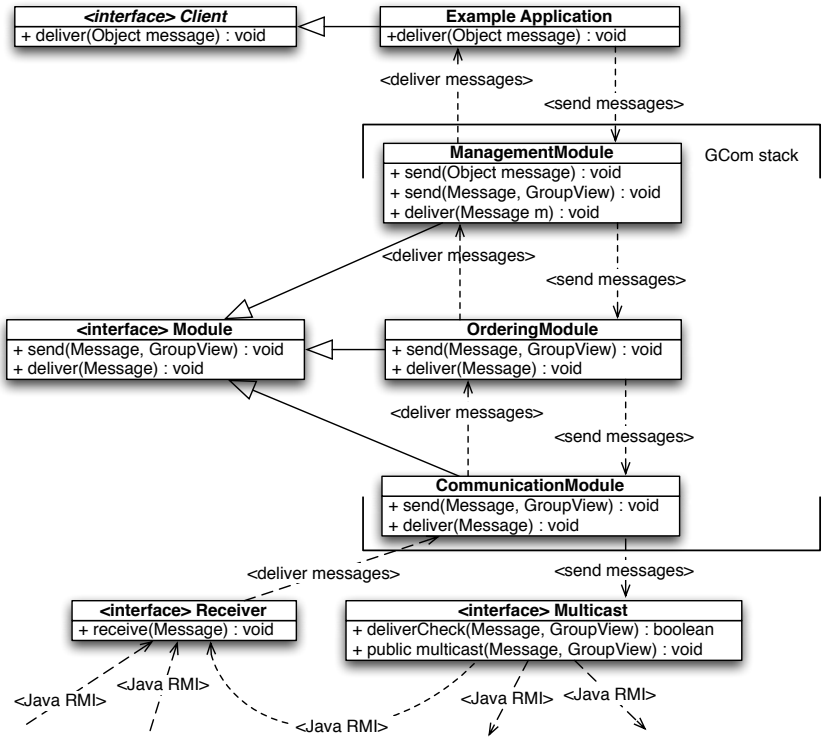

Figure 1: GCom stack

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.

4.2.1 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 reciving 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 4.3.4.

4.2.2 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.

4.3 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

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

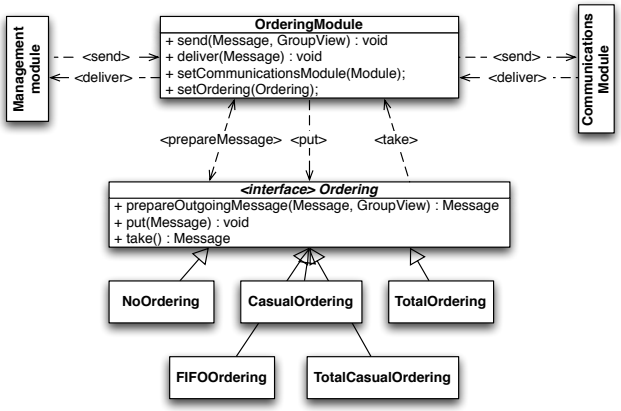

Figure 2: Ordering module

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.

4.3.1 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.

4.3.2 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.

4.3.3 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 [DKC05]: *"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 [DKC05].

When preparing a message for sending every process piggybacks 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.

4.3.4 Total ordering

4.3.5 Causal-Total ordering

4.4 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 3.

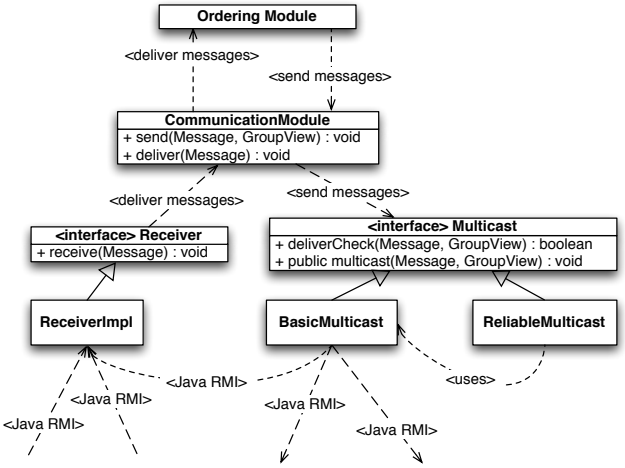

Figure 3: Communications module

4.4.1 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*.

4.4.2 Reliable multicast

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

4.5 Debugger

5 Limitations

6 Tests

6.1 Test protocol

p1	p2	p3 (hold all messages)
ett		
	2	
tre		
	4	
		(Release messages in reverse order)
		2
		4
		ett
		tre

References

[DKC05] Jean Dollimore, Tim Kindberg, and George Coulouris. *Distributed Systems: Concepts and Design (4th Edition) (International Computer Science Series)*. Addison Wesley, May 2005.

A Appendix