Final Report
Integration Project 'Let's talk'

Wouter Bos, Tim Hintzbergen, Gerben Meijer, Tristan de Boer
Group number 2

Moduleproject 3: Let's Talk is an application that supports chatting over a WiFi network without use of an actual WiFi connection or a server. The whole network is peer-to-peer!

Getting started

In order to setup the network, your network card must be set to ad-hoc. The script to do this is located in "Project files/adhoc". Run adhoc_setup in root or sudo. Note that the group number should be changed to your own group number or an random number. Your computer number should also be changed to an unique number.
Now you're all set up! Open the program by running the Main-class!

Simulation setup

Not the room and space or computers to test this entire network? You can run this application multiple times with different computer numbers (and let computers 'ignore' eachother). Add numbers to of nodes you do not want to be neighbours of eachother to the following in line in network/NetworkManager.java:

```
        private final byte[] excluded = new byte[]{};
```
to
```
        private final byte[] excluded = new byte[]{1, 2, X};
```

You can manually set the computer id by uncommenting and changing the number of the following line in network/NetworkManager.java:

```
        //Protocol.CLIENT_ID = X;
```

Requirements

Must

- Packets can follow a path between multiple nodes without looping (passing a node twice)
- A connect can connect to all other nodes that are within his range or within the range of the other nodes
- Nodes should periodical ping showing that they are present in the network
- Nodes should adapt to changes in the network (moving and disconnecting nodes)
- Packets cannot loop the network
- Nodes should be able to send simple chatmessages
- The application does not use a central server, but is using a peer-to-peer network.
- The application supports at least four clients
- Unicast and broadcast should be available

Should

- Message choose the most efficient route
- We make use of a GUI.

Could

- Encryption
- Multicast
- Support for emoticons
- Support for profile images
- Sounds whenever you receive a file
- Sending files such as
    - Text files (.doc, .docx, .txt etc.)
    - Audio files (.mp3, .flac, .wav etc.)
    - Video files (.mp4, .wmv, .avi etc.)

Won't

- Live video chat support
- Teamspeak/Whatsapp/Facebook/Telegram/Mumble/Twitter integration.
- Pirates

Network Layer

The network layer handles everything that has something to do with the network. We can identify the following classes in our network layer:

NetworkManager
The main network class
- Sets up connection to multicast group.
- Initializes PacketHandler.
- Is used to send packets over the network.

IncomingPacketHandler
Handles incoming packets
- Checks if packets were already received.
- Drops cloned(/double) packets.
- Calls NetworkManager when a new (non-cloned) packet has arrived.

OutgoingPacketHandler
Sends packets and ensures that a packet is received correctly.
- After a timeout it should resend a packet.
- Holds a map of all the “floating” packets, packets that are sent, but not yet acknowledged.

Packet
Sets up a basic packet, for construction and reading.
- A packet can be constructed from incoming data.
- A packet can be transformed into a byte[] for sending.
- A Packet should be an easy way to create and read packets.

Protocol
Holds constants used in the Protocol and network.
- Holds Multicast group IP, port.
- May hold various methods for handling packets.

PacketListener
A PacketListener implements a function onReceive and listens to the PacketHandler.
- onReceive is called if a new (non-duplicate) packet comes in.
- onReceive is called with the new packet as parameter.
- onReceive runs on a separate thread.

FloatingPacket
An extension of a Packet, this Packet has a timestamp and is used to ensure that a packet is received correctly on the receiving side.

Class diagram

A class diagram of each is available. Links between some of these packets aren't described in this class diagram. They are described at *communication between network layer and (gui)controller*. The Visual Paradigm files are available in "Project files" in the root of our Java-project, images are available in "Project files/Class-Diagrams".

Protocol

Communication packets

+	0	1	2	3
0	packet type (2 = communication, etc.)	data type (2 = text)	Source (id)	Destination (id)
4	seq	seq	seq	seq
8	flags	data length	data length	next hop
12	data	data	data	data

Number	Packet Types:	Data Types:
1	Discovery (Like distance vector)	ping
2	Communication (Like TCP)	text
3		file

The diagram above describes a packet for text transmission. The packet type should be 2 (indicating it is a communication packet). The data type should also be 2, indicating it contains text. Source and destination both represent the last integer of a host.
The sequence number is client_id * 2^24 + seqnumber. In that way we prevent clients to use the same sequence number when sending a message (no sequence number collisions will occur). The seqnumber starts at 0, and increments every time a new sequence number is needed.

name	**integer**	**byte**
Data	1	1
Ack	2	10
Broadcast	4	100

Flags can be set to identify the contents (or use) of an packet. Setting the flag to 2 means it is an acknowledgement of the sequence number provided in the data of the packet.

The length of the data describes the length of the data (duh), because it can be of a variable length. The next hop describes the node that should forward the message to the destination or to a hop that has a lower cost to the destination.

Discovery packets

Discovery packets are broadcasted to all nodes that are within range and has type 1. The packet contains a table length and some sort of sequence number and the DVR-table of the node.

+	0	1	2	3
0	type (1 = discovery, etc.)	Table length	seq	seq
4	tableData	tableData	tableData	tableData
8	tableData	tableData	tableData	tableData
12	tableData	tableData	tableData	tableData

The Discovery Data Layout (or table data) exists of the following elements:

Destination id	cost/hops required	next hop

In that way, a node that has received a discovery packet can check whether the other node has a shorter (more efficient) route to the desired node.

Distance Vector Routing

On the network layer packets can follow a path without passing a client twice or looping the network. This feature is implemented by the Distance Vector Routing algorithm, which allows us to find the shortest, most efficient route (calculated by hops) between different nodes that are within range of one another node.

Discovery algorithm

The Distance Vector Routing algorithm is called the Discovery algorithm in our implementation. It operates over the lowest network layer available to us, it does not use our TCP like Communication protocol.

Our Discovery algorithm operates mostly like a normal DVR implementation, it holds tables of all known destinations with their cost and a next hop. When a better route to a certain node comes in, it will replace the slower route with this one. The Discovery algorithm treats all links as a link with a cost of one. Packets are flooded over the network, if a node received information that caused a change to its tables it will forward its own tables again. This way the changes will reach all clients in the network, without running in infinite circles.

Dropping tables

When the tables are dropped, the dicoverySequenceNumber is inceased and the tables will be sent. Other clients will receive the packet and detect the higher sequence number, this will make them drop tables and set their sequence number to the one they received. This way a table drop will flood the network

Changes in the network (like disconnects and moving nodes) can be detected and corrected. When a node detects a change in the network (like a disconnected node), tables are dropped and new tables are generated. Besides every 30 seconds all tables are dropped.

The detection of changes is handled by the pings. The network layer keeps a list of connected clients and if for some reason a couple of pings are lost, the tables are dropped and the algorithm will reconverge. After a table drop there will be a pause in the resending of packets, this to give the Discovery algorithm some time to converge. Other packets might be able to send, but will also be stopped if there is no route to the destination.

Exclusions
To test multi hop file transfer, we created the exclusions system. Adding a id to this byte[] in a *NetworkManager* will exclude entries with this id as nextHop to be entered in the table, thus simulating a disconnection.

Improvements and discussion

- The Discovery protocol should have had a cost mechanism. At the start of the project tests indicated that ping would vary no matter the distance and we concluded that a system would work fine when taking every link as a cost of one. It was only on the last day that this was proven to be wrong. This caused file transfers to break over long distances because the more unstable connections without hops were prefered over the more stable multi-hop links.

- The Discovery protocol could have a system that only dropped routes that were affected by a change, thus reducing the time spent reconverging after a loss of connection. This would not make a large difference in a small network, but it would certainly be worth to explore this idea more if the system would have to work over tens of laptops.

The Communication protocol

The Communication protocol is our TCP-like protocol. It uses ACKs and sequence numbers to guarantee the arrival of packets on a working connection. In all but the most extreme conditions the Communication protocol can guarantee to an application that:

- Packets will arrive
- Packets will not arrive twice

Difference to TCP

Other than with normal TCP, ACKs are only acknowledging individual packets. This is due to the fact that we are not having a communication between 2 clients, but between multiple clients. It could still be made like a TCP connection between every client but we chose to not implement this for two reasons. The first reason was that we did not really need it. The only application that could have used it was file transfer, which is now solved at higher layers. The second reason was that the performance would suffer. It would have to wait on certain packets to proceed, stopping the filetransfers and chat even harder than it does right now at a packetloss.

Outgoing traffic

The *OutgoingPacketHandler* handles the sending of outgoing packets. It has a Map of all packets currently “floating” on the network. In the event of an ACK packet coming from another client it will delete this packet from the Map. If the packet has been in there for long enough, a timeout will occur and the packet will be sent again.

We have some sort of reversed Quality of Service for file transfers. File packets will be put in a separate buffer if the number of “floating” packets exceeds the Protocol.FILE_SEND_BUFFER_SIZE. This makes sure that the network is not flooded with file packets, while all other traffic is not delayed. It could be seen as some sort of send window like TCP although its size is static.

The packets have a maximum size dictated by the buffer size: Protocol.RECEIVE_BUFFER_BYTES_SIZE which is 2048 bytes in out application. The header sizes are constant , so the maximum payload of a packet is constant.

Broadcast

A packet is broadcasted if its destination is 0 when passed to the send method on the *OutgoingPacketHandler*. It will then be sent in unicast to all clients currently connected according to the *NetworkManager*. This means that a packet is not guaranteed to arrive at all clients that seem connected via the UI. The UI will display someone as Online if there has been at least one ping in the last 10 seconds, while the *NetworkManager* will drop a client if it has not reacted after *Protocol.MAX_MISSED_PINGROUNDS,* is 4 in our application. This is also the reason that you can’t send files to the lobby. If your client is disconnected from the sender during transfer, it might not send the whole file.

Incoming traffic
The *IncomingPacketHandler* handles all incoming packets. It has different methods for handling Discovery and Communication packets. It will filter packets so only the right packets are forwarded to the higher layers. At first the Discovery and Communication packets will be separated, Discovery will be handled and Communication will be filtered even more.

After deciding that a packet is a Communication packet, it will be filtered further. Now the packets will be split in 3 types:

- Packets directed at this client (*destination* == *Protocol.CLIENT_ID*), these will be filtered even more.
- Packets routed via this client (*nextHop* == *Protocol.CLIENT_ID*), these will be routed further.
- Packets not related to this client, these will be neglected.

A packet directed to this client will most likely be a packet with either the DATA or the ACK flag set (and sometimes the BROADCAST flag too). The DATA packets are normal packets, directed at delivering some form of data like text or file data to this client. They will be ACKed and will be delivered to all *DataListener*s if the packet is not duplicate. All ACKs will be handled in the *OutgoingPacketHandler* and will forward the original packet acknowledged by the ACK to the *AckListener*s.

As can be seen above, other layers can listen to Data and Ack packets by subscribing to the *IncomingPacketHandler* as a listener. This is how information is forwarded to the higher layers.

Duplicates will be ACKed again, but will not be forwarded to the application layer. The duplicates are checked by an ArrayList of the unique identifier of a packet, the so called floatingKey. In short this is the sequence number plus the id of the original sender of the packet. This list holds up to *Protocol.MAX_PACKET_BUFFER_SIZE* packets. In theory this could mean that a packet is not labeled as a duplicate while it is a duplicate. In practice this does not happen.

The Packet class
Because we did not want to create each packet directly as a byte[], we needed a way to create packets with ease and store them in an object oriented matter. We made a class Packet to convert from and to a byte[]. This class has all the features a normal communication packet in our protocol has. The packet class automates some features, like setting the correct data length and thus allows for easy construction in higher layers.

By making seperate construction methods in the NetworkManager, we can easily create DATA, ACK and PING packets. These will hold the correct sequence numbers and other data for easy transmission. The correct way for a higher layer to construct a Packet is via these methods, settings the destination, data and datatype. Ack packets can easily be constructed by entering the corresponding DATA Packet instance in the constructACK.

Ping

Devices ping the network periodically by sending a ping to all clients in the network. This ping contains the name of the client, so a client can broadcast his name. The name of the client will be changed in the GUI.

Pings run on the Communication protocol, meaning that they are guaranteed to arrive if we have a usable connection.

Improvements and discussion

- The *OutgoingPacketHandler* should have been an *AckListener*, but is not because these were implemented long after the *OutgoingPacketHandler.*

- During a file transfer, the clients available in the lobby at the start of transfer should be kept. We could not find a way to implement this easily and decided that this feature should be left out. Being able to force a file to the whole lobby was not a feature we found usefull.

- Our Protocol is not adaptive to slow connections, making it a huge mess on these connections. This could be solved if we had had more time, but for now this won't be fixed.

File Transfer

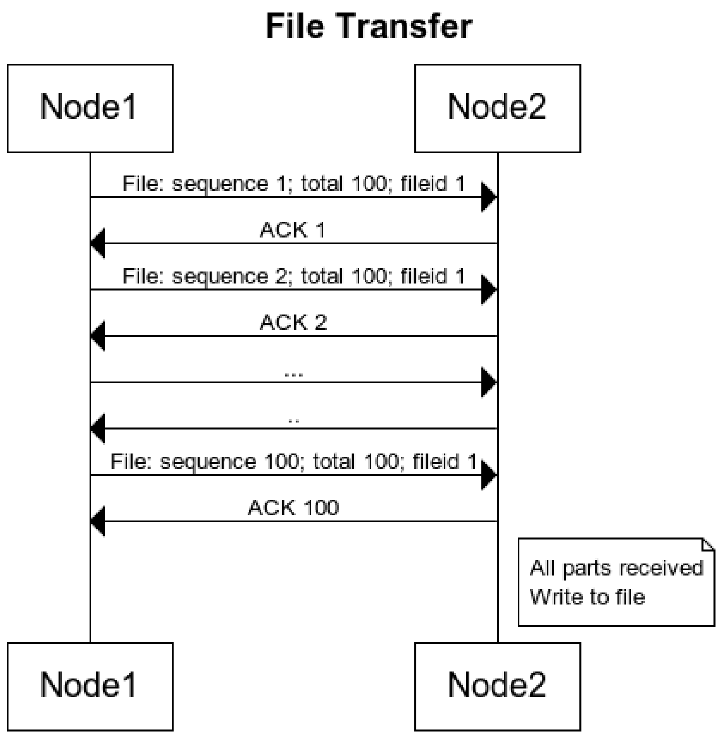
Clients can send simple text messages and files. A single data packets contains the data of a text message or a file.

packet type (2 = communication, etc.)	data type (3 = file)	Source (id)	Destination (id)
seq	seq	seq	seq
flags	data length	data length	next hop
dataseq	dataseq	seqtotal	seqtotal
fileid	file	data	data

The packet that sends the part of a file is a data packet. It contains an extra layer describing the number of the part of data being send, the total amount of data parts to send, the file number and data.

Sending a file
Whenever a clients wants to send a file, the file is read to an bytearray. The byte array is split up to multiple byte arrays. Headers are added to all byte arrays, containing the number of that byte array (sequence number), the total amount of byte arrays to expect and the identifier of the file being sent.

Receiving a file
When a file packet arrives, the headers are checked. The identifier and the source of the packet form a 'key' for the file. When an other file arrives, the key of this packet is compared to the other packets. If the keys are the same, they describe the same file. Together all packets are placed in a sorted map with the sequence number of the byte array as key and the byte array without headers as value.
If the total amount of entries in the map is the same as the total amount of byte arrays to expect the file transfer is complete, thus can be written to a file. The file is named by an appended byte array, containing the filename and extension of the file. The file is written to a new folder in /home/user/Penguin.


File Transfer
Node1
Node2
File: sequence 1; total 100; fileid 1
ACK 1
File: sequence 2; total 100; fileid 1
ACK 2
...
..
File: sequence 100; total 100; fileid 1
ACK 100
All parts received
Write to file
Node1
Node2

The advantage of our protocol is that we do not have to wait for the first packet to transmit, and it doesn't matter in which order the packets are transmitted.
However, some overhead is generated because the id of the file and the total amount of packets is transmitted every time.
Every time a new part of a file arrives, the progress of the file is checked. The progress can be determined by:
(*total_amount_of_arrived_data / total_amount_to_expect) * 100%.*
The protocol doesn't allow us to send bigger files. Our implementation is not really efficient. On top of that, the Java Virtual Machine can get slower because of the files loaded in the virtual memory.

File progress of sending client

Checking the progress of a file works the same as receiving a file. Our implementation sends the packet to all ackListeners whenever a packet has been ack'ed. The client listening to this acks does the same as when a file has been received, however it does not generate a new file, it only displays a progress message in the queue of the node it is sending a file to.

Graphical user interface

Communication between Networklayer and (Gui)controller

IncomingPacketHandler contains two fields: ackListener and dataListener. Whenever a file arrives, all listeners are notified and a packet is sent as a parameter.
When such a notification arrives in the *MessageController*, the method onReceive (or onAck) determines what to do. This decision is made on base of the DataType of a packet (see *Protocol.DataType*).
Protocol.DataType.PING is a ping message, which updates the clients list. The client is added if its ID is not found in the list. If his ID is found, its name, last seen date and route are updated (the nexthop for the client to reach the host that has sent the ping).

Protocol.DataType.TEXT is a text message. Using the source id and client list we find the name of the user that has sent the message. The message is added to the queue of that user , or (when destination is 0, thus broadcast), it is added to the 'Lobby', by creating a new ChatMessage and adding that Chatmessage to a DefaultListModel.
Protocol.Datatype.File is sent to a FileReceiver. This is described in *File Transfer*.

Sending a message

Whenever a user wants to send a message, he enters a message and pushes enter (or the 'send' button). A packet is constructed using NetworkManager and sent to the node with id currentView (the id that describes the queue the user is currently viewing, thus typing a message to). The sent message is also added to the queue.

Sending a file

If a user has selected a file to sent, the path of this file is sent to a FileHandler. For more information see *'File Transfer'*.

JLists/ListCells

A JList can be populated by a DefaultListModel. The entries of this models for the JLists in the Gui are instanceof ChatMessage or Client. These classes (implementing interface Message) allows us to easily store information about the message, the user or the file process (ProcessMessage extends ChatMessage).
Cells for the JList are generated in ListRenderer, a class that implements a ListCellRenderer and uses information of the the entries of a DefaultListModel to generate a ListCell.
Chatmessages and Progressmessages contain a profile picture of the user and the name of the user that has sent the message and the time the message has been sent. ChatMessage also contains the message the user has sent. ProcessMessage contains the progress of the file transfer (represented as a String).
An entry Client contains information about the name of the node, the last time the node was seen, via which node the node is routing (if it isn't directly connected to the node) and whether the user has read all (new) messages of the node.

List of features of the GUI

- Choose your own name (and change your name if you reconnect).
- Enable/disable sound
- Auto scroll
- Select a file and send it to other clients
    - View the progress of a file transfer
    - Multithreaded
- Receive files and view the progress of a transfer
- See if a user is online or how long he has not been seen.
- Send messages to other users or to the broadcast.
- Receive (broadcast)messages
- View the time and date a message and the sender of a file/message.
- 'Poke' other users by sending '!pinguplay'.
- Get visually notified when you get a new message.
- Get notified by a sound message when you get a new message.
- Different icons for different users (we've only added icons for users 1 to 4, more can be added later).
- View message you've sent to other users.

However, some improvements can be made. It would be awesome if a user could set his own profile picture (and it wouldn't be difficult). However, due to limited time we did not want to do this.

Unit Tests

To ensure that the classes work as they should work, we have made a couple of unit test classes with the JUnit4 framework. These unit test are especially focussed on the network-part of our application. A description of the tests can be found in the javadoc, but for clarity we will briefly discuss each separate test class and their coverage, measured with the built-in coverage framework in IntelliJ IDEA. The test classes can be run together in one run, but an exception has to be made for the NetworkManagerTest class because of the close relation with the GUI and the size of this test.
NB: The FileHandlerTest class has not been made to be used in the JUnit4 framework. Therefore it should not be tested in one run together with the other test classes. This class contains furthermore a hardcoded path. To use this test you should change this path!

Exception tests
All exception tests are small tests made for testing the getMessage() method. This to ensure that a proper message is coupled with the right exception.

FileHandlerTest
Tests the FileHandler class. Ensures that the FileHandler handles with files correctly.

PacketTest
Comprehensive test for the Packet class. Covers 21 out of 27 methods and 75 out of 100 code lines. Tests all getters and setters, except the addFlag(), removeFlag and the hasFlag() methods. Furthermore we are aware of the deepCopy(), toString() and the print() methods not beïng covered. We made the choice of not testing these methods, because they are trivial and / or adding another test would cost too many time.

FloatingPacketTest
Small test for the FloatingPacket class. Only the extended methods are beïng tested here.

ProtocolTest
Tests the static method in our Protocol class.

PacketHandlersTest
Because the PacketHandlers are closely related to other classes we have decided to put them in one test. Especially the methods of the IncomingPacketHandler class is beïng tested by this class.

NetworkManagerTest
Last but not least the extensive NetworkManagerTest. Tests 23 out of 24 methods and 112 out of 141 code lines. Because of the fact that this class is crucial for a properly working application there is a massive amount of assertions in the test class. Only the shouldBeExcluded() method is not beïng tested because of the lack of time.

(System) tests

Testing of the GUI

The GUI is tested, however no Unit Tests are available. The tests used were working before the user interface was connected to the NetworkLayer. However, these tests aren't available anymore.
However, by using system tests we could see that every text message that was sent to the GUI was displaying provided that the user had sent a ping message with his name. During the System tests, all errors were fixed and we assume that no further errors occur.

Testing of the FileHandler

In order to test that FileHandler and FileReceiver would work, we created a test that would open a file, split it up in multiple byte arrays, add headers, generate a packet, send that packet to the filereceiver, remove all headers and put all byte arrays in a list in the original order, generate one byte array out of all those arrays and write that bytearray to a file. That file should be the same file as the original file. The test tests if the total length of all byte arrays (minus the added headers) is the same as the original byte array size.

Testing of the NetworkLayer

To test the networklayer, we used a variance of different approaches. The first approach was to just start the NetworkManager and log certain parts of the network to test their functionality. After the GUI was linked to the *NetworkManager* we could test from the chat itself. At first we simply tried chatting to see if our chat functionality worked. These tests still had many print statements to log the different parts at work.

After these stages the testing started focusing more on the file transfer, these stressed the network a lot more and delivered a better representation of the network performance. We performed multiple tests with a chain of laptops, only able to reach their neighbour over a slow connection. In these tests the chat functionality seemed to work quite robust, except for the lobby chat. The file transfer blocked off the whole network and gave a lot of problems.

In the last stages we built in an exclusion system so we could simulate different setups without having to move through half of the Horst building.

In all of these tests we relied on Communication packets being sent over the network seeing if and how fast they would arrive at their destination.

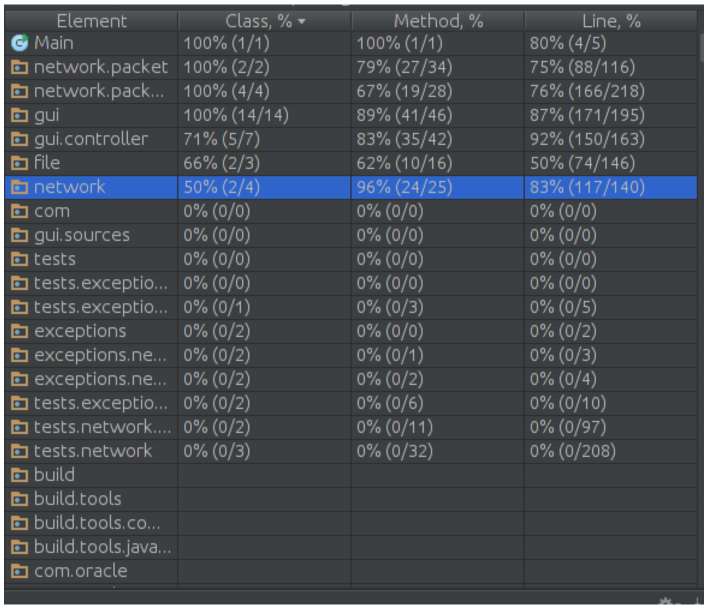
Performance evaluation

We measured performance between two (0 hops), three nodes (1 hop) and four nodes (2 hops). This was done by sending files in a private chat.

Hops	File size (byte)	Average Transfer Time (seconds)	Speed (KiBps)
0	419.961	81	5 KiBps
1	419.961	9	45 KiBps
1	9.810.348	330	29 KiBps
2	419.961	162	2.5 KiBps

We can conclude that (according to our performance) the most efficient node to send a file to is 1 hop away. One of the possible reasons of this could be that our timeout is optimal for a 1 hop file transfer.
When we try to transfer to a 2 hop node, the node often disconnects, so the transfer rate is fairly low.

Coverage



Element	Class, %	Method, %	Line, %
Main	100% (1/1)	100% (1/1)	80% (4/5)
network.packet	100% (2/2)	79% (27/34)	75% (88/116)
network.pack...	100% (4/4)	67% (19/28)	76% (166/218)
gui	100% (14/14)	89% (41/46)	87% (171/195)
gui.controller	71% (5/7)	83% (35/42)	92% (150/163)
file	66% (2/3)	62% (10/16)	50% (74/146)
network	50% (2/4)	96% (24/25)	83% (117/140)
com	0% (0/0)	0% (0/0)	0% (0/0)
gui.sources	0% (0/0)	0% (0/0)	0% (0/0)
tests	0% (0/0)	0% (0/0)	0% (0/0)
tests.exceptio...	0% (0/0)	0% (0/0)	0% (0/0)
tests.exceptio...	0% (0/1)	0% (0/3)	0% (0/5)
exceptions	0% (0/2)	0% (0/0)	0% (0/2)
exceptions.ne...	0% (0/2)	0% (0/1)	0% (0/3)
exceptions.ne...	0% (0/2)	0% (0/2)	0% (0/4)
tests.exceptio...	0% (0/2)	0% (0/6)	0% (0/10)
tests.network....	0% (0/2)	0% (0/11)	0% (0/97)
tests.network	0% (0/3)	0% (0/32)	0% (0/208)
build			
build.tools			
build.tools.co...			
build.tools.java...			
com.oracle			

Our project has got a pretty good coverage. The main reason the file coverage is low, is because not every feature was tested, so we could have a higher coverage. The class coverage of network is low, but this is caused by the two interfaces. This also applies to gui.controller. So we have got a pretty good coverage!

Repository

For further information regarding fixed and open issues and over 400 commits see https://github.com/wouwouwou/project_module_3.

Javadoc

Javadoc is available in the folder "doc" in the root of our Java-project.