Systems Software

Spotibook (network course work)

Christopher Mack (N0576565)

Benjamin Doe (N0561281)

Contents

[Introduction: 3](#_Toc447319571)

[Network protocol: 3](#_Toc447319572)

[Examples: 4](#_Toc447319573)

[Network Handling: 5](#_Toc447319574)

[Network overview: 5](#_Toc447319575)

[Client: 6](#_Toc447319577)

[Server: 6](#_Toc447319578)

[Common: 7](#_Toc447319579)

[Data Structure: 8](#_Toc447319580)

[References: 8](#_Toc447319581)

[Appendix A: 9](#_Toc447319582)

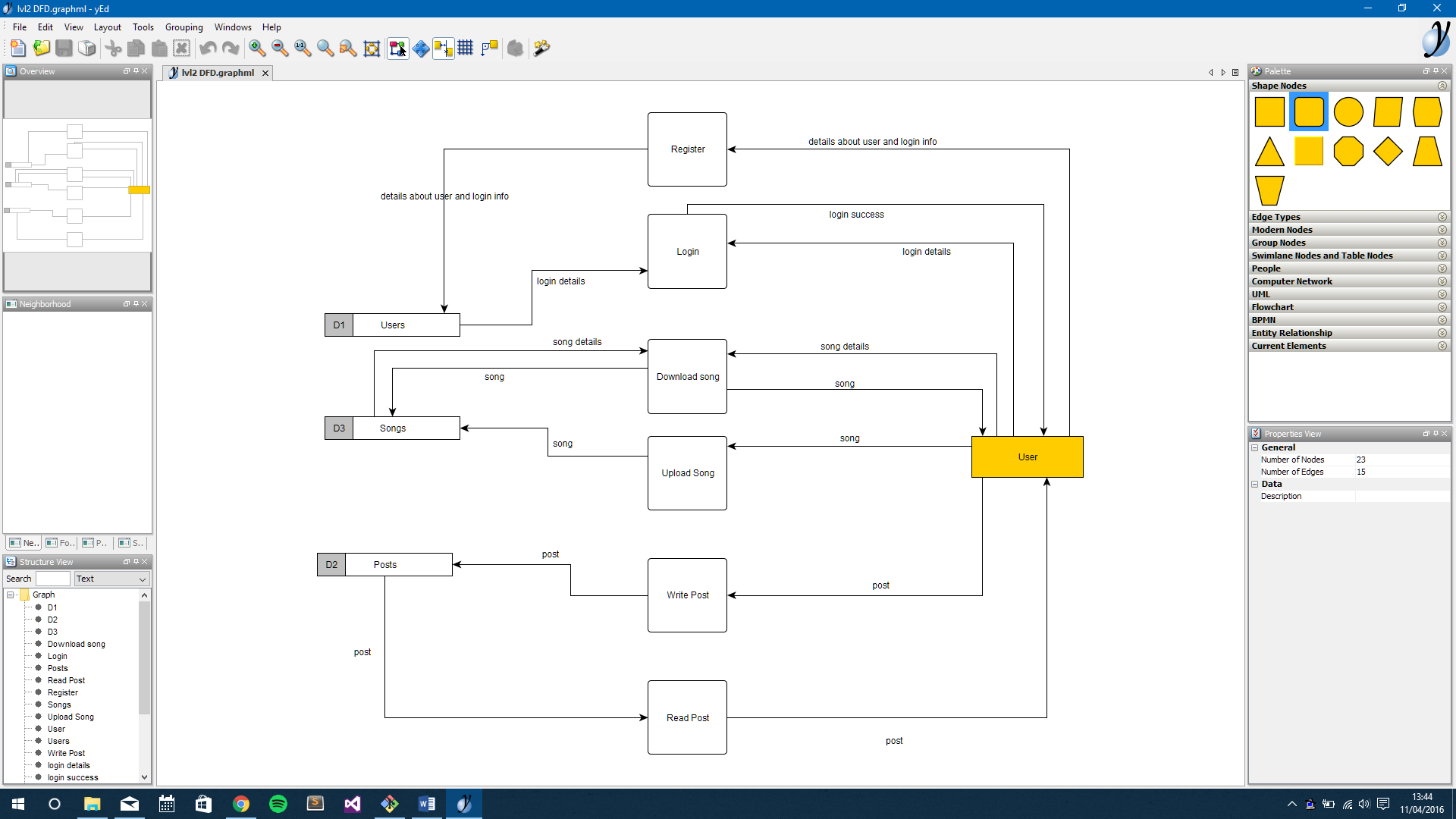
# Introduction:

For this project we we’re asked to create a social network similar to Spotify / Facebook where users would be able to; add friends, post messages, chat to users, and share songs with their friends.

The project had to be built from the ground up using [1] Java using our own network handling to send messages between A client and a server and vice versa. As we are making our own network we had to create our own protocol for the Server and the client to communicate effectively.

A graphical user interface had to be created for the Client and the Server to allow users to easily interact with the program.

## System Design:

Following the specification, the Client would get the data and put data on the database via the server. The server itself would handle all of the communicating with the database. 

The chat server is a separate server to the main server and the client connects both servers running however it can still function if only the main server is running. The client connects to both servers individually however a connection to the main server is required for the client to do things. The chat server is more of a secondary server and is not mandatory for the client to run.

# Security:

The passwords are stored securely on the database using the hash and salt method. The password is sent to the server as raw text; this isn’t the most secure way but if the client is connected securely using a secure protocol with TCP/IP then it shouldn’t matter. After the server receives the raw text, it compares the password to a hashed version of the password that is in the database. The hashing algorithm used is called BCrpyt and is adapted from the Blowfish algorithm. The algorithm has the ability to generate a random salt and the hash can be stored without storing the salt in the database which gives another layer of security to the password. When the user registers, the password is passed into the algorithm and a hash is generated with is then stored into the database for later use to compare a candidate password.

# Network protocol:

The protocol that was implemented has been built upon the TCP/IP protocol to send messages to different system on a network. The protocol used for our application works by using a two-part header system.

Data

Request Header

Routing Header

The first header tells the system where the packet needs to go within the system to be properly handled. As the protocol works on a trust basis it is presumed that the client or server will send messages with the correct headers, if this routing header is not recognised by the system the packet is discarded. Once the packet has then been transferred to the next location the Routing header is discarded.

Request Header

Data

The second part of the header is a specific request for an operation to be carried out. This header defines how the information that follows the header will be processed.

Once this header has been received and properly interpreted it like the last header will be removed leaving just the data.

Data

When the packet has reached its location all of the information will be processed. If data is null, no operation will be performed. Data that is sent is comma separated so arrays of data can be stored in the remaining header.

This protocol also allows for just one Header to be sent, this is when the packet is very simple and has no data following and is just one simple command.

The Structure for packets is as follows <Routing Header>**:**<Request Header>**=**<data**,**data>

## Examples:

Header with data attached:

“LOGIN:FRIENDS=Phil,x,b,Chris,w,c,f,e,g,h”

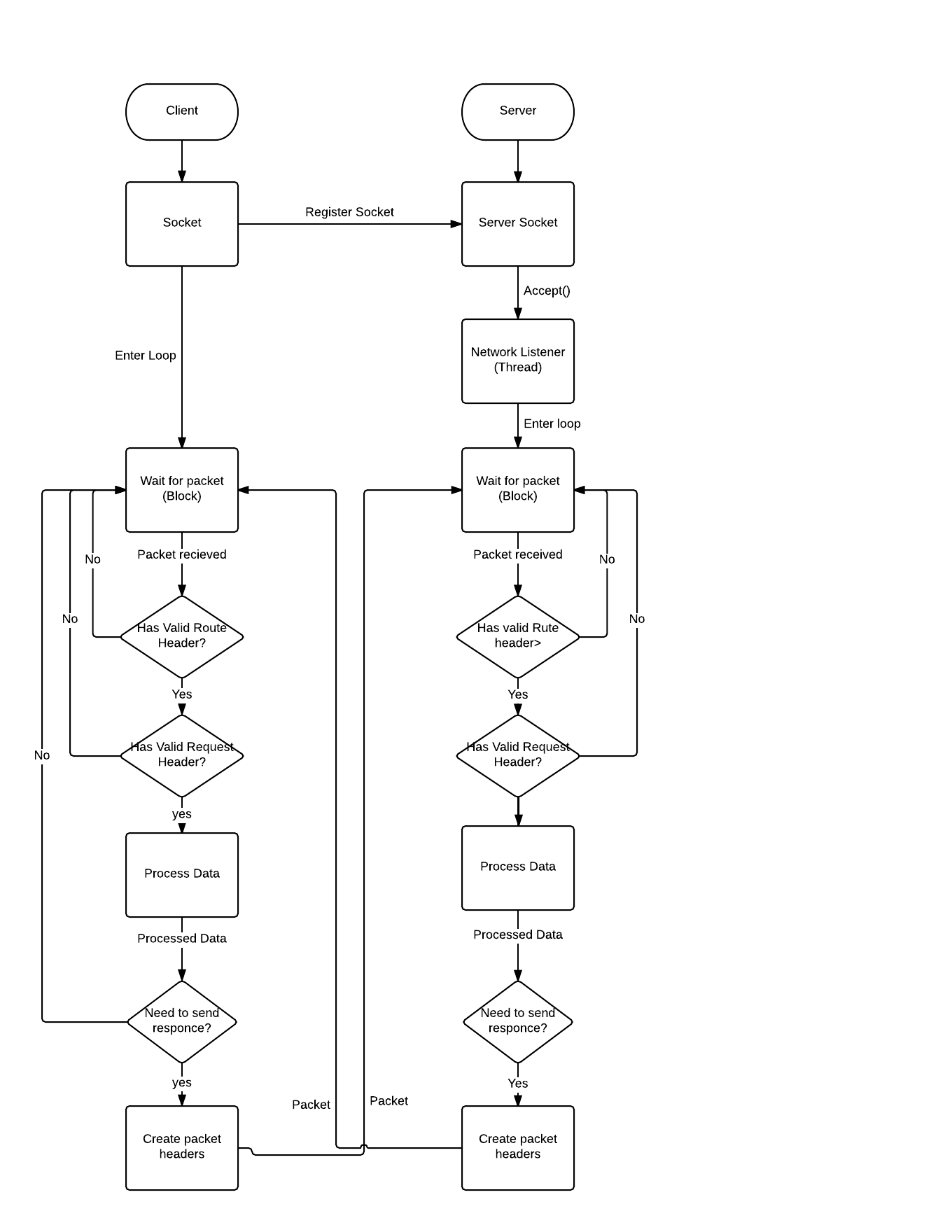
Data less packet:

“LOGOFF:”

(not single header packet still require the ‘:’ at the end to specify the end of the header)

# Network Handling:

## Network overview:



Server is started and starts the Network Handler class

Client is started and starts the Network Handler class

If socket accepted create listener for client’s socket.  
Each socket has its own thread.

Server socket listens for connections

Process information revived

If valid header exists check for valid request header

Send response if processed data requires it

Protocol Checks for valid header

While loop starts listening for data on socket input stream  
Loop waits until new data is received

Client opens new socket with the server using the server address and port

## Client:

The client and the server share much of the same code for handling packets. However as the client is only listening to once socket the network handler only needs on thread listening to the packets sent over the socket. Unlike the server which has to listen to multiple (see Network Handling / Server for more information).

The client handle the Request header and data in each GUI as this is typically where the data needs to be displayed. Each GUI implements INetworkHandler.

Depending on the Route Header that is received the function in INetworkHandler is called in a particular GUI by getting a singleton instance of that GUI class. Each GUI extends GuiScreen which keeps a static instance of the current GUI to make it easy to obtain an instance of the GUI.

The GUI will then check for a valid Request Header, strip it from the string and process the remaining data.

## Server:

As the server has to listen to multiple socket at once a slightly different approach needs to be taken to listen to all connections simultaneously. The network handler class registers Server Socket. This is in a while loop that will continually listen for new connections.

If a new connection is received A new instance of Network Listener is created, this is a threaded class so each new connection has its own thread to listen for incoming messages from the specified socket.

serverSocket = **new** ServerSocket(***PORT***);

**while**(**true**)

{

**try**

{

**while**(**true**)

{

**new** NetworkListener(serverSocket.accept()).start();

}

}

**finally**

{

serverSocket.close();

}

}

This code registers the Server socket and waits for incoming connections and will create a new thread if a new connection is made. serverSocker.accept() passes the clients socket to the network listener.

List the Client to handle the incoming Request Headers and data Classes that are called Processors listen to packets with specific routing headers. In these processors the Request Headers and the data is processed

## Common:

To listen for packet over the socket we listen to the input stream in an infinite loop

‘While ((count = socket.getInputStream.read(bytes)) > 0)’

This line listens to the input stream which contains a blocking function [2], so the loop will only fire when data has been received.

**Count** (Integer) is set to the number of bytes that have been received  
**Bytes** (Byte array) has all of the bytes that have been received in the packet copied into the byte array

If the number of bytes received is more than 0 then the packet is processed

Packet headers are handled by searching through the start of the packet looking for a ‘:’ this is the end of the first header.

message += **new** String(bytes, 0, count);

INetworkMessage messageProcessor = **null**;

**switch** (dataType)

{

**case** "REG":

messageProcessor = **new** RegistrationProcessor(sqlHandler);

**break**;

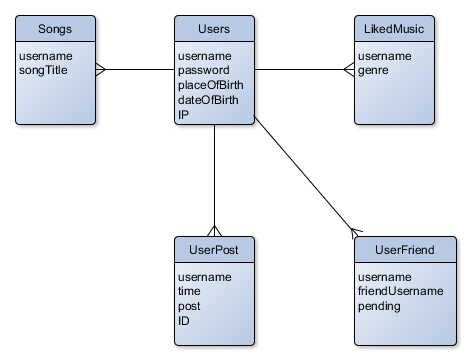
This code snippet is from the servers Network listener class, here it is listening for the routing header and setting the Processor class to the corresponding class.

Once all the data in the packet has been received messageProcessor.processMessage is called and will process the remaining data in the header.

**if**(messageProcessor != **null**)

messageProcessor.processMessage(message, socket);

# Data Structure:



Users : This is where the main information of each user is stored, the primary key is the username as each username has to be unique.

LikeMusic : This is also part of the user’s registration but is in a different table as one User may like many genres of music, the primary key is a compound of username and genre as a user can like multiple genres but one like them once.

This is also used for the searching of users as users can be searched depending on if they like a specific genre.

UsersFriend : This stores all users friends and friend requests, the username is the user that sent the request, the friendUsername is the person that is receiving the friend request and pending is whether it has been accepted by the friendUsername or not. The primary key is a compound of username and friendUsername

UserPost : This is a list of posts that users have made username is the name of the person that made the post, time is the Unix time that the post was made and Post is the post itself, ID is an auto increment value, this is used to easily sort between the different posts. The primary key is a compound of username and ID

Songs : This is a list of all the song files that exist on the server, along with a list of users that have posted the songs. The primary key for this table is a compound of username and songTitle as one user can post many songs but it only needs to be posted once.

# References:

* [1] Download Free Java Software". *Java.com*. N.p., 2016. Web. 1 Apr. 2016.
* [2] Haines, Steven. "Socket Programming For Scalable Systems". *JavaWorld*. N.p., 2016. Web. 2 Apr. 2016.