

School of Science and Technology

Java Server Programming Report

**Barry O’Connor – N0813926 – 2**

**Kieran Olney – N0813911 – 2**

**Nathan Lewin – N0809215 – 2**

**Finlay Craig – N0806748 – 2**

# Introduction

The main objective of the project was to create a TCP based network to enable a farmer to monitor the weather conditions on his fields. This would comprise of several weather sensors which would connect to a central server to deliver their data. This data will be shown through a client GUI (Graphical User Interface) which can be used to log into the server and will be used to display information about each sensor. Once logged in the server will then send all of the data about the field which will include – temperature, barometric pressure, relative humidity and wind force.

You’ll also be able to log into the main server this will allow you to create new workstations and be able to see an up to date list of connections from different weather stations and be able to download their respective data on request.

This network infrastructure will be written in java and will be using client-server model. The application will be built using NetBeans allowing the team to create the required classes and GUI to complete this task.

# Features, design and implementation

The application features a central Server which facilitates the communication between the different parts of the system. The server is implemented using a while loop which constantly checks for connections from the various parts. Once a connection is made, the server spawns a ServerWorker in a separate thread which will then bind with the client component to create a long lasting connection to handle all data. This connection will be between the specific instances of the ServerWorker and the client or sensor, allowing for multiple connections to be handled at once.

Upon connection, each element identifies itself using a predefined command, as a sensor or a client component. A sensor component will register a unique ID as part of this initialisation and will be added to a list of sensors on the server. Once connected the sensor will operate a timed event and will post new randomly generated data every 5 seconds to emulate real world conditions as closely as possible.

Each client application will initially identify but will be made to perform a login action before any data will be transferred. This is handled via a login screen in the client application and uses a text file database on the server to check login details. This component will also register the username of the current user as part of this initialisation and will be added to a list of clients on the server.

As mentioned previously, the sensor will generate random data every 5 seconds and send this data to the server via a command. This data has been researched to be within “normal” parameters for the UK and includes temperature, humidity, wind speed, wind direction, precipitation and snow fall. The team also wanted to include information about the health of the sensor itself and have added in information regarding the battery and storage limits of the device. This may help the farmer identify issues with the batteries or hard drive on the device. Functionality to simulate compressing the hard drive for proof of concept has been included in the software and this could be expanded to include more hard drive management tools.

When a client application has logged in, this user gains access to the data from each sensor, and the application is updated to include a list of sensors automatically. The application uses java listeners to listen out for various events like the selected sensor sending new data, or for the addition of new sensors to the network which will appear automatically for the user.

Upon changing sensors, the application sends a request to the server which forwards to the relevant sensor and asks it to post a new set of data immediately. Once this happens, the data is forwarded through the server and updates the user’s GUI when it arrives.

The client GUI uses a table to display the data as it arrives and some basic bar graphics display the changing values of the data in a graphical format.

As well as the weather sensor information, the client receives up to date information about the field when they have successfully logged in and can view this using the GUI.

The system will be able to automatically display up to date information without the user having to manually request updates. This means that whenever the user connects to the system they will be able to see up to date information about the weather for example they will be able to see the up to date temperature, wind speed or wind direction etc. which means the user will not have to manually request an update from the server to get the most recent information.

# TCP Choice and Implementation

The team chose to use TCP because of the relatively small amount of data that needed to be passed. This allowed for the additional overhead incurred by the TCP protocol implementing handshaking and enabled the use of connection-based communication where each element in the network could have constant 2 way communication with the server. This also allowed for the additional features of TCP such as reliability, acknowledgement of data etc.

It should be mentioned that the system would operate as well using UDP, however the team felt that, for the reasons stated above, there was no need to send large amounts of data quickly and that the additional overhead and reliability TCP offered would be more suitable.

Port “9090” was selected for use within this project and all sockets within the project use this as the port value. This allows the client to be able to link and connect with the server through the port. The server and client bind to the connection and communicate with each other across that connection.

The group has implemented multithreading by using ServerWorker classes. When a connection is made to the server, a new thread containing a ServerWorker is spawned. This handles all communication between the server and the client it is bound to. Both the weather sensors and client GUI’s make use of a ServerWorker and the class accommodates both types of client.

Both types of client employ a thread which simply listens to any messages from the server. This allows the client to continue, since listening to the server commands is a while loop and therefore will stop program flow until it is completed. By moving the listener into a separate thread, we are able to instantly respond to server messages regarding new Data being sent by the sensor or a new sensor registering itself on the system.

The team originally designed the system to work with two types of worker thread instead of one – one for Sensors and one for Users. However the process of identifying which was which took valuable server time and could have led to the server being busy when a new connection was made. Having the server create a new thread instantly and then letting the thread identify the type of unit at its own pace makes for a smoother experience.

## Commands

The communication is handled mainly by the use of commands. These are prefixes to a data stream and allow for the data to be handled in various ways. Each enf of the communication has a function which implements a Switch command depending on what the command is and then performs actions based on those commands.

For example, the command “LOGIN|<username>|<password>” is passed from the user client when a login is attempted. The server receives this command, splits the string and checks the username and password exist in the database. If they do it responds sending “LOGIN SUCCESSFUL”

Another example is the “INIT|<client type>|<id>” command which both the sensor and user clients employ. This introduces the sensor or user to the server, giving a type and an ID to use which allows the server to handle both differently. This information allows basic data to be stored and for the ServerWorker to be able to handle both types of connection.

# Classes and their Roles in the system

The team implemented several core classes to make the system operate, these will be briefly discussed to give a better understanding of how they interact.

## Server

The server class runs the basic loop, waiting for connections from client applications. When a new sensor or client connects, this class spawns a ServerWorker object to contain the communication. A reference to the server class is passed as part of this process so that each ServerWorker only has a single parent. This allows the server to store an array of ServerWorkers connected to Users or Sensors, both to update the server GUI with these but also as a way of communicating since the ServerWorker classes all have a “send()” method, meaning it is possible to broadcast messages or send a message to a specific client.

The server also handles some ancillary functionality – it is able to perform basic admin on users, allowing an admin to add, edit or delete user accounts. An interface for this is provided in the server GUI. This interface also allows for some basic information to be stored about the field, including crop, area, who manages the field and some notes to allow recording of additional information.

## ServerWorker

The ServerWorker class is the backbone of the “Server” part of the client-server relationship. Both types of client will connect and communicate exclusively with a unique ServerWorker object. When a Sensor registers with the server, the corresponding ServerWorker is given the ID of the Sensor and is added to a list of currently connected sensors on the server, this is true of the User also however the ID and instance data is slightly different since a Sensor has GPS etc.

This list can be accessed by a ServerWorker at any point, and using a simple loop to parse the list, it is possible to find a specific instance of a Sensor or a User. This coupled with the use of Java’s Interface or “listener” classes to listen out for specific events, such as a new Sensor registering, or a User choosing to look at a different Sensor.

This is best explained with an example. If there were 10 users registered and active on the system, each would need to be notified when a new sensor is installed so that they can connect to this new sensor and receive the data it generates. To do this, the initialise command from the Sensor will trigger a procedure where the ServerWorker will retrieve the list of other ServerWorkers connected to a User client. Since the ServerWorker uniquely communicates with each client and has a listener for the initialise event, it is simple to send a message through the ServerWorker to the client.

This happens in reverse for a User changing the selected Sensor to monitor. The User client sends a message to its ServerWorker partner which then finds the SensorWorker connected to a Sensor with a matching Sensor ID. The it messages the Sensor to resend data and in a process exactly like the previous paragraph, that data is then passed to the User clients who are listening for a data change from that particular Sensor.

## Sensor

The sensor class is the simplest of the three parts to the system. Objects of this class simply register with the server then proceed to transfer data. Our implementation of this class uses a Timer event to send data every 5 seconds, in reality this short a period would not make sense.

The team tried to emulate a real sensor as closely as we could. Most sensors are microcomputers with an array of sensors, a hard drive for storage, battery for power, and (generally) something like a small solar panel to recharge the battery. Being able to compress the hard drive if it got too full or simply being able to notice that a particular unit is not charging were felt to be valuable features. These were implemented as a way of simulating some possibilities of the system.

## ClientApplication

The “client” side of the client-server relationship is handled by the ClientApplication class. This interfaces with both the Login screen and the main Client Application GUI so that users can have a simple way of interacting with the various components of the system.

The ClientApplication facilitates initial connection with the server however, unlike the Sensor, this is not automatic entry but simply an initial process to allow for the creation of a ServerWorker thread. To gain full communication ability, the user running the application must perform a login using authorised credentials.

Once logged in, the user gains access to the various Sensors and can request a stream of data from a Sensor. Once selected the data will be sent automatically at the given interval and each piece of data is represented using a basic set of bars to display the information. These have been configured to work with the maximum and minimum values within the range of data being generated. Battery charge and free space on the Sensor hard drive are also simulated here and the user has the opportunity to compress a hard drive if the drive becomes too full. This action simply resets the value but could be expanded to implement an archiving system for example.

The ClientApplication class makes heavy use of Java’s Interface classes which allow it to listen out for events such as a new Sensor being added to the system. This allows for the ClientApplication object to respond to these events automatically, allowing for a seamless flow of data without the user having to do much. When a new Sensor is added for example, the GUI automatically is populated with the Sensor ID and all users can access that Sensor immediately.

## Additional Features

One of the extra features implemented is the functionality to allow system to simulate battery life for the weather and also to allow the client to see how full the hard drive is. This can be useful because if the hard drive becomes full then they will not be able to get the most recent weather information. Without a way of viewing both pieces of information, the farmer may never realise there is a problem. In terms of battery, there are many possible causes, ranging from vandalism to a bird nesting directly on the solar panel.

Another extra feature which we have implemented is the ability for the client to administer accounts from the text file database. This is achieved through an interface on the Server GUI.

# Conclusions

The system meets all requirements and with testing can handle many clients of both types reliably.

Sensors can register with the server and upload their data to the server as needed. Users can log in to the system and use the GUI to access information about the connected Sensors on the network. This all happens within a multithreaded environment so many of each type of client.

The project went well and was enjoyable to work on, most of the team have used TCP but have never implemented a system based on it. However, there was a steep learning curve in understanding how to communicate between threads and this caused a lot of confusion. Java’s Static methods caused a lot of issues for the group and understanding how to reference objects within these methods (generally the Main function) took several days to understand.

## Future work

During the course of the project, the team found many areas for improvement of the project with several ideas that would bring value to the farmer.

As mentioned previously, the hard drive free space feature could be used to archive files from the sensors and these could be stored centrally. This would allow for the various sensors to continue for longer periods of time and the information could be backed up regularly in this manner.

If the Sensors were actual microcomputers, it would be possible to store the specific sensor ID’s within the sensor itself. This would mean that every time a sensor disconnects and reconnects, the server itself would know that this was a specific sensor and could use this to append new data to previous data, allowing for less of an issue should the network need to be taken offline.

In a real system, the microcomputers in the sensor would be able to work out averages for each value and could present the data with those averages attached. This was discussed by the team but because our Sensors post data every 5 seconds, there was no consensus on what would constitute a month and we were unsure of how to demonstrate that successfully in a demonstration.

The system as it stands allows for broadcasts. This is in effect how the data reaches the various clients attached to the system. This could easily be expanded to allow for some basic communication between users. This would be of use if more than one user works on a field or if the farmer is very wealthy and has several fields across the country. Notifications could be sent by the farmer or between users to pass on information or just to chat.

Finally it would have been much nicer to have a displayed graph of all of the data as part of the client GUI. However this involved using one of several methods which each had their negative aspects. Java Swing library allows for drawing of 2D objects and several tutorials were available and understandable but this only seemed to allow for one set of data, so while we would have been able to create a single graph for a single set of data, this felt too simple to be worthwhile. jFreeChart is a module which allows for complicated ways of displaying data including scatter charts, pie charts and more. This library involved installation into NetBeans and had several steps that the team felt would cause issues when marking the software as we were unsure whether the additional libraries would be contained within the final software and might cause issues with compilation during the marking process. However, these graphs would have given the software a really great way of displaying the data.