TECHNICAL REPORT

ROBOCUP RESCUE SOFTWARE DESIGN SECTION

CONTENTS

1 Introduction 3

1 Software Development Methodology 4

2 Software Requirements and Specification 5

2.1 Functional Requirements 5

2.2 Non-Functional Requirements 6

3 Software Design 6

3.1 Choice of Architecture 6

3.2 Choice of Programming Language 6

3.2.1 Java 6

3.2.2 C++ 7

3.2.3 C# 7

3.2.4 The Decision 7

3.3 Choice of Operating System 8

3.3.1 Linux 8

3.3.2 Windows 8

3.3.3 The Decision 8

3.4 Safety Considerations 9

3.5 Communications 10

3.5.1 Communications Design 10

3.5.2 Security Considerations 10

3.5.3 Robot ‘Commands’ Library 11

3.6 Server Design 14

3.6.1 UML Model 14

3.6.1 Multi-Threaded Design 15

3.6.2 Control Brokering ... 15
3.6.3 Mapping ... 16
3.6.4 Communications Server Design ... 17
3.6.5 Data Storage and Simulation ... 18
3.6.6 Artificial Intelligence ... 20
3.6.7 Configuration File ... 20
3.6.8 Device Interfaces ... 21
3.7 Safety Controller Design ... 24
3.8 Client Design ... 26
4 Documentation ... 29
5 Software Testing and Validation ... 29
6 Conclusions ... 30
7 Recommendations for Future Work ... 31
8 References & Bibliography ... 33
9 Appendices ... 34
9.1 Appendix A: CD containing Code and JavaDoc Documentation ... 34
9.2 Appendix B: Software Bug Register ... 35
9.3 Appendix C: Safety Test Cases ... 38

1 .0 INTRODUCTION

The purpose of this report is to cover the technical details involved in the software development carried out for the Search & Rescue Robot as part of the Warwick Mobile Robotics (WMR) 4th Year Group Project. It aims to provide sufficient technical detail to allow the reader to undertake further development, fixes or maintenance of the software. It forms part of the documentation for the software.

The Search & Rescue Robot has been designed to enter the RoboCup rescue competition, which aims to simulate the terrain encountered in a building that has been struck by an earthquake. It aims to encourage the development of robots that are capable of traversing the terrain (shown in Figure 1) and finding trapped victims, with obvious real world application. This task can be undertaken autonomously by the robot or by an operator driving tele-operated. Points are scored for locating and identifying victims (with additional multiplier effects if performed autonomously), traversing increasingly difficult terrain and producing a map of the arena which conforms to international standards.

Figure 1: Photograph of the competition arena

This report will cover the methodology used for the development and how it was implemented, the requirements placed on the design of the software, the actual design of the software, technical details involving the implementation of the design.

Being the first in a projected three year development period, the report also details how far the software has progressed against the requirements and suggestions for improvements, which were mostly gained from the experience of entering the competition this year.

2 .0 SOFTWARE DEVELOPMENT METHODOLOGY

Traditional methods of software development use the Software Development Life Cycle (SDLC), commonly referred to as the waterfall technique. This method relies on progressing the project through stages of requirements capture, design, implementation, testing, installation and maintenance. In practice for many software projects this is too inflexible with its defined stages. Requirements may not be fully known up front, software designs can end up proving impractical to implement requiring modification at a later stage. A more appropriate methodology for this project to adopt would be that of Agile development. This will allow the flexibility and adaptability required, where tight timescales and parallel development of the hardware platform require the ability to make rapid modification and feature additions. More specifically the eXtreme Programming (XP) methodology will be used, which focuses on creating the simplest system possible for a given problem and then making lots of frequent improvements or feature additions to build up the level of complexity required. For each feature that is to be added, four stages are identified, design, coding, testing and listening. Each feature addition is tested fully before integration, such that the whole system will work with the new addition. XP is particularly suited to small teams of developers (up to about 12), which is ideal for this project.

Figure 2: The Remotec development/testing platform

In order to facilitate the rapid progress and testing cycle that an Agile methodology requires and permit the software development activity to proceed in parallel to the design and manufacture of a new chassis, a separate testing platform was required. For this purpose, a bomb disposal robot chassis was kindly provided to the team by Remotec, as shown in Figure 2. This platform although physically very different from the intended chassis will allow the mounting of the chosen sensors and control systems (such as speed controllers) to allow proper testing. This should minimise the amount of time required for software integration testing on the new chassis, which is desirable given the tight time scales within the project. The Remotec platform does have a few drawbacks as a testing platform however, in so far as the heights and distances between sensors are different meaning any trigonometric calculations will be wrong. This can be overcome by ensuring that these distances can be easily changed (possibly through use of a configuration file).

To allow easy auditing of the code’s progression and facilitate the development activity of multiple programmers on the same project there is a need for a system to be in place to manage change control on the software code. To assist with this, a software package called Subversion will be used. This package creates a ‘code repository’ from which code can be checked out, modified and then the changes can be updated to the repository. This package will also alert developers when others have changed parts of the code and allow them to update their copies of the code. In this way conflicts

between developers are avoided. Subversion also allows all changes to be tracked, making it possible to 'roll-back' the code to any previous version, as well as recording which changes were made and by whom. To make successful use of Subversion, a work ethic needs to be agreed upon and adhered to with regards to its use. To tie in with this projects use of an 'Agile' development methodology, it was agreed that code should be uploaded (known as committing) into the repository with every cycle of the development process, such as when a new feature has been added and tested or a bug fixed and tested.

3 .0 SOFTWARE REQUIREMENTS AND SPECIFICATION

The RoboCup Rescue Software is primarily concerned with helping to meet the following overall project objectives:

- Build a test environment for the robot
- Produce a sensor array capable of mapping the environment
- Provide support for tele-operation and autonomous navigation
- Investigate the implementation of victim identification using thermal imaging, motion, sound and CO2 sensors
- Compile a handbook detailing the rules and regulations of the challenge as well as the learning experiences of the team for the benefit of the next team
- Compete at the European RoboCup Rescue league within 3 years

This has been broken down into the requirements that the software must fulfill to meet these objectives (system boundaries have been drawn between what requirements the hardware will meet and what is the responsibility of the software). These requirements have been further broken down into functional and non-functional requirements. It should be noted that is not expected for this year's efforts to achieve all these goals.

3.1 FUNCTIONAL REQUIREMENTS

- Software must allow Tele-Operation of the robot
    - Must be capable of being controlled by a single operator
    - Must present sufficient sensory data to allow remote operation
    - Must supply as much information as possible (the more detailed information they supply, the more confident the incident commander will be)
    - Must provide a means of communication between operator station and robot
    - Must allow control of all actuators on the robot
- Software should produce a map (not a focus for this year)
    - Map must be in the GeoTiff map format
    - Map must indicate the locations of each victim
    - Must not have prior knowledge of the arena
- Software should allow autonomous operation in the simpler areas of the arena (not a focus for this year)
    - Should autonomously identify victims
    - Should be capable of navigating through uneven terrain
    - Does not need to be capable of navigating steep slopes, stairs or step fields

3.2 NON-FUNCTIONAL REQUIREMENTS

- Software should ensure the robot is under control (safe) at all times
- Software should be reliable enough to minimise the number of resets that are required during operation
- Software should be secure enough to only permit authorised persons to control the robot
- Sufficient documentation must be in place to permit future teams to expand, fix and maintain the software
- The software should be designed to aid future extension of its functionality

4 .0 SOFTWARE DESIGN

The software design aims to produce a system that will meet all of the requirements. There are various decisions that have had to be made along with the system design. These are discussed in this section.

4.1 CHOICE OF ARCHITECTURE

The obvious choice of architecture for this system is the Client – Server model, whereby one server program, in this case running on the robot, allows multiple client programs to connect to it. This model can be beneficial in so far as several clients can connect, allowing the operator to show the data that is being received to various specialists (such as Doctors) to get second opinions very easily.

In future, it may be beneficial to implement a peer-to-peer topology between the robots, with a single operator (client) controlling multiple robots. This should be considered during the server design as far as possible (without making it a focus) to make any future transition to this form of 'swarm' topology as simple as possible.

4.2 CHOICE OF PROGRAMMING LANGUAGE

Three languages were considered when deciding which language the software should be written in namely Java, C++ and C#.

4.2.1 JAVA

Java is a good cross platform language, meaning that we can run it on a variety of platforms (including some embedded platforms which are capable of running Linux). It also means we can develop on a different platform to the one that the software is deployed to. One of Java's strengths is it has very good support for networking, with very good documentation, which will doubtless prove useful for this project. It is a good high level language which is useful for tackling higher level problems, such as mapping and AI, although is weaker when it comes to interfacing with lower level devices as it doesn't have the open access to the systems resources that a language like C++ does. It has a large user base, with lots of sample code and libraries available to carry out more common (and slightly less common) tasks which could save development time. It does suffer from slightly higher CPU and memory costs

than C++ due to it being partially interpreted through a virtual machine (although this virtual machine is also the source of its strengths).

4.2.2 C++

C++ is a very strong at interfacing with devices on a low level, which will be useful with the variety of sensors and actuators that will be on the robot platform. It is however more complex and difficult to program, particular when it comes to the networking and threading of the program, making it harder to use on the intended scale of this development. One of its benefits is that it is very fast to execute, being completely compiled, though this does make it platform dependent. There is also the OpenCV vision processing library available for C++, written by Intel and used by Stanley in the DARPA grand challenge, this could be a very useful aid to development in that field.

4.2.3 C#

C# is similar to Java, with its ease in developing on a high level, with good support for the networking and threading features that this project will require. It is better at handling event driven design than Java (though Java is also capable of this). There are many packages available which could aid development, such as the MS Robotics Studio. It does however suffer the same (slight) performance problems as Java due to the .NET virtual machine, however is not cross-platform compatible in the same way as Java, since .NET is designed for windows, although there is an emulator available for Linux, its reliability and feature set are unknown. Finally use of the full Microsoft IDE for C# development will incur licensing costs.

4.2.4 THE DECISION

Due to the very complex nature of the task in hand, and the cross platform nature of the development and testing environments, Java has been chosen as the main language for the Robot Server and Client.

For the Safety Controller system, C++ has been chosen. This due to its suitability for embedded systems, where complexity is not required (indeed is to be actively discouraged for safety reasons). There is also scope for the re-use of code created for MiroSot Robot Football which uses the same microcontrollers. It should be noted that C++ is not capable of verification using formal methods for various reasons, one of which being it's use of pointers. This could present a problem for proving it's safety, however this robot is not judged as requiring Safety Integrity Level (SIL) 3 or above, where such formal methods would be required (SIL 3+ is a requirement for systems where potential for large-scale loss of life exists, such as in the aerospace or nuclear industries).

4.3 CHOICE OF OPERATING SYSTEM

There were two operating systems that were considered, both for running the on-board robot server software and the remote client application. Since Java has been chosen as the programming language for the project, with cross-platform compatibility as one of its advantages, this aspect of the choice has been eliminated.

4.3.1 LINUX

Linux is an open source operating system. It has a text based shell which can run a graphical user interface (GUI) on top of it. It can be run without this GUI, which wouldn't be required for the robot's onboard computer, making for very efficient use of the available resources. Additionally, because it is an entirely text based shell, a technology known as secure shell (SSH) makes it possible to operate the computer from across the network in exactly the same capacity as a local user would be able to and in a very bandwidth efficient manner (the traffic is purely text based). Further, the text based shell presents the options for scripts to be written that can make full use of the operating system. Finally being open source means there are no licensing fees to add to the cost of the robot.

The major drawback to using Linux would be the lack of experience with using this operating system amongst the team. In addition, all sensors would need to be compatible with Linux, although this shouldn't be a problem since most of them use a serial interface - which is universal.

4.3.2 WINDOWS

Windows is a commercial operating system (OS) with which everyone on the team is familiar. A lot of the sensors and other hardware are supplied with APIs and demo code to run on Windows, which would speed up the development time.

The major drawbacks of Windows are that its natively a GUI based operating system, meaning that it cannot be run without the GUI engine, which is not required onboard the robot since there is no keyboard, monitor or mouse attached for a local user to use. As a result of this a large amount of system resources are required for the base OS alone. There is no inbuilt remote command interface, although remote desktop would provide a solution, this is incredibly inefficient compared to the Linux secure shell (SSH). Finally, being a commercial OS, the source code is closed meaning that no modifications are possible to the operation of the OS if this is required.

4.3.3 THE DECISION

Due to the resource friendly nature of Linux and it's suitability to being run without a keyboard, monitor or mouse attached to the computer, this has been chosen as the operating system for the robot's onboard computer. For the client software, it has been decided to leave it as a choice for the operator, since the client software is written in Java and will not be platform dependant. As a result of this, it makes sense for the operator to use whatever platform they are most familiar with. In our case this will be Windows.

4.4 SAFETY CONSIDERATIONS

As a mobile robot, safe operation is a large concern, to help ensure safe operation when designing the software, a Failure Modes and Effects Analysis (FMEA) was conducted for the software design. This is a process of identifying the many ways in which the components of a system can fail and analysing the effects that these would have. This analysis was updated every time the design of the software was modified to ensure its continued safe operation (although it also has to account for the impact of hardware failures on the software. The latest such analysis is included in Table 1: Latest FMEA analysis for the robot softwareTable 1 below.

Table 1: Latest FMEA analysis for the robot software

Failure Mode	Effect	Solution
The main robot software crashes	The speed controllers would continue at the present speed, probably resulting in a collision.	Have a secondary PIC microcontroller that monitors the software (via a heartbeat) and stops the robot if the software crashes.
The onboard computer crashes	The speed controllers would continue at the present speed, probably resulting in a collision.	Have a secondary PIC microcontroller that monitors the software (via a heartbeat) and stops the robot if the computer crashes.
The serial cable to the speed controllers becomes disconnected.	The speed controllers would continue at the present speed, probably resulting in a collision.	Use the heartbeat feature on the speed controllers to ensure that the serial cable remains connected.
The network communications drops out.	The server wouldn't change the speed controllers from their present speed, probably resulting in a collision.	Implement a heartbeat between the client and server to ensure the client is still connected while it's in control
The USB controller on the client becomes disconnected during operation.	No change of speed will be sent, resulting in the current speed being maintained, probably resulting in a collision.	Monitor whether the USB is connected or not and issue an E-Stop when it is disconnected.
The sticks on the USB controller are inadvertently pushed while the client is in control of the robot.	The robot would move in an unexpected manner, with a risk of collision	Implement a dead-mans handle on the control that must be pressed before any controls are sent.
The safety controller is rendered in-operable.	Could the robot be rendered operable due to a safety controller failure?	Safety controller has been designed such that the robot will only be capable of operation if it is fully functional.

To ensure that these measures are effective, a series of test cases has been developed to test each of these failure modes, which is included as appendix C. These tests will be run every time the robot is to undertake a live run under remote operation after a software modification has taken place.

4.5 COMMUNICATIONS

4.5.1 COMMUNICATIONS DESIGN

For the communications design, it has been decided to use the TCP/IP protocol suite, since this is independent of the physical transport layer and as such allows the physical communications system to be modified, without needing to change the software. Also, it may allow use of existing network infrastructure to communicate, for example allowing remote control or monitoring of the robot from any computer connected to the network or even the internet (provided the available bandwidth is high enough). This brings about the possibility to simultaneously relay in real-time any gathered information from a disaster situation to specialists in various fields (such as doctors, structural engineers, rescue workers, disaster managers). Additionally, since these protocols are designed for networking, their use will make the program capable of easy expansion onto a more network type topology, with multiple computers capable of connecting to one robot, multiple robots or even possibly multiple robots communicating together as a swarm.

4.5.2 SECURITY CONSIDERATIONS

Security has been a strong consideration during the development of the software. Since the communications channel will be wireless, and therefore publically accessible, there exists the possibility of unauthorised parties attempting to gain control of the robot remotely. This poses risks with regards to safety, as well as our competitiveness. To combat this risk, a two-stage approach has been taken to ensure user verification, before the robot will respond to any communications.

The first stage prevents the on-board robot computer from being contacted across the network, by having a firewall which blocks virtually all communications and running intrusion detection software to detect attempts at circumventing this. With these systems in place, the only way to communicate with the on-board computer is by using a Secure Shell (SSH) Tunnel.

SSH Tunnels make use of asymmetric cryptography to provide a secure means of communicating across an insecure network such as the internet or the robots wireless network. Asymmetric encryption involves the use of two dissimilar keys, known as public and private keys. Anything that is encrypted with the public key can only be decrypted by the private key and vice-versa. This allows for secure private communications by encryption using the public key and for authentication by the encryption of a message digest (unique summary of the message) of the communication using the private key of the client. This prevents man-in-the-middle attacks through packet insertion (since the private key is not known and therefore the message signature would be incorrect) or packet tampering (since the message digest would be rendered inaccurate, making the message signature incorrect).

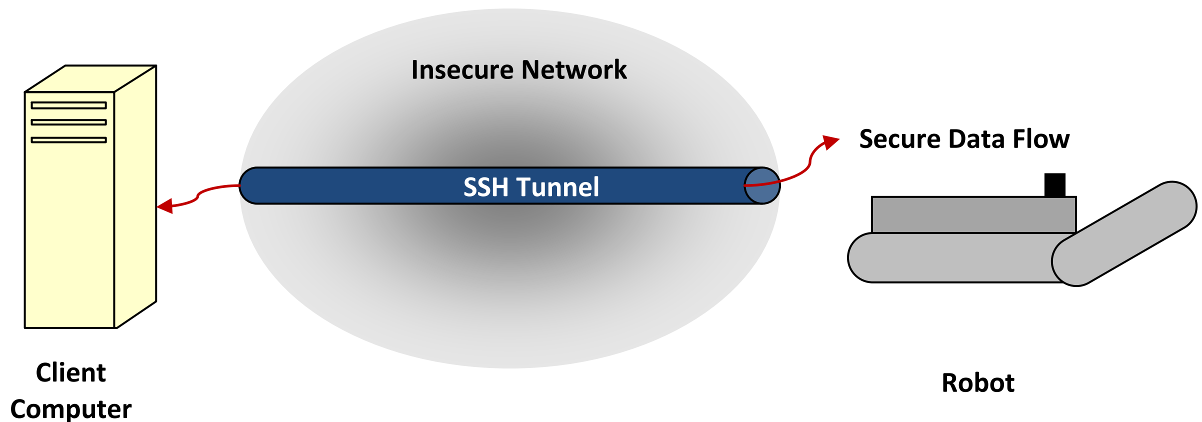


Figure 3: Representation of an SSH Tunnel

This technique could be achieved within the software itself, doing away with the need for SSH tunnels. This would however be a waste of time due to duplicated effort. Additionally, the user base for SSH (using OpenSSH) is vast. It is included with several major operating systems (including all Linux systems, Mac OS X and several BSD distributions) as well as in various hardware devices like firewalls and routers (from vendors such as Novell, Cisco, Nokia and Sun). The benefit of such a large user base and of being open source is that it has been open to widespread inspection and testing and as such confidence in its security and performance is very high, especially when compared to that of a custom implementation.

The second stage is user authentication carried out by the robot software itself. This consists of username and password pairs, which are compared to a database of users in the software. This approach allows different levels of access to be assigned to different users, allowing the potential for 'guest' access to be granted, allowing a user to be given access to the readout from the sensors but not to control the robot. This access mechanism is discussed further, later in the report.

4.5.3 ROBOT 'COMMANDS' LIBRARY

In order to control the robot, a group of commands will be required which cause different actions to be undertaken. These commands should be strictly validated to ensure that they don't contain any bad data which could cause erratic behaviour. One option for passing these commands would be to agree on a set of text based instructions that are sent as characters over the network stream (as would be used in telnet applications) and then carry out validation on these when they are received. A better method (and the one that was adopted) is to make use of Java's serialisation abilities, whereby an instance of a class can be serialised to a byte stream and sent over any form of I/O stream (including network streams). The instance of the class can then be reconstructed the other end. In this manner the validation on the commands is carried out on their creation at the client (the class cannot be created without all the required variables), and the data that they contain cannot be modified after creation, just read and acted on. Another benefit of this method is that there is scope to include much more

complex data types (such as image data, or multiple variables of different types) within these command objects, which are capable of self-describing the data they contain, making the processing of the received data much easier. One drawback to this method is that it increases the amount of data to be transmitted, putting greater strain on the network communications, however this increased load is marginal compared to the amount of data being transmitted. For this to work, a number of 'command' objects needs to be created and included with both the client and server programs. To achieve this, a new 'RobotLibrary' was created, containing all the commands, which are shown in Figure 4 and included in the class path for both projects.

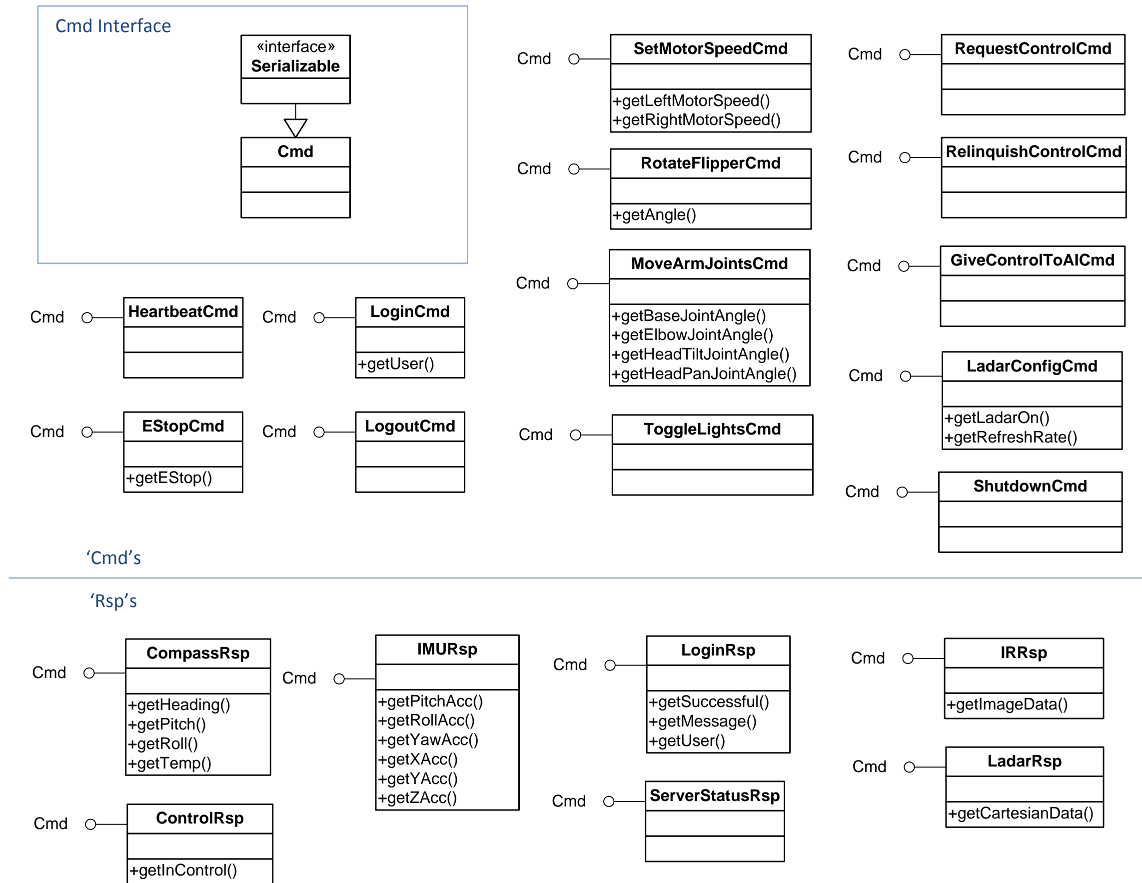


Figure 4: UML representation of the RobotLibrary

As can be seen, a base interface 'Cmd' was created, which all classes in the 'RobotLibrary' implement, although at the minute all the 'Cmd' interface does is extend the 'Serializable' interface, it provides scope for enforcing other methods that might be necessary across all command objects in the future, through the editing of just one interface. The commands themselves are split into 'Cmd's (commands) and 'Rsp's (responses), where the commands are sent from the client to the server and the responses are sent from the server to the client, this distinction is made clear at the end of the class name, although they all implement the same Cmd interface.

With this technique, the compiled 'RobotLibrary' project is included with the compiled versions of each of the other programs. This can lead to issues when edits are made to any of the command objects and

only one of the programs is recompiled with the new library. This is because the client and server will end up trying to communicate using different objects, causing errors to occur (the object that has been changed will not be recognised when it passes between programs). The solution to this is to include a version number with each object, allowing the programs to recognise that the problem is being caused by different library versions, producing a much more meaningful error message that allows the problem to be fixed a lot quicker. The code included in each class for this ID number is;

```
public static final long serialVersionUID = 0;
```

4.6 SERVER DESIGN

This section talks about the design of the software that is run on the robot's onboard computer, referred to in a client/server topology as the server software.

4.6.1 UML MODEL

To help design the onboard robot software, UML was used, since this is a good method for object orientated design. To help break the problem down into manageable chunks, a high level design was produced using packages (displayed on the model as a folder icon) to represent the different functional areas of the software. These packages will then be broken down later to produce a more full design. This high level design can be seen in Figure 5.

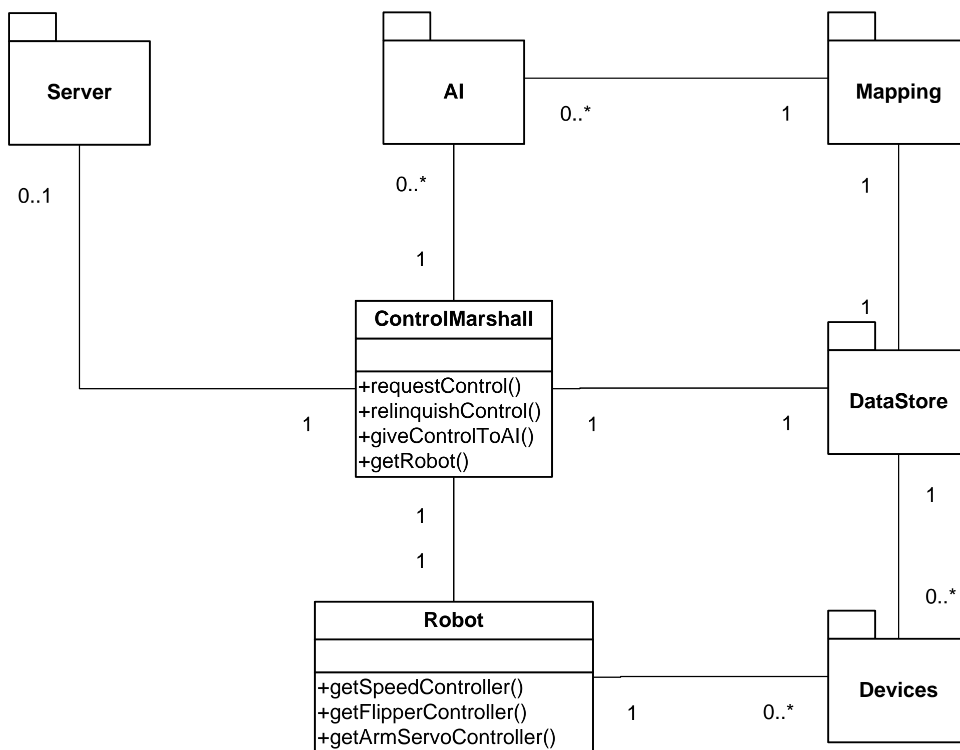


Figure 5: High level UML model of the Server design

As can be seen, there is a base 'Robot' class, to which is linked a number of 'Devices', this package will include all the actuators and sensors on the robot, such as the motor speed controllers, LADAR and SONAR sensors and so on. Each of these will update the 'DataStore' with any new information as it becomes available. The 'ControlMarshall' class provides access to the 'DataStore' and the 'Robot' (and indirectly the Devices linked to the Robot). Any 'clients' then sit on top of this 'ControlMarshall' class (the purpose of which, though fairly self explanatory will be covered in a later section), which includes an AI or a communications server to link to remote operators. Multiple AIs or communications servers and clients can be handled in this configuration. The 'Mapping' package will require data from the 'DataStore' for its operation. It will need to allow access to its results to any AIs that are in the system.

Should the processing requirements for this software become too great for one computer to fill, there is potential to adapt this model to make the AI and Mapping packages sit on different computers, acting as remote clients on the same server that is used by remote operators (in software terms the two are relatively indistinguishable). This would however increase network traffic and server load, which is undesirable unless absolutely necessary, though these additional computers would be local to the robot and capable of being connected on high speed cable network links.

4.6.1 MULTI-THREADED DESIGN

The software has been designed to make use of threading. Threading is a process of allowing parallel execution of different parts of the software, otherwise known as threads. This parallel execution can be achieved both in reality, by actually running the separate threads on the separate cores of a multi-core processor (as is used by the robot) and by virtual multi-threading implemented on each processor core, whereby successive clock cycles are given to each of the different threads in turn, giving a good approximation to parallel execution.

The use of threading is useful to allow the regular polling of devices and is made widespread use of within this software. This has the added benefit of ensuring that the program is capable of making best use of the processing power available to it.

4.6.2 CONTROL BROKERING

The software is capable of having multiple controlling 'clients' connected to it simultaneously, whether artificial intelligence or remote operators. It is essential for there to only be one entity actually in control of the robot at any one point in time, otherwise conflicting commands from two parties will result in erratic unsafe behaviour. Additionally it is necessary to ensure that the robot is made safe (stopped from moving) for the period when control is being passed from one entity to another.

The 'ControlMarshall' class has been developed specifically to solve this problem. It contains a reference to an entity which is currently in control of the robot, only this entity will be allowed access to the robot and its attached devices (through which control is achieved). Any entity attempting to control the robot without being in control will receive a 'NotInControlException' which can be caught and acted upon. There are various actions available to clients, request control, relinquish control and pass control to the AI. Only 'human' entities can request control, which will be given if no-one is currently controlling the robot, or the requesting entity has a higher privilege level (see section about 'Communications Server Design') than the one currently in control ('human' operators are placed higher than 'AI' operators). When control has been removed from a client, they will receive a 'NotInControlRsp' response from the server to notify them of this (as well as receiving exceptions if further control attempts are made).

4.6.3 MAPPING

This academic year the focus of WMR has been on creating a tele-operated robot capable of international competition. Despite this, future expandability has been considered at every stage of the design process. For this project to successfully continue in subsequent years, teams will have to pay significant consideration on creating a versatile mapping solution.

For the rescue robot to exhibit true autonomy and fulfill some of the more advanced goals of the competition, it must be able to localize its position, retrace its steps and produce a detailed map of its environment including the locations of any survivors.

This is a highly advanced topic; no one single solution exists to solve this challenge. Future teams will have to investigate mathematical studies, cutting edge research as well as the efforts of the WIMRC work in this field.

The current team has investigated the area and selected some topics that would prove advantageous as a starting point for future teams.

The first topic is that of SLAM (an acronym for Simultaneous Localization and Mapping). This is an issue for all mobile robots, who have to carry out localization, the process of identifying one's position within an environment and mapping which involves working out the environment around oneself. When tackled separately, for example building a map when you know your position, or localizing yourself when you know what the environment is, is a fairly simple task, however when combined they become very difficult to solve, this is SLAM. Many consider the solution to this problem to be the pre-requisite for "true autonomy".

SLAM uses probabilistic techniques to tackle the problem of noisy signals generated from real world data sensing units. A combination of active, passive, relative and absolute sensors work alongside each other and the outputs are processed together. SLAM can utilize various different methods to process this information, one of which can involve a Kalman filter. The goal is to provide an estimation of the current location state in a dynamic environment from a sequence of noisy data.

This is a highly advanced topic, with significant research being conducted worldwide. One of the more reported examples of SLAM in action is in the annual DARPA challenge competitions. The theory underpins the cars navigation systems, allowing them to map, localize and traverse a course in real time, solely based on the inputs from the on-board sensors.

SLAM is essentially the overall concept that future teams should be striving to implement over the next number of years.

Since this is quite a wide field, some starting point suggestions include researching Hausdorff distance and image correlation to start building up a map from successive LADAR scans. When a map is produced, it will need to conform to the GeoTiff mapping format (although this should be simple enough in comparison).

4.6.4 COMMUNICATIONS SERVER DESIGN

The communications server is responsible for handling all communications to remotely connected clients. It needs to allow multiple clients to connect to it simultaneously. The design for the server is shown in Figure 6 below.

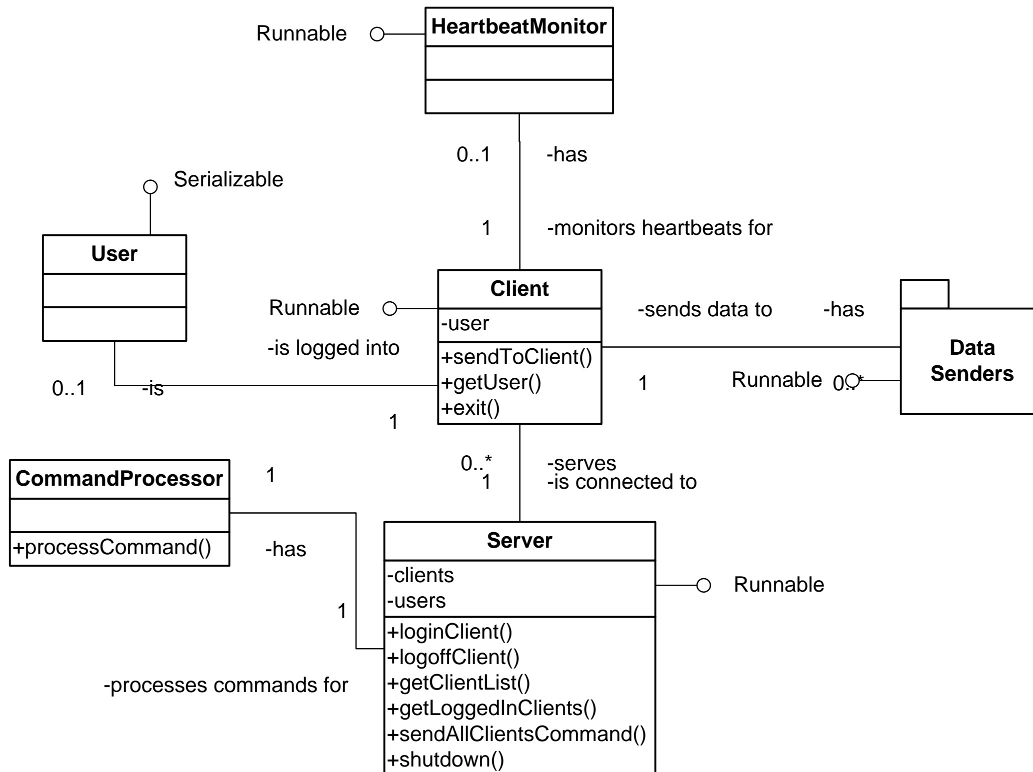


Figure 6: Communications Server UML model

The base class for this implementation is the 'Server' class, which listens for incoming connections on a specified port and sits on its own thread (by implementing the 'Runnable' interface). When a remote client makes a connection to the port, the server creates a new socket connection specifically for that client; it then creates a new thread to deal with that specific client, which is an instance of the 'Client' class after which it goes back to listening for other remote clients to connect. Each of these client threads then handles the two way communications with a single client, constantly polling the socket to listen for incoming communications and providing a method to allow sending of data back to the client. As previously mentioned, the communications are achieved by serializing of 'command' objects over the network streams. When a command is received, it is passed through the server to the 'CommandProcessor', which then interprets it and acts accordingly. In order to ensure that the client is connected when it is in control of the robot, a 'HeartBeatMonitor' thread is also spawned which checks that a heartbeat has been received from the client at least once a second; otherwise it will relinquish the client's control, setting the robot into a safe mode.

In order for sensor data to be sent back, a variety of data sender classes have been created. The major benefit of having multiple sender classes as opposed to a single sender class which sends back everything at once is that it allows different sensors to have different refresh rates, allowing more control over consumption of network resources and the ability to prioritise critical sensors whilst slowing down or completely disabling superfluous sensory information (also useful if one of the sensors malfunctions to reduce the network traffic).

As previously mentioned, there is a need for any clients that connect remotely to be authenticated. To do this, a 'User' class was created, which is serializable, allowing it to be stored to file. The server contains a list of these 'User' objects, which its stores to file to allow persistence of data between sessions. When a client attempts to login, the username and password that provide is compared to the list, if one of the matches, then the client session becomes associated with that user, otherwise the login attempt fails. For security, the passwords themselves are not stored; instead a message digest of the password is stored. A message digest is a unique one-way conversion from which it is impossible to get back to the original password. A password can be compared to the one that is stored by taking the message digest of the password and comparing it to the one that is stored (each message digest is unique and repeatable). This means that if it were possible for someone to get hold of the stored 'User' objects, the clear-text passwords would remain secure and it would still not be possible to login as one of the stored users.

Whenever a command is passed to the 'CommandProcessor', the user object associated with the 'Client' that received it is passed with it. The 'CommandProcessor' then uses the privilege level of the user to determine whether or not the command can be executed. If there is no user associated with the 'Client' (i.e. they haven't logged in yet) then the only command that is permitted is the 'LoginCmd'.

There are three privilege levels that have been implemented, Admin, Controller and Observer. Observers can see all sensor information, but may not control the robot. Controllers may control the robot as well as viewing the sensor data. Admin can also control the robot but may also make fundamental changes to the robot's systems (such as adding new users, or modifying the configuration file). These levels provide a lot of flexibility in real world situations where many people might be involved in the rescue operation, however in practice the only one that has been used so far is Admin (though all of them have been implemented) as the competition only requires one operator.

4.6.5 DATA STORAGE AND SIMULATION

A Data Store class has been created, which is used to store all the data that the robot generates, this means that any classes which control the robot, such as remote clients or the on-board A.I. program do not need direct access to the classes which control the devices on the robot, allowing access to the data, but not to any control features (allowing the Control Marshall class mentioned in the previous section to distribute control without any method of bypassing it). The Data Store allows classes to register 'listeners' on it, which 'listen' for changes to any data in the store (or to specific areas of the data store), this means that classes don't continuously have to poll the data store to watch for updates, instead they

are notified by the store when fresh data is available, increasing the efficiency and the ease of integration of new components (such as the AI) into the software.

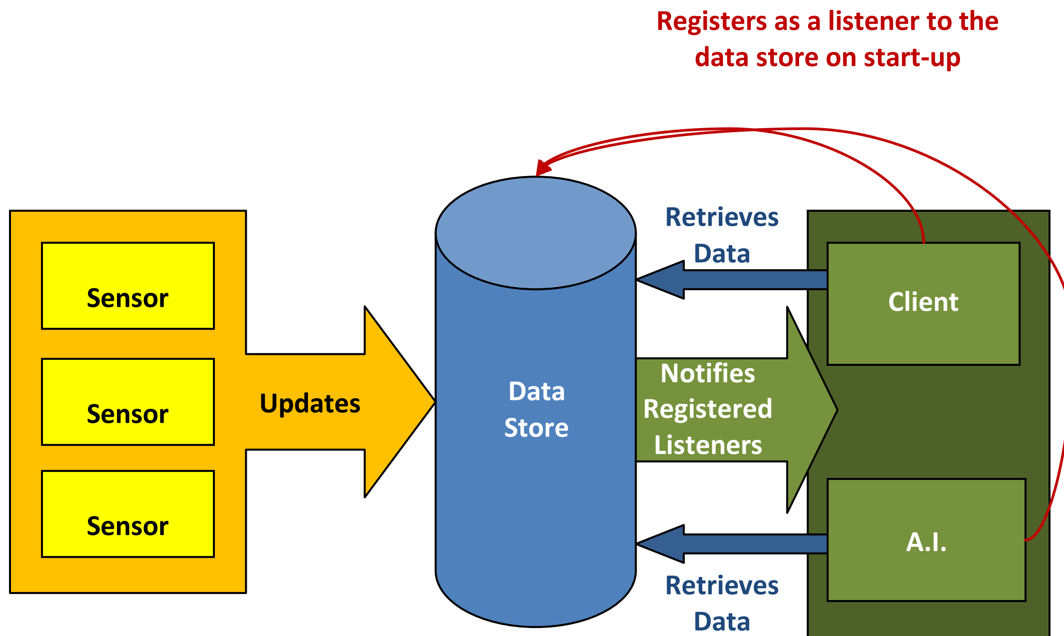


Figure 7: Diagram showing the interactions with the Data Store

The Data Store can be run as a thread, which if recording is desired will write the complete data store to file at a particular frequency, complete with timestamps, allowing all the sensory data gathered about a particular test run (such as our first attempt in the competition) to be stored. This recording functionality has been complemented with the functionality to be capable of re-playing or simulating a pre-recorded run, which will re-load all the recorded sensor readings back into the Data Store at the appropriate frequency, effectively simulating the recorded session for anything that listens to the Data Store for information to run on, such as the AI. This allows for easier development and testing of such components.

Finally the data store concept allows for better control over data synchronisation issues which occur due to the large number of threads running in the program, whereby one thread is attempting to read a value, whilst another thread is attempting to write to it, resulting in incorrect data being read. This is solved by the use of 'lock' objects in the data store, which only allow one thread to access a particular object at a time.

4.6.6 ARTIFICIAL INTELLIGENCE

The development of the Artificial Intelligence (AI) for the robot is beyond the scope of this year's work, though it is an essential part of the design for competition success and therefore the work on all the other sections of the software design has had compatibility with an AI in mind. It has been aimed to allow the AI to be a standalone project (much like the strategy is for the robot football software, as this model has proved successful in allowing successive teams to work on it). There is currently a proposal being put forward to Computer Science to create a masters group project focused on this objective, this is only possible because of the clear boundaries between this system from the rest of the software.

4.6.7 CONFIGURATION FILE

Within the robot server system, there are a lot of variables that are desirable to be able to change with ease. These range from things such as the serial ports particular devices are attached to, the port number that the server should listen to for remote client connections, the maximum speed at which remote clients or the A.I. are allowed to drive at and much more. To facilitate this, it was decided to create a configuration file for the software, which is essentially a text file and can be edited as such. This configuration file is parsed on start-up and stored in the software as a HashMap of key-value pairs.

Figure 8: The Robot Server Configuration File

The format of the configuration file is a common one used in open-source software, whereby the variable name appears first on the line, separated from its value by a space. A '#' character may be placed at the beginning of a line to denote a comment, which is ignored by the parser.

This method makes it very easy to add new variables into the configuration file which is achieved by simply adding another line, containing a variable name that has not been used before and a value. This will then automatically be made available within the program. This approach could however cause problems if certain key variables are not found within the configuration file. For this reason, immediately after parsing of the file, validation occurs to check that all necessary variables have been set, if this is not the case, then a custom 'InvalidConfigurationFile' error is thrown, allowing the user to be alerted to the problem and ceasing the program from running in a safe way.

4.6.8 DEVICE INTERFACES

This section describes the fundamentals of the interfaces with all the devices (both sensors and actuators on the robot). For full details, the relevant code should be consulted (include on the CD in Appendix A), which is fully commented to answer any questions. The majority of these devices use serial communications, so the first stage in interfacing to these devices was to create a generic 'SerialDevice' class, with methods useful for serial communications (such as connect() and send()). This generic base class is then extended by any other device which uses serial.

4.6.8.1 LADAR

The data that is retrieved from the LADAR device is received as a series of 681 polar coordinates covering a field of view of 212 degrees. The full range of the LADAR actually covers 240 degrees (and gives back 769 coordinates), however the first and last 14 degrees of this range are a dead zone where readings are returned from the inside of the devices' case. The LADAR works on a request system, whereby you send it a command, the string "G04472501" in this case, and it will then send back the points. This request is made by the software at a frequency of 10Hz.

4.6.8.2 SONAR

The SONAR devices are analogue sensors that return a voltage proportional to the distance that is detected. The interface to these sensors is through the Phidgets I/O board. The Phidgets device library provide a listener that will fire an event whenever any of the inputs changes, this makes it very easy to read the SONAR devices.

4.6.8.3 MAIN TRACK SPEED CONTROLLER

The Main Tracks are controlled by a dual channel speed controller, one track per channel. This controller takes commands over the serial port. These commands consist of exactly four characters, the first being an '!' to indicate the start of a command. The next character indicates which channel the command is destined for, either 'a' or 'b'. The next two characters of the command give the value of speed to set the motors to, given in hexadecimal; these can take a value from 0 to 127. The channel indicator can be upper case or lower case, where lower case indicates the next two characters are negative and upper case positive, giving a range from -127 to 127. A method in the 'MainTrackSpeedController' class converts a speed from -100% to 100% and a track name into an appropriately formatted command and sends it to the device, allowing for easier visualisation of the process.

4.6.8.4 FLIPPER SPEED CONTROLLER

The flipper speed controller is the same model as the main track speed controller and operates using the same style o f commands. The main difference is that the hardware in the speed controller is configured to give positional feedback, through the connection of a rotary encoder to the end of the flipper arm

shaft that is wired into the feedback terminal of the controller. In this setup, instead of the range -127 to 127 presenting a speed to run at, it instead represents a position (from -180 to 180 degrees) to hold the motor at.

4.6.8.5 VISION CAMERAS

The vision cameras are Axis IP webcams. The imagery from this is not routed through the robot server software; instead the client software grabs the imagery direct from the cameras themselves. Each camera hosts a static image of the current view in the directory '/jpg/1/image.jpg', the client simply navigates to this directory and embeds that image in it imagery window. This image is updated at 15 Hz (dependent upon network loading), giving the video feed.

4.6.8.6 THERMAL IMAGING CAMERA (IR)

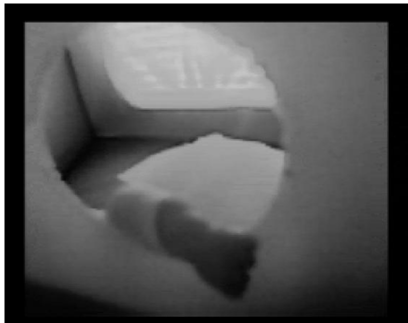


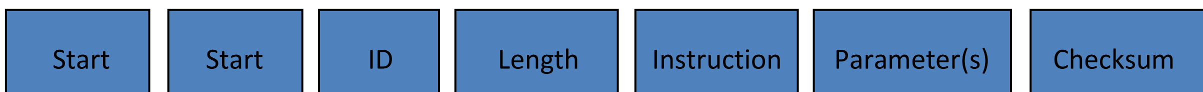
Figure 9: An Image from the IR Camera

The IR Camera is connected to the computer through a TV capture card. It was necessary to implement a 'frame grabber' to get images from this video. To speed up development, a standalone C++ program was created which catches the image and makes it available on a network port. This program re-used a lot of code from previous MiroSot projects. This image (shown in Figure 9) is then read into the server program, which in turn transmits to any connected clients.

4.6.8.7 ROBOT ARM

The purpose of the robot arm is to allow the IR camera and forward facing vision camera to be manipulated into positions that provide a better viewpoint of either a victim or an obstacle within the competition. The arm itself is powered by five servo motors, two powering the base joint (for increased torque), one on the elbow joint and two on the head joint, providing pan and tilt motion.

Although no overall kinematics of the robot arm was implemented, the control systems for the servo motors that control each joint were implemented. The servos operate on a single four-wire bus. Commands are sent over this bus to control the servos. Each servo has a unique ID, by which it can be addressed. The actual control of the servos is achieved by writing data to a control table on the servo, which is then read and acted upon. The format of the commands that are to be sent over the bus is:



The command packet starts off with a start byte (value of 255) issued twice. This is then followed by a byte giving the ID of the servo for which the command is intended. The next byte gives the length of the remainder of the command packet and is equal to the number of parameter bytes plus two. The next byte gives the instruction, which can be 1-6 in value, relating to the following operations, ping (1), read data (2), write data (3), reg write (4), action (5) and reset (6). The next bytes give the parameters for this operation, in the case of writing data (to set the position of the servo), the first byte is an address, and the next bytes are the data to be written starting at this address (and moving to the next address for the next byte of data to be written). Finally a checksum byte is added to ensure the integrity of the command; if this is incorrect the servo will not act on the command. The checksum is calculated as the bitwise NOT operation conducted on the sum of the value of the ID, length, instruction and all parameter bytes, if this sum is larger than 255, then the remainder over that number is used as the sum.

To move most of the joints, a 2 byte position value is written to address 30 in the control table of the servo using instruction 3 (write data). For the two base servos however, there is a synchronisation issue, since they are acting on the same joint, but in opposite directions. To control these, instruction 4 (reg write) is used, which writes the data to the servo, but the servo doesn't act upon it until an action command (instruction number 5) is sent. Having used instruction 4 to write the appropriate data to both servos, a multicast command (using an ID of 254) is sent containing an 'action' command, which causes both servos to move simultaneously.

4.6.8.8 INERTIAL MEASUREMENT UNIT (IMU)

The IMU operates on a request/send basis. It sends a string of data whenever it receives a request for it. In our application, this device is polled as quickly as possible. The received data string is tab delimited, with the values in the following order; accelerations in x, y, z, rotational accelerations about x, y, z and finally the battery voltage (the unit can be run off its own battery, though we don't use this feature).

4.6.8.9 COMPASS

The compass constantly sends out its data as a string over the serial port. This string conforms to the NMEA standard output for heading data, with the heading being given first, followed by the pitch, roll and temperature. Since a standards based compass is used, this device can be swapped for any other NMEA based compass without a need to change the software.

4.6.8.10 LIGHTING

The lighting on/off relay was wired into the auxiliary output on the first speed controller (the same one the main tracks are run with). It is controlled using a two character string sent over serial. '!C' will turn the lights on and '!c' will turn the lights off.

4.7 SAFETY CONTROLLER DESIGN

The safety controller software is to be run on a separate PIC microprocessor. This microcontroller has two outputs which trigger two different kinds of emergency stop (E-Stop). The first kind is referred to as a 'Soft E-Stop' as it is purely software controlled, meaning that the power supply for the actuators on the robot is cut by a relay when the output is pulled low, such that the robot can no longer move and cause damage, however this can be reset by the safety controller by reasserting the output to be high. This is desirable since it provides a means for the robot to continue with its work if the problem can be overcome remotely. The second kind of E-Stop, known as a 'Hard E-Stop' is triggered in a similar manner, however can only be reset by pushing the physical reset button on top of the robot; this is identical in effect to pushing the E-Stop button on top of the robot. This type of E-Stop will have to be reset in software as well as with the physical button if it was triggered by software.

As a Safety system, it is desirable to limit the complexity of the program to the bare minimum, since this will minimise the potential for errors to occur in its implementation. It is for this reason that two microcontrollers were included in the electronic design of the chassis, one purely for safety and the other to handle the switching, power cycling and power management of the many devices. The functionality of the safety controller has been limited such that it is at minimum. It must accept commands over the serial port, it must poll the computer for a heartbeat to check it is still active and it must provide capability for the two types of E-Stops to be forcibly triggered by the computer, and finally it must allow for the E-Stop state to be reset. As another measure, the safety controller side is programmed by a different programmer to the safety side of the main server software. This should reduce systematic errors and is recommended practice for backup safety systems.

There are four commands that can be sent to the safety controller by the computer:

- '**H**' – Is the character for a heartbeat, this must be received at least every 100ms, otherwise the safety controller will carry out a soft E-Stop of the robot
- '**S**' – Is the character for forcing a soft E-Stop, this will be enacted within 100ms of the receipt of this command
- '**E**' – Is the character for forcing a hard E-Stop, this will be enacted within 100ms of the receipt of the command
- '**R**' – Is the character for resetting the safety controller from an E-Stop state. If the E-Stop was a hard E-Stop, then the physical reset button will also need to be pushed.

The safety controller will start up in the soft E-Stop state, such that the PC will need to reset it before any motion can occur, in this way, the safety controller can make sure that the program has been started before enabling any actuators. Figure 10 shows the desired logic flow for the safety controller, the data flows have been displayed in green.

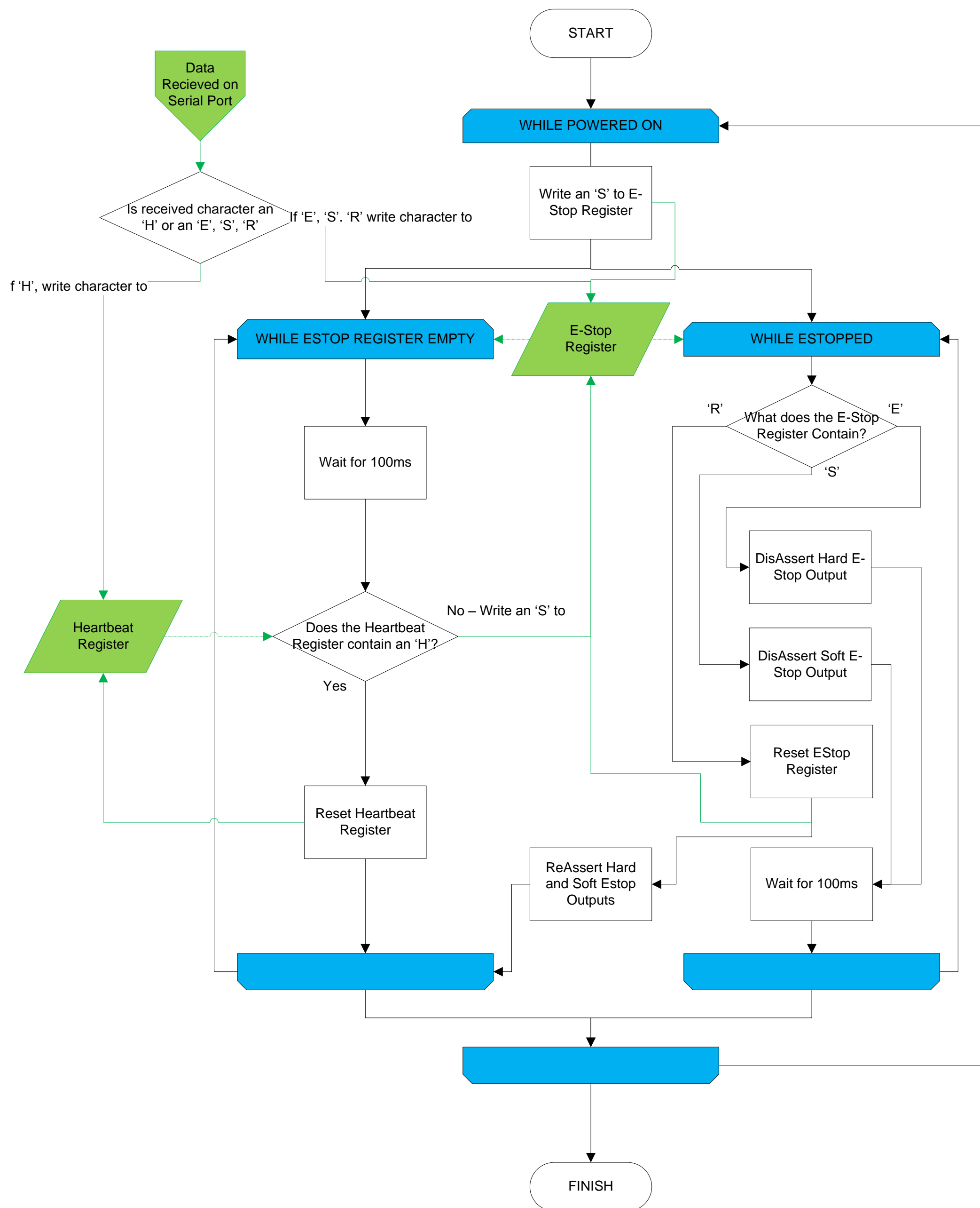


Figure 10: Program Flowchart for the Safety Controller

As can be seen, there are two loops that the program will sit in, depending upon whether it is in an E-Stopped state or not. If it is not, then every 100ms it will check that a heart beat has been received, if not, then it will carry out an E-Stop and switch to the other loop. If a heartbeat has been received, then the heartbeat register is cleared ready to detect the next heartbeat. If the program is in the E-Stopped loop, then every 100ms it will check whether a reset command has been received and makes sure that the E-Stop outputs are grounded. If a reset signal is received, then the E-Stop register is cleared, the E-Stop outputs are reasserted and program control moves back to the other loop.

4.8 CLIENT DESIGN

The client program is intended to be run by the person controlling the robot. Its primary objectives are to transmit the controls from the controller to the robot in a safe, timely manner and to present the available data from the robot's various sensors in the best way possible to aid the operator in their task.

To assist in the design of the client program, a UML (Unified Modelling Language) diagram was drawn up to display the intended design. This process enabled the developers to get a clear understanding of how the system will fit together, allowing for better collaboration. This diagram is shown in Figure 11 (on the next page).

The design has been split into two logical parts, the client part, which deals with the actual network communications, data storage and interpretation of the USB controller device and the GUI (Graphical User Interface) part which deals with displaying the information from the client and providing controls to control the client's behaviour.

The client part of this program makes extensive use of events and listeners to convey information to the GUI part of the program. There are many event/listener combinations which allow any part of the clients behaviour to be acted upon, such as when the client connects, logs in successfully, logs out, disconnects, data store is updated, the client gains or loses control of the robot or when the USB controller is connected or disconnected. The GUI is then managed by registering listeners for these various events and acting when they occur. The Client and DataStore classes have various public methods that can be called by the GUI to ether get data or manage the connection. Every class in the GUI part of the program has a direct reference to the Client object (and therefore implicitly the DataStore).

Any responses that are received by the client from the server are passed to the response processor, which interprets the responses and either writes any data contained within into the DataStore or fires an appropriate event. As previously mentioned in the design of the server system, whenever a remote client is in control of the robot a heartbeat has to be received by the server at least every second. To facilitate this, the 'HeartBeatSender' class sends a heartbeat every 400ms whenever the client is in control. This allows for one of the packets to be dropped each second.

In the UML model, the boxes represent classes, an arrow off a class represents an event that can be fired, and a circle with a line represents an interface that the class implements (often a listener for an event being fired). The lines connecting the classes represent the links between classes with the numbers on the line showing the relationship (i.e. 1 – 1 means a 1 to 1 relationship, whereas 1 – 0...* means one to none or many, as is the case with the vision camera window relationship). The folder icon is used to denote a sub-package of classes in the model, which indicates a collection of classes rather than one single class.

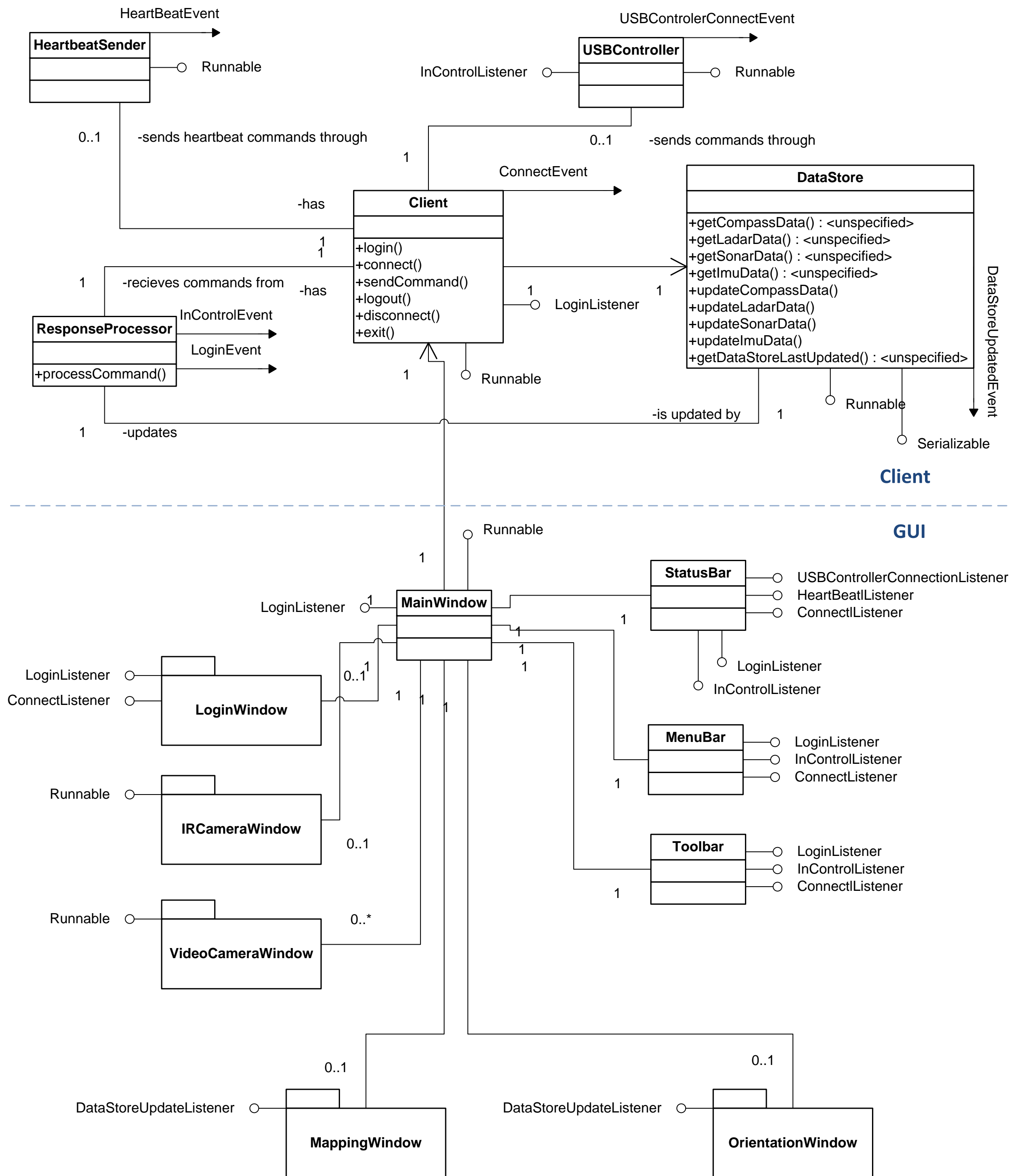


Figure 11: UML Model for the design of the Client

The result of this is the client window shown in Figure 12. It has been designed to provide a customizable view of the available data, allowing the operator to pick the various screens that they wish to use and display them in the size and position that they want.

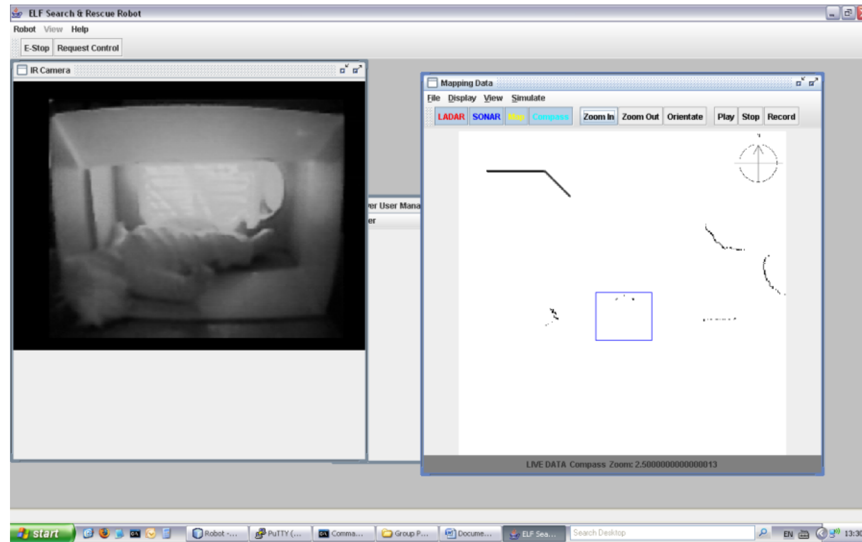


Figure 12: Client window displaying the mapping and IR camera windows

A toolbar has been added to the top of the window to allow quick access to important controls and a status bar has been added to the bottom of the window to allow important information (such as the connection status or whether the client is currently in control of the robot) to be displayed .

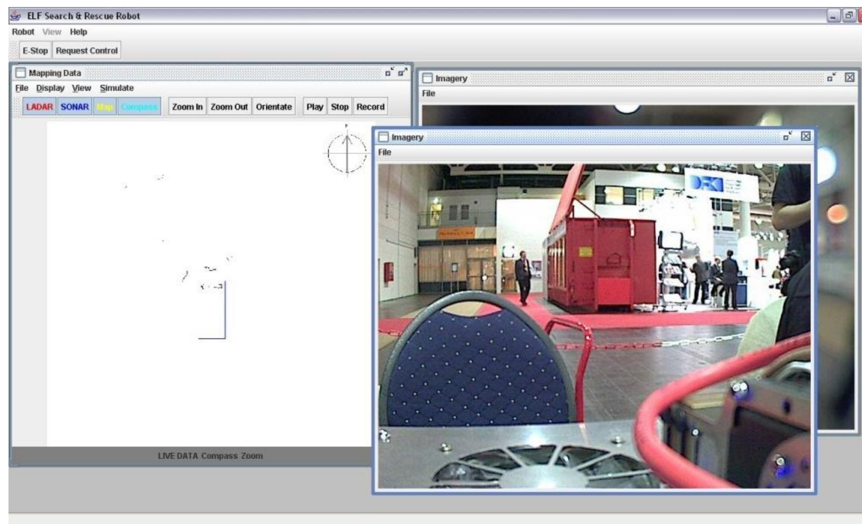


Figure 13: Client window displaying the mapping and vision camera windows

Figure 12 and Figure 13 show two of the possible configurations that are available with the client program.

5 .0 DOCUMENTATION

Documentation is a key requirement for this software as the development is expected to take place, the transfer of knowledge is vital. To this end the following measures have been taken:

1. High Level Documentation has been produced about the overall design (this report)
2. Class Documentation has been kept by way of the production of JavaDoc's (included on the CD as Appendix A)
3. Detailed commenting of the source code allows developer to follow the in-depth program logic (source code can be viewed on the CD included as Appendix A).
4. The Bug Register contains details of all the problems that are known with the current code (See Appendix B)

6 .0 SOFTWARE TESTING AND VALIDATION

Continuous testing is a major part of the development process; therefore there is high confidence in the correct operation of the parts that have been fully developed so far. To keep tracks of any problems in the current code release, a 'bug' register has been kept (see Appendix B). As this is the first year of a projected three year project lifespan, it is not possible to carry out system wide tests and integration testing. For safety, a series of test cases has been produced (see Appendix C); these test the most severe failure modes of the software to ensure it is safe to run. These tests have been conducted regularly after any changes to the software before any live run of the robot (such as before the competition).

Figure 14: The robot tackling the most difficult obstacle in the arena

In the competition the software proved to be sufficient to allow the robot to reach the most difficult part of the arena successfully in a tele-operated mode (shown in Figure 14). Although it was sufficient for the task, there are a lot of improvements that would make it more effective and easier to control; these are covered later in the section 'Recommendations for Future Work'. No autonomous operation was attempted.

In terms of monitoring progress against the requirements and the general success of the design, the following components have been completed:

- A system of software has been produced that allows tele-operation of the robot, presenting sufficient sensory data to allow remote operation. More information can be supplied about the robot (not all the sensor data that is available is relayed back to the operator yet). The main tracks and flipper arms are controlled, however the robot arm is not (though it should be a relatively trivial task to get it working with the work that has already been done).
- Through its use of heartbeats, safety controller and dead-mans handle, the software ensures that the robot is under control (safe) at all times.
- The design of the communications server is robust enough to minimize (virtually eliminate) required number of resets to the software. Under load testing it has failed to crash even with multiple machines connected. Even if a client thread does crash, it is designed to allow re-connection to the server to regain control.
- Security systems are in place to ensure only authorised users gain access to the robot.
- There is sufficient documentation to allow future teams to carry on the work

As can be seen, all of the non-functional requirements for the software have been met, while it has gone a good deal of the way to meeting all of the functional requirements that were targeted for this year. Three out of five of these functional requirements were fully met, whilst the two that weren't ("Must supply as much information as possible" and "Must allow control of all actuators on the robot") were largely met.

7 .0 CONCLUSIONS

This year's efforts have produced a good outcome in terms of the robot software. A sound base has been produced, that is more than capable of being built on by future teams. An overall design for the system has been produced, which meets the requirements of the competition, but also aims to meet some of the challenges that more 'real-world' applications might demand.

The software proved itself in the RoboCup rescue competition at Hannover in April 2008, allowing the remote operator to navigate his way through the arena to the toughest to reach point, something which was not matched by any other team. During this competition, lots of lessons were learnt, from which plenty of recommendations were generated and documented.

8 .0 RECOMMENDATIONS FOR FUTURE WORK

As this is a new project, with a projected development cycle of several years, there is a lot of scope for extending and improving upon the work that has been undertaken. The work so far has concentrated on providing a platform for remote clients to connect to the robot and control it. This has involved a lot of work on low level device drivers and the writing of a communications server and client program. These areas are fairly well optimised and developed, however the competition presented possibilities for improvement. Additionally, there is a distinction to be made between what improvements could be made to enhance the robot's competitiveness within the RoboCup Rescue competition and those which would improve its capabilities as a 'real world' general search and rescue robot and would be of more interest to companies involved in the field.

For the competition, there is quite a lot of focus on complete autonomy of the robot. Points for identifying victims in the first (yellow) section of the arena can only be gained if it done completely autonomously. There are also extra points to be gained for autonomous navigation and victim identification throughout the rest of the arena, although this would be incredibly difficult to achieve. Autonomy within the yellow section should be relatively easy as there are no large slopes or step-fields to contend with. The platform that has been developed has been designed to allow for an 'AI' platform to sit on top of it, with all data being presented to the AI in an event driven manner and control of the robot accessed through a single class which restricts access to a single commanding entity (i.e. the AI will not be able to interfere with a tele-operator's command by trying to control the robot simultaneously). Secondly, mapping should be fully implemented, taking the research and device interfacing with the LADAR and SONAR sensors that have already been undertaken to produce a map, the main challenge with this is the undulating nature of the terrain, there appeared to be little in the arena that would confuse either sensors (about which we had been previously misinformed).

For real world applications, the interest is a lot more focused on having an operator being involved with the robots deployment. Any 'intelligence' to be added to the robot should be more involved with assisting the operator with his job or improving the resilience of the robot to harsh conditions. Most of these improvements would also be of benefit in the competition.

Here is a list of such potential improvements:

- It would be beneficial to monitor the signal strength, bandwidth and latency properties of the communications channel; these can be displayed in the client window to alert the operator to any potential problems with the communications. This could be taken a stage further to allow the robot to detect when communications have been dropped to trigger it to autonomously try to regain the communications with its operator. Other solutions to this problem could include the dropping of signal boosters or if multiple robots are being used in the same area, using the communications systems of the other robots to relay the signal.

- To simplify the control of the robot, it may be beneficial to provide a way-point form of navigation control, whereby the operator can click on certain points on a map that a robot will then try to follow autonomously, alerting the operator when it gets there or if it encounters difficulty in doing so. This would allow a single operator to control multiple robots with relative ease, speeding up any rescue efforts.

- The ability within the Client window to take snapshots (or possibly even video clips) of any of the sensory data, such as the infra-red camera, the vision cameras or the map and save it to file would help document the rescue operation.

- A timer integrated within the Client window to provide quick feedback to the operator about how long is left for the competition run, or in real-world applications to monitor how long the robot has been active.

- A major issue experienced in the competition was not the ability to see the terrain to the front of the robot, but to try and work out what configuration the robot was in. This problem had been foreseen and lots of sensors are in place, such as the Inertial Measurement Unit (IMU), that will provide sufficient data for this purpose. The initial concept was to create a 3D model of the robot to display the position of the flippers and the arm and use the IMU data to tilt it to give the operator feedback, the compass heading could also be displayed. Implementing this would be a great help to the operator.

- Whilst the ability to rearrange the window to suit the operator’s preference is very useful, it can be very difficult to arrange them all back onto one view again when necessary. It may be useful to provide a default window layout where everything is on display that can be reverted to with the click of a button.

9 .0 REFERENCES & BIBLIOGRAPHY

1. "Teach Yourself UML in 24 Hours", J.Schmuller, Sams Publishing, 1999
2. "Java for Dummies", D. Lowe, Wiley Publishing Inc., 2005
3. "Real-Time Java Platform Programming", P. C. Dibble, Sun Microsystems Press, 2002
4. "Swing Hacks", J Marinacci & C. Adamson, O'Reilly Media Inc., 2005
5. "Professional C++", N. A. Solter & S. J. Kleper, Wiley Publishing Inc., 2005
6. "The Java Native Interface: Programmers Guide and Specification", S. Liang, Addison Wesley Longman Inc., 2003
7. "Safety Critical Computer Systems", N. Storey, Pearson Education Ltd., 1996
8. "Introduction to Robotics", J. J. Craig, Pearson Education Inc., 2005

10 .0 APPENDICES

10.1 APPENDIX A: CD CONTAINING CODE AND JAVADOC DOCUMENTATION

10.2 APPENDIX B: SOFTWARE BUG REGISTER

This register contains a description of all the known bugs that have been reported in the software that have yet to be fixed (N.B. ‘bugs’ are distinctly different to features that have yet to be implemented).

<u>Bug # 10</u>

Reported By:	A. Barnes
Date:	24/01/08
Severity:	Low
Description:	The USB controller cannot be connected (or reconnected) once the software is started; the software must be restarted with the controller connected.

<u>Bug # 14</u>

Reported By:	R. Sanders
Date:	15/02/08
Severity:	Medium
Description:	The connect button on the USB controller needs de-bouncing, it regularly registers two presses rather than one, causing control to be gained and lost in the same push.

<u>Bug # 20</u>

Reported By:	A. Smith
Date:	09/03/08
Severity:	Low
Description:	Exiting the client using the Robot-> Exit Menu doesn’t actually stop all threads from running properly.

<u>Bug # 21</u>

Reported By:	A. Barnes
Date:	09/03/08
Severity:	Low
Description:	Shutting down the Server using the Shutdown command doesn’t stop all threads from running properly, causing issues with serial port locks being persisted between sessions and blocking reconnection attempts on subsequent runs.

<u>Bug # 22</u>

Reported By:	A. Smith
Date:	10/03/08
Severity:	Medium
Description:	Program exit attempts via Ctrl-C aren’t caught properly by the server program in Linux (works in Windows) – acts as normal rather than calling the custom shutdown routine.

Bug # 23

Reported By: A. Barnes
Date: 13/03/08
Severity: Medium
Description: On server start-up several device threads are blocking during their construction, resulting in a slower start-up time.

Bug # 24

Reported By: A. Barnes
Date: 16/03/08
Severity: Medium
Description: The Client program won't reconnect properly after it disconnects from the server.

Bug # 27

Reported By: A. Barnes
Date: 01/04/08
Severity: Low/Medium
Description: The vision camera window in the Client program blocks and throws an exception if it cannot connect to the camera.

Bug # 31

Reported By: A. Barnes
Date: 16/04/08
Severity: Low
Description: The DataStore Recorder functionality doesn't work because the file output stream to write to is contained in the DataStore and therefore the DataStore cannot be serialised (streams can't be serialized.

Bug # 33

Reported By: A. Smith
Date: 20/04/08
Severity: Low
Description: The SpeedController Class in the Server doesn't ensure that the SpeedControllers receive a heartbeat every 0.5 seconds (though in practice with the frequency of controller updates, this is irrelevant).

Bug # 34

Reported By: A. Smith
Date: 20/04/08
Severity: Medium/High
Description: The Port IDs of all attached serial devices are prone to change every time the PC starts.

Bug # 35

Reported By: A. Barnes
Date: 20/04/08
Severity: Low
Description: The IR Camera Image occasionally goes out of synchronisation with the image grabbing server, causing the image to be fragmented.

Bug # 36

Reported By: A. Barnes
Date: 21/04/08
Severity: High
Description: The IR Camera Image Server causes a network stream buffer overflow after a constant period of time (probably after transmitting a certain amount of data).

10.3 APPENDIX C: SAFETY TEST CASES

Test: Use Ctrl-C in a shell window to stop the robot server software whilst the robot is driving forward.
Expected Result: The robot should come to a stop within 1 second.

Test: Turn off the robot's onboard PC whilst the server software is running and the robot is driving forward.
Expected Result: The robot should come to a stop within 1 second.

Test: Disconnect the serial cable to the main tracks speed controller whilst the server software is running and the robot is driving forwards.
Expected Result: The main tracks should stop within 0.5 seconds.

Test: Disconnect the serial cable to the flipper rotation speed controller whilst the server software is running and the flippers are being rotated.
Expected Result: The flippers should stop moving within 0.5 seconds.

Test: Disconnect the network cable from the PC whilst the robot is driving forward
Expected Result: The robot should come to a stop within 1 second.

Test: Disconnect the USB controller from the client pc while the robot is driving forward
Expected Result: The robot should come to a stop within 1 second.

Test: Press each of the buttons on the controller in turn (one button at a time)
Expected Result: Nothing should happen (it requires a minimum of two buttons to be pressed for any action to occur)

Test: Try driving with the safety controller disconnected from its power source
Expected Result: Nothing should happen