Customisable non-graphical interface to start/stop activities on mobile robots

CS39440 Major Project Report

Author: Sam Matthews (sam82@aber.ac.uk)
Supervisor: Dr. Frédéric Labrosse (ffl@aber.ac.uk)

3rd May 2019
Version: 1.0 (Release)

This report was submitted as partial fulfilment of a BSc degree in Artificial Intelligence And Robotics (GH76)

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB
Wales, U.K.

Declaration of originality

I confirm that:

- This submission is my own work, except where clearly indicated.
- I understand that there are severe penalties for Unacceptable Academic Practice, which can lead to loss of marks or even the withholding of a degree.
- I have read the regulations on Unacceptable Academic Practice from the University's Academic Registry (AR) and the relevant sections of the current Student Handbook of the Department of Computer Science.
- In submitting this work I understand and agree to abide by the University's regulations governing these issues.

Name Sam Matthews

Date 3rd May 2019

Acknowledgements

I would like to thank my supervisor Fred Labrosse for his support throughout the entire project.

Abstract

The main goal of this project was to solve various problems associated with operating the large robots in the department. These robots have screens attached to them that display a non-graphical environment running Linux and can be controlled using mounted keyboards. The most notable problem with this is that the output of only one process can be monitored in a single `TTY`, meaning that it was necessary to log in to separate `TTY` instances. Additionally, using an entirely terminal-based interface is inconvienient due to the necessity to type every command. To streamline this process, it seemed appropriate to create a ”tabbed” interface that allows users to run commands from a list as well as switch between different processes using a tab bar. This can run any executables including `ROS` nodes that are defined in a configuration file beforehand. In addition to this, it was also decided to include a method for teleoperation within the application as well as a configurable way to display `ROS` topics in a menu. The main outcome of this work is a form of convienience and flexibility for anyone performing experiments using the robots and provides a form of redundancy in the event that a laptop connected to the robot is unavailable/infeasable.

Contents

List of Figures

List of Tables

Chapter 1

Background & Objectives

1.1 Background

The main objective of this work was to provide an application for operating robots that emphasised efficient use of resources as well as usability. The personal motivation for this project was to create something that can have a real benefit towards future research and will actually be used by others in their own work to make their lives easier/better. This is in addition to personal interest and experience in using low level, non-graphical applications that emphasise speed, extensibility and usability. This project provided an opportunity to understand how these sorts of systems work as well as the potential to make more of these types of systems.

1.1.1 Similar Systems

There are plentiful and various open source systems to assess in order to gain an understanding of non-graphical applications. One example that was assessed was `tmux` [1], a terminal multiplexer written in `C` using `ncurses`. This application runs in the terminal and can divide a terminal window into tiles running separate shell sessions. It can also display different ”windows” which are similar to virtual workspaces and displays tabs at the bottom of the screen to represent these workspaces. This demonstrated the possibilities regarding the monitoring of program output and preserving it in tabs. The main problem with `tmux` as an alternative is that there is no menu to select applications to run from built in as each window is a shell prompt.

Another example that was looked at was `ranger` [2], a terminal file manager written in `Python` using the `curses` module. `Ranger` shows a usage of menus that can be navigated using key bindings as well as mouse input. Similarly to `tmux`, `ranger` divides the terminal into sections and has ”tabbed” functionality. However, unlike `tmux`, the sections show menus that display the current and parent directory as well as a preview of the currently selected file/directory. Also, the tabs are essentially different `ranger` sessions that help to multi-task within the same window. The general layout of `ranger` is very intuitive for this kind of application and also demonstrates further possibilities in terms of capturing

process output and manipulating text.

Finally, a system that is already being used for controlling these robots is a `qt` application made by Fred Labrosse. This application has dedicated tabs for stopping, teleoperation and status monitoring. This application was the basis of what this project is trying to recreate in the terminal aside from some extra features. As a result, the layout and functionality of the relevant sections of my application are very similar to his.

1.1.2 Reading and Preparation

The initial preparation for this project involved considering the various ways to make a non-graphical application. Creating a custom UI library would have been infeasible so utilising existing tools was necessary. A very common tool for this is `ncurses` [3], however, alternative tools were considered such as `termbox` [4] and even `BASH` [5].

The Curses Development Environment (CDK) [6] was also tested in order to make the design less complex. This library adds `ncurses` widgets such as menus and buttons and can help to construct a TUI much quicker. Unfortunately, this did not work during initial testing and so was disregarded in favour of writing similar code from scratch. Additionally, there is no tab bar widget included meaning that some of the UI would have to be hand coded regardless.

The primary reading for this area mostly consists of reading manual pages for standard Unix utilities. There are some exceptions such as this comprehensive guide on `ncurses` [7]. Some prototypes were made in order to get a better understanding of the admittedly sparse documentation. Additionally, this work illustrated how `ncurses` works in terms of displaying text which informed later design decisions.

As preparation for the report, a journal was kept and updated daily indicating progess and difficulties. This also acted as a way to organise new features to add based upon weekly meetings. This journal was not written as a blog and is not publicly available.

1.2 Analysis

The background work resulted in the selection of `ncurses` as the method of creating a non-graphical interface. This was decided as `ncurses` is a standard utility used everywhere and has more useful features than the more minimalist alternatives such as built in panes for splitting the terminal window. It is also the same tool used by the similar systems that were assessed. This will allow the application to be run in most terminal emulators as well as the `TTY`.

Due to the low-level nature of `ncurses`, there was an understanding that not many UI features would be implemented. This is because more is required in order to achieve similar results in a more graphical API. This analysis showed that the final system had to be relatively simple so that other requirements could be feasibly met within the scope of the project.

1.2.1 User Stories

The requirements were identified as user stories after the background work was done and a better understanding of the project had been achieved. The stories that were identified are as follows:

- The main menu is generated from a configuration file that the user can specify.
- The user can run processes by selecting them from the menu. The process will appear in a tab and the output is displayed in the main window.
- The user can switch tabs to view different processes being outputted.
- Users can close the currently views process which kills it and removes it from the tab bar.
- The user can scroll through menu items or output using a scrollbar. This behaviour is based on the size of the window and the limit of the buffer.
- The robot can be controlled by the user using the keyboard or touch screen
- There will be a part of the interface that will display robot status information

Each requirement was also associated with relevant classes as depicted in the class diagram in Appendix D.

1.2.2 Project Components

The problem was broken up into three main systems, each of which represent a distinct stage in development as well as its own set of requirements. The ordering of these sections was based mainly on personal preference towards developing a user interface before the back-end. It was also foreseen that each of theses sections would take up a similarly significant partition of the code.

1.2.2.1 User Interface

The UI was seen as a substantial component of the system as it was made up of various interconnected sub-components. For most, projects, the user interface could be considered the simplest part, however, creating a comprehensive user interface is more involved with the tools that were used. The `ncurses` library definitely makes this process simpler, however, it is still low level and more tightly integrated into the back-end. The user interface was broken up further into windows that each serve a related yet distinct purpose. The main windows consist of the main output area, the process menu and the tab bar.

1.2.2.2 Process Management and Multitasking

This section concerns the ability to run pre-defined processes from a menu and view the output of each of them. To achieve this, the application needs to keep track of all open processes as well as be able to redirect and capture output. Additionally, it should be ensured that all processes are updated at equal intervals and that there is no performance impact on the user experience. There should also be functionality for closing a process tab that would also effectively kill the process.

1.2.2.3 Teleoperation and Monitoring

Teleoperation refers to the ability to control a robot using its own keyboard. This is arguably not teleoperation as it is not remote controlled, however, it can be used as such if the application itself is run on a separate machine and publishes messages to the topic the robot is listening to. Therefore, this feature will be known and referred to as teleoperation. This will involve creating a dedicated window layout with buttons that can send different velocity messages as well as being able to control the robot using the keyboard.

The monitoring feature will have another dedicated window layout that displays status information of the robot. The topics to display will be inspired by the existing `qt` application that was assessed in Section 1.1.1.

1.2.3 Security and Safety Issues

Due to the nature of this project, there are little to no security issues as there are already security measures put into place on the robot platforms being used. Also, the application does not deal with private information nor does it interface with networking protocols at all.

However, there are certainly issues of safety to consider as this application will be directly and indirectly controlling large, heavy machinery. While this is not strictly related to security, measures still need to be taken during the project as well as considered during the design of the software. To combat these potential issues, all hardware testing is done under supervision and is accompanied by the existing teleoperation application. This is running on a laptop and can stop the robot immediately at any time in case anything goes wrong. Additionally, it is important to close programs opened in the application carefully to ensure that messages are not still being sent upon shutdown.

1.3 Process

The development process throughout the project can be described as a modified version of Scrum that is more appropriate for one individual rather than a team. This adaptation removes the idea of roles and instead allows for all work, planning and assessment to fall to a single developer as well as the client where necessary. What remains is an

iterative development approach with an emphasis on rapid prototyping and integration as well as quick response to a change in the client's requirements. The justification for this approach is due to inexperience with the chosen technologies as well as the fact that the requirements were constantly subject to change.

Each sprint lasted for one week, at the end of which was a meeting with the project supervisor/client. This meeting would comprise of reporting progress, discussing issues and new features as well as planning the objectives for the following sprint. The start of a sprint would include either an analysis of a new feature to be completed or prioritising the most urgent/important items in the backlog. The amount of features to be completed in each sprint depends on individual understanding of personal velocity. If all designated features were finished before the end of the sprint, the next item on the backlog would be addressed.

Chapter 2

Overall Final Design

The evolution of the design of the application is inherent to the agile methodology adopted. The design outlined in this section will illustrate the final design as well as talk about how each design element has evolved when appropriate. Some of the design has been consistent from the beginning of the project such as the general user interface design. However, other parts needed to evolve due to a change in the understanding of the requirements and the technical limitations of the technologies used. There is also discussion of some decisions made that are not relevant to the content in Chapter 3.

Much of the initial design consisted of user stories arranged on a story board alongside basic class descriptions. These items were arranged on the story board in such a way that the classes would cluster around relevant stories and create spatial CRC ”cards”.

2.1 Overall Architecture

The final design of this project uses an object oriented approach to development making use of inheritance and encapsulation. The class diagram illustrating this design is included in Appendix D.

As is common for object-oriented programs, the main method is usually small and merely exists to construct an object that will itself run the application. Regardless of this convention, it was decided that this is wasteful and that the main function should be used appropriately. The usefulness of objects within the design is arguably restricted to the utilisation of inheritance for the front end classes. This is likely due to a personal bias towards procedural programming paradigms which is why C was initially used as noted in Section 2.2. However, the re-implementation in C++ affected the initial design as it was decided that making effective use of the language’s features (including objects) was important, regardless of personal preference.

Each section of the screen is represented by a Window object that defines its co-ordinates as well as dictates the behaviour of printing. The Menu inherits from the Window to take advantage of the printing functionality as well as implement menu selection. The TabBar then inherits from the menu as it is effectively the same aside from the orientation and the

ability to iteratively refresh processes. It seemed more appropriate to make the tab bar act as the container/manager for the running processes as opposed to having a dedicated object for such activity. This provides a way to bridge the gap between the front and back ends. The teleoperation interface was planned to inherit from the window class as this seemed to make the most sense at the time. However, it was deemed to be unnecessary as much of the functionality was unrelated to the other windows. As a result, the front end of the teleoperation system is a function in Window, whereas the back end encompasses its own independent class.

The Button class does not inherit from window even though much of its functionality is the same. The reason for this is that the window class automatically spawns with a list of buttons. This could be potentially changed to reduce duplicated code, although the benefits gained are not substantially worth it.

A class that was omitted was the ScrollBar class which would have been used for spawning and updating the scrollbars. It would also scroll through the printed output and maintain the current scrolling position. This was also found to be redundant due to the simplicity of scrolling and how easy it would be to integrate it directly into the main printing function.

The back end classes mainly consist of Monitor, Teleop and Process. The monitoring and teleoperation classes are mostly used to manage `ROS` messages in a way that can be interfaced with the from end. The process class is integral for process management and stores information about each process as well as provides tools to start/stop them.

In terms of the overall behaviour, the application is based around the main loop that accepts user input and acts accordingly. Within this loop, each refresh subroutine occurs once such as the teleoperation update or updating process output buffers. This sort of structure can cause issues in terms of temporary stalling during touch input due to the delay needed to register clicks properly and can cause some latency. However, the behaviour effectively spreads the load of each minimal task to the point where there is no performance impact. As a minimal system focused on multitasking, this kind of behavioural structure is optimal when operating on a single thread. Multi-threading was briefly considered, but was deemed to be unnecessary and would likely lead to more problems than it would potentially solve.

2.2 Implementation Tools

The application is written in `C++11` using a combination of `C` and `C++` libraries. The choice to use `C++` is primarily based on the need to program using `ROS` [8]. `ROS` wrappers only exist for `C++`, `Python` and `LISP`, making these the only choices for languages to use. It was decided to use `C++` over the other `ROS` supported alternatives simply due to increased runtime speed as well as existing knowledge and experience in using `C++`. This decision is one that evolved during development as the initial application was written in `C`. This was due to the mistaken understanding that all interactions with `ROS` would have been through using scripts or executables. However, due to a change in the understanding of the requirements regarding how `ROS` information should be displayed, it was made clear that using `C` would have been less than ideal.

In terms of development tools and environments used, `neovim` [9] was used as a text editor as opposed to using an IDE. This is mainly due to past experience using `neovim` as well as the enhanced editting and runtime speed a minimal modal editor can afford. All work was backed up daily to a `GitLab` repository as well as automatically synchronised between two computers using `Syncthing` [10].

2.3 Configuration File

The decision to use `YAML` as a configuration syntax is detailed in Section 3.2.1 and was mainly due to convenience. The general layout consists of two sections, those being ”command” and ”teleop”. The command section lets the user define a set of executables that they want to run using the application. The teleop section contains information on the behaviour of the teleoperation tab. Each executable needs to be referenced using its absolute path as it was decided to not take into account the path environment variable in order to have a consistent method of invoking executables. It was decided to have the location of the configuration file be in ” /.config/robot_manager/config.yaml” as this is standard for Unix programs. It is also beneficial for the user as their own configurations can remain constant even when updating the software. An example of the configuration file can be found in Appendix C.

2.4 User Interface

Various elements of the design of the user interface were specified before the start of the project by the client. Such elements include the addition of a tab bar showing all running processes and an area to monitor process output. An intuitive decision was to have the tab bar placed at the top of the screen and have the vast majority of the screen real estate be taken up by the main output section. This is in accordance with the standard layout of tabbed graphical applications but simply adapted to a text interface.

The design of the user interface was loosely inspired by existing applications such as `ranger` [2]. The goal of the design was to fit as much as possible onto one static screen as possible. This is mostly due to the constraints of working in an entirely text based environment as well as the usage of small display devices. As a result, the design is deliberately simple where each section is divided up into rectangles and remains consistent for the duration of the application runtime.

Additionally, the consideration of ease of use had an impact on certain design choices. For instance, the area displaying available processes to run was integrated as a side bar as opposed to its own dedicated screen. This saves time if the user wants to run several processes in rapid succession without having to repeatedly switch back to the main menu.

The design of the interface also accounts for touch input as well as keyboard control. This requirement was designed for with the inclusion of extra buttons on screen such as a button for closing tabs, scrollbars for viewing output history as well as onscreen buttons for teleoperation.

Initial designs can be seen in Appendix D and illustrate the change in design as a result of discussing the diagrams with the client. Figure D.3 shows the addition of scroll bars as well as the importance of keyboard input. Figure D.2 shows that the initial idea was to have monitoring information replace the commands list when in teleoperation mode and that each of those items would go to a separate screen. This was seen to be too cumbersome and involved too many unnecessary clicks according to the client. The response to this was to have everything available on one screen and to treat everything as just another tab.

2.5 Other Design Decisions

One decision that was made was to deliberately disregard changes in terminal window size during runtime. The reason for this is that when using a plain `TTY` with no window manager, the terminal window size will remain constant. This obviously has unintended effects during testing in a graphical environment, however, during real use, this will not be an issue. This also means that it is not necessary to dynamically change the UI which would be a waste of resources and development time.

A similar decision was to maintain a fixed sidebar width of 25 characters. This amount was determined to be big enough to fit any reasonably named command onto one line, but simultaneously small enough to be unobtrusive. It also means that the sizing does not have to be calculated manually which saves a step in initialisation. One potential concern early in the project was that there would be issues with displaying on smaller screen sizes. However, as noted in 4.2.3, this was fortunately not a problem.

Chapter 3

Implementation and Design Changes

As an agile project, there was little upfront design and a heavy focus on iterative development. As a result, this chapter will not only talk about what was implemented and the issues faced but also describe how the design was affected by these issues. This will provide an understanding of how the final design detailed in Chapter 2 came to be as well as explain the relationship between it and the initial design decisions noted in the same chapter.

3.1 User Interface

The interface is primarily made up of a series of windows that divide the terminal window into tiled rectangles. This is particularly easy to achieve using `ncurses` as it has functionality for this built in. The major advantage of using windows is that when manipulating the content within them, all coordinates are relative to the origin (top-left corner) of the window. This made developing printing functions seamless as there is no need to keep referring to absolute positions and instead assume that everything starts from zero.

One interesting aspect of the window implementation to note is the calculation of the window sizes. Every window dimension is based upon the height of the tab bar and the width of the side menu. As a result of this there is very little hard coding of values which makes it easier to adjust the sizing of the entire interface.

One challenge, or more accurately annoyance, when using `ncurses` is that the printing function requires the passing of the vertical co-ordinate value before the horizontal. This is counter to common convention and was occasionally cause for small mistakes.

3.1.1 Main Output Window

The main window is treated as the basic foundations of the other windows. It is mostly used to display a list of lines as well as have the ability to scroll through this output. The other window types inherit from this window and share much of its functionality.

The fundamental functionality of each window in the application is to print line by line. There is no existing function for this in `ncurses` so it was necessary to create a way to do this. The function implemented takes a vector of strings where each string is a line to be printed. It iterates over this starting from the size of the window subtracted from the size of the list and simply prints each string until the window is full or the list runs out.

In order to account for scrolling through output, the iteration actually starts from the current scroll position. When the list is updated, the scroll position increments unless the user is currently scrolling. This is referred to as scrolling mode which prevents the scroll position from changing so that the output can be viewed conveniently.

The initial design for the printing mechanism utilised the idea of calculating an offset based upon the difference between the height of the window and the length of the output buffer of the currently selected process. This offset was calculated in real time and would then be added on to an incrementing index starting at 0 to select the appropriate line to be displayed at the top of the window. Additionally, the scrolling/printing was performed in relation to the bottom of the screen rather than the top. This made the implementation more confusing and also created issues when it came to scrolling multiple lines at a time. Notably, it was impossible to scroll to the top of the buffer without under/over shooting the first line. This was primarily due to the unnecessary complexity of the function leading to mistakes.

The solution to this was to simplify the printing method to start at the offset itself as opposed to trying to calculate everything at once in the same function. Also, the scroll position is treated as being relative to the top of the window as this was more intuitive. This ended up making the printing and scrolling far simpler and more reliable than the initial implementation. Even though the scrolling is functioning, the bars themselves are not being displayed properly. This is likely due to the overhaul in the design of the printing and the over-complication of the original method of displaying bars.

3.1.2 Side Bar Menu

Due to the low level of the `ncurses` library, the idea of what a menu is had to be abstracted and implemented from scratch. The idea is that the currently selected item is based on a current index and that the line corresponding to this index will be highlighted. When a menu item is chosen, the index dictates which process to run from the stored list. The way this works in practice is to modify the Window printing function to invert the background and foreground colours when the line number being printed matches the selection index. This creates the illusion of having ”selected” an item in the list. To make referring to the index easier, this number starts from one as opposed to zero. The reason for this is that the zero co-ordinate refers to the window border outline. This means that accessing list elements requires subtracting 1 from the index to retrieve the current item which did cause

some minor issues due to confusion.

In order to move this selection index, keyboard control was implemented as a simple switch statement in the main program loop. The keys ”up” and ”down”, for instance would decrement and increment the index respectively, thus changing which line is highlighted and which process is started upon confirmation.

Mouse input was added later in development but was a simple addition to the existing switch statement. A mouse event is treated as a keyboard event with some extra meta-data such as screen co-ordinates. This then expanded into checking which window the mouse press was registered on and subsequently retrieving the correct menu/button index. This was based on whether or not the vertical co-ordinates of the mouse input was less than or equal to the size of the list. Unfortunately, this design resulted in some cases having to be repeated for both key and mouse input rather that effectively reuse the code.

There were some issues regarding mouse input, namely that if the application was restarted a few times, trying to use the mouse can lead to it crashing. Also, after one restart, using the mouse caused all process output to be upper case. The reasons for this behaviour was to do with how pseudo-terminals were being invoked as described in Section 3.2.2 and are presently resolved.

The way that touch input will work in practice involves using `gpm` in order to have a mouse cursor in the `TTY`. Input on the touch screen is registered as a mouse click at a given co-ordinate and will be managed by the application user interface appropriately.

3.1.3 Tab Bar

The tab bar inherits from the Menu class and performs in much the same way aside from some important differences. One of which is that the contents of the tab bar are dynamic as processes are added/removed during runtime. This results in the addition of the ability to add a tab (open a process) and close a tab (kill a process). There is not much to discuss for these elements aside from having to also update the list of currently open processes stored as as a vector. However, much of the process related features of the tab bar a explained in Section 3.2.

An unfortunate issue with the tab bar is that it does not scroll properly like the other windows. This means that while new tabs will spawn, they will be invisible as they will be outside of the viewable window. They can still be reached using the keyboard input but touch control will not be possible after a certain amount of tabs.

3.1.4 Teleoperation Tab

This addition to the user interface came about during the development of the teleoperation feature described in Section 3.3. This window is rather distinct from the others as it does not deal with lists of lines. Alternatively, it houses buttons that are clickable and are arranged as depicted below.

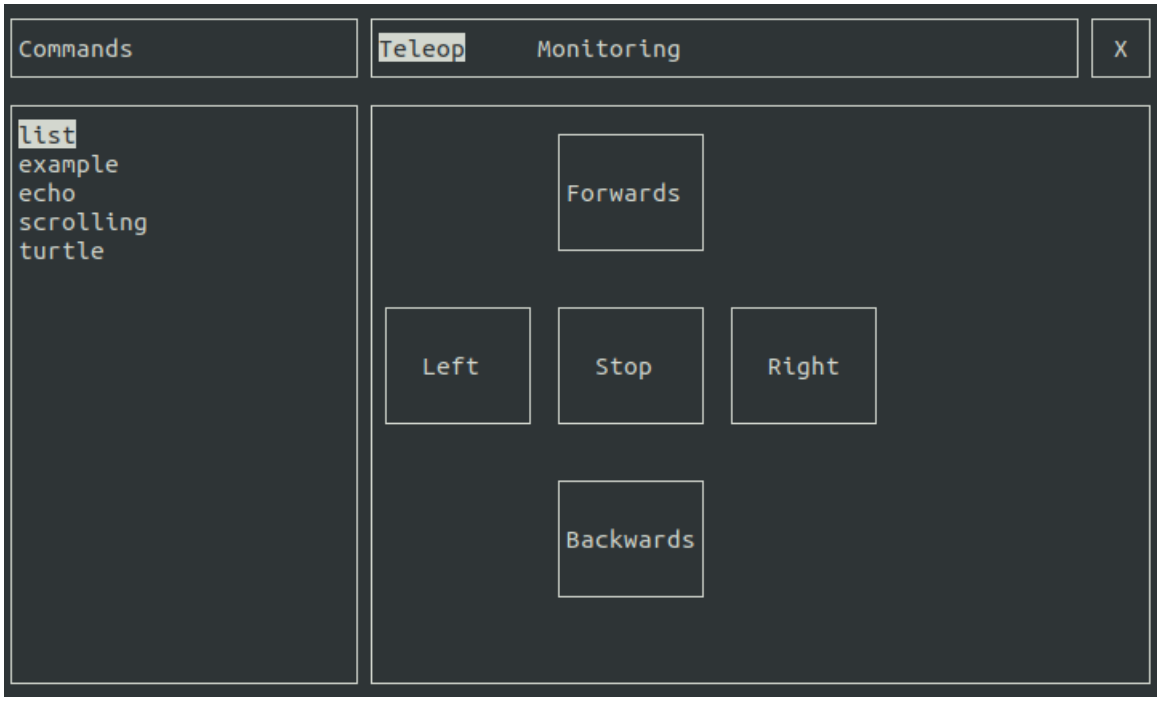

Figure 3.1: Screenshot of application showing the teleoperation interface

The buttons themselves are sub-windows of the main output window. The reason for this is that it was easier to draw a box around them. Also, this makes it easier to determine if a mouse click was registered inside a button due to the `ncurses` function `wmouse_trafo`. The function that defines the positions of each button is unfortunately long and overly complex. It also uses hard coded values for the button dimensions and the button labels. However, it effectively sizes the buttons appropriately and ensures that the label is kept in the middle of the button by forcing the vertical size to be an odd number.

In order to integrate the teleoperation tab into the tab bar, it was decided to keep it as the first index at all times. The menu printing function was modified to accept an extra vector of strings to prefix the start of the list. Additionally, it was hard coded that if the bar index was at 1, this indicated that teleoperation mode was active.

The user interface responds to teleoperation mode by displaying the teleoperation tab in the main window. Additionally, this allows the user to control the robot using the keyboard or on-screen buttons.

3.1.5 Monitoring Tab

As explained in Section 3.4, the monitoring tab was created as a static output displaying pre-defined sets of data. The UI comprises of two sections (Telemetry and Safety) containing lists of message data with preceding labels. As the interface is hard coded, the method of printing is not exactly elaborate, apart from the way that coloured boxes are displayed.

Colour-coded boxes are used for some items as opposed to raw data for a more visual representation of the robot status. This works by defining colour pairs for green, yellow and red, and assigning each to numbers. The number that is used is then based on the message value provided. As a precaution, any number outside of the expected range is treated as green by default. A box is then drawn by enabling the correct colour pair before printing two spaces which creates a square. A screenshot of the final version is shown below:

Figure 3.2: Screenshot of the monitoring interface

3.2 Process Management

3.2.1 Starting Processes

The application reads the `YAML` configuration file during its initialisation. Part of this file defines the commands to be made available in the menu and run within the program. The file is parsed using `yaml-cpp` [11] and each process is constructed with the data extracted from there. The file was originally using a custom format until it was decided that this would be too complicated and unnecessary. Also `YAML` is used heavily in `ROS` so using it for this application seemed more suitable than alternatives for the sake of consistency. Additionally, `yaml-cpp` is the actual library used by `ROS` to interpret `YAML` files. This means that on some systems, there is no need to install it as a separate dependency (although this was not consistently true). It was also decided to separate the program executable option from the parameter option in the file. This makes parsing easier due to the increased ability to deal with commands that do not use parameters.

The location of the YAML file is initially presumed to be in the ".config" directory as this is standard for most Unix applications. However, if this is not present, the provided sample file is read as a temporary alternative.

In order to start a process, the strings in the YAML file need to be split in order to extract the executable name and the separate parameters to be passed to execv [12]. This was done using the boost [13] split algorithm, splitting by spaces or "/" characters where appropriate. This was planned to be done manually using strtok, however, due to complications, it was decided to make use of pre-made libraries. This made the relevant sections of code simpler and ultimately saved development time. There were also some initial problems regarding accepting zero or multiple parameters. However, these were alleviated by making use of both yaml-cpp and the boost split algorithm and is now functioning properly.

When the necessary process information has been extracted from the configuration file, the desired process is run when it is selected from the menu. This is done by forking the program and running the execv function. Specifically, the program uses forkpty [14] which forks as well as creates a new pseudo-terminal simultaneously. This streamlines the common practice of doing this manually with arguably verbose code. Before introducing pseudo-terminals as described in Section 3.2.2, the regular fork function was used originally before the change in design decision.

3.2.2 Emulating Teletype Interfaces

An unforeseen issue surfaced while testing the application using ROS commands such as roscore and roslaunch. This issue was primarily to do with how ROS utilities (as well as other programs) display and manipulate data in the terminal. In order to display text in colour without explicitly using ASCII escape codes, as well as accept standard input as raw input, these programs change the termios attributes of their respective terminal devices. If a program does this to suit its own needs, the parent application does not accept user input as the ncurses settings were being overridden by the children.

The solution to this problem was to essentially emulate a Teletype interface (TTY) using pseudo-terminals (PTY's) [15] for each child process so that they act as if they are running in a regular terminal environment. Additionally, it was necessary to disable standard input for each PTY so that the parent can take over. Much of the understanding and the actual code for this came from a project called tty8 [16], an example of which can be found in Appendix C.

The example shown illustrates how standard input as well as blocking were disabled as the process was being started. As well as this, during the creation of the PTY, the original signal mask was passed as a parameter in order to preserve the terminal attributes of the parent. This resulted in a positive outcome in which ROS processes were not interrupting the activity of the parent, allowing for effective multitasking and the ability to use the application as intended.

A subsequent benefit of this problem being solved was that the colour data that ROS outputs was being captured in its raw form where previously it was being omitted from the text. This creates the opportunity to parse and convert text into its intended colour in the main output screen using ncurses colour pairs during the printing stage. Unfortunately,

due to time constraints this was not implemented and instead the codes are deleted using a regular expression. This makes use of the `regex` [17] standard library and does remove most of the undesired meta-data. There are some matches that are not consistent with other testing that was performed using a `regex` tester [18] for unknown reasons. As a result there are some visible escape codes that are not too noticeable but are not removable for some unknown reason.

Another improvement that using pseudo-terminals made was the resolution of an issue identified during earlier development. The issue occurred when there were multiple tabs open and the user closes a tab that is to the left of any other tab. This resulted in the output of the tab that was to the right of the closed tab to stall indefinitely. The issue was non-existent after introducing pseudo-terminals even though the direct cause was never determined.

One problem with the way pseudo-terminals were being created was discovered later in development and was found to be the cause of major issues. Essentially, the original terminal settings that were passed to `forkpty` was being stored before the `ncurses` environment was initialised. This means that it was not using the terminal attributes set by `ncurses` but rather the settings used before initialisation. This, combined with not properly setting signal settings on the side of the parent, lead to undefined behaviour and complete crashing. This usually occurred when trying to use the mouse as described in Section 3.1.2 and made any hope of using the mouse potentially dangerous. The issue was also leading to inconsistent output behaviour such as randomly making all output upper-case. The issue has since been properly resolved and the mouse input behaves as expected.

3.2.3 Capturing Process Output

An initial design decision was to use `popen` [19] in order to combine forking and running `exec` and `fdopen` [20] into one function. However, due to a misunderstanding as to how `popen` works, it was found to be unhelpful for certain features of the application. The reason for this is that `popen` does not return the process ID of the forked process which means that it is impossible to reliably kill it. The only way to use this would be to kill processes with `pkill` [21] which kills all processes matching a regular expression which is not ideal. This meant that it was necessary to re-implement the functionality of `popen` in order to gain access to both the standard output and the `PID` of each process. This served to delay development as well as make the code more complicated than initially predicted. However, this re-implementation meant that solving a later issue 3.2.2 was easier due to more granular control over the functionality.

In order to display process output properly, it needs to be redirected, captured and stored appropriately. This was done using `fdopen` to create a file stream that captures the file descriptor of a pseudo-terminal. This is done for each respective process that is run. In order to get the data from the file stream, `fgets` was used as it terminates either at a newline character or after a specified amount of characters. This is helpful because if the size of the window is given to it, this results in line wrapping built in to the function. This is performed for every process in the tab bar iteratively. An example of line wrapping is shown in the screenshot below:

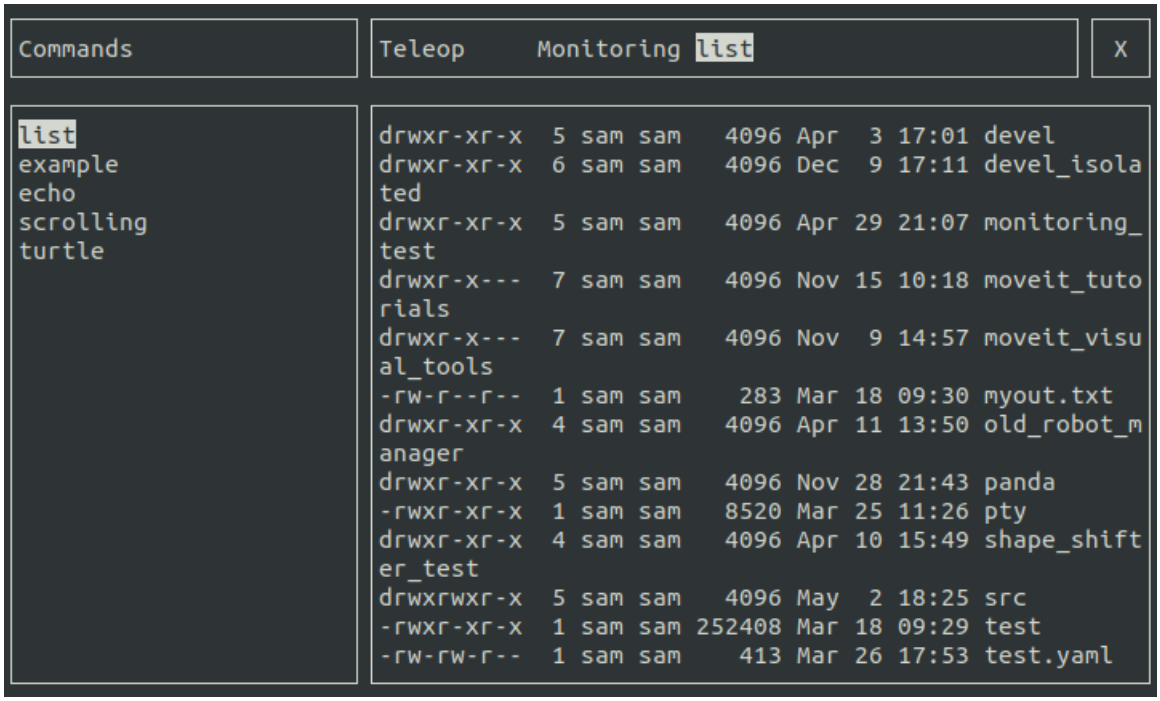

Figure 3.3: Screenshot of the `ls` program output being displayed and wrapped

Before the implementation of pseudo-terminals as described in Section 3.2.2, the output redirection was being done with pipes and the `dup2` function. There were some initial issues with redirecting the file descriptors correctly as it is not well explained in the documentation. Also, before the discovery of the `fdopen` utility, there was an attempt to recreate `fgets` using the read function. This turned out to be a waste of time and was not successful regardless of the inclusion of `fdopen`.

3.2.4 Killing Processes

An integral and simultaneously temperamental aspect of this project is to be able to kill processes easily. The way this was being done initially was to send a kill signal using `SIGINT` and then wait for the given process to terminate. The advantage of this is that the program can take note of the death of a process and prevent it from remaining in a zombie state. However, a particular problem with `ROS` applications is that they need to shut down safely and thus take a long time to terminate. The result of this is that the parent application that is waiting for them to terminate is completely frozen. However, since learning about options that can be passed to `waitpid`, a subroutine was added to keep track of terminated processes and subsequently wait for each of them and remove them once each process has fully terminated. This made killing `ROS` processes safer, more reliable and less likely to halt the parent application.

3.3 Teleoperation

This section describes the back-end implementation of the teleoperation feature. The general function of this is to construct and send geometry twist messages to a given ROS topic. The message that is sent will be adjusted based on user input, changing the linear velocity and the turning increment. Each button press will add an amount to the relevant velocities in question.

In order to ensure that the robot can be straightened, it was made so the turning value will degrade to zero over time. This allows the robot to continue to drive in a straight line if no alterations are made. This occurs iteratively after a given amount of time and subtracts a given amount from the velocity before sending a message. Without this delay, it was found that turning is impossible due to the main loop being too fast.

If the stop function is called, a linear and rotational velocity of 0 will be published ten times. This was a suggested amount given by the project supervisor based on the behaviour of the systems already in place. Sending multiple stop messages simply ensures that at least one will be received by the robot and is a point of precaution.

One issue discovered during hardware testing was that velocity messages were being sent out far too quickly, leading to lag and a large message queue. This was due to the fact that messages were being re-sent during each iteration of the main loop without any sort of delay. Messages have to be sent constantly during teleoperation in order to maintain control, however the lack of delay was causing problems for other applications. The solution to this was to add a timer to the refresh function that will tick over after a configured amount of seconds. The value of the amount of time to wait is defined in the configuration file which gives the user the ability to adjust it according to the needs of the hardware being used. Additionally, it was found that the queue for the ROS publisher was too large at 100 and should have been set to 10 to further limit the amount of messages that are sent at any one time.

An emergent behaviour of having two message sending timers is that, of course only one of them is going to be useful. That would be the time that is the smallest which does make the second timer redundant. That being said, the functionality of each is technically different given that one timer reduces rotational velocity as well as sends messages. So, depending on the system, it may be worth setting one higher than the other, meaning that the option is there if needed.

Much of the teleoperation behaviour is dependent on the configuration file as it defines the following attributes:

- The amount of time to wait between sending regular messages
- The topic to publish to
- The magnitude of the rotational degradation
- The amount of time to wait between degrading the rotation
- The magnitude of the linear and rotational velocities to be added on for every button press

An example set of values can be viewed in the teleop section of the example configuration file in Appendix C.

A feature that was also implemented was a teleoperation mode that would disable when not on the teleoperation tab of the interface. Disabling this mode means that no messages will be sent which allows other ROS processes to take control of the robot without interruption. There was an issue with this function where, while switching tabs will send a stop signal, the previous velocity before the switch was somehow being stored. This lead to the robot resuming movement when switching back to the teleoperation tab. This issue was solved by sending more stop signals and being more strict when entering and leaving teleoperation mode.

An unfortunate consequence of having one application that can control robots as well as manage processes is that it still has to be a ROS node. As such, it requires roscore to be running before it is started which is unfortunate as it means that the output of roscore cannot be monitored in a tab. This issue was discovered late in development and is unfortunately not fixable given the nature of ROS. A suggested solution was to use roslaunch to launch the application as it automatically runs roscore. Unfortunately, due to the way that roslaunch changes the terminal settings, the application simply froze and did not work at all.

3.4 Monitoring

The monitoring interface was initially designed to be a variable number of tabs that could be configured extensively to display any ROS topic data. However, due to a combination of factors, the interface was implemented as a static layout that displays pre-defined message types. One reason for this was time constraints nearing the end of development as well as complications involving subscribing to generic message types. Syntactically, ROS has no way to do this in C++ as the callback message parameter needs to have a declaration of its message type.

There was an attempt at using shapeshifters [22], however, the dependencies for doing this were not functional. While accomplishing something similar to this is possible, it was decided to create a monitoring tab that was suitable enough for Idris. The topics to subscribe to are defined in the configuration file and messages are checked for iteratively in the main loop. The back end of this process not exactly complex, although there are some interesting front end concepts described in Section 3.1.5.

Chapter 4

Testing

4.1 Overall Approach to Testing

As an agile project, tests evolved alongside the codebase as new features were introduced. Testing was conducted manually after each project build utilising a sample configuration file as well as various example programs (see Appendix C). As a minimalist program, it was appropriate to perform full build tests regularly in order to test each individual feature as it is being developed. Once a feature is finished, the overall functionality of the existing system is evaluated. This ensures that implementing a new feature did not compromise what has already been done.

Usually when a bigger part of the software was to be added, a separate program with the same functionality of the subsystem was constructed and later integrated if it performed as expected. This is useful when using new technologies or when a section of code needs to be isolated in order to be tested. Isolation was necessary due to the fact that in ncurses, it is not possible to view output printed to the console during runtime as a form of debugging. The alternative to isolating code segments would be to change the printing function which did not always work due to the encapsulated, object-oriented nature of the design. This would also involve changing significant parts of the code and thus is not ideal.

Tests that involved interacting with robots were performed using a simulation, specifically `turtlesim` [23]. This simulation listens to and interprets the same messages as the robots used in department in a similar way. It also runs well which is ideal for testing scenarios involving opening and closing the simulation as well as viewing `ROS` output which proved to be challenging at first . However, when the project was mature in its development, tests were being performed on actual hardware.

4.2 Hardware Testing

Opportunities for testing on real hardware were unfortunately limited and reserved for the end of the project. Testing was carried out using Idris as the screen was already working for that robot. Idris also has plenty of safety features to avoid accidental collisions.

Other safety considerations and measures that were taken can be seen in Section 1.2.3. The overall goal of this testing was mainly preparation for the final demonstration and assurance that everything would work without trouble. The aspects of the system that were tested are described in the following sections.

4.2.1 All dependencies Satisfied

There are a number of dependencies that are required at compile time such as `yaml-cpp` and the `boost` libraries, among others. The only dependency not satisfied at the time of testing was `yaml-cpp` which was then subsequently installed on Idris.

4.2.2 Successful Code Compilation

There were no issues here aside from an error regarding the minimum version of CMake required. This was resolved by simply changing the requirements in the CMake configuration file.

4.2.3 User Interface Displaying Properly

In some terminal emulators, the user interface does not display as it should. This is potentially due to character encoding errors. Prior to testing on Idris itself, the application was tested in various terminal environments, the results of which are shown below:

Test Environment	Pass/Fail
`st` [24]	Pass
`xterm` [25]	Pass
`urxvt` [26]	Pass
The `GNOME` Terminal	Pass
`Konsole` (KDE Terminal) [27]	Fail
Plain `TTY` running on `Arch`, `Parabola` and `Ubuntu` Linux	Pass
Tmux running in a plain `Ubuntu TTY`	Pass

Table 4.1: Test results of running application in various terminal emulators

Idris is running a plain `TTY` in `Gentoo` Linux which was able to display the application properly with no issues. There were also concerns regarding whether or not the sidebar would be too big as the width is hard coded and the mounted screens are reasonably small. However, the resolution of the screen was appropriately sized meaning that all output in each window looked sensible.

4.2.4 Processes Terminating Correctly

The particular processes used on Idris had not yet been tested (nor were they known exactly beforehand) and so it was important to test managing them. This provides a good example of a real life scenario. Due to one of the issues outlined in Section 3.2.4, it was discovered through this testing that waiting for a process to close was causing problems. This lead to multiple processes having to be painstakingly killed manually from a separate computer.

4.2.5 Appropriate Signals Sent from the Teleoperation Tab

This test lead to the discovery of the issue detailed in Section 3.3 regarding the teleoperation tab sending too many signals in rapid succession. The consequence of this was very noticeable latency when monitoring the velocity topic using `rostopic echo`.

4.3 User Interface Testing

The UI was tested using some example programs in Appendix C, most notably the second example. This helped to test how the scroll bars were being drawn as well as scrolling in general by changing the amount of lines printed. Additionally, some third party programs were used such as the `GNU ls` command as well as `neofetch` [28]. The `ls` command was used as a simple test with a usually small output stream that was guaranteed to stop outputting new lines. `Neofetch` was used for similar reasons as well as having the ability to test how colour output is dealt with.

`Neofetch` changes several attributes (such as coloured and bold text) for some lines, creating several instances of concatenated escape codes which is an unusual test case for omitting these codes generally. This served to test how robust the printing and output capturing systems were due to its complexity. These programs were included in the configuration file and were run as part of regular testing for new features.

4.4 Stress Testing

The application is deliberately very minimal in order to deal with multiple process output streams without impacting performance. However, processing this output is done in real time in one loop so stress testing the refresh rate of the program is important. The main form of stress testing employed was an example program in Appendix C which prints numbers incrementing from 0 to any arbitrary amount infinitely. Several instances of this program were run at one time within the application. This allowed for testing the refresh rate of the printing function as well as how the program dealt with updating several tabs at a high refresh rate.

There used to be no delay used in this program so the update rate was as fast as the hardware allowed it. However, this lead to 100% of the CPU thread being used during testing and was dealt with using a five millisecond delay. A higher delay makes the interface feel sluggish and any smaller delay greatly increases the load on the CPU. The current load sits at approximately 20% on an `Intel i7 4790K` at `4.4GHz` running `Ubuntu`.

4.5 User Testing

Each sprint focused on delivering demonstrable features that have an observable impact on the application from the perspective of the user. At the end of a sprint, during the weekly meeting with the project supervisor (who is also the client), there was an opportunity for them to try newly developed features on a laptop. Although, later in the project, this became impossible as the application required `ROS` in order to run which was not installed on said laptop. This was mainly due to the difficulty of installing `ROS` on Linux distributions not based on `Ubuntu` such as `Arch`.

The replacement for this was to show screen casts of the new features alongside commentary and further discussion. These screen casts included a log of every keystroke using `screenkey` [29] in order to make it clear what is happening in the video. These demonstrations usually lead to new features/requirements being added due to a better understanding of how the software was developing from a usability perspective. User tests were deliberately frequent and varied in order to respond to change more effectively and to test the latest and more appropriate aspects of the system.

Chapter 5

Evaluation

5.1 Evaluation of Agile Approach

Due to the uncertainty of how the requirements would manifest as well as the knowledge that plans would inevitably change, an agile approach seemed most appropriate. This initial decision proved to be most effective as it allowed for rapid prototyping and proving concepts before moving forward with potentially uninformed rigorous design. This was also closely aligned with the fact that meetings with the project supervisor occurred regularly and frequently. Given that the supervisor was also the client/user of the application, it was suitable to show regular prototypes as a form of user testing. The ability to respond to change combined with the technical experience of the project supervisor meant that the correct decisions were taken even without a formal design.

5.2 Identifying the Requirements

The requirements in Section 1.2.1 were identified at the start of the project and remained consistent throughout the project. However, these requirements do not describe the entire system in detail as they are specifically user-oriented stories. The advantage of this is a greater focus upon implementation time, however, the downside is the lack of direction from the offset. This could have been improved and may have helped to mitigate issues faced during development. However, the approach was (and is) viewed as more appropriate for the individual project undertaken.

5.3 Design Decisions

As an agile projects, it was expected that some design decisions would have to be remedied later on in the project. This is due to change in requirements from the client, dealing with integration issues or responding to a change in understanding of the technologies used. Such examples have been documented in earlier sections, namely 2.2, 2.4, 3.2.2

and 3.2.3. However, these issues were appropriately dealt with and are only a subset of design decisions that were made. Other various decisions outlined in previous sections such as 2.5 required no changes and the substance of what was produced suffered no ill effects due to design.

5.4 Appropriateness of Tools Used

The C++ language was the most appropriate tool due to the reasons outlined in Section 2.2. The decision to use ncurses was also appropriate over the other alternative tools noted in Section 1.1.2. It is certainly more appropriate than any graphical library for the purposes of the project, despite its low-level and verbose nature.

In terms of editing tools, the choice to use neovim over a more fully featured IDE arguably had some impact on development time. However, the only tangible benefit gained from using an IDE for this project is automatic code completion and refactoring. Using any integrated console is unsuitable as it would make it harder to view the application running. This means that many of the features of an IDE would be unavailable without spending time trying to make it work. Neovim and other similar editors are also much faster regardless of hardware specifications in many cases. Therefore, the appropriateness of this tool is not entirely non-existent.

The appropriateness of using YAML as well as the yaml-cpp library is noted in Section 3.2.1. Using other formats or creating a custom parser would have been less suitable and more time consuming.

5.5 Relevance to Degree Scheme

The relevance of this project to my degree scheme (Artificial Intelligence and Robotics) was questioned before the start of the semester. This was primarily due to the lack of robotics involved in the initial specifications of the project. The argument was made that it was more associated with either a software engineering or system management project. Even if the application will be running on robots, the substance of the task did not involve interacting with them them. Due to this decision, it was decided to include additional requirements. These were to create a way to operate the robot from the application as well as display ROS topic information.

5.6 Application and Impact

As evidenced by the relatively successful hardware testing as well as feedback from the project supervisor/client, the usefulness of the developed software is certainly apparent. The fundamental requirements were satisfied and function correctly and the application itself is easy to use and runs effectively. There were other discussions during meetings

regarding optional features that would make the system more like a pseudo desktop environment. Features such as colour-coded tabs and tabs being opened using specified shortcut keys would definitely add to the overall usability of the application. However, these features were strictly classified as optional and were reserved for the end of the project if there was any spare time remaining.

The impact of the completion of this project has the potential to be very beneficial for day-to-day research, providing more flexibility and redundancy for controlling the rovers. There is also the possibility of this software being used for other non-robotics systems if the ROS features were removed and the terminal emulation was more robust. It can be used for demonstrations or possibly a normal terminal multiplexer with built-in executable shortcuts.

5.7 Time Management

At the beginning of a sprint, tasks were chosen in accordance with the assessment of velocity. As a result, it was foreseen how much should be completed within the sprint which provided a measure of how well time was being managed. On a day-to-day basis, due to the ability to carry out work at any location (ie. from home) for most of the project, it was possible to start working first thing in the morning. This provided motivation to complete more work as the day progressed and resulted in more intense focus on the allotted tasks at hand. There were unfortunately long periods where little to work was completed due to illness. However, on days where the usual methods of working were not sufficient, techniques such as the pomodoro technique [30] were employed to help manage time.

There was no use of a schedule which has advantages and disadvantages. The obvious disadvantage is the inherent disorganisation that comes with a lack of a formal schedule. However, an advantage of this is the ability to have more intense focus on the project as a whole instead of constantly switching between tasks. Also, this approach provides plenty of opportunity to prioritise and re-organise work if this is indeed necessary. These reasons create a time management methodology that maintains focus and ultimately fits into the agile nature of the project.

Due to these methods, time management was adequate enough to finish the main project. However, some steps could have been taken to manage time more effectively with more order and organisation to the day. That being said, what was employed was suitable and necessary for me as an individual as opposed to alternative approaches that may not have provided any benefit to me.

5.8 Future Work

As noted in Section 5.6, there is the potential for this system to be a fully complete terminal emulator with its own truly unique features. The elements to be addressed in order to achieve this are noted below:

5.8.1 Extra Usability Features

Section 5.6 describes optional features that were suggested such as shortcut keys. Providing other way to interact with the application can make it more usable as a replacement for a graphical desktop environment. This is important for both other use cases as well as the intended use on robots if it is used for long periods of time.

5.8.2 Signal Handling

The application currently only displays standard output and standard error. There is no way to accept standard input from a child process as this was not necessary for the original intended purpose. This is the main reason why it is ineffective as a terminal emulator and would take high priority on the list of additions. There are also many other signals to emulate which would have been impossible to implement within the scope of the project.

5.8.3 More Customisability

The key-bindings are currently hard coded and are not configurable. This would be a valuable feature to add in order to accommodate for different user preferences. Also, there should be multiple keys mapped to the same function so that people collaborating can use their own layouts simultaneously without confusion.

Also, due to time constraints, the monitoring tab is hard coded which means that it is potentially useless on other machines without the same messages. An ideal system would allow easy customisation using any potential message type with any number of menu items.

5.8.4 Help Menu

Something that was brought up during testing was the addition of a help menu for new users. This is somewhat related to the lack of key customisation as different users will expect different default settings based on their own experience.

Annotated Bibliography

[1] “Tmux github home page.” [Online]. Available: https://github.com/tmux/tmux/wiki

Home page for the terminal multiplexer Tmux which was used as inspiration for parts of this project

[2] “ranger home page.” [Online]. Available: https://ranger.github.io/

Home page of the file manager ’ranger’ used as inspiration for the design of the user interface

[3] “Ncurses man page.” [Online]. Available: https://invisible-island.net/ncurses/man/ncurses.3x.html

The official manual for ncurses usage. This is also freely available on Unix systems. It provides compilation instructions, basic usage and a complete list of functions with their own man pages.

[4] “Termbox github page.” [Online]. Available: https://github.com/nsf/termbox

Github page for termbox, a TUI programming library similar to ncurses

[5] D. Araps, “Bash tui guide.” [Online]. Available: https://github.com/dylanaraps/writing-a-tui-in-bash

A guide showing how to write a non-graphical interface using pure bash

[6] “Cdk home page,” 1999. [Online]. Available: https://invisible-island.net/cdk/

The homepage for Curses Development Kit information and downloads. There is also information on ncurses in general.

[7] “Ncurses programming howto,” 2005. [Online]. Available: http://www.tldp.org/HOWTO/NCURSES-Programming-HOWTO/

This website provides an extensive tutorial on NCURSES programming and features. It includes detailed explanations of core concepts and basic examples of these as well as more advanced libraries and techniques.

[8] “Ros.org.” [Online]. Available: www.ros.org

The official ROS website homepage. The Robotics Operating System is a middlewear for programming and controlling robots and is available for Linux.

[9] "Neovim homepage." [Online]. Available: https://neovim.io/

Neovim is a fork of the modal text editor vim

[10] "Synthing homepage." [Online]. Available: https://syncthing.net/

Syncthing is an open source cloud service used for file synchronisation.

[11] "Yaml-cpp github page." [Online]. Available: https://github.com/jbeder/yaml-cpp

Yaml-cpp is a C++ library for reading YAML files and is also used by ROS

[12] "Exec manual page." [Online]. Available: https://linux.die.net/man/3/execv

The exec family of functions are used to run separate executables within a C program

[13] "Boost libraries hom page." [Online]. Available: https://www.boost.org/

The boost libraries provide useful utilities for C++ programming and are available under the Boost License

[14] "Forkpty manual page." [Online]. Available: https://linux.die.net/man/3/forkpty

Forkpty is used to fork a C program while simultaneously creating a pseudo-terminal in order to streamline the process

[15] "Manual page for pseudo-terminals." [Online]. Available: https://linux.die.net/man/7/pty

The man page for pty's describing their usage and history

[16] D. Krasner, "tty8 github page." [Online]. Available: https://github.com/dimkr/tty8

Github page for a simple terminal multiplexer used as inspiration for parts of this project.

[17] "Regex c++ library." [Online]. Available: http://www.cplusplus.com/reference/regex/

Regex is a part of the C++ standard library and is used to manipulate strings using regular expressions

[18] "Regex tester." [Online]. Available: https://www.regexpal.com/

Online tool for visualising the effects of regular expressions on test data

[19] "Popen man page." [Online]. Available: https://linux.die.net/man/3/popen

The manual page for the popen function under the stdio.h C library

[20] "fdopen man page." [Online]. Available: https://linux.die.net/man/3/fdopen

fdopen is used to open a given file descriptor as a file stream in C

[21] "Pkill man page." [Online]. Available: https://linux.die.net/man/1/pkill

The manual page for the pkill UNIX utility

[22] "Ros package ros_type_introspection." [Online]. Available: https://wiki.ros.org/ros_type_introspection

Package information regarding shapeshifters and other generic message utilities

[23] "Turtlesim." [Online]. Available: https://wiki.ros.org/turtlesim

The ROS package webpage for the package turtlesim used for testing

[24] "The suckless terminal." [Online]. Available: https://st.suckless.org/

ST is a minimal terminal emulator for X

[25] "Xterm." [Online]. Available: https://invisible-island.net/xterm/

xterm is a terminal emulator for the X window system

[26] "Rxvt-unicode arch wiki page." [Online]. Available: https://wiki.archlinux.org/index.php/Rxvt-unicode

An X terminal emulator forked from rxvt

[27] "Konsole." [Online]. Available: https://konsole.kde.org/

Konsole is the default KDE terminal application

[28] D. Araps, "Neofetch github page." [Online]. Available: https://github.com/dylanaraps/neofetch

Neofetch is a command line system information tool that displays colour formatted textual output

[29] "Homepage for the program screenkey." [Online]. Available: https://www.thregr.org/~wavexx/software/screenkey/

Website for screenkey, a program used to display on screen what keystrokes are being pressed

[30] F. Cirillo, "The pomodoro technique." [Online]. Available: https://francescocirillo.com/pages/pomodoro-technique

Primary information page on the pomodoro time saving technique

[31] "The 3-clause bsd license." [Online]. Available: https://opensource.org/licenses/BSD-3-Clause

This is a template describing the usage of the 3-clause BSD license

[32] “Ncurses license.” [Online]. Available: https://invisible-island.net/ncurses/ncurses-license.html

A comprehensive description of the ncurses license regarding its history and complications

[33] “tty8 liscense.” [Online]. Available: https://github.com/dimkr/tty8/blob/master/COPYING

Licence for the tty8 terminal multiplexer included in the respective Github repository. This software uses the MIT license.

[34] “Yaml-cpp license.” [Online]. Available: https://github.com/jbeder/yaml-cpp/blob/master/LICENSE

Licensing and copyright information for yaml-cpp

[35] “Boost license.” [Online]. Available: https://www.boost.org/LICENSE_1_0.txt

The licensing for the Boost C++ libraries

Appendices

Appendix A

Third-Party Code and Libraries

Boost C++ Libraries - This was used to parse and split string data from a YAML file. Version 1.69.0 was used. The library is open source and is available from the Boost Organisation. [13]. The library is released under the Boost license [35]. This library was used without modification.

yaml-cpp library - This was used to parse a configuration file using the YAML syntax. Version 0.6.2 was used. The library is open source and is available from the yaml-cpp Github Page. [11]. The library is released under the MIT license [34]. This library was used without modification.

ROS - This was used to interact with robotic systems also running ROS. The library is open source and is available from the ROS Website. [8]. The library is released under the 3-Clause BSD License [31]. This library was used without modification.

ncurses - This was used to develop a text-based user interface. The library is open source and is available from the ncurses homepage. [3]. The library is released under a permissive MIT-style license [32]. This library was used without modification.

Appendix B

Ethics Submission

23/02/2019

For your information, please find below a copy of your recently completed online ethics assessment

Next steps

Please refer to the email accompanying this attachment for details on the correct ethical approval route for this project. You should also review the content below for any ethical issues which have been flagged for your attention

Staff research - if you have completed this assessment for a grant application, you are not required to obtain approval until you have received confirmation that the grant has been awarded.

Please remember that collection must not commence until approval has been confirmed.

In case of any further queries, please visit www.aber.ac.uk/ethics or contact ethics@aber.ac.uk quoting reference number **12234**.

Assesment Details

AU Status
Undergraduate or PG Taught

Your aber.ac.uk email address
sam82@aber.ac.uk

Full Name
Sam Matthews

Please enter the name of the person responsible for reviewing your assessment.
Reyer Zwiggelaar

Please enter the aber.ac.uk email address of the person responsible for reviewing your assessment
rrz@aber.ac.uk

Supervisor or Institute Director of Research Department

cs

Module code (Only enter if you have been asked to do so)
CS39440

Proposed Study Title
Customisable non-graphical interface to start/stop activities on mobile robots

Proposed Start Date
28/01/2019

Proposed Completion Date
03/05/2019

Are you conducting a quantitative or qualitative research project?
Mixed Methods

Does your research require external ethical approval under the Health Research Authority?
No

Does your research involve animals?
No

Are you completing this form for your own research?
Yes

Does your research involve human participants?
No

Institute
IMPACS

Please provide a brief summary of your project (150 word max)
This is a software engineering project that aims to develop an application that will run on the large robots in the department in order to assist in process management during usage. The software also lets the user control the robot's movement from its onboard computer using keyboard or touch input.

Where appropriate, do you have consent for the publication, reproduction or use of any unpublished material?
Not applicable

Will appropriate measures be put in place for the secure and confidential storage of data?
Yes

Does the research pose more than minimal and predictable risk to the researcher?
No

Will you be travelling, as a foreign national, in to any areas that the UK Foreign and Commonwealth Office advise against travel to?
No

Please include any further relevant information for this section here:

If you are to be working alone with vulnerable people or children, you may need a DBS (CRB) check. Tick to confirm that you will ensure you comply with this requirement should you identify that you require one.

Yes

Declaration: Please tick to confirm that you have completed this form to the best of your knowledge and that you will inform your department should the proposal significantly change.
Yes

Please include any further relevant information for this section here:

Appendix C

Code Examples

3.1 Altering Child IO Signals

```
/*These are a modified versioa of functions taken from the tty8 project
Copyright (c) 2015 Dima Krasner*/

/*Found in the Process class*/
int flags = fcntl(pty,F_GETFL);
fcntl(pty, F_SETSIG, SIGRTMIN + 1);
fcntl(pty, F_SETFL, O_ASYNC | O_NONBLOCK | flags);
fcntl(pty, F_SETOWN, pid);

/*Found in main*/
void setMask(sigset_t * m, sigset_t * o) {
  sigemptyset(m);
  sigaddset(m, SIGCHLD);
  sigaddset(m, SIGINT);
  sigaddset(m, SIGTERM);
  sigaddset(m, SIGWINCH);
  sigaddset(m, SIGRTMIN);
  sigaddset(m, SIGRTMIN + 1);
}
```

3.2 Example Configuration File

This YAML file was used as testing data throughout the project.

```
command:
  - name: "list"
    exec: "/bin/ls"
```

```
    param: "-l"
  - name: "example"
    exec: "example"
  - name: "echo"
    exec: "/bin/echo"
    param: "Hello World"
  - name: "scrolling"
    exec: "scrolltest"
    param: "50"
  - name: "turtle"
    exec: "/opt/ros/melodic/bin/rosrun"
    param: "turtlesim turtlesim_node"

teleop:
  topic: "turtle1/cmd_vel"
  degradation: 0.01
  turning_increment: 0.5
  linear_increment: 2
  refresh_timeout: 0.5
  degradation_timeout: 0.05

monitoring:
  telemetry_topic: "/idris/telemetry"
  safety_topic: "/idris/safety_status"
```

3.3 Test Programs

The following are example programs used for testing the main application and were written separately from it.

3.3.1 Example Program 01

```
/*An example program used for stress testing the application*/

#include <stdio.h>
#include <unistd.h>

int main() {
  int i = 0;
  while(1) {
    printf("%d\n",i);
    i++;
  }
}
```

3.3.2 Example Program 02

```
/*An example Program used to test scrolling functionality*/

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char * argv[]) {
  if (argc < 2)
    printf("No args\n");
  else {
    for(int i = 0; i < atoi(argv[1]); i++)
      printf("%d\n",i);
  }
  return 0;
}
```

3.3.3 Example Program 03

```
/*An example program used for testing file reading and YAML parsing*/

#include <yaml-cpp/yaml.h>
#include <iostream>
#include <string>
#include <vector>
#include "Process.h"
#include "FileReader.h"

main() {
  FileReader reader("test.yaml");
  std::vector<Process> processes = reader.getProcesses();
  processes[0].start();
  while(1) {
    processes[0].refreshBuffer(100);
    for (auto it : processes[0].getBuffer()) {
      std::cout << it;
    }
  }
}
```

Appendix D

Diagrams

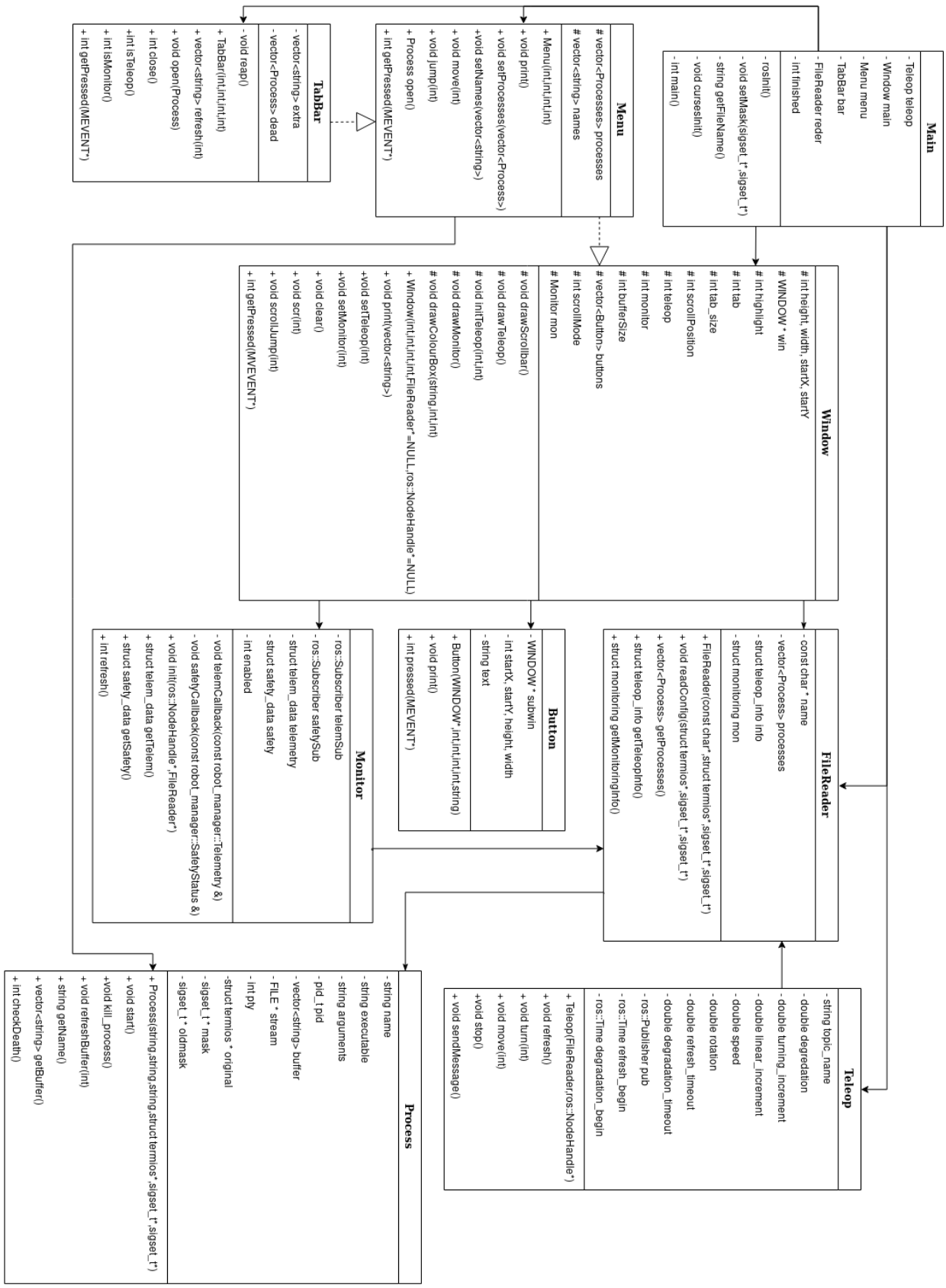

Figure D.1: UML class diagram of the final system

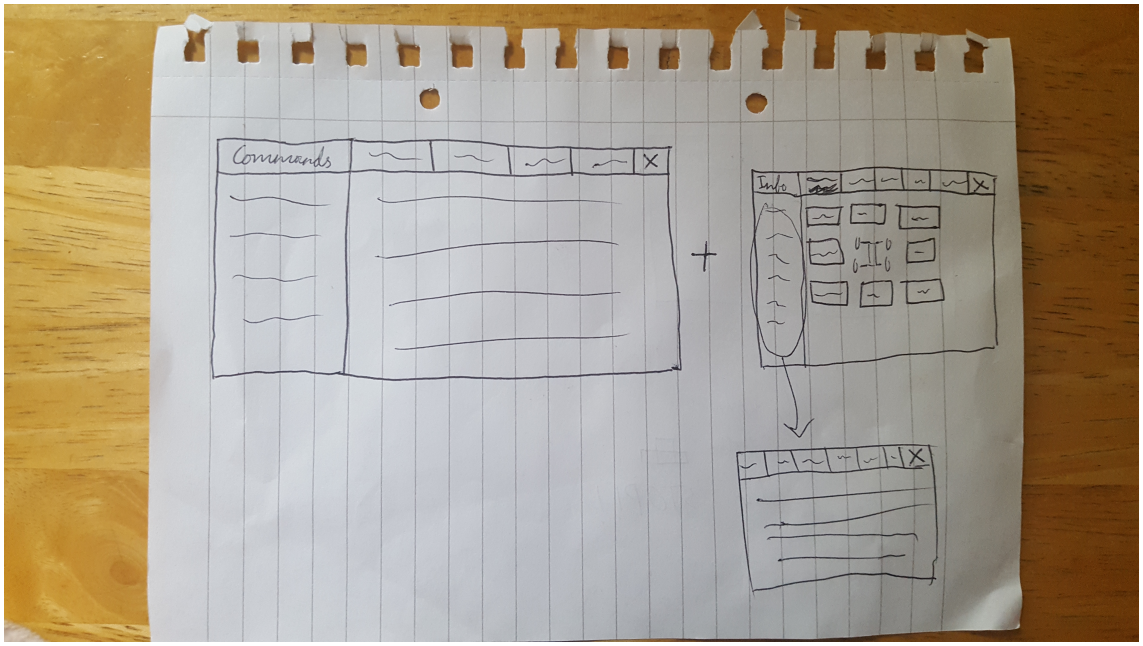

Figure D.2: Initial drawing of what the user interface was planned to look like

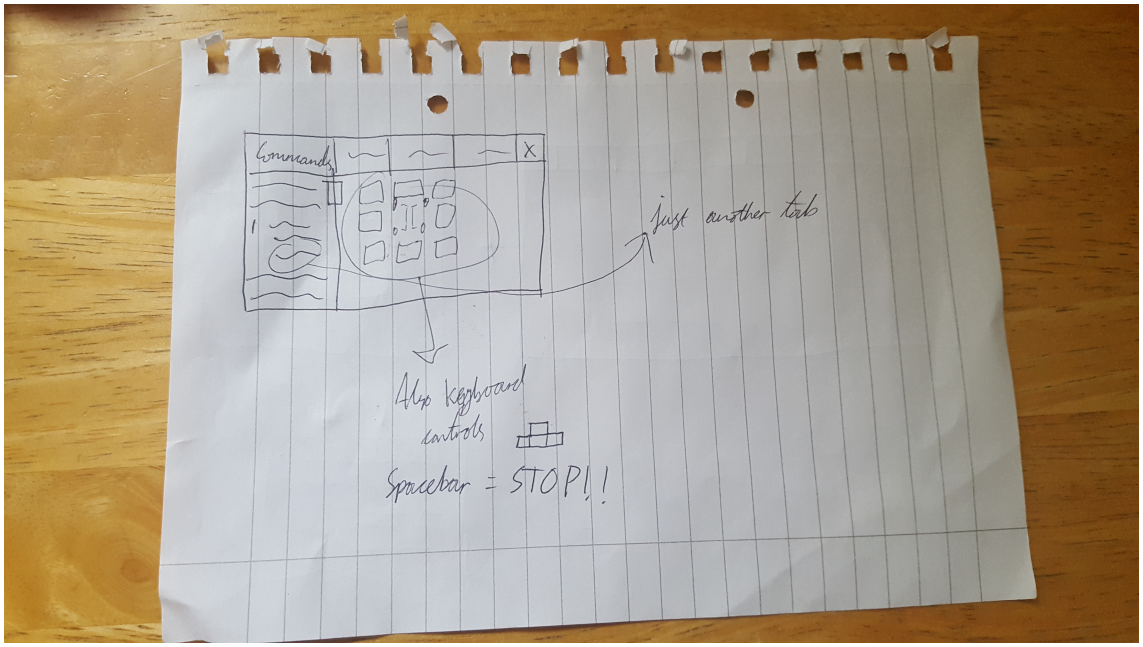

Figure D.3: Altered design to account for changes in client requirements and to have as much as possible on one screen