Remote control of Autonomous Surface Vessels

Aaron Goldsworthy, 21108324

October 31, 2018

Supervisor
Dr Thomas Bräunl

Abstract

This thesis concerns two different autonomous surface vehicle (ASV) platforms, the first being the Solar Powered Autonomous Boat (SPAB) constructed as a previous student project at UWA and the second being a Liquid Robotics Wave Glider owned by L3 Oceania. The challenges and approaches involved in both projects differed but the ultimate goal in both was create and enhance the remote control and automation capability of the two vehicles. Both vehicles are fitted with 3G mobile telecommunications systems, and the WG is additionally fitted with an Iridium satellite communications modem. In the case of the Wave Glider, remote communications are achieved using a control protocol using simple messaging service (SMS) over 3G and Iridium. In the case of the SPAB, autonomous control is done using the mobile data capability of a 3G modem. The SPAB regularly logs telemetry data, retrieves commands and parameters from a remote control server using a Representational State Transfer (REST) application programming interface.

Acknowledgements

This research is supported by L3 Oceania Pty Limited, Fremantle WA

Word count: 7281

Contents

List of Figures

List of Tables

Listings

Table 1: Table of Abbreviations

Abbreviation	Definition
API	Application Programming Interface
CGI	Common Gateway Interface
GPIO	General Purpose Input/Output
HTTP(S)	Hyper Text Transfer Protocol (Secure); Common document transfer protocol
INS	Inertial Navigation system
IP	Internet Protocol
JS	Java Script; programming language supported by web browsers
JSON	Java-Script Object Notation; a method of describing binary data with text
KF	Kalman Filter
MATLAB	MATrix LABoratory; program for numerical analysis
PPP	Point-to-Point Protocol
REST	Representational State Transfer; a web application architecture
RFC	Request For Comments; document describing a technical standard
RPi	Raspberry Pi
SLIP	Serial Line Internet Protocol
SMS	Simple Messaging Service
SOAP	Simple Object Access Protocol
SPAB	Solar Powered Autonomous Boat
TLS	Transport Layer Security; A protocol which provides encryption and server authentication. Replaced Secure Sockets Layer (SSL)
UDP	Universal Datagram Protocol; Connectionless Internet protocol
USB OTG	USB On The Go; a USB device that can act as both a master and a slave
WAFO	Wave Analysis for Fatigue and Oceanography; module for MATLAB
W3C	World Wide Web Consortium
WG	Wave Glider
WGMS	Wave Glider Management System
XML	Extensible Markup Language

Part I

Solar Powered Autonomous Boat

1 Introduction

1.1 Completed Work and Background

The UWA Solar Powered Autonomous Boat (SPAB) is an ongoing masters student project under the supervision of Thomas Bräunl. Initial construction, automation and short range wireless communication has been completed by students John Hodge and John Borella. Hodge's design report[1] and Borella's design proposal[2] both provide detailed descriptions of these systems. The main design features are differential drive using two electric propeller modules, a marine grade solar panel, a battery management system for solar panels, 4 sealed lead acid batteries and automation using an Ardupilot micro-controller. The SPAB itself is pictured in Figure 1.

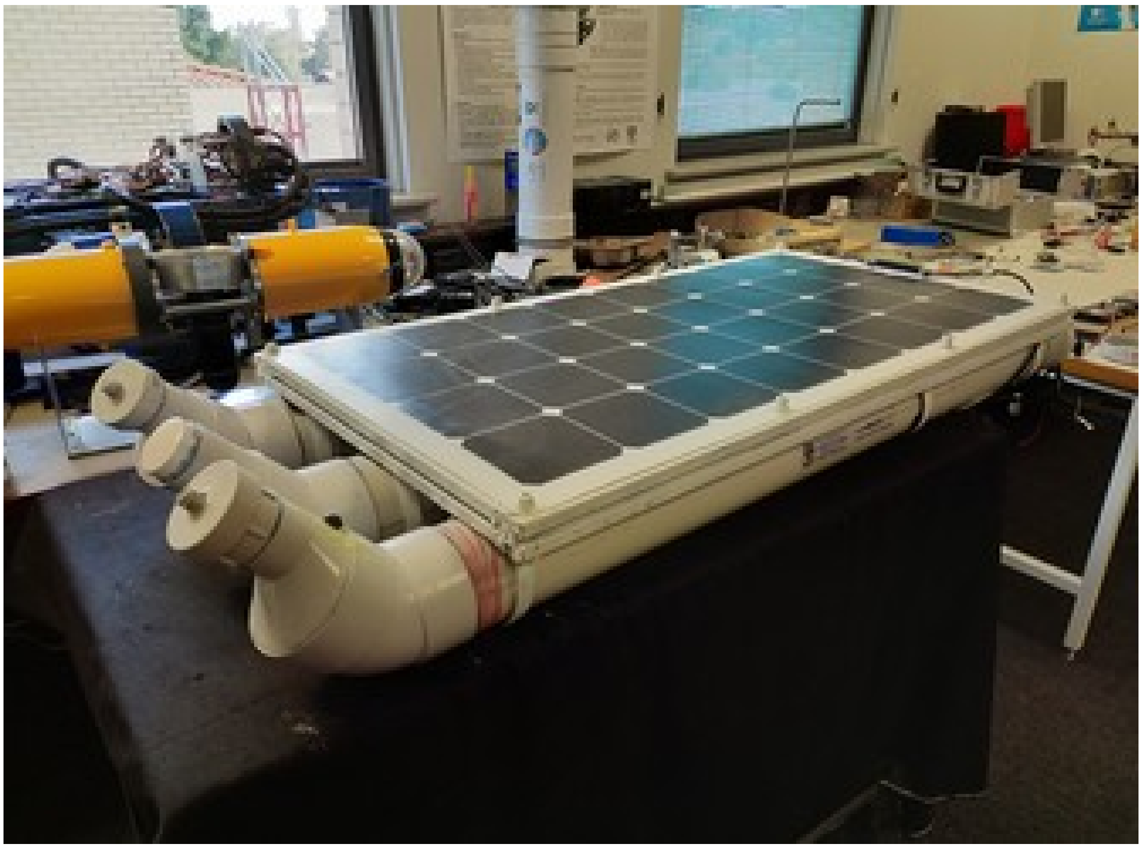


Figure 1: The UWA Solar Powered Autonomous Boat

1.2 Problem Statement

The goal of the SPAB project is to create a long range, endurance, autonomous water craft for research purposes. The solar powered design so far has been conducive to that aim. However, in order to maximize the use of the long mission life a method of updating missions and retrieving mission data needs to be established over long range communications without user action. Without this, the platform is limited in its long range capabilities due to the requirement of having an operator monitor it nearby. The goal of this part of the project is then to create a system enabling the SPAB to communicate over mobile networks and eventually over satellite networks or any other way that internet access can be obtained. In order to achieve this some additional processing and communications hardware was added along with the necessary server and client components. These three broad components are covered in sections 2, 3 and 4.

1.3 Literature Review and Similar Projects

1.3.1 REST

REST is a stateless protocol that is designed to be compatible with and make use of HTTP methods to upload and download data, typically as JSON [3]. The standards governing REST are numerous and extensive and are best summarized at https://restfulapi.net/. REST was originally conceived by Roy Fielding in his 2000 dissertation [4]. Fielding described his REST architecture as being designed around leveraging the constraints of the World Wide Web. Those constraints and therefore principals of a good REST API are:

1. Uniform interface - Resources should be synonymised with a web page.
2. Client-server - Client application and server application must be able to evolve separately without any dependency on each other.
3. Stateless - Each and every request is treated as new.
4. Cacheable - Caching should be applied to resources when possible.
5. Layered System - Different layers can be handled by different physical systems.
6. (Optional) Code on Demand - Executable code can be returned.

1.3.2 SOAP

SOAP is a competing web application standard based on XML. SOAP did not achieve RFC standard status but a specification is maintained by the W3C [5]. SOAP consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined data types, and a convention for representing remote procedure calls and responses. SOAP is designed to be protocol independent, but is primarily associated with HTTP. Using SOAP was undesirable due to increased complexity compared to REST.

1.3.3 Model-View-Controller

MVC is a general software engineering architecture for designing user interactive systems. It is not a standard per-se rather an design pattern[6]. It splits an application into the components: The Model, the View and the Controller. I received influence from the MVC architecture to design the SPAB web server. The Model refers to a unified method of storing the current state of the interactive system. All other interfaces interact with the model. The View, refers to how information is presented to the user and the Controller refers to the method in which the user acts on the model. By splitting the application into these three components independence and encapsulation can be achieved. For example different visualizations of the same data can be done by having multiple different Views and multiple users can interact with multiple Controllers.

2 Hardware Upgrades

Some additional hardware has been added to the SPAB in order to support the integration of remote command and data logging from the SPAB. Figure 2 and 3 displays the location and mounting of the new hardware components.

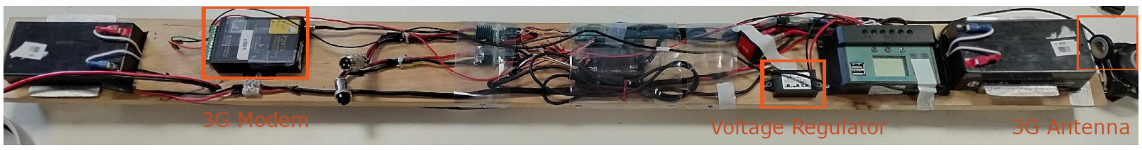


Figure 2: New hardware components

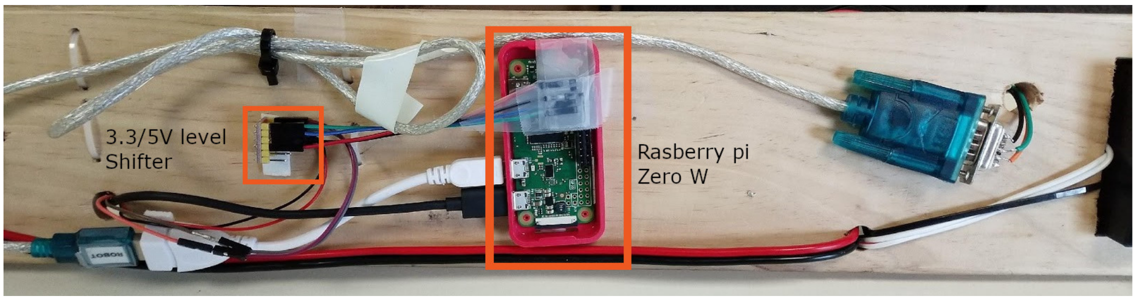


Figure 3: New hardware components

2.1 3G Modem

A F2414 Four-Faith 3G modem was added as the remote communications radio. 3G was selected as it is known that 3G coverage extends around Rottnest Island and because the 2G network in Australia is being decommissioned. This modem is designed for transparent remote monitoring of sensors.
This modem is configured in 'TELNET' and data trigger mode. In TELNET mode, the modem acts as a transparent and complete network and transport stack. Bytes which arrive at the serial interface are sent to remote location without being altered. In data trigger mode, the modem remains in standby mode until it receives the byte sequence 'don'. When it receives that trigger it connects to the remote server and begins transmitting data. At the end of the transaction the connection is torn down by the remote server as is typical of Apache web server. In order to simplify development, this modem is managed in software using the python class `F2414Modem.py`. Documentation for this class can be found within the python module as doc-strings. The modem is best configured using a Windows application provided by the manufacturer and downloaded for free from the product page listed in Table 2. A text file `ModemConfig.txt` found at the local server repository in Table 2 can be used by the configuration application to restore the configuration of the modem if the configuration becomes undone.

2.2 Raspberry Pi

A Raspberry Pi Zero W has been added to the SPAB. This RPi was added for a number of reasons. The first reason is to make interfacing the with the 3G modem easier and secondarily to interface with other sensors in the future, for example salinity and temperature sensors. The RPi is powered using a 12V to 5V USB regulator. The modem is connected using a USB to serial cable and the Ardupilot and is connected to the RPi using the UART found in the GPIO header. Between the Ardupilot and the RPi is a 5 V to 3.3 V logic level converter. It is important that this component remains in place as the RPi will be damaged if exposed to voltage levels above 3.3V on the GPIO pins.

3 Remote Server Creation

The remote management server is implemented as a stack of the Linux, Apache, SQLite3 and Python components. Together these components form a REST API. Section 13 describes a complete tutorial on bringing these components together to form a functional server on a host purchased from a third-party. Not all possible features of a REST API have been implemented so far, but having the architecture of the system designed around REST means that additional REST features could be implemented without having to change the architecture. The architecture of components are outlined in figure 4. All code and documentation for the remote server can be found in Table 2.

3.1 JSON Message Format

The SPAB application client should issue POST requests containing logging data to `data.cgi` in JSON either periodically, when it reaches a way point or both. An example for this format is found in Listing 1. Similarly, the SPAB application client should perform a GET request to

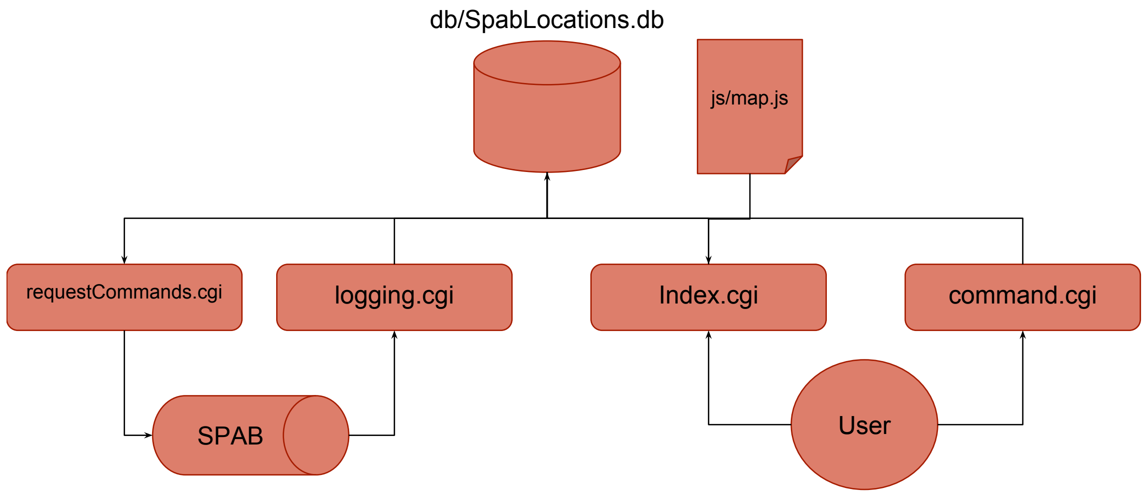


Figure 4: Remote Server Architecture

`requestCommands.cgi` periodically in order to receive new commands from the server in JSON. An example command JSON string is given in Listing 2.

```
[
   {
   timestamp: 1536551550,
   latitude: -31.9505,
   longitude: 115.8605,
   temperature: 301.3
   salinity: 35.5
   },
   {
   timestamp: 1536552678,
   latitude: -30.1505,
   longitude: 116.8705,
   temperature: 299.1
   salinity: 35.5
   }
]
```

Listing 1: Logging JSON

```
[
   {"type":"command"},
   {
   id: 1,
   action: "moveTo",
   timestamp: 1536551550,
   latitude: -31.9505,
   longitude: 115.8605
   },
   {
   id: 2,
   action: "moveTo",
   timestamp: 1536552678,
   latitude: -30.1505,
   longitude: 116.8705
   }
]
```

Listing 2: Command JSON

3.2 Index.cgi

This is the first page presented to the user accessing the web interface. It displays the history of recorded locations on an interactive JS map. The map is created using the Leaflet.js plugin. Information on how to develop with this plugin can be found at `https://leafletjs.com`.

3.3 Command.cgi and requestCommands.cgi

`Command.cgi` presents a form for the user to enter commands to be stored in the web server database. Commands stored as pending in the database will be given to the SPAB the next time the SPAB requests new commands from `requestCommands.cgi`.

3.4 data.cgi

This page is primarily for interaction with the SPAB only. This is the page that handles logging data from the SPAB and stores it in an SQLite3 database.

3.5 db/spabLocation.db

This SQLite3 database contains all location, command and other data that might be of interest to the SPAB or web application. An additional script file initSpabDB.sql found in the remote server repository can be used to recreate this database if required. It is also useful for getting an idea of the database schema.

4 Local Server Creation

The client application than runs on the RPi embedded in the project is split into three main parts. The Telemetry Manager, the Mavlink manager and the SPAB model. Each of these respective parts is covered in detail in this section. In addition to the python scripts, some Linux system configuration needs to be done to ensure correct functioning of the application. The current architecture of the python client is shown in Figure 5.

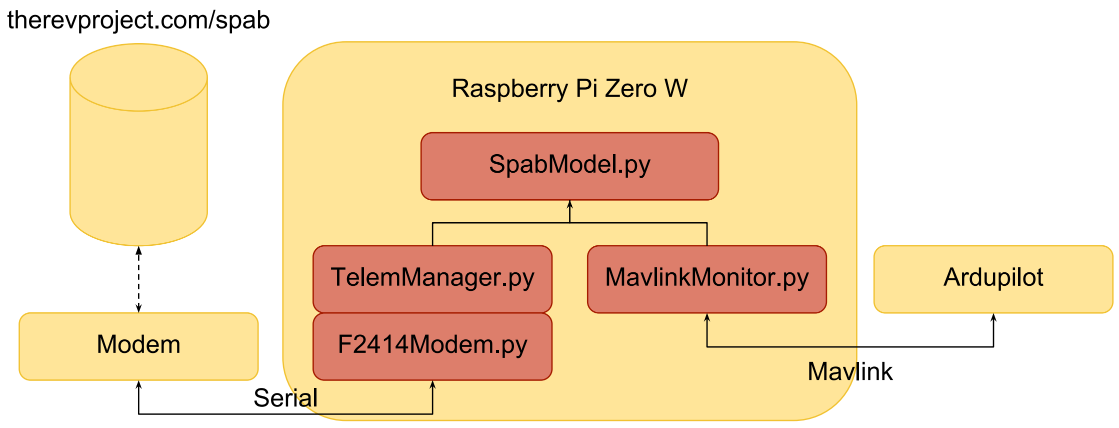


Figure 5: Client Server Architecture

4.1 Firmware Repository

The Ardupilot firmware is based on the work of previous students work. I have repackaged and re-hosted this firmware at the location indicated in 2 in order to simplify the build process. The previous version required a number of bug fixes to the code, checking out older commits and patches applied to the code that are no longer used. This version of the firmware is hard forked from the original Ardupilot repository at the appropriate commit and has all the relevant patches included. In order to upload to the Ardupilot on the SPAB, connect to the Ardupilot using a micro-USB cable. Specify the location of the USB port using `export PORT=/dev/ttyACM0` on a

Linux system. Then use `make apm2-upload` while in the `APMRover2` directory to build and upload the firmware.

4.2 Telemetry Manager

The `TelemManager.py` class defines the actions the SPAB should take in response to the modem. It subscribes to the `F2414Modem.py` class, decodes the JSON from the command server and executes the corresponding commands. It also regularly packages logging data such as last GPS location and uploads it to the remote monitoring server.

4.3 Mavlink Manager

communication with the Ardupilot is performed using the Mavlink protocol described in this online documentation [7]. The pymavlink module installed as part of this repository closely implements the protocol and is the basis for the class which responds to mavlink messages. More message handlers can be created by writing a handler function and registering the function in the list of delegates.

4.4 Operating System configuration

There are a number of small tasks that need to be done in order to prepare the RPi. `Cron` is used to automatically start the client application as soon as it boots. This is done by running `crontab -e`, following the prompts and adding the task `@reboot /home/pi/spab.sh`. `spab.sh` is a Bash script which maintains the functioning of python application and redirects outputs to a text logging file. In order to use the RPi UART for a purpose other than the Linux console, the block `console=serial0,115200` needs to be removed from `/boot/cmdline.txt`. Add the blocks `enable_uart=1` and `dtoverlay=pi3-disable-bt` to the file `/boot/config.txt` this will ensure the hardware UART in enabled and the Bluetooth module is disabled for power saving reasons. `sudo systemctl disable hciuart` should also be run in order to disable the Bluetooth configuration service since the Bluetooth module is disabled. These steps are also covered in the `README.md` file for repository.
As the previously mentioned commands will detach the Linux console from the GPIO headers, it is recommended to develop on the RPi Zero by connecting a USB hub to the USB OTG port and attaching peripherals including the F2414 modem to the USB hub.

5 Results

Pictured in Figure 6 is a screen shot taken during a functionality test at Matilda Bay to the South of the UWA campus. In this test the SPAB began to successfully send telemetry data to the remote server as soon as it was powered on without any further interaction by the user. To confirm the commanding functionality, new commands were entered in the `command.cgi` page and the receipt of these commands was confirmed using the APM Planner application - A third-party real time application designed for the control of Ardupilot drones. The short range radio and APM Planner compatibility remains in place for short range control, however new commands from the remote server will overwrite commands sent by APM Planner.

6 Conclusions and Future Improvements

This project lays the ground work for development. While not all possible mission features have been added yet, the current implementation and architecture has proven robust on each occasion it has been tested. Extending each component should be possible without causing incompatibility with any of the other components, allowing the task to be split among a number of future students. Many possible improvements to the SPAB could be done in the future with the following being a sample.

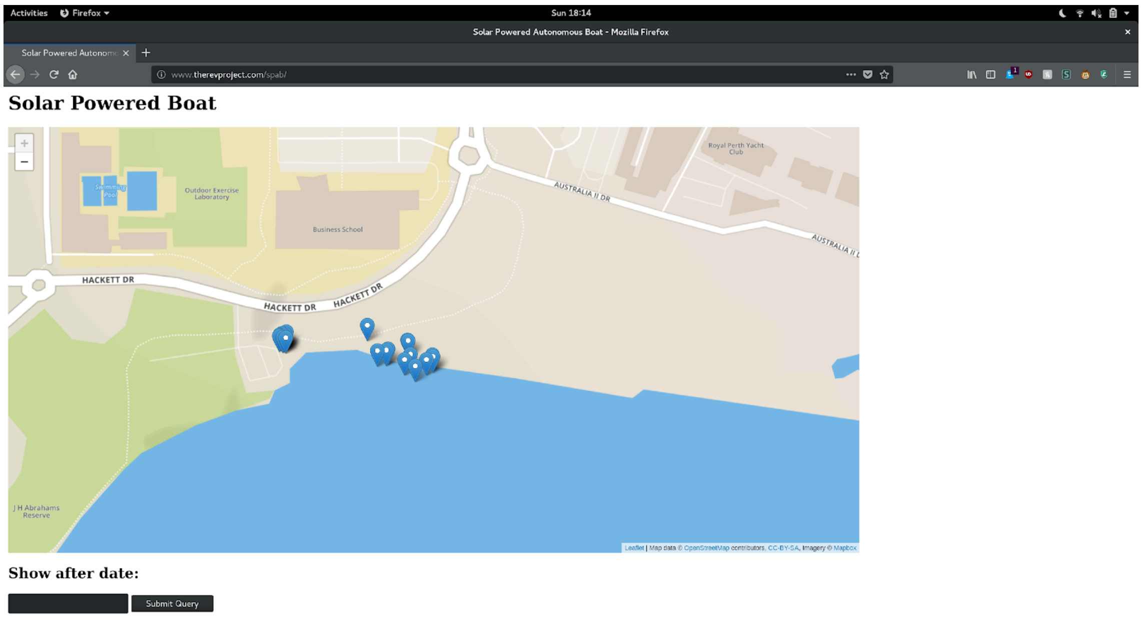


Figure 6: Remote Telemetry From SPAB

6.1 Suggested Hardware Improvements

The F2414 modem presents some limitations in that it is both a transport and network layer device. It is not very flexible and most python libraries expect to act on a network interface instead of a serial port. It may be possible to emulate a network interface using a serial connection using either a SLIP or PPP compatible modem. It maybe possible to use SLIP with the F2414 modem by changing the configuration. Using this method, integration of security like encryption and the construction of HTTP messages could be simplified over the current method of manually constructing HTTP strings. SLIP is described in RFC 1055 [8].
Additional stasis sensors could be added. For example a humidity sensor could be used to potentially detect a hull breach. Battery voltage sensors would also be useful to give an indication of remaining mission time to the remote server.

6.2 Suggested Server Improvements

The application could be improved to provide filtering functionality, for example only showing markers in a certain date range, displaying additional information such as battery levels and sensor readings at each location. Achieving this will require multiple edits to both receiving pages `data.cgi` and displaying pages `index.cgi`. Currently, the only command supported by the `command.cgi` page is to go to a location. Additional commands and entire missions could be created from this page and sent to the SPAB provided there is an appropriate Mavlink approximation for the commands. The command interface could be made more interactive, for example a clickable map to set way points, rather than manually entering decimal degrees.
Security has so far been limited for the remote server. Currently all pages are public where only authorized users should be able to send commands to the SPAB. This could be achieved by adding HTTP Basic Authentication to the command page and additionally adding it to the logging page. Getting HTTP Basic Authentication to work through the F2414 modem has so far proved difficult, but it should be possible.

6.3 Suggested Client Improvements

The client application should be extended to handle more Mavlink packet types and extend the number of commands that can be written to the Ardupilot. Additionally, as new sensors are added to the SPAB the readings of these sensors should uploaded to the server.

7 Source and Resources

The following table 2, gives the location of all existing resources for this project. The 'Old Repository' contains the work of previous students in building the SPAB and may prove to be useful for future work. In particular it contains schematics, drawings, bill of materials and mechanical design documentation for the project. Other third-party resources previously mentioned in this document are also here for convenience.

Component	Location
Remote Server	https://github.com/formahult/Spab-cgi
Client Application	https://github.com/formahult/pyspab
Ardupilot Firmware	https://github.com/formahult/spab-pilot
Old Repository	https://github.com/thepowersgang/fyp-boat
This Document	https://www.overleaf.com/project/5bbae1bbfeddef2f871912c0
Leaflet	https://leafletjs.com
F2414 Modem	https://en.four-faith.com/f2414-wcdma-ip-modem.html

Table 2: Useful source Repositories

Part II
L3 Oceania Wave Glider

8 Background

The Liquid Robotics Wave-glider is an ASV with a unique method of passive propulsion. The Wave Glider consists of a surface float containing the main payload and a tethered underwater glider section. Hydrofoils on the underwater glider convert vertical motion of the float due to waves into forward motion as depicted in Figure 7. A rudder on the glider that can be used to change the course of the wave-glider is the sole actuator. The motion of the Wave Glider is therefore dependent on the surrounding sea conditions. The float section includes dry boxes for multiple internal user payloads, once of which is occupied by the payload board being developed by L3 Oceania. A central dry box contains an integral control server included with purchase of the wave glider which manages the charging of batteries and routes data between user payloads and payloads in the underwater glider including the servo which controls the rudder. There is no scope to alter this control server, so all development must be done on the user payload board being developed by L3 Oceania. The surface float is also fitted with masts for antennas and sensors, and photo-voltaic panels to recharge the on board batteries[9].

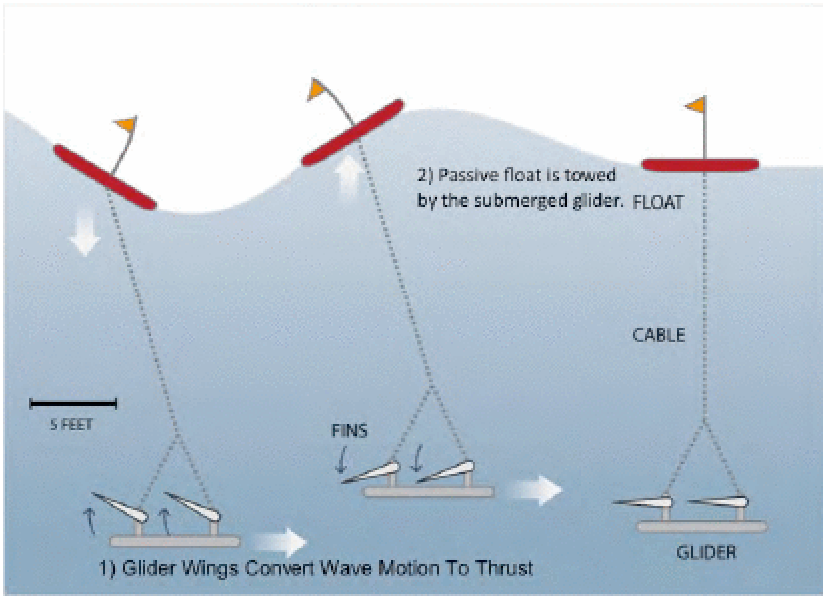

2) Passive float is towed by the submerged glider.
FLOAT
CABLE
5 FEET
FINS
1) Glider Wings Convert Wave Motion To Thrust
GLIDER

Figure 7: Description of Wave Glider propulsion

The Wave Glider to be used in this thesis is provided by L3 Oceania and is fitted with a proprietary payload board providing satellite communications over the Iridium network, cellular communications over 3/4G, and close range radio communication using XBee. A GPM300 acoustic modem also developed by L3 Oceania can be fitted to the glider for underwater communications. The float contains a payload board developed by L3 Oceania in the aft payload bay. The Wave Glider also contains a proprietary command and control payload contained in a central dry box. The purpose of this controller is to route communications between multiple payload devices on the the float and the underwater glider. Commands to the underwater glider must therefore be routed through this control server. The surface payload contains an Inertial Measurement Unit(IMU), a Global Navigation Satellite System (GNSS) receiver, temperature, pressure and humidity sensors for stasis monitoring. The sensors and communications data is routed using micro-controllers on the payload board. High level on board decision making and control is intended to be performed by a Raspberry Pi (RPi) attached to the payload board.

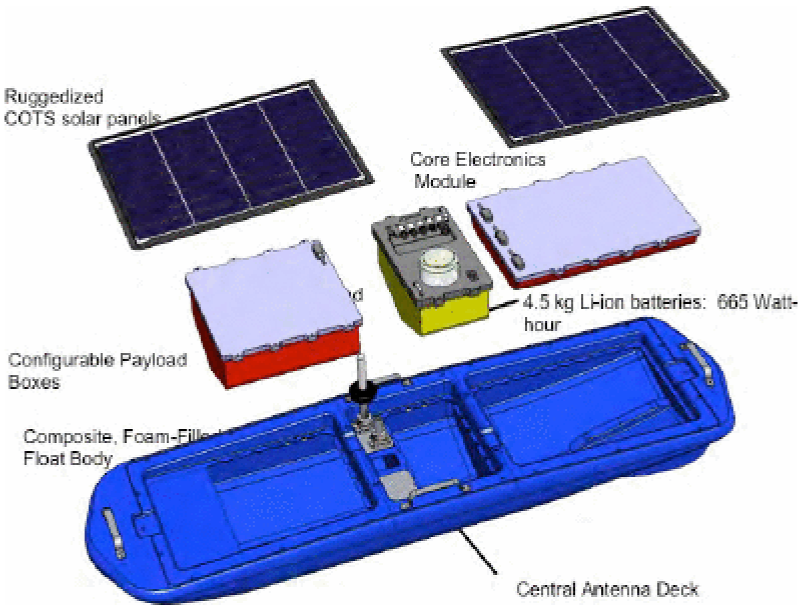

Ruggedized COTS solar panels
Core Electronics Module
4.5 kg Li-ion batteries: 665 Watt-hour
Configurable Payload Boxes
Composite, Foam-Filled Float Body
Central Antenna Deck

Figure 8: Structure of the float as provided by Liquid Robotics.

8.1 L3 Oceania Payload Board

L3 Oceania had previously created a custom circuit board to integrate the various hardware components together. The key components are the RPi 3 running a Mono C♯ stack, an Ethernet microcontroller, a serial and sensor management microcontroller, a connection for an Iridium modem and a 3G modem. Figure 9 points out each of these components.

8.2 Problem Statement

The intended use of the Wave Glider is the CUUUWi project to create a gateway between subsurface acoustic communications and surface radio communications. Liquid Robotics provides an Internet based infrastructure for remotely controlling a fleet of Wave Gliders using their proprietary system. Although functional, the system is reliant on Liquid Robotics owned servers and is somewhat cumbersome to use. As one of the purposes of the system is to function as a mobile gateway it was desirable for the control interface to make use of the already existing SMS interface. The task was therefore two fold: to create a compatible implementation of the protocols used by Liquid Robotics to communicate with the Command and Control box and to extend the SMS interface to give control access of the Wave Glider to users over SMS.

9 Payload Upgrades

The payload board provides a number of UDP servers for each of the various types of data it produces. Subscribing to these servers is simply done by sending a packet to the relevant port number on the payload board. The payload board will send data using UDP to whichever IP address last contacted it. Each of the management classes subscribe using this method to the data servers. Figure 10 describes the architecture of the CUUUWi system components that are relevant to my involvement in the project.

9.1 Wave Glider Model

This singleton class contains the current state of the Wave Glider for reference by the various other components in the system. Sensor data and locations are stored here for example. This ensures that there is only one single representation of the Wave Glider at a given time and data sources don't fall out of sync.

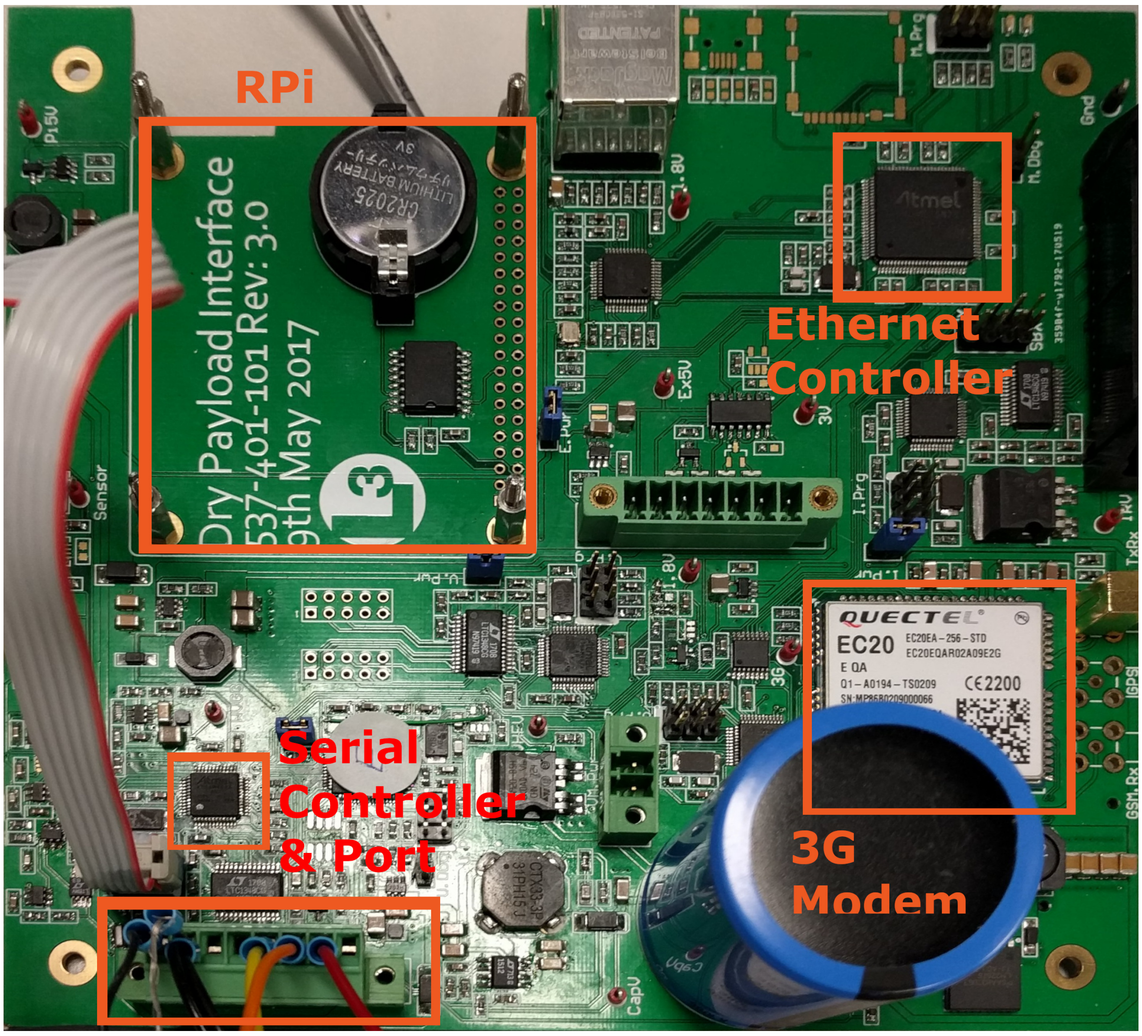


Figure 9: Payload Board Diagram

9.2 Sensor Manager

This is the simplest management class added to the RPi server. A binary structure containing the most recent sensor readings is received from the Ethernet channel and decoded into the relevant data structures stored in the Wave Glider model class. The payload board needs to regularly respond to status requests from the command and control box. When this occurs the stored sensor data is returned to the command and control box.

9.3 Wave Glider Manager

The most complex class, this class is responsible for appropriately responding to decoded Level 2 protocol packets. It has many tasks including notifying the RPi operating system of power down messages, responding correctly to enumeration and status requests and also to send commands to the command and control box if new commands have been queued.

9.4 SMS command parser

The SMS command parser decodes strings into abstract command objects along with their arguments whenever a SMS message is received. The abstract command objects are passed to the appropriate module for execution. This parser is also capable of composing responses from the same abstract objects in order to return acknowledgements and information back to the caller. In order to provide a degree of security, the SMS controller will only respond to messages from users and phone numbers that have previously been stored in the authorisation database.

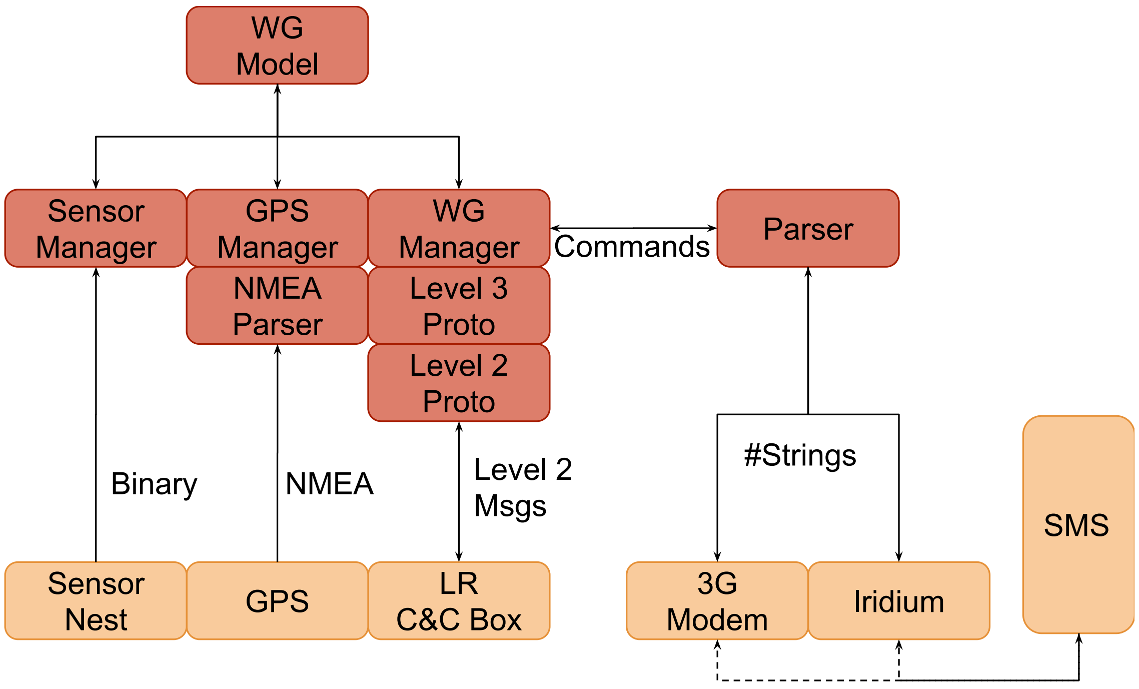


Figure 10: Wave Glider Architecture

9.5 Liquid Robotics Protocols

There are two main types of protocols used by Liquid Robotics named in the documentation as Level 2 and Level 3. The Level 2 protocol functions on the serial connection between the command and control box and a number of payload boards. The Level 2 protocol is designed around enumeration and discovery of payloads, passing payload data and power control. The Level 3 protocol is designed primarily for remote communications and control of the command and control box itself. The Level 3 protocol is what is typically passed over IP or XBee channels. In order to achieve control of the command and control box from the payload one has to create Level 3 packets and encapsulate them in Level 2 packets. As the Level 2 protocol is strictly a master/slave relationship, there is unfortunately little real time control and the payload must wait until it is polled before it can pass a command. Additionally, the architecture of the command and control box doesn't seem to allow executed Level 3 commands to return data through Level 2 messages. This limits the possible functionality of the system but objectives remain achieved. The Level 2 Liquid robotics protocol needed to be additionally implemented in C++ in order for the serial and Ethernet controller chips to interpret and route the packets.

10 Documentation

In addition to the functionality itself I completed documentation on implementation details. These documents are stored on the L3 Oceania intranet and are not provided since it's not likely another UWA student will be working on the project. I also wrote unit testing scripts using the Microsoft Visual Studio Unit Test framework to ensure that future changes and improvements do not affect the previous functionality of the application. These unit tests are stored alongside the soufce code repository.

10.1 User Interface

Table 3 lists the user interface available over SMS for controlling the Wave Glider. The simple protocol consists of a command and a number of parameters. Typically, listing a command without parameters instead queries the state of the paramete, rather than setting it. The CUUUWi gateway supports many more commands than listed which have been implemented by other teams and projects and so aren't relevant to my thesis.

Command	Arguments	Description
#FHEAD	[0-359]	Wave Glider continues in the bearing given.
#LIGHT	[ON|OFF]	Get or Set state of visible marker light.
#IR	[ON|OFF]	Get or Set state of Infra-red marker light.
#HOLD	none	Remains at given location; ceases other navigation
#WAYPOINT	[INDEX] [LAT] [LON]	Set latitude and longitude in decimal degrees at the specified index
#HOLDWP	[INDEX]	Go to and stay at the way point at the specified index (1-254).
#FSEQ	[TARGET] [BEGIN] [END]	Wave Glider visit target then loops defined by the sequence of way points between 'begin' and 'end'.
#RUDDER	[-30 to 30]	Get or set the rudder angle
#FCOURSE	[LOOP|HOLD] [index] ...	Follow course defined by given way points and then stop or loop again

Table 3: Supported Wave Glider Commands

11 Test Deployment

Testing of the Wave glider was performed on the 31st of August 2018. A number of tests were scheduled including my integration tests when the wave glider was launched from *Whale Song*. While I was unable to attend, the tests were carried out regardless according to a test procedure document that I wrote previously.
The following notes were taken from the test deployment. As indicated in table 4, there were a couple of initial problems. When I returned, however I was able to fix the problems indicated in the notes. The issues were minor and resulted from incorrect command flags being set.

Time	Command	Result	Success
11:18	#FHEAD 230	Replied and confirmed success using WGMS	Y
11:49	#HOLD	Replied and confirmed success using WGMS	Y
12:09	#WAYPOINT 1 -32.0380 115.3325	Replied, assume success (no visible action)	Y
12:09	#HOLDWP 1	Replied, glider did not seem to move, perhaps misinterpreted as #HOLD	N
12:10	#WAYPOINT 2 -32.0345 115.3326	Replied, assume success (no visible action)	Y
12:11	#WAYPOINT 3 -32.0345 115.3326	Replied, assume success (no visible action)	Y
12:12	#WAYPOINT 4 -32.0345 115.3326	Replied, assume success (no visible action)	Y
12:28	#FSEQ 1 1 4	Replied but with "#FSEQ 1 1" missing " 4" at the end. Could not confirm due to slow WGMS update rate. Logs indicate the incorrect command was accepted	N
13:03	#FCOURSE HOLD 3 4 1	Replied. Could not confirm due to slow WGMS update rate. Logs indicate the command was accepted	Y

Table 4: Results from Wave Glider Field Trial

12 Conclusion and Future Improvements

The SMS command functionality achieves the stated requirements and at the time of writing is being used on a deployed Wave Glider being tested by L3 and the Royal Australian Navy. Originally there were ambitions for greater (non-critical) functionality however the slow master/slave architecture of the system means that it would be difficult if not impossible to achieve without replacing the command and control box entirely. One of the later desired goals of the Wave Glider project is to be able to characterize sea state using the on board INU. Some work has already been done on creating simulated data and processing it using MATLAB and WAFO, but a lack of real data has stalled development for the moment. It is likely that this feature will be brought with future developments of the project.

Part III

Apache Server Creation

13 Apache Web Server Common Gateway Interface(CGI) tutorial

The Common Gateway Interface (CGI) is a standard protocol web servers can use to run a normal command line script or program and return the result to the client. This allows you to create and run a server script on a normal web server without having to manage the details of networking, connections, encryption and other common problems. Secondarily it allows you to make use of an existing web server, staying within the security limitations that may be imposed on you by a hosting provider. Hosting providers will typically not allow you to open arbitrary ports for security reasons and may only allow access to the web server they have provided for you. This tutorial will be familiar to anyone who has done server scripting in PHP, however the advantage of CGI scripts is you can do server programming in a language you are already familiar in.

13.1 Pre-requisites

I will not be covering the details of all these items. Many tutorials exist for these subjects on the Internet if you feel you need them. This tutorial focuses on how to integrate these items into a simple web application server.

1. Linux command line knowledge (bash): It is assumed for this tutorial that you have a basic understanding of using the Linux command line, opening files, navigating directories what a 'home' folder is etc. Regardless of what operating system you choose to use, many if not most hosting providers run Linux. You will need to be able to edit files on the host server in order to configure it correctly. If you need some help here, there are many tutorials and 'cheat sheets' available online on how to use the Linux command line.

2. Basic scripting knowledge: CGI allows for any command line program to run, even binaries compiled from C/C++. However I strongly recommend that you use a scripting language that has inbuilt support for the CGI. Popular examples for this are PHP, Python, Perl and Ruby. I will be giving any examples in **Python 2**, however I will make note of what configuration differences you will need to provide to use the other languages. A Python specific tutorial can be found at `https://docs.python.org/2.7/tutorial/`. Other scripting language tutorial can be found if you search for them.

3. HTML: I will assume you know how to write HTML documents, forms and including external media files and script files. Server side scripts output the HTML which is ultimately served to the client. If you are new to this tutorials can be found at `https://www.w3schools.com/html/`.

4. JavaScript: If you want client side scripts to be run you will still need to write these in JS. A tutorial on JS can be found at `https://www.w3schools.com/js/`.

5. SQL or similar database language: You may want to store, retrieve and search for items from a database. Particularly if some clients store data and other clients retrieve them. The database engine you wish to use will depend on what is installed on your web server and what libraries are available for your chosen scripting language. In this tutorial I will be using SQLite3. Again, an SQL tutorial can be found at `www.w3schools.com/sql`.

13.2 Secure Shell

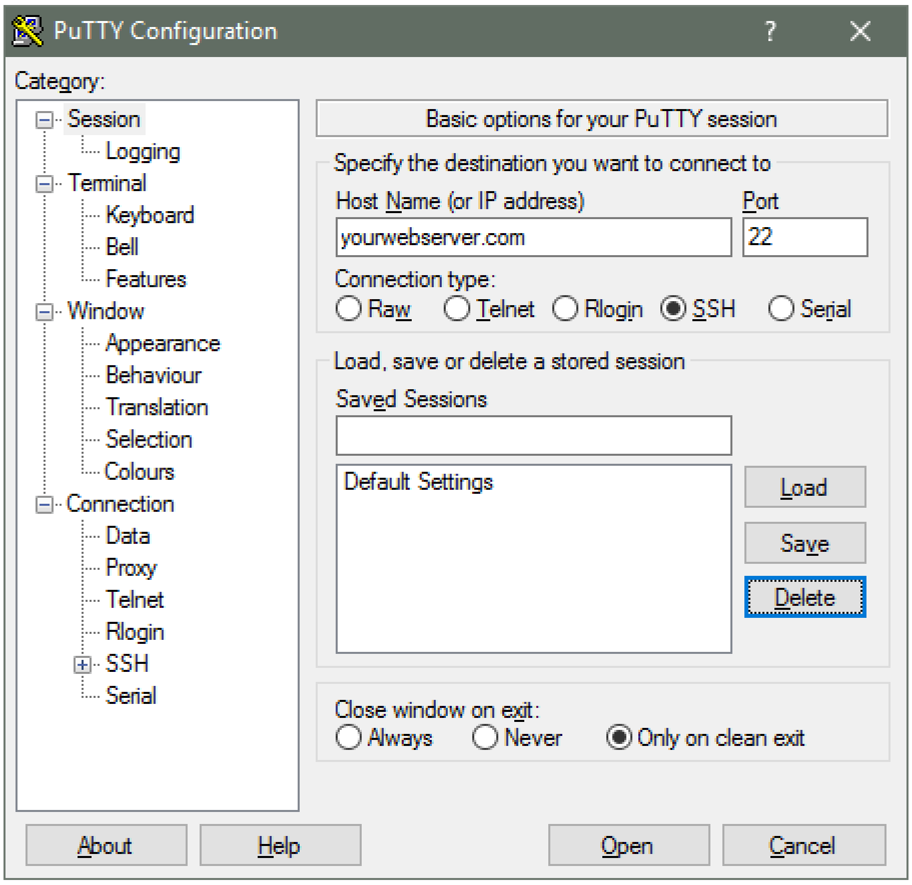
Secure Shell (SSH) allows you to securely log on to a remote computer or server and run commands on it as if you were physically typing commands on it's own keyboard. Many web hosts provide their own unique interfaces for uploading files and managing the web server, meaning you can avoid using SSH if you want. However these interfaces are all unique and can't be covered in one document. You may find it useful to look at your web hosts tutorials on how to use their particular interface.

13.2.1 Linux and OSX

Most Linux distributions come prepackaged with SSH. If it is not installed by default it can be freely installed using your package manager. OSX should also have SSH available by default. log on to you host server using the user name and password provided to you by the hosting provider using the command `ssh username@hostserver.com`. If this is the first time you have connected, SSH will indicate that it has never seen the server's public key before. Type 'yes' to continue and it will then ask for a password. Once the session is established you can interact remotely with the server.

13.2.2 Windows

Widows does not have SSH installed and installing it is not easy or necessary. The windows application PuTTY is compatible with SSH so you can use that instead. Once you log on however you will still be in a Linux environment so Linux knowledge is still necessary. Putty can be downloaded from https://www.putty.org/. Once installed, run it and type your user name and host address in the field shown in Figure 11. Ensure that you click the SSH button to indicate you want to start an SSH session.



Host Name (or IP address): yourwebserver.com
Port: 22
Connection type: Raw Telnet Rlogin [✓ SSH] Serial
Close window on exit: Always Never [✓ Only on clean exit]

Figure 11: PuTTY User Interface

PuTTY will also show a dialog prompt indicating it has never seen the server key before. Click 'yes' to continue and then type the password.

13.3 Apache and NGiNX

Most hosting providers will have either Apache Web Server or NGiNX. They are compatible and you shouldn't notice the difference nor will it change this tutorial.

13.3.1 Web Root

Typically within the home folder will be your web root. This folder is typically called 'www' or 'webroot'. Everything within the folder can be accessed by anyone through the web server unless the file is hidden, inaccessible by Apache or some security configuration is performed (I will cover this later). As such do not put sensitive files here unless you *really* mean to put them here. Important files can be safely stored out side of the webroot and Apache will refuse to serve them. Folders are also accessible here, for example the file placed `www/myproject/file.txt` will be accessible from the Internet as `http://hostserver.com/myproject/file.txt`. A special case is any file called Index.html. Apache will serve Index.html if you don't specify a file, so `www/myproject/index.html` will be served when you go to `http://hostserver.com/myproject`.

13.3.2 cgi-bin

`www/cgi-bin` is the traditional place CGI scripts are placed. Scripts can be called from anywhere if configured to do so but for now place all your scripts here. Traditionally CGI scripts have the .cgi file extension regardless of what language is actually used but any extension or no extension is acceptable (Linux doesn't treat extensions specially).

13.4 Uploading Files

There are many ways to upload files on all systems. I will only provide one simple example way for each operating system.

13.4.1 Linux & OSX

Both these systems should also have a program called Secure Copy (SCP). This will allow you to securely upload files to the web server including your scripts. Simply calling `scp /localdirectory/myfile.txt username@host:/remotedirectory/myfile.txt` will transfer the file. It should prompt you for a password to complete the command. Reading the SCP manual will describe how to upload multiple files and directories or you can upload a single .zip file containing multiple files. Some linux environments also allow you to seamlessly connect to the remote server using SFTP from with the file explorer but this process varies for different systems.

13.4.2 Windows

WinSCP https://winscp.net/eng/download.php provides a graphical interface for the SFTP protocol allowing you to upload to the webserver using SFTP. The start up prompt will again ask for the host details and a username and password. Then you can transfer local files from the left window pane to remote folders shown in the right window pane.

13.4.3 Permissions

Folders and files that you create and upload on the server will only be accessible by you. However Apache runs as a separate user and you need to give it access to files you wish for it to serve or run. Scripts should typically be user mode 755. This allows Apache to read and execute your scripts, but not edit them. Changing the permissions on your script once you have uploaded it is done by typing `chmod 755 myscript.cgi` from within a SSH session.

13.5 Writing a Script

13.5.1 Shebang

`#!` called a 'shebang' or 'hash-bang' is a special symbol placed as the very first characters in a file that tells the server what interpreter to use to run the script. Immediately after the shebang type the path to the interpreter you wish to use. Hosts may place their interpreters where ever they wish and you may be able to install some yourself. The easiest way to find the path to an interpreter is to use the `which` command. For example to find the location of the python interpreter on the system type `which python`. similarly `which perl` and `which ruby` will give you the locations of those interpreters. Binary files will not need this added to them, nor will PHP scripts.

13.5.2 CGI packages

Common scripting languages will typically have some packages to assist with making CGI scripts. These will allow you to easily access GET and POST variables and other things. In Python simply starting with `import cgi` will get you this functionality. More information about using python as a CGI script can be found at https://docs.python.org/2/library/cgi.html.

13.5.3 HTTP header

Your web browser expects some metadata describing what kind of content is being returned by the server. The HTTP header can contain a lot of information but at the bare minimum you will need to print the line "Content-type: text/html" followed by two new line characters. Without this header line, the client web browser will not render the HTML document correctly.

13.5.4 Displaying HTML

Apache will send all standard output from your script back to the client requesting it. Therefore you can create complex web pages by simply printing HTML in the same way that you would normally print to the command line. Code which does not produce an output will not be displayed. In Python, multi-line quotes using 3 " (double quote) characters can be used to easily print a large block of HTML for structuring your page. Use a single block quote to encapsulate a logical block of HTML, for example the header of your page.

13.5.5 Example Python CGI script

The following minimal python script will display a basic web page with a title and then display all arguments passed to it, demonstrating use of the `cgi.FieldStorage` object which contains all arguments passed to your script. `cgi.FieldStorage` will contain arguments passed both from GET requests and POST requests. Using python block quotes it's easy to print static HTML elements in a way similar to PHP. You can also use string methods to include dynamic data into your web page. In this case the key/value pairs of any arguments are printed in the HTML document.

```python
#!/usr/bin/python
import cgi

print "Content-type: text/html"
print

print """
<meta charset="UTF-8">
<html>
<head>
<title>My project title</title>
</head>
"""

print """
<body>
<h3>Title</h3>
"""

print "<p>Args</p>"
arguments = cgi.FieldStorage()
for key in arguments.keys():
   print key + ":" + arguments[key].value + "<br>"

print """
</body>
</html>
"""
```

13.6 Accessing a Database

You will be able to access any database that you are able to access through your scripting language. In this example we will be accessing a SQLite3 database using the python module of the same name.

13.6.1 Creating the database

Place the database somewhere convenient on the server. It should be stored somewhere like `/db` and you should disallow anyone from accessing this directory. Otherwise this leaves the potential for a malicious person to simply download the database and inspect it's contents. There is almost no situation where someone should be able to do this. Particularly if the database contains sensitive data. Simply create a .htaccess file containing only the line `deny from all` in the `/db` folder to prevent anyone from directly accessing the contents of the folder.

I recommend creating an SQL script to create and initialize tables in your database for maintenance purposes. You may want to reset and create an identical database in the future.

13.7 Client Side Scripting

Server scripts are executed on the server and have so far been the focus of this tutorial. However, you may additionally wish to serve code which is executed in the client user's browser. Client side scripting is almost universally done in JavaScript.

Your client-side scripts can be included in the web page in the same way as any other HTML document. For example an external script can be included like this:

```
print """
<script type="text/javascript" src="/js/script.js"></script>
"""
```

You can also use the server side script to generate inline script which will also be executed by the client. This is particularly useful for transferring variables from your server script and allowing them to be accessed by the client scripts. For example:

```
myInt = 100
print '<script type="text/javascript">'
print 'var myJSVariable = ' + myInt + ';'
print '</script>'
```

When this is executed by the client web browser the value of 100 will be available in JavaScript as myJSVariable.

For ease of maintenance I recommend that the majority of your client scripts be created in a separate file as in the first example. Only use the server side script to place values into variables before including the main JavaScript file. This will make finding bugs easier and avoid confusion about which statements get executed when.

13.8 Sending Data to the Server

The advantage of leveraging a normal HTTP server stack is that you can interact with the server using normal HTTP requests. There are two main requests: GET and POST requests. A GET request is issued by a client (including web browsers) in order to obtain a file. In addition to specifying the file the client wishes to obtain, it also specify GET variables that influence what is returned. GET requests resemble
`http://hostserver.com/pathto/file?getVariable1=value&getVariable2=value2`. In this example two variables 'getVariable1' and 'getVariable2' are sent.

13.8.1 Webforms

HTML can be used to markup forms like text-boxes, radio buttons, drop-down menus etc that you are probably already familiar with. These form fields can be configured to send data to the server as either GET or POST requests. The form attributes 'action' and 'method' define the script to submit to and whether to use GET or POST respectively. The 'name' attribute define the key that a variable will be sent with and the 'value' attribute contains the actual user input. Predefining the 'value' attribute allows you to set defaults that the user can override.

```
<form action="/action_page.cgi" method="get">
 First name:<br>
 <input type="text" name="firstname" value="Mickey"><br>
 Last name:<br>
 <input type="text" name="lastname" value="Mouse"><br><br>
 <input type="submit" value="Submit">
</form>
```

13.8.2 Sending POST/GET requests from an external script

Most scripting languages have libraries for sending POST and GET requests to a server. This will probably be the preferred solution where you need a program to autonomously interact with the server without having to go through a web browser. Python has the `requests` library for interacting with HTTP servers and is a very easy and straight forward library to use.

```
import requests

# Create a dictionary of key/value pairs you wish to send
payload = {'key1': 'value1', 'key2': 'value2'}

# Send a get request to the server
response = requests.get("https://someserver.com/pathto/script", data=payload)
print(response.text)

# send the same in POST
response = requests.post("https://someserver.com/pathto/script", data=payload)

# Do basic authentication
credentials = requests.auth.HTTPBasicAuth('username', 'password')
response = requests.post("https://someserver.com/pathto/secureDir/script", data=payload,
    auth=credentials)
```

The `requests` library contains support for many more HTTP methods, authentication methods and support for multi-part encoding for large files.

13.8.3 Java Script Object Notation (JSON)

If you want to send data more complex than built-in types like strings, floats and int consider using JSON. JSON is a standard method for defining complex data structures as strings. This allows you to send these objects, such as a class containing multiple fields of strings, ints, list etc easily as strings and then recreate those data structures on the server without having to define and parse multiple variables.

```
import json
# Send the payload as JSON
payload = {'some': 'data'}
response = requests.post("https://someserver.com/pathto/secureDir/script", json=payload)
```

On the other side, decoding the sent JSON values can be done in a similar way

```
import json
stringFromClient = '["foo", {"bar":["baz", null, 1.0, 2]}]'
list = json.loads(stringFromClient)
# list now contains a list object which can be indexed like any other list object
```

13.9 Security

13.9.1 Be aware of Threats

Your server will be accessible by the public including malicious users therefore you will need to take care beyond what you might usually take in order to avoid common security vulnerabilities. While important these security concerns are too vast, sometimes complicated and not always relevant to every use case, therefore I will not be going into detail about them in this tutorial. However, the Open Web Application Security Project (OWASP) is a very useful resource that explains in detail about how various attacks are performed and how they can be defended against. I highly recommend you look up each of these topics on https://www.owasp.org/.

1. SQL Injections: Most SQL libraries will have some function to sanitise inputs in order to avoid common SQL injections. Learn what those functions are and how to use them. Basic string functions part of the built-in Python library are vulnerable to injections for example.

2. Cross site Scripting (XSS) Attacks: If your application stores data that is entered by one user and then displays it some where on a page, XSS are possible. XSS vulnerabilities allow an attacker to execute malicious actions on a victim's machine, often without any victim input.

Be aware of what items should and shouldn't be public. If you deny access to resource folders such as /images, /js or /css clients won't be able to download those resource files and your application won't work as intended.

13.9.2 Basic Authentication

Often you may want to create a script or folder which isn't publicly accessible. This can be done using the .htaccess file and the .htpasswd file. Let's say you want to password protect the mydomain.com/secure directory. In the the /secure directory you should place a .htacess file containing the following. Note '#' indicate comments in configuration files and don't affect the configuration of the server.

```
# Optional, text which is displayed in the prompt
AuthName "Secure Area"
# Path to the .htpasswd file
AuthUserFile /home/username/www/secure/.htpasswd
AuthType Basic
Require valid-user
```

These configuration options indicate that the containing folder requires authentication and indicates how authentication is performed. Additionally a .htpasswd file will be required which contains the credentials (usernames and password hashes) used for authentication. It is important that passwords are stored as hashes so even if the .htpasswd file is stolen the password behind the hash is difficult to recover. You can generate password hashes in the correct format multiple ways. The easiest method is using the `openssl` command line utility that is installed on most linux distributions, meaning your web host likely already has it installed. Use the command `openssl passwd -apr1 YourSecretPassword -o .htpasswd` to generate a mostly complete .htpasswd file. Finally you need to prepend a user name to the hash. Open the .htpasswd for editing using `nano .htpasswd` or another editor and add a username and colon to the beginning of the password. Your final .htpasswd file should resemble something like this.

```
# hash for 'password' very insecure. _Do not copy this example_
username:$apr1$tvIrD/Hp$EsNN5cLtvwLugB/7k0bKh1
```

You can have multiple users and passwords, just add each new entry on a new line of the .htpasswd file. Whereever you store your `.htpasswd` file make sure you make a `.htaccess` file with `deny from all` in that folder so that the password file isn't stolen separately.

13.9.3 HTTPS

Secure HTTP (HTTPS) provides encryption and authentication between the web browser and the web server. Without HTTPS any traffic which is sent between the server and client can be read *including Basic Authentication exchanges*. There are multiple ways of enabling HTTPS but all require a valid certificate. Certificates can cost money to obtain them from a Certifying Authority (CA), however the service Let's Encrypt is a CA that provided free certificates. Placing a certificate on a web server requires administrative access and therefore will usually require going through the web hosting provider. For example instructions for using a free Let's Encrypt certificate for the BlueHost hosting provider can be found at `https://my.bluehost.com/hosting/ssl_certs`. The last option is to use a self-signed certificate. This is free and gives you a lot of control on the configuration, but requires explicit administrative privileges this is only really an option if you are hosting yourself or your hosting provider allows you to do it.

References

[1] J. Hodge, “Project report: Autonomous solar powered boat,” Master’s thesis, University of Western Australia, 2017.

[2] J. Borella, “Solar powered autonomous boat (spab) thesis proposal,” Master’s thesis, University of Western Australia, 2017.

[3] D. Crockford, “The application/json Media Type for JavaScript Object Notation (JSON),” Internet Requests for Comments, RFC Editor, RFC 4627, July 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4627.txt

[4] R. Fielding, “Architectural styles and the design of network-based software architectures,” Ph.D. dissertation, University of California, Irvine, 2000.

[5] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer, “Simple Object Access Protocol (SOAP) 1.1,” Technical Note, W3C, W3C Note, May 2000. [Online]. Available: https://www.w3.org/TR/2000/NOTE-SOAP-20000508/

[6] G. E. Krasner and S. T. Pope, “A cookbook for using the model view controller user interface paradigm in smalltalk-80,” *Journal of Object Oriented Programming*, vol. 1, no. 3, pp. 26–49, 1988.

[7] M. LLC. (2018) Mavlink developer guide. [Online]. Available: https://mavlink.io/en/

[8] J. Romkey, “A NONSTANDARD FOR TRANSMISSION OF IP DATAGRAMS OVER SERIAL LINES: SLIP,” Internet Requests for Comments, RFC Editor, RFC 1055, June 1988. [Online]. Available: https://tools.ietf.org/rfc/rfc1055.txt

[9] R. Hine, S. Willcox, G. Hine, and T. Richardson, “The wave glider: A wave-powered autonomous marine vehicle,” in *OCEANS 2009*, Oct 2009, pp. 1–6.