A Generalised Communication Framework for the Remote Control of Robots

Submitted in partial fulfilment of the requirements of the degree of Bachelor of Science (Honours) of Rhodes University

Dale Tristram

Grahamstown, South Africa
November 2011
Abstract

The robotics industry is plagued with proprietary interfaces for computer-to-robot communication. Open source projects have emerged in an attempt to create robot communication standards that will benefit the industry as a whole, but have thus far only satisfied particular use cases. This project investigates the feasibility of creating a generalised computer-to-robot communication framework that provides a uniform interface to a robot’s functionality over HTTP, is easy to use, and is accessible by smartphone and tablet devices. This is achieved by developing a prototype framework that satisfies the above objective. Analysis of the framework reveals that it is easy to deploy, and significantly reduces the complexity and code required to perform simple operations on a test robot. Therefore, it can be concluded that it is both feasible and beneficial to robot users and developers to use a framework such as the one created.
ACM Computing Classification System Classification


D.3.3 [Language Constructs and Features]: Frameworks
I.2.9 [Robotics]: Operator interfaces
H.3.5 [Online Information Services]: Web-based services

General-Terms: Robot Interface, Framework
Acknowledgements

I would like to acknowledge the financial and technical support of Telkom, Tellabs, Stortech, Eastel, Bright Ideas Project 39, and THRIP through the Telkom Centre of Excellence in the Department of Computer Science at Rhodes University.

I would also like to acknowledge the financial assistance of the Andrew W. Mellon Foundation and the Rhodes University Postgraduate Funding Division for the Prestigious Honours Scholarship. Additionally, I acknowledge the financial assistance of the National Research Foundation (NRF) for the NRF Prestigious Honours Scholarship awarded to me. It must be noted that the opinions expressed and conclusions arrived at are not necessarily to be attributed to the above parties.

The completion of this thesis would not have been possible if it were not for the support from people around me. In particular, I would like to thank my parents, Tony Tristram and Deborah Moore, my step-father, Alan Moore, and my girlfriend, Lindsey Gouws, for their continued love, support, and encouragement.

Finally, I would like to thank and acknowledge my supervisor, Dr. Karen Bradshaw. Her help and support throughout the year made this thesis possible.
# Contents

1 Introduction .................................................. 1
   1.0.1 Problem Statement and Research Goals ................. 1
   1.0.2 Thesis Organisation .................................. 2

2 Background ................................................. 3
   2.1 Introduction ........................................... 3
   2.2 Generalised Communication in Robotics .................. 3
   2.3 Considerations for a New Framework ...................... 5
      2.3.1 Mobile Platform .................................... 5
      2.3.2 Web Services ........................................ 7
      2.3.3 Web Server Setup ................................... 12
      2.3.4 Message Passing ...................................... 14
   2.4 The WifibotLab LIDAR ................................... 17
   2.5 Acer Iconia Tab A500 .................................... 18
   2.6 Summary ................................................ 18
3 GRF Design and Implementation

3.1 Introduction ............................................ 19
3.2 GRF Requirements ...................................... 19
3.3 GRF Overview .......................................... 20
3.4 Design Decisions ...................................... 20
  3.4.1 Suitability of HTTP ................................. 20
  3.4.2 GRF API ............................................. 21
  3.4.3 Web Server and Programming Language .......... 21
  3.4.4 GRF Coupling with the Robot Instance .......... 23
  3.4.5 Socket Type ........................................ 23
  3.4.6 Message Serialisation Format ..................... 24
3.5 Robot Instance Specification .......................... 25
  3.5.1 Use Case 1: Developing the Entire Robot Instance 25
  3.5.2 Use Case 2: Using the Existing GRFInterface Module 30
  3.5.3 The Graphical User Interface ...................... 32
3.6 Project Modules ....................................... 32
  3.6.1 The Web Interface ................................ 32
  3.6.2 The Robot Interface ............................... 33
  3.6.3 The Robot Instance ................................ 35
3.7 Software Requirements ................................. 36
  3.7.1 Node.js ............................................ 36
  3.7.2 Protocol Buffers ................................ 36
  3.7.3 NPM .............................................. 36
### CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.7.4 Node.js Modules</td>
<td>36</td>
</tr>
<tr>
<td>3.8 File Structure</td>
<td>37</td>
</tr>
<tr>
<td>3.8.1 Files</td>
<td>37</td>
</tr>
<tr>
<td>3.8.2 Organisation</td>
<td>39</td>
</tr>
<tr>
<td>3.9 Summary</td>
<td>39</td>
</tr>
<tr>
<td><strong>4 Analysis</strong></td>
<td>40</td>
</tr>
<tr>
<td>4.1 Introduction</td>
<td>40</td>
</tr>
<tr>
<td>4.2 Case Study: The Wifibot Lab LIDAR</td>
<td>40</td>
</tr>
<tr>
<td>4.2.1 Accessing the Wifibot’s Functionality</td>
<td>40</td>
</tr>
<tr>
<td>4.2.2 Writing the Wifibot Module</td>
<td>42</td>
</tr>
<tr>
<td>4.3 Usability Tests</td>
<td>46</td>
</tr>
<tr>
<td>4.3.1 Performing a 180° Turn</td>
<td>46</td>
</tr>
<tr>
<td>4.3.2 Retrieving the Status from the Wifibot</td>
<td>46</td>
</tr>
<tr>
<td>4.3.3 Streaming LIDAR Data</td>
<td>47</td>
</tr>
<tr>
<td>4.3.4 Comparison</td>
<td>47</td>
</tr>
<tr>
<td>4.4 Performance Tests</td>
<td>50</td>
</tr>
<tr>
<td>4.4.1 Methodology</td>
<td>50</td>
</tr>
<tr>
<td>4.4.2 Test 1: Repeatedly Executing a Command</td>
<td>50</td>
</tr>
<tr>
<td>4.4.3 Test 2: Moving in a Square</td>
<td>54</td>
</tr>
<tr>
<td>4.5 Assessment of Generality</td>
<td>57</td>
</tr>
<tr>
<td>4.6 Summary</td>
<td>58</td>
</tr>
</tbody>
</table>
## 5 Conclusions

5.1 Project Summary ............................................... 59

5.2 Revisiting the Objectives ....................................... 59

5.3 Possible Extensions ............................................. 60

## A Code Listings

A.1 Protocol Buffer Protocol ....................................... 65

A.2 Simple Robot Instance .......................................... 65
List of Figures

3.1 GRF overview .......................................................... 20
3.2 RobotRequest/Response messages .................................. 26
3.3 The high level operation of the Robot Instance .................. 29
3.4 The processing of RoboRequest messages ............................ 30
3.5 Module overview .......................................................... 32
3.6 File structure ............................................................. 37
4.1 Throughput graph for Test 1 .............................................. 53
Chapter 1

Introduction

The field of computer-to-robot communication has traditionally been one where each robot manufacturer has its own proprietary computer-to-robot communication framework, thereby limiting interoperability and preventing standardisation. With robotic devices becoming more accessible to researchers, businessmen, and the general public, it is becoming increasingly important to simplify and generalise the interface we use to communicate with robots. Furthermore, with the increased usage of smartphone and tablet devices as a control platform, it is important that robot interfaces are easily accessible by these types of devices.

In an ideal scenario, the functionality of all robots would be exposed via a standard web accessible interface, greatly simplifying computer-to-robot communication. This would make robot functionality more accessible, both to a variety of Internet-enabled devices, and by a large number of network configurations through the use of the widely accepted Hypertext Transfer Protocol (HTTP).

1.0.1 Problem Statement and Research Goals

This research project investigates the feasibility and practicality of creating a generalised computer-to-robot communication framework that exposes robot functionality over HTTP. The aim of the project is to create a prototype framework, called the Generalised Robot Framework (GRF), which is sufficiently general to be used with any kind of robot that meets the software requirements, is easy to use and deploy, and exposes all the robot’s functionality over HTTP.
The framework should be as light on resources as possible to ensure that it works adequately on low resource robots, and it should render robot functionality easily accessible on a range of devices, particularly smartphone and tablet devices. Thus, the research objectives for this project are:

**Primary Objectives**

- Build a framework general enough to be usable on a large variety of robots.
- Expose robot functionality through the framework using HTTP.
- Ensure that robot functionality is easily accessible via smartphone and tablet devices.

**Secondary Objectives**

- Ensure that any latency introduced by the framework does not adversely affect robot functionality.

**1.0.2 Thesis Organisation**

The remaining chapters of this thesis are described below:

**Chapter 2** describes existing attempts at generalised robot control frameworks, and discusses technologies that could be used in creating a new framework, which satisfies the objectives outlined above.

**Chapter 3** describes the prototype design and implementation of the GRF, and explains the design decisions that were made for the system.

**Chapter 4** analyses the work involved in deploying the GRF, its usability, its performance, and its generality.

**Chapter 5** summarises this thesis and outlines the conclusions that were drawn from this research. Finally, it identifies possible extensions to the project.
Chapter 2

Background

2.1 Introduction

Attempts have been made to create open source, generalised frameworks that provide a uniform interface to robot functionality for computer-to-robot communication. This chapter surveys the existing attempts at creating such a framework, as well as the technologies that could be used in creating a new framework. This new framework should provide an HTTP interface to a robot’s functionality that is usable on both smartphone and tablet devices.

2.2 Generalised Communication in Robotics

The search for literature on generalised robotic communication frameworks yielded few results. The robotic industry software practices are such that every manufacturer has its own proprietary software and data structures [23], making information about the communication techniques used inaccessible. Two open source projects that were discovered are Open Robot Control Software (Orocos) and Open Robot Interface for the Network (ORiN).

2.2.0.1 Orocos

Started in 2001, the Orocos project originally had the goal to become a generalised robot control software package [23][12]. However, over time the real-time Orocos framework
evolved into the machine control field, outgrowing its roots in robotics [12]. The evaluation of the literature on Orocos failed to identify a module that provided a generalised network accessible interface for robot interaction.

### 2.2.0.2 ORiN

ORiN is a project with the goal of creating open standards for machine or robot-to-computer communication [34][15]. The motive behind this goal was to provide a standard method to communicate with a variety of different robots, and to create a multi-vendor robot-task executing system [34]. According to Mizukawa et al. [34], ORiN is expected to:

- provide a standard method to exchange data between a robot and a computer;
- enable software vendors to easily develop robotic software packages for the robotic industry;
- enable system integrators to build high quality manufacturing systems at lower cost;
- enable end-users to construct and use a multi-vendor manufacturing system easily; and
- reduce the cost of development of hardware and software for the purpose of communication in robotics, and allow different robot manufacturers to use technologies from other manufacturers.

The project aims to achieve this by creating a three-layered framework consisting of an Application layer, a Kernel layer, and a Provider layer [34][15]. Client applications can access a robot’s functionality through the Controller Access Object (CAO) Engine within the Kernel layer [34][15]. The CAO Engine provides a standard interface to the robot’s functionality by communicating with the CAO Provider [34][15]. The CAO Provider is tasked with managing exchanges of information between the CAO Engine and the vendor specific robot controller [34][15]. One of the components of the CAO Provider is the Controller Resource Definition [34][15]. The Controller Resource Definition uses Extensible Markup Language (XML) to define robot resource information specific to a particular robot model, thereby allowing the creation of a unified framework for robot manufacturers [34][15].
2.3 CONSIDERATIONS FOR A NEW FRAMEWORK

The review of the literature on ORiN has led to the conclusion that the framework targets the use case of creating software that is able to communicate seamlessly with different industrial robots in a similar manner, but is not intended as a generalised robot control framework. There are two observations that have contributed to this conclusion: 1) none of the use cases mentioned in [34] and [15] relate to non-industrial robots, and 2) the framework is located on a controlling computer rather than the robot itself [15].

2.3 Considerations for a New Framework

In this section, the mobile platform is discussed in the context of the GRF, and various web services, web server setups, and message passing options are investigated for use in the GRF.

2.3.1 Mobile Platform

Smartphones and tablets are becoming increasingly pervasive in our society, and because of their mobility and increasing power, they are attractive devices for the remote control of robots. The web browser on smartphones and tablets should allow access to robot functionality exposed via a web interface. However, there are over 21 different mobile browsers of various qualities [32], resulting in inconsistent support for HTML, Cascading Style Sheets (CSS), and JavaScript features. Browser support for touch interactions, HTML5, and Motion JPEG (M-JPEG) are investigated below.

2.3.1.1 Touch Interaction Support

Tablets and most smartphones have touch screens [19]; this results in interaction with web pages that is different from typical mouse interaction [33]. Traditionally, touch interactions have been normalised and interpreted as mouse events, but such an approach can have undesirable results [22]. For this reason, different implementations of JavaScript touch events have been added to several web browsers to better support this new kind of interaction [32]. An attempt to standardise touch events is being made; the W3C

---

1 http://quirksmode.org/mobile/mobilemarket.html
2 http://www.quirksmode.org/mobile/tableTouch.html. On this page, Koch states that he made a mistake in his tests. I have confirmed with him via email that the first table following that statement contains valid results.
2.3. CONSIDERATIONS FOR A NEW FRAMEWORK

published the specifications for a working draft of Touch Events version 1 in September 2011 [22].

The working draft specification describes four new document events, \textit{touchstart}, \textit{touchend}, \textit{touchmove}, and \textit{touchcancel} [22]. According to tests performed by Koch [32], these new events are supported by the Dolphin and Phantom browsers, as well as the native browser on Android 2.1 and 2.2, iPhone 3.1, and BlackBerry Torch devices. His tests further show that Firefox 1.1 on Maemo, Opera Mobile 10, IE Mobile, Net Front, MicroB and older BlackBerry devices do not support touch events. Through testing, Firefox Mobile 7 was also found to support these events.

2.3.1.2 HTML5

HTML5 is the latest version of HTTP [37], and is still currently under development as of October 2011. It has better support for multimedia than previous versions [37], enabling the creation of rich graphical user interfaces without third party add-ons such as Adobe Flash or Silverlight. HTML5 is therefore important for the extensibility of the GRF, as robot functionality is exposed via a web based user interface, which may require the multimedia additions to function as desired.

According to the compatibility tests done by Firtman [28], the HTML5 multimedia features are supported by all recent smartphone and tablet browsers.

2.3.1.3 M-JPEG Support

Limited testing has revealed that M-JPEG HTTP streams are supported in Firefox 3 - 7, Chrome 10 - 15, Firefox Mobile 7, Opera Mobile 11.5, and Safari 4 - 5. This testing has further revealed that M-JPEG is not supported in Internet Explorer 8, Dolphin 6.2, Opera 11, and the default browser in Android 2.1 - 3.2.

2.3.1.4 GRF Support

One of the goals of this project is to use the GRF to stream video from a camera mounted on a robot to an Android powered device to aid in its remote control. Touch events would most likely be helpful for robot control using a web interface. Based on research and experimentation, Firefox Mobile 7 appears to be the only mobile web browser that
adequately supports both touch events and M-JPEG streaming. According to Mozilla [16], Firefox Mobile is only available on Android devices with an ARMv7 processor and Android 2.1 or above.

2.3.2 Web Services

Web services provide a means to expose functionality over HTTP, and could therefore be used to expose robot functionality. The three kinds of web services that have been investigated are RESTful services, the WS-* stack, and XML-RPC.

2.3.2.1 RESTful Services

Originally introduced as an architectural style for large-scale distributed hypermedia systems, Representative State Transfer (REST) principles have since been adopted for the creation of RESTful web services [36]. The following four principles form the commonly recognised basis of the RESTful web service architectural style [36]:

- **Identification of resources through Uniform Resource Identifiers (URI).** Services are exposed as a set of resources identified by URIs.

- **Uniform interface.** Manipulation of resources is achieved by using one of the HTTP GET, PUT, POST or DELETE operations. GET returns a representation of the resource; PUT modifies a resource if it already exists, or creates a new resource if it does not already exist; POST creates a new resource, and DELETE removes a resource.

- **Self-descriptive messages.** Resources are decoupled from their representation, which allows them to be represented in a number of different formats, such as HTML, Extensible Markup Language (XML), or JavaScript Object Notation (JSON). Resource metadata is also used to perform authentication, control caching, and determining the most appropriate representation format.

- **Stateful interactions through the use of hypermedia.** Client state is not stored on the server, but rather transferred from the client to the server with every resource request. In REST over HTTP, state can be transferred using features such as hidden form fields, cookies and Uniform Resource Locator (URL) query parameters.
Zur Muehlen et al. [42] state that REST has these self-declared advantages:

- a client can receive notifications through one generic listener;
- the only routing information needed by a client is the initial URI;
- it is possible to perform late binding; and
- process instances for dealing with client to server communication are created explicitly.

Zur Muehlen et al. [42] further state that REST possibly has these disadvantages:

- the URI namespace can become cumbersome to manage if many resources are exposed; and
- the number of objects created on the server could be substantial.

Two additional weaknesses that relate particularly to RESTful web services, as identified out by Pautasso et al. [36], are the lack of commonly accepted standards or best practices for building RESTful web services, and the limitations on GET requests.

**Lack of commonly accepted best practices**

Part of this problem can be attributed to the lack of support for HTTP verbs other than GET and POST in certain proxies and firewalls [36]. Hi-REST and Lo-REST implementations have emerged, where Hi-REST utilises all four HTTP verbs (GET, PUT, POST and DELETE), while Lo-REST implementations only utilise the GET and POST verbs [36]. Additionally, the Hi-REST approach advocates the use of “nice” URIs and using Plain Old XML (POX) for message formatting, while the Lo-REST approach enforces the use of MIME-types to indicate the format of the data in a message [36].

**Limitations on GET requests**

Current implementations of HTTP do not allow more than four kilobytes of data to be encoded into a URI [36]. Such a limitation renders the strict following of REST principles unachievable in certain cases, as an idempotent resource request that would exceed the four kilobyte URI limitation would need to be encoded as a POST request instead [36].
Additionally, the lack of a standard marshalling mechanism for complex data structures makes the encoding of such data structures problematic [36].

Web Resource Formats

Two common formats for representing RESTful web application resources are JSON and XML.

**JSON** is a language independent, compact, and text-based data-interchange format based on a subset of the JavaScript programming language [24]. It has been designed to be easily read by humans, and simple for computers to parse [35]. This design has resulted in parsing speeds estimated at 100 times faster than XML [35], although the results of the case study carried out by Nurseitov et al. [35] indicate that it is closer to 40 - 50 times faster. According to Nurseitov et al. [35], there are a number of debatable arguments against the use of JSON, including lack of input validation, lack of namespace support, and weaknesses in extensibility. Another possible drawback of using JSON is the limited set of natively supported data types; it is able to serialise strings, numbers, booleans, null, and two types of structures [24]. The first structure is a collection of key/value pairs, otherwise known as an object, hash table, dictionary, or associative array [24]. The second structure is an ordered list of values, otherwise known as an array, list, vector or sequence [24].

**XML** is a subset of the Standard Generalised Markup Language, and includes human readability and simplicity in its design goals [35]. Unlike JSON, the intent of an XML document forms part of its structure, and as such, it is able to represent user-defined data formats by allowing user-defined markups and encoding schemes [35]. It is thus more flexible than JSON. Nurseitov et al. [35] state that the primary uses of XML are object serialisation and Remote Procedure Call (RPC). However, the use of XML for object serialisation (instead of a simpler format such as JSON) does come with a significant performance penalty, as revealed in the discussion of JSON above.

2.3.2.2 The WS-* Stack

The WS-* stack enables web services to be flexible, loosely coupled between service provider and consumer, and addresses many of the requirements of enterprise computing, such as reliable messaging, quality of service and secure communication [36]. SOAP, a message format that uses XML to encode messages and remote procedure calls,
2.3. CONSIDERATIONS FOR A NEW FRAMEWORK

the basis of communication for the WS-* stack [25]. The SOAP format allows messages to be transport-neutral, and hence be communicated over existing transports such as HTTP and SMTP [25]. The use of the Web Service Description Language (WSDL) to describe web service interfaces aids in abstracting the implementation details by providing a formalised specification on how clients should interact with a service [36][25]. Proxy generators are able to use the WSDL associated with a web service to generate client code that correctly communicates with the remote service [25], simplifying the consumption of web services.

Zur Muehlen et al. [42] state that SOAP has these self-declared advantages:

- components within the system are tightly coupled;
- its structure makes debugging possible;
- it has support for increased privacy;
- existing Application Programming Interfaces (APIs) can be exposed easily; and
- the façade can hide complex operations.

Zur Muehlen et al. [42] further state that SOAP has these potential disadvantages:

- each type of client notification needs its own dedicated port;
- the operations and their associated semantics of the web service need to be known by the client prior to their use; and
- process instances for dealing with client to server communication are created implicitly.

A disadvantage of SOAP not mentioned by zur Muehlen et al. is the significant overhead brought about by SOAP messages. As shown by an experiment carried out by Tian et al. [40], a request-reply pair carrying a total of 589 bytes of data produced 3363 bytes of web service overhead in the form of HTTP and SOAP headers, the XML schema, and brackets. This data overhead, coupled with the performance overhead of serialising and deserialising the SOAP XML, can be significant when consuming web services on mobile devices. In tests performed by Hamad et al. [30], the response times of consuming RESTful web services on a mobile device were less than half the response times of the same services exposed using SOAP over HTTP, with message sizes up to 10 times smaller.
When comparing WS-* web services to RESTful services, Pautasso et al. conclude that WS-* web services are best suited for professional enterprise application scenarios, whereas RESTful services are better suited for ad-hoc, tactical integration over the Internet [36].

### 2.3.2.3 XML-RPC

XML-RPC is a technology specification that allows software in diverse environments to make remote procedure calls over the Internet [18][26]. A Remote Procedure Call (RPC) is formatted as an XML message and submitted to the remote server using an HTTP-POST request, after which the server returns an XML response [26]. One of the primary design concerns of XML-RPC was to make XML-RPC as simple and effective as possible without compromising its ability to transfer complex data [26]. It supports six primitive data types: int, double, string, boolean, datetime, and base64, and two complex data types: struct and array [26].

According to Allman [18], the advantages of XML-RPC are that:

- method requests are constructed dynamically, obviating the need for generating method stubs for the remote services;
- all data is encoded as text before transmission, removing the problem of argument marshalling;
- the use of standard XML encoding techniques makes the system highly interoperable between hosts using the same RPC system; and
- as communication is done in plain text, XML-RPC can be layered above existing application protocols such as HTTP.

According to Allman [18], disadvantages of XML-RPC include the fact that:

- using XML introduces overhead during encoding and decoding operations;
- due to the absence of generated stub methods for remote procedures, a more in-depth knowledge of the underlying system is required of the programmer; and
- XML-RPC’s flexibility and generality comes at the cost of performance.

Dissanaike et al. [26] conclude that XML-RPC is better suited for use in embedded devices than SOAP as it is simpler to use and understand, and has significantly smaller message sizes.
2.3. CONSIDERATIONS FOR A NEW FRAMEWORK

2.3.3 Web Server Setup

The GRF will require a web server to expose robot functionality over HTTP and host an HTML Graphical User Interface (GUI). In order for all communication between the GRF and a web client to occur over HTTP, the web server must support both the pushing of information to a client, and the usual request-reply pattern. This is necessary because clients may want to subscribe to certain robot events. A web service architecture that uses the Comet technology pattern can be used to achieve this. In the Comet model, the server sends data to the client when appropriate rather than the client requesting it [21].

Four open source web servers were investigated for use in this project, namely Jetty, Apache Tomcat, GlassFish, and Node.js.

The Jetty project was started in 1995, and has grown into a reputable web server used by companies such as Google and Yahoo [8][9]. According to the website [9], Jetty provides:

- an asynchronous HTTP server,
- a standards based servlet container,
- an asynchronous HTTP client, and
- support for the Open Services Gateway initiative (OSGi) framework, Java Naming and Directory Interface, Java Management Extensions, Java Speech API, and Apache Java Protocol.

Its developers claim that Jetty has a small footprint and is embeddable [8], which is of particular importance to this project.

Apache Tomcat implements the Java Servlet and JavaServer Pages technologies [2]. It is used by a large variety of organisations to run mission-critical web applications, as can be observed by the organisations listed on the Apache Tomcat PoweredBy page.

GlassFish is a production implementation of the Java EE 6 specification and benefits from many of its advantages, such as flexibility, extensibility, and ease of use [17]. It has a modular architecture based on the OSGi framework, and has support for third-party languages such as Groovy, JRuby and Jython [17].

[^3]: http://wiki.apache.org/tomcat/PoweredBy
**2.3. CONSIDERATIONS FOR A NEW FRAMEWORK**

**Node.js** is a JavaScript event-driven input/output (I/O) framework that runs on the V8 JavaScript Engine [11]. It has extensive support for HTTP, making it a considerable contender as a web server [11]. Unlike many other web servers that utilise threads, it is event-driven and almost never performs I/O directly [11]. When coupled with libraries such as Express\(^4\), it can become a high performance web server that contributes to high class web development [5]. Node.js can be easily installed on Linux, Macintosh, Solaris, and Windows [3]. It can also be installed on other Unix platforms with extra effort [3].

Two Java REST frameworks, Jersey and RESTEasy, were investigated for use in this project. Express was investigated as the Node.js REST framework.

**Jersey** is a production quality open source framework for developing RESTful web services [7]. It was built as the reference implementation for the JAX-RS\(^5\) specification, but also provides an API that allows it to be extended as needed by developers [7]. One of its goals is to make it easy to build RESTful web services using Java [7]. Dynamically adding RESTful resources at runtime does not appear to be possible using the standard annotation method since annotations must be declared with compile-time constants [1].

**RESTEasy** is a fully certified portable JAX-RS implementation that has been created to help build RESTful web services and Java applications [13]. It also includes a client-side framework that makes it possible to use JAX-RS annotations and interface proxies to map outgoing HTTP requests to remote servers [13]. Nevertheless, it appears as though dynamically adding RESTful resources at runtime is not possible for the same reason as stated for Jersey.

**Express** is a framework that facilitates high performance web development for the Node.js platform [5]. It is not advertised as a REST framework, but its features facilitate the creation of RESTful web services. According to the website[5], some of these features include:

- robust routing,
- redirection helpers,
- dynamic view helpers,
- content negotiation,

\(^4\)http://expressjs.com/
\(^5\)http://jcp.org/en/jsr/detail?id=311
2.3. CONSIDERATIONS FOR A NEW FRAMEWORK

- view rendering and partials support.

The two Comet frameworks considered for a Java server are Cometd and the Atmosphere Framework:

**Cometd** is a scalable framework that uses an HTTP-based event routing bus to provide Comet services [4]. It provides implementations of the Bayeux\(^6\) protocol in languages such as Python, Perl, JavaScript and Java [4].

The **Atmosphere Framework** simplifies the building of Comet based web applications that also include RESTful behaviour [20]. It can be used with web servers that support the Java Servlet Specification 2.3 [20], but also contains its own web server, the Atmosphere Spade Server, which contains the Grizzly Web Server, Jersey, and the Atmosphere components [20]. Since it uses the Jersey framework, it too has the restriction of only being able to define RESTful resources before runtime.

2.3.4 Message Passing

Message passing is commonly used for implementing communication between two or more processes in a system. A variety of message passing implementations exist, but the socket protocol and data format options to be considered for a socket-based message passing system for the GRF are described below.

2.3.4.1 Socket Protocol

A number of different socket protocols exist for communication over network sockets. The socket protocol affects how messages on the socket are transported. Different socket protocols differ with attributes such as reliability, congestion control, and state management, as seen below.

**Transmission Control Protocol**

Transmission Control Protocol (TCP) is a reliable, connection-oriented full-duplex protocol for end-to-end communication [27, p. 239]. TCP also provides congestion and flow control on an established connection [27, pp. 261-264].

\(^6\)http://svn.cometd.com/trunk/bayeux/bayeux.html
**User Datagram Protocol**

User Datagram Protocol (UDP) is a very minimalist protocol; a UDP datagram only contains the IP header, source port, destination port, UDP length, UDP checksum and UDP Data [27, p. 269]. UDP is connectionless and unreliable, thus once a UDP datagram has been transmitted, it is forgotten [27, p. 269].

**ØMQ**

A ØMQ socket is built on a TCP socket [31], and hence has the same base attributes. However, unlike TCP sockets, ØMQ sockets provide added functionality such as connection retrying, message queuing, and receiving whole messages as they were sent [31]. There are nine different types of ØMQ sockets, and each ØMQ socket type is only compatible with certain other ØMQ socket types. The different types of ØMQ sockets are PUB, SUB, REQ, REP, ROUTER, DEALER, PUSH, PULL, and PAIR [31].

The only ØMQ socket types that would be of interest are those that deal with one-to-one socket communication. This narrows the list down to REQ, REP, and PAIR [31]. There are only two combinations in which these sockets can be used, REQ–REP and PAIR–PAIR [31].

REQ–REP sockets are used in a request-reply pattern where every reply must first be preceded by an associated request [31]. PAIR–PAIR sockets are used to connect one peer to another peer, and thus the order in which messages are sent to each other does not matter [31]. However, Sustrick and Lucina state that PAIR sockets have been designed specifically with inter-thread communication across an internal ØMQ communication transport in mind, and are thought of as experimental, possibly with broken or missing functionality [39].

**2.3.4.2 Data Format**

Network sockets simply transmit and receive bytes of data, and hence structured data that needs to be sent over a network socket needs to be marshalled into a suitable format before being sent, and parsed back into its original format once it has been received. There are a number of libraries that are able to do this, four of which are discussed below:

**JSON**

Refer to Section 2.3.2.1 for a discussion of JSON.
2.3. CONSIDERATIONS FOR A NEW FRAMEWORK

Protocol Buffers
Protocol Buffers are an open source mechanism created by Google engineers for automated and efficient serialisation of structured data [6]. According to Google [6], they are language-neutral, platform-neutral, and extensible. Use of Protocol Buffers requires the structure of Protocol Buffers messages to first be defined in a protocol file, or .proto file [6]. This file stores information such as message type names, the properties and their value types, whether the property is optional, required or repeated, and lastly the unique id of each property [6].

The types of values allowed in Protocol Buffer messages are integers, floats, booleans, strings, raw bytes, and other Protocol Buffer message types [6]. Once the protocol file has been created, a Protocol Buffer compiler is used to generate classes or modules for a particular programming language. These classes (or modules) provide accessors for each field in a Protocol Buffer message, methods to serialise data into a binary Protocol Buffer message, and methods to parse a binary Protocol Buffer message back into a usable programming structure [6].

MessagePack
MessagePack is a fast and efficient object serialisation and deserialisation library [10]. Objects are dynamically serialised into binary without a pre-defined message template by using embedded type information [29]. The library also provides RPC functionality with synchronous, asynchronous, and callback styles [29]. In a particular benchmark test, MessagePack was found to be up to four times faster than Protocol Buffers, and almost 11 times faster than JSON [10].

Thrift
Thrift was built for cross-language service deployment by Facebook [14]. It was open sourced in April 2007, and later entered the Apache Incubator in May 2008 [14]. Much like Protocol Buffers, the structure of Thrift messages is pre-defined and input into code generation software to build language specific classes to handle data specific to those Thrift structures [38]. The RPC-style services exposed to Thrift clients must also be defined in this file [38]. According to the maintainers [14], their goal was to make “reliable, performant communication and data serialization across languages as efficient and seamless as possible”.

2.4 The WifibotLab LIDAR

The WifibotLab LIDAR\(^7\), hereafter referred to as the Wifibot, was the robot used as a testbed for the GRF, and is the target of the sample implementation.

The version of the Wifibot we used has the following specification:

- Commell LE-376 Embedded Intel Atom Miniboard
- Intel Atom\(^{TM}\) D510, 1.66GhHz, 1 MB cache
- 2 GB of DDR2-667 memory
- 4 m URG-04LX Light Detection and Ranging (LIDAR)
- 2 x Infrared (IR) sensors
- Atheros AR5413 a/b/g WiFi card
- Logitech QuickCam\(\copyright\) Orbit AF
- 4 x 12 V motors
- 2 x Hall encoders

The Wifibot’s operating system is Ubuntu 10.04 LTS. On boot, the Wifibot launches a daemon which listens on TCP port 15020. This daemon allows a connected user to send movement commands to the Wifibot, which in turn responds with Wifibot status information. This status information includes IR readings, the battery voltage, odometry readings, and wheel speeds.

Webcam access and control is provided by a web application that listens on port 8080. No access to the LIDAR readings was provided by software on the Wifibot. Access to this data was made possible by downloading a URG C++ library that communicates directly with the LIDAR through a USB-to-Serial communication port.

\(^7\)http://www.wifibot.com/wifibot-wifibotlab.html
2.5 Acer Iconia Tab A500

A tablet device, the Acer Iconia Tab A500, was used to test the usability of the GRF on a mobile device.

It has the following specification:

- 10.1" multi-touch screen
- Dual-core NVIDIA Tegra™ 2 CPU
- NVIDIA Geforce GPU
- 1 GB RAM
- Android Honeycomb 3.2 OS
- 16 GB internal storage
- WiFi 802.11 a/b/g/n card
- A-GPS

2.6 Summary

The current attempts at generalised frameworks for computer-to-robot communication are too specialised for mass adoption and standardisation. This provides an opportunity to exploit modern technologies and development patterns, such as the ones described in this chapter, to create a new computer-to-robot communication framework that is more generalised, and exposes robot functionality over HTTP.
Chapter 3

GRF Design and Implementation

3.1 Introduction

The aim of this project is to build a generalised, easy-to-use robot control framework that exposes robot functionality over HTTP. This aim imposes certain requirements on the design of the GRF, which are discussed in the first section of this chapter. This is followed by a broad overview of the GRF's design, and a discussion of the design decisions made given the options investigated in Chapter 2. A detailed specification of how to implement a GRF compatible robot module is then given. The chapter continues with a brief description of each module in the project and how it was implemented. The software requirements for running the GRF are then listed and briefly explained. Finally, a description of each file in the project is given and their relationships illustrated.

The GRF design was created using the rapid prototyping methodology. This methodology was chosen because it allowed the design to be easily changed in the face of any unforeseen technical problems, and ensured the design was kept as simple as possible.

3.2 GRF Requirements

The requirements of this framework are that it should expose robot functionality over HTTP, be efficient, strictly open source, generalised, and easy to use. The goal of these requirements is to ensure that the GRF can be used to provide the same easy-to-use interface to robot functionality on a variety of robots running different platforms.
3.3 GRF Overview

The GRF is split into three primary components: the Web Interface, the Robot Interface, and the Robot Instance, the relationship between which is illustrated in Figure 3.1.

![GRF Overview Diagram](image)

Figure 3.1: GRF overview

3.4 Design Decisions

In this section, we explain the reasoning behind the choices made for the GRF design.

3.4.1 Suitability of HTTP

One of the primary goals of this project is to expose robot functionality over HTTP. A number of advantages and disadvantages of using HTTP have been identified and should be noted.

Advantages

- It is widely used and understood as it is the foundation of communication over the World Wide Web.
3.4. DESIGN DECISIONS

- Software that uses HTTP (such as a web browser) is prevalent.
- The success of the protocol has resulted in it being unblocked on most Internet firewalls if recognised HTTP ports are used.
- HTTP communication is done in plain text, enabling it to be easily read and understood.

Disadvantages

- The HTTP header can add significant overhead to otherwise small requests or responses.
- An HTTP server can only send data to a client in response to a client request.
- It performs poorly compared with a UDP-based binary protocol when streaming loss-tolerant data such as audio and video.

3.4.2 GRF API

In Chapter 2, three options were investigated for exposing robot functionality over HTTP, namely RESTful web services, the WS-* stack, and XML-RPC. The WS-* stack was found to be too complicated, data heavy, and slow compared to the alternatives. The experiments performed by Hamad et al. [30] showed that RESTful services responded twice as fast as SOAP-based services with data transfers as much as 10 times smaller. RESTful web services use the inbuilt functionality of HTTP. This obviates the need for specialised client-side software (other than a web browser), and makes it simple and fast.

In contrast, XML-RPC uses HTTP as a transport for an XML encoded RPC request or response, thereby requiring additional client software and extra processing. A RESTful API was therefore favoured over XML-RPC and SOAP-based web services.

3.4.3 Web Server and Programming Language

A number of web servers were investigated in Chapter 2, namely Jetty, Apache Tomcat, GlassFish and Node.js. Jetty, Apache Tomcat, and GlassFish are all Java web server containers, and hence rely on other projects to provide REST and Comet patterns. The
Java REST frameworks investigated were RESTEasy and Jersey. Research into these frameworks revealed no easy way to add RESTful resources dynamically at runtime. The ability to add request handlers to URIs during runtime is a necessary requirement of the GRF because it will only have knowledge of the resources it needs to expose once it has connected to a Robot Instance and received its resource list. It may be possible to circumvent this requirement by adding catch-all resources for each type of HTTP request, and handling requests to these resources based on the URL path requested. However, such a solution is undesirable as it would require developing a custom URL pattern matching engine and parser to extract URL parameters and match a requested URL to a robot resource. This is all done by the web server if the resource URLs are known beforehand.

Node.js allows the creation of a very simple web server, and coupled with the Express module, it can expose RESTful resources at runtime very elegantly, as shown in Listing 3.1. For comparison, the Java equivalent with a statically defined RESTful resource is shown in Listing 3.2. Since Node.js is an event-based programming language, no extra functionality is required to support the Comet pattern. This is because writing the response to a web request can be placed in a callback function that is only executed once the response to the robot request has been received.

Thus, given that Node.js coupled with Express provides the functionality needed by the GRF in a simple manner, it was chosen over the Java solutions, which had complications in dynamically exposing the robot resources.

```javascript
var app = require('express').createServer();
app.listen(80);
var resourceURL = '/My/Resource';
app.get(resourceURL, function(req, res) {
    res.send('Hello World');
});
```

Listing 3.1: Simple resource exposed using Node.js and the Express module

```java
//The contents of @Path must be a compile-time constant
@Path("/My/Resource")
public class HelloWorldResource {
    @GET
    @Produces("text/plain")
    public String getResource() {
        return "Hello World";
    }
```
3.4. DESIGN DECISIONS

3.4.4 GRF Coupling with the Robot Instance

In an early design of the GRF, the Robot Interface imported an exposed GRF module and communicated with it directly. It was decided that this should be changed since it restricts the implementation of the Robot Instance to the same programming language as the GRF, namely JavaScript. In the final design, the Robot Instance communicates with the Robot Interface over a TCP socket with messages serialised using Protocol Buffers. The programming language for the Robot Instance is thus only restricted to languages with support for Protocol Buffers.

3.4.5 Socket Type

The socket types investigated in Chapter 2 were UDP sockets, TCP sockets, and ØMQ sockets. Reliable messaging is a desirable attribute of the socket transport since we do not want requests failing to arrive at the Robot Instance, or responses failing to get back to the Web Interface. Out of the three choices, TCP sockets, and by extension ØMQ sockets, provide reliable messaging. Compared to TCP sockets, ØMQ sockets require less management since they will auto-retry connections, queue messages, and receive whole messages as they were sent [31]. However, their usage pattern only permits one response per request. This becomes problematic when data needs to be streamed from the Robot Instance to the Robot Interface. In such a scenario, the Robot Instance needs to send multiple responses (chunks of the stream) to a single request. The only way to handle this use case with ØMQ sockets is to have a REQ - REP socket pair for requests, where the REQ socket is in the Robot Interface and the REP socket is in the Robot Instance, and the reverse combination (REP - REQ) for responses. Although this would work, it requires the use of two communication ports, thereby adding more network connection management code and complexity. TCP sockets do not have this problem, and were therefore chosen as the socket type for communication between the GRF and the Robot Instance.
3.4.6 Message Serialisation Format

The options for message serialisation reviewed in Section 2.3.4.2 were JSON, Protocol Buffers, Thrift and MessagePack. Given the simplicity and speed of MessagePack, it would have been the most suitable serialisation library to use. However evaluation of node-msgpack\(^1\), the Node.js module for MessagePack, revealed that it has a flaw that causes it to deserialise what was initially a binary buffer into a Unicode string. This flaw renders the transfer of binary problematic, and transferring binary is crucial for functionality such as camera streaming.

JSON is a text-based serialisation format \([24]\). This format is not able to represent binary data since it would be interpreted as a string, thus possibly containing reserved characters. Binary data therefore needs to be encoded into a different format before being added to a JSON message. The encoding and decoding of binary data would increase the size of JSON messages and decrease the performance of serialising and parsing. Since potentially large quantities of time-sensitive binary data would need to be transported between the GRF and the Robot Instance for streaming resources, this increase in message size and decrease in performance is undesirable.

Thrift is similar to Protocol Buffers, except RPC services form part of its core usage. To use Thrift, a Thrift server must be created that exposes one or more RPC methods \([38]\). A Thrift client is then able to call one of these RPC methods and receive a response, but the client cannot expose RPC methods of its own \([38]\). This means that two-way communication between two running programs can only be achieved if they both have Thrift servers as well as clients, resulting in added complexity and the use of two communication ports.

The Protocol Buffer format places little emphasis on RPC, and therefore more closely fits the role of a message serialisation format. Protocol Buffer messages can be sent or received from either side of a TCP connection since transport of serialised messages is left to the user. Requests made from the Web Interface to the Robot Instance for a streaming resource requires the robot to send multiple replies. Since sending multiple replies to a single request is not possible using Thrift RPC, it was decided that Protocol Buffers best fits the usage requirements of the GRF. The Protocol Buffer message structures used in the GRF can be seen in Appendix A.1.

\(^1\)https://github.com/pgriess/node-msgpack
3.5 Robot Instance Specification

There are two methods that can be used to develop the Robot Instance. The first method, covered by Use Case 1 discussed below, is to develop the entire Robot Instance. This entails writing the code to communicate with the GRF, as well as the code to interface with the robot. The second method, covered by Use Case 2 discussed in Section 3.5.2, is to develop only the Robot module and use an existing module to handle communication with the GRF. In the GRF prototype, this option is only viable if Node.js is used as the programming language for the Robot Instance.

3.5.1 Use Case 1: Developing the Entire Robot Instance

Developing the entire Robot Instance is necessary if the GRFInterface module is not available for a selected programming language. In this section, we describe what needs to be done to realise network communication and subsequent operation of a Robot Instance.

Network Communication

3.5.1.1 Connection Setup

- A TCP socket must listen on a particular port; the standard port is 9999. If a non-standard port is chosen, the WebInterface instance’s appParams.roboInstancePort field must be set to the new port.

- Robot Interface connections to the above port must be accepted and kept alive.

3.5.1.2 RoboRequest Message

The RoboRequest message is sent by the GRF to the Robot Instance and has the following structure:

**id** : integer - Represents the id number of the request.

**operation** : string - Represents the operation requested. This could be either a GRF specific operation or a robot specific operation.
3.5. ROBOT INSTANCE SPECIFICATION

**parameters**: string - This is a JSON serialised object containing the parameters specified to this operation. For a robot operation, this object will always contain a ‘method’ property, which indicates the type of HTTP request made by the user.

### 3.5.1.3 RoboResponse Message

The RoboResponse message is sent from the Robot Instance to the GRF in response to a RoboRequest message. It has the following structure:

- **id**: integer - This should be the same as the id value in the corresponding RoboRequest message.

- **response**: string - This is a JSON serialised object containing the response of the RoboRequest.

- **binary**: byte[] (optional) - This field is used when the response is only binary. If this field has a value, the value in the **result** field will be ignored.

The JSON object in the **response** field has the following structure:

- **result**: string - This specifies whether the robot request succeeded or failed. If it succeeded, this field should be ‘success’, otherwise it should be ‘failed’.

- **data**: string (optional) - This field represents the response data that will be sent back to the requesting user.

- **error**: string (optional) - When the **result** field is ‘failed’, this should be set to the reason for failure.

### 3.5.1.4 Receiving Messages

![Figure 3.2: RobotRequest/Response messages](image)

- A full message contains a 4-byte integer followed by the RoboRequest message. The 4-byte integer represents the size of the RoboRequest message to follow. This is illustrated in *Figure 3.2*. 
3.5. ROBOT INSTANCE SPECIFICATION

- The 4-byte integer must be read from the socket, and then the exact number of bytes indicated by the integer must be read from the socket and parsed as a RoboRequest message.

- The structure of RoboRequest messages is described in Section 3.5.1.2.

3.5.1.5 Sending Messages

- Messages to be sent must be formatted as described in Section 3.5.1.3, and serialised as RoboResponse Protocol Buffer messages.

- The size of the RoboResponse message must first be written onto the socket as a 4-byte integer, followed by the RoboResponse message itself.

- The \textit{id} field in the RoboResponse message must match the id of a previously received RoboRequest message. The same id cannot be used more than once if the RoboRequest it represents references a non-persistent robot resource.

Operation

3.5.1.6 Handling Requests

When a RoboRequest is received, the \textit{operation} field of the message should be inspected to determine how the request should be handled. The operation should first be checked against the list of special GRF requests described in Section 3.5.1.7, and if no match is found, checked against the list of exposed robot resources. If a match is found, the appropriate handling method should be called. The response received from the handling method should then be wrapped as a RoboResponse message, as described in Section 3.5.1.3, and sent to the Robot Interface.

3.5.1.7 Special Requests

The following special requests may be made by the GRF, and will be specified in the \textit{operation} field of a RobotRequest message:

- 'InstanceInfo': This must return information about the Robot Instance in the form of a JSON object. The object must have the following properties:
3.5. ROBOT INSTANCE SPECIFICATION

robotName : string - This should be set to the name of the robot that the Robot Instance represents.

version : string - The version number of the Robot Instance.

author : string - The author of the Robot Instance.

contact : string (optional) - Contact information for the author.

extra : string (optional) - A JSON object containing extra information about the Robot Instance that the author would like to provide.

• ‘GetResources’: This must return the robot resources in the form of a JSON array of objects. Each object must have the following properties:

  path : string - Represents the RESTful resource path, with parameters denoted by a colon followed by the parameter name. For example /Sensors/:id would match /Sensors/1, /Sensors/2, and so forth. Regular expressions can be used to restrict valid parameter values\(^2\). If a regular expression is used, the entire URL must be defined in terms of a regular expression.

  regex : boolean - If the path is defined as a regular expression, this must be set to true.

  persistent : boolean - Represents whether the resource requires multiple responses. True represents multiple responses.

  contentType : string (optional) - The HTML mime type of the resource. This is defaults to ‘application/json’.

  method : string - The HTML method that must be used to access the resource. It must be one of ‘GET’, ‘POST’, ‘PUT’, or ‘DELETE’.

• ‘CloseStream’: This must stop the robot from streaming responses back to the Robot Interface for a persistent request identified by the \(id\) field.

3.5.1.8 High-Level Flow

The Robot Instance needs to follow the high level steps given below.

1. Start listening on a TCP socket and accept incoming GRF connections.

\(^2\)Examples of valid regular expressions can be found here: [http://expressjs.com/guide.html#routing](http://expressjs.com/guide.html#routing)
2. Continuously read from the socket as specified by Section 3.5.1.4. Follow steps 3 - 4 for RoboRequest messages received.

3. Inspect the operation field of the message and determine whether it is a special operation. If it is, call the corresponding method, otherwise call the robot method to which it refers, passing it the parameters object specified in the parameters field as a parameter.

4. The response to the request must be wrapped into the format specified by Section 3.5.1.5. The response id must be the same as the id specified in the request.

The operation of the Robot Instance is illustrated in Figures 3.3 and 3.4 below.

Figure 3.3: The high level operation of the Robot Instance
3.5. ROBOT INSTANCE SPECIFICATION

3.5.2 Use Case 2: Using the Existing GRFInterface Module

The GRFInterface module for Node.js can be found in the GRFInterface.js file. This module manages the connection and communication with the Robot Interface, thereby simplifying the creation of a Robot Instance. To build a Robot Instance using this module, the following needs to be done:

1. A module must be created that provides access to all of the functionality of the robot for which the instance is being built. Each robot resource must have an associated prototype method that accepts the following arguments:

   self : A reference to the current object instance.
   id : integer - The id of the resource request.
3.5. ROBOT INSTANCE SPECIFICATION

parameters : object - contains the parameters (if any) specified in the resource URL.

cb : function - This must be called with the result of the robot request. It accepts
an object as a parameter with fields id, responseObject, and an optional binary
field. These fields correspond to the fields in the RoboResponse message defined
in Section 3.5.1.3.

closed : function (optional) - This function must be called with a function as
a parameter for resources that are persistent. The function passed in as a
parameter will be called when the request for the stream has ended.

2. The module must import and create an instance of the GRFInterface module, called
GRFComm here for ease of reference.

3. GRFComm’s RobotObj attribute must be set to the current instance of the module
that provides the robot functionality.

4. GRFComm’s ResourcesList attribute must be set to an object that contains the
robot resources in the following form:

Path : { regex: true/false, GET : ResourceProperties, POST: ResourceProperties,
PUT: ResourceProperties, DELETE: ResourceProperties }

where ‘Path’ is the resource URL, and all of the properties within Path are individually
optional, but at least one HTTP method must be defined. ‘ResourceProperties’
must be an object with the same fields as those described in Section 3.5.1.7 under
‘GetResources’, but without the path and method fields, and including a new field:

fn : string - Represents the name of the method that must be used to handle
requests for this robot resource.

5. GRFComm’s InstanceInfo attribute must be set to an object with the same structure
as described in Section 3.5.1.7 under ‘InstanceInfo’.

6. Starting the instance is achieved by calling GRFComm’s start method.

A sample implementation is provided in Appendix A.2.
3.5.3 The Graphical User Interface

By default, the GRF’s Web Interface will attempt to serve a file in the current directory named ‘robotUI.html’ when a request is made to the base URL ‘/’. Thus, a GUI can be created by writing an HTML web interface with its starting page named ‘robotUI.html’. The Web Interface allows CSS files to be placed in a subdirectory called ‘Styles’, script files to be placed in a subdirectory called ‘Scripts’, and images to be placed in a subdirectory called ‘Images’.

3.6 Project Modules

A graphical illustration of the modules within the GRF and Robot Instance, and their interaction, is shown in Figure 3.5. This section provides a brief description of each module, and details on how important methods within each module were implemented. Following a particular JavaScript programming convention, methods that should only be called from within the module (otherwise known as private methods) have a preceding underscore in their name.

3.6.1 The Web Interface

The Web Interface provides the interface to the robot’s functionality through RESTful resources. It does this by exposing the list of resources it receives from the Robot Interface as web resources. If a request is made to one of these resources, the web server passes information about the request to the Robot Interface and waits for a callback with the response. Once the response is received, it sends the response to the requesting client. The implementation details of important methods within the module are described below.

WebInterface()

The WebInterface constructor sets the initial parameters, creates an instance of the RobotInterface module, specifies the onConnect and onError event handler
functions for the RobotInterface, and calls the init method of the RobotInterface instance.

_getResources(cb)

This method requests the resource list from the Robot Instance, and adds the resources it receives to its internal resource array whilst trying to ensure that duplicate resources are not added. Once completed, the callback function (given as a parameter) is called.

_exposeResources()

This method is responsible for exposing the resources contained in the internal resource array as requestable RESTful resources using the Express Node.js module. The path attribute of each resource object specifies the unique resource URL, and the method attribute specifies the HTTP method that must be used to interact with the URL. The request handler for these resources is specified as being the _handleRequest method.

_handleRequest(req, res, resource)

The _handleRequest method is called for every RESTful resource request. Its purpose is to forward the request to the Robot Instance and to write the appropriate response to the requesting user once the robot’s response is received. If the requesting user terminates a connection to a persistent RESTful resource, the closed callback is called to signal the Robot Instance to stop sending data for that particular persistent request.

startWebServer()

This public method starts the web server. Requests to the base URL ‘/’ return a file named ‘robotUI.html’ by default. However, if the Robot Interface has not connected to a Robot Instance, requests to the base URL will return a service unavailable message.

3.6.2 The Robot Interface

The Robot Interface is a proxy for the Robot Instance. Upon initialisation, it attempts to connect to the Robot Interface over a TCP socket. Once a connection has been established,
the Web Interface can use the Robot Interface to send and receive requests to and from the Robot Instance, respectively. Thus, its primary purpose is to perform connection and state management. The implementation details of important methods within the module are described below.

RobotInterface()

This is the RobotInterface constructor. It creates the TCP socket that connects to the Robot Instance, creates callback functions for socket events, and sets several module attributes.

init(errorCb, connectCb)

The init method is called by the Web Interface to initiate the connection to the Robot Instance. It requires an error callback and a connect callback as parameters. The error callback is used to pass any errors encountered to the Web Interface. The connect callback is called once a connection has been successfully established to a Robot Instance.

_sendRequest(persistent, cb, operation, parameters)

This method sends a request to the Robot Instance. It creates a RobotRequest with the given input parameters, and adds an object containing the cb and persistent input parameters into the PendingRequests dictionary using a local request counter, id, as the index. This is done so that the callback can be used once the response to the request is received. The id request counter value is incremented in preparation for the next request, and the current request is written onto the socket using the _writeRequest method.

_onConnected(socket)

This method is called once the connection to the Robot Instance has been established. Its purpose is to read the network stream and extract RobotResponse messages by reading the message size followed by the message itself, as described in Section 3.5.1.4. Once the message has been read from the socket and parsed as a RobotResponse message, the callback stored in the PendingRequests object for the associated RobotRequest id is used to pass the contents of the RobotResponse to the Web Interface.
3.6.3 The Robot Instance

The Robot Instance implementation is not part of the GRF as it is robot specific. However, a skeleton implementation is provided for Node.js. The skeleton implementation is separated into two modules.

**GRF Interface**

This module handles all the communication with the Robot Interface. It listens for connections from the Robot Interface on a specified address and port. Once it is connected to the Robot Interface, it simply passes robot commands or requests to the `Robot` module with an associated callback for the response. The implementation details of important methods within the module are described below.

**GRFInterface()**

The constructor sets the default properties and application parameters. It further creates a TCP server, specifies the event handlers for the TCP socket once it has connected, and specifies how RoboRequest messages are extracted from the socket stream. RoboRequest messages received on the socket are passed as a parameter to the `_handleRequest` method.

**_sendResponse(result)**

This method wraps the response from a robot request into a RoboResponse message, and sends it to the Robot Interface as described in Section 3.5.1.4.

**_handleRequest(req)**

The `_handleRequest` method is responsible for handling incoming RobotRequest messages. It accepts a RobotRequest message as a parameter and determines whether the request is a GRF request or a robot request. If the request is a GRF request, it responds with the requested information immediately (if a response is warranted), otherwise it calls the relevant robot module function as specified by the resource object to which that particular request relates. Requests that cannot be matched as robot or GRF requests are logged.

**_init()**

The `_init` method processes the `ResourcesList` object set by the `Robot` module, and creates a separate resource list, which is formatted to match the resources specification in Section 3.5.1.7. This new list will be returned when the GRF requests the list of robot resources.
Robot Module
This module is specific to a particular type of robot. The GRF Interface maps requests from the Robot Interface to methods exposed by this module, enabling robot control. It is also responsible for giving the GRF Interface information about itself and the robot functionality.

3.7 Software Requirements

3.7.1 Node.js

Node.js is the cross-platform server-side JavaScript runtime that is used to run the GRF. At the time of writing, Node.js version 0.4.12 was used; earlier versions are not guaranteed to work.

3.7.2 Protocol Buffers

Protocol Buffers are a cross-platform mechanism for serialising structured data into an efficient binary format, and deserialising the binary result back into structured data. This is used for all network communication in the GRF as it greatly simplifies the transfer of structured objects.

3.7.3 NPM

The Node Package Manager (NPM)\(^3\) is used to install Node.js modules.

3.7.4 Node.js Modules

The Node.js modules listed below are used by the GRF.

- Binary: This is used to extract entire Protocol Buffer messages from the network stream.

\(^3\)http://npmjs.org
• Buffers: This module exposes a new data structure that has the ability to store and concatenate multiple Node.js Buffer objects. It is used for aggregating stream data.

• Express: This is used to expose robot functionality as web resources.

• protobuf-for-node: This module uses the system Protocol Buffers libraries to add Protocol Buffers support to Node.js. It is used to serialise and parse messages sent and received over the network between the Robot Instance and the Robot Interface.

3.8 File Structure

The files contained within the project are described below, followed by their directory organisation. The file structure is illustrated in Figure 3.6.

3.8.1 Files

3.8.1.1 WebInterface.js

The WebInterface.js file contains the WebInterface module, which exposes the robot resources it receives as RESTful web resources. It requires the RobotInterface.js file to function. If needed, this file could be swapped out with another in order to expose robot resources differently.
3.8.1.2 RobotInterface.js

The RobotInterface.js file contains the RobotInterface module, which handles the communication between the GRF and the Robot Interface. Since the WebInterface.js file is dependent on it, any changes to the file must be made with caution as they could break functionality within the WebInterface and Robot modules.

3.8.1.3 GRFInterface.js

This file contains the GRFInterface module, which provides a reference implementation of how communication can be done with the GRF. The file’s intended use is to be imported by another module which has, or can call, methods that interact with a robot in order to expose that functionality to the GRF. Modifying the file can therefore break any robot implementation that uses it.

3.8.1.4 Wifibot.js

This file forms part of a Robot Instance and contains the Wifibot module. It imports the GRFInterface module from the GRFInterface.js file to handle the communication with the GRF. The requirements placed on this file are listed in Section 3.5.2.

3.8.1.5 robocomm.desc

This is the compiled Protocol Buffers data structure file that describes the RoboRequest and RoboResponse data structures.

3.8.1.6 ShellImplementation.js

The ShellImplementation.js file is a skeleton robot module that can be used as a base for a Robot Instance implementation.

3.8.1.7 robotUI.html

This file is what the WebInterface module presents to clients visiting the base URL of the web interface, such as http://RobotAddress/. It is completely independent of the GRF, but it is expected to expose the robot’s functionality via an HTML GUI.
3.8.2 Organisation

In order for Node.js to recognise the modules within WebInterface.js, RobotInterface.js, GRFInterface.js and Wifibot.js, these files need to be placed into a node_modules directory. This can be the global Node.js node_modules directory, or a node_modules directory located in the current directory. The robotUI.html file needs to be placed in the same directory as the file that imports the WebInterface module. Similarly, the robocomm.desc file must be located in the same directory of any file that imports the RobotInterface or GRFInterface module.

3.9 Summary

This chapter focused on the design and implementation of the GRF. The most suitable design for the GRF was found to be one where the robot specific code (the Robot Instance) runs as a separate process to the GRF. This design was chosen because it allows the Robot Instance to be developed in any programming language, and increases the framework’s flexibility. Further design decisions include the use of a RESTful API to expose robot functionality, use of the Node.js programming language coupled with the Express module to host the RESTful API, use of TCP sockets for communication between the GRF and the Robot Instance, and finally the use of Protocol Buffers to serialise messages for transport between the GRF and the Robot Instance.

The chapter continued by providing the specification for developing a Robot Instance. This was followed by descriptions and implementation details of the project modules, and the software requirements of the GRF. Finally, an overview of the file structure for the project was given.
Chapter 4

Analysis

4.1 Introduction

In this chapter, we investigate various attributes of the GRF, such as ease of implementation, usability, performance, and generality. To evaluate the work involved in deploying the GRF, a case study was done by walking through the deployment of the GRF on the Wifibot Lab LIDAR. The usability of the GRF was evaluated by comparing the code needed to execute simple robot commands using the GRF versus executing the same commands without using the GRF. The performance was tested by measuring the time taken to execute robot commands using the GRF versus not using the GRF, and making deductions thereof. Finally, the generality of the GRF was assessed.

4.2 Case Study: The Wifibot Lab LIDAR

As described in Chapter 2, the Wifibot Lab LIDAR is a robot capable of movement, with a camera, IR sensors and a LIDAR. Using this robot, a case study was done to evaluate the work involved in exposing its full functionality though the GRF, and the benefits of doing so.

4.2.1 Accessing the Wifibot’s Functionality

The first task was to determine how to communicate with the robot using its existing API. The robot’s functionality is not consolidated into a single API, and hence accessing
all of it required interacting with three different interfaces.

4.2.1.1 Movement and Sensors

The documentation specifies that the Wifibot listens for connections on TCP port 15020 and UDP ports 15000 and 15010 [41]. The TCP port provides the same amount of access to the robot’s functionality as the UDP ports, and so it was preferred over the UDP ports as it provides reliable transport and only requires one socket. The documentation states that the Wifibot accepts a nine-byte command which specifies the speed of the motors, to which it will reply with a 19 byte robot status message if the command was valid [41]. The last two bytes of the command are specified to be the Cyclic Redundancy Check (CRC) bytes for the message, but the documentation states that these bytes are only used for UDP connections [41]. After many unsuccessful attempts at communicating with the robot on the TCP port using the given specification, the robot source code was investigated, and it was found that the CRC bytes are in fact checked on TCP connections. Once the CRC bytes were calculated correctly, movement commands could be successfully sent to the Wifibot, and the corresponding Wifibot status replies read.

The Wifibot status message contains general information such as the battery voltage, the battery current, and the robot server version. The rest of the information contained in the message pertains to either the left or right side of the robot. Each side has information about the motor speed, the first IR sensor, the second IR sensor, and the odometry reading for the motors driving the wheels on that side.

4.2.1.2 Camera

Access to the camera stream is provided by an existing M-JPEG camera server running on the Wifibot on port 8080, powered by a Common Gateway Interface (CGI) application. Camera commands, such as panning and tilting, are issued to the camera through the CGI application located at the ‘/?action’ URL resource. For example, to tilt the camera upwards, the following GET request must be issued:

http://wifibot:8080/?action=command&command=tilt_minus

A direct HTTP M-JPEG camera stream can be established by issuing a GET request to:

http://wifibot:8080/?action=stream
4.2. CASE STUDY: THE WIFIBOT LAB LIDAR

4.2.1.3 LIDAR

The robot did not come with any software to access the LIDAR data. In order to access this data, the LIDAR manufacturer’s API source code was downloaded and compiled on the Wifibot. The source code package contained examples that dumped LIDAR data to standard output, and this is what was used to gather LIDAR data.

4.2.2 Writing the Wifibot Module

The GRFInterface module was used for the implementation, which greatly simplified the work needed to expose the robot’s functionality to the GRF. The greatest amount of the implementation work involved writing the Wifibot module, which contains methods for accessing all of the robot’s functionality. The subsections that follow describe a particular aspect of the Wifibot implementation.

4.2.2.1 Wifibot API Connection

As sending movement commands to the robot requires a TCP connection to the Wifibot API, the Wifibot module either has to ensure an established connection is always available, or establish a connection whenever one is needed. The latter approach was found to be unsuitable because the Wifibot API rejected new connections that closely followed a connection teardown. This meant that retrying to connect caused an unacceptable delay of one or two seconds before a new connection was established successfully.

The first option was to always keep a connection alive. This approach requires a ‘ping’ message to be sent to the Wifibot at a regular interval during periods of inactivity to ensure the connection does not timeout, and this is what is done by the _pingWifibot method within the Wifibot module. However, since the Wifibot API will tear down the TCP socket if it receives an invalid command, the ping message needs to be a valid command. The ping message was therefore set to command the Wifibot to stop moving. This solution was found to be perfectly adequate provided the right ping interval was chosen (we used 320 ms), and the latency between successive movement commands is relatively low. When the latency between successive commands is higher than the ping interval, the robot movement becomes erratic due to ping commands being sent to the Wifibot in between multiple movement commands, temporarily setting the motor speeds to zero. This issue surfaced often when controlling the Wifibot over the Internet, or using a slow device such as an older generation smartphone to control it.
4.2.2.2 Movement

The `Move` method within the `Wifibot` module exposes movement functionality to the GRF. It was implemented by encoding the left and right wheel speed input by the user into a nine-byte array, as specified by the Wifibot protocol [41], and writing the array to the Wifibot communication socket.

4.2.2.3 Sensor Data

The Wifibot API does not allow querying of information from a particular sensor. Instead, the Wifibot sends all of the sensor data in reply to any valid robot command it receives. Thus, it was decided that the sensor information should be cached locally in the `Wifibot` module, and updated with the new status information received following the execution of a Wifibot command. Since a command will be sent to the Wifibot at least every 320 ms, the sensor data is updated regularly. The `Status`, `IR`, `Odometry`, and `Speed` methods within the `Wifibot` module provide access to the Wifibot status information and were made to return the cached information.

4.2.2.4 Camera Stream

The Wifibot already exposes the camera stream over HTTP, thus the `Wifibot` module merely needs to pass data it retrieves from the Wifibot’s M-JPEG server to a requesting client. In order to do this efficiently, an object called `cameraStreamer` was created. The `cameraStreamer` object contains the logic that continuously sends camera data to the stream listeners (the list of clients that have requested the camera stream resource). It does this by connecting to the existing M-JPEG stream on port 8080, and sending the data it receives to all of the stream listeners. The camera stream data is binary, and so the separate binary field available in response messages is used to carry the stream data. To prevent new camera stream listeners from receiving partial initial images, they only start receiving data once a new image starts. The `cameraStreamer` disconnects from the existing M-JPEG stream when there are no more listeners, and reconnects as soon as a listener is added.

This approach to camera streaming has the drawback that the camera stream is slightly delayed due to the indirect path the stream data takes to get to the end-user. It has also been found to be unstable, with clients reporting that the stream connection has been
terminated by the remote server after a seemingly random period of time, and the server reporting that the client aborted the connection. A better approach might have been to transcode raw camera data into a video format supported by HTML5 directly from the camera port (perhaps using software such as ffmpeg\(^1\)), and send it to connected clients. However, doing so is beyond the scope of this project as the Wifibot Robot Instance is only meant to be a simple sample implementation.

Access to the camera stream is provided by the CameraFeed method within the Wifibot module. This method simply adds an object (containing pertinent information about the camera feed request) to the stream listeners dictionary within the cameraStreamer object. It also specifies how to remove a stream requestor from the stream listeners dictionary in the case that the request for the stream ends.

### 4.2.2.5 LIDAR

The LIDAR was exposed by the LIDAR and LIDARStream methods within the Wifibot module. The LIDAR method responds to LIDAR data requests with the output of a program available on the Wifibot that prints LIDAR data to standard output. Streaming of LIDAR data is handled by the LIDARStream method by simply requesting LIDAR data at a regular interval and sending the chunks of LIDAR data to the requesting client.

### 4.2.2.6 Connecting the Wifibot Module to the GRF

Once the Wifibot module provided access to all of the Wifibot functionality, we merely needed to set several attributes of a GRFInterface instance to expose the Wifibot functionality to the GRF. This was done in the following five steps.

1. An instance of the GRFInterface module was created, called GRFInt.
2. GRFInt’s ‘RobotObj’ attribute was set to the current Wifibot instance.
3. GRFInt’s ‘ResourcesList’ attribute was set to an object that described all of the Wifibot’s resources as specified in Section 3.5.2 under the fourth point.
4. GRFInt’s ‘InstanceInfo’ attribute was set to an object describing the Wifibot Instance as specified in Section 3.5.2 under the fifth point.

\(^1\)ffmpeg is an application that can stream, record and convert audio and video.
4.2. CASE STUDY: THE WIFIBOT LAB LIDAR

5. GRFInt’s start method was called.

All but two of the above steps only required one line of code. The exceptions were steps 3 and 4, which required twelve and six lines of code respectively. Step 3 required the greatest consideration; it required the specification of the resource path, the type of HTTP method, and the content-type of the response for every robot resource.

4.2.2.7 Developing the Graphical User Interface

The GUI was simple to create using HTML, CSS and JavaScript. The jQuery\textsuperscript{2} library was particularly useful for dynamically updating elements on the page, and requesting the appropriate RESTful resource in the background when the user of the interface performed a particular action. In order to support touch based devices such as smartphones and tablets, the touchstart, touchmove, touchend, and touchcancel JavaScript document events needed to be used.

There were no problems while using the GUI on a desktop or laptop computer. However, it only worked properly on an Android device if Firefox Mobile 7 was used. This is because it is the only browser we found to support both M-JPEG streams and touch events.

4.2.2.8 Analysis

The lack of a comprehensive and mature Wifibot API made implementing the Robot Interface more troublesome than expected. Once the methods that utilised the existing Wifibot API had been written, writing the code to meet the requirements of the GRFInterface was straightforward. Using the GRF with the completed Robot Instance provided a clean and uniform interface to all of the Wifibot’s functionality whilst largely hiding the problems with the existing API. One of the reasons for using HTTP to expose robot functionality was because it would make robot functionality more accessible over the Internet. Controlling the Wifibot with the GRF over a potentially high latency network like the Internet is not likely to be a good experience due to the problems highlighted in Section 4.2.2.1 about sending ping commands to the Wifibot. However, this problem is unique to the Wifibot and it is unlikely that an identical problem would occur with other robots.

\textsuperscript{2}http://jquery.com
4.3 Usability Tests

A number of sample programs were developed to illustrate how simple it is for a script or application to use the RESTful API exposed by the GRF.

4.3.1 Performing a 180° Turn

```python
import urllib.request
import time

for i in range(8):
    req = urllib.request.Request(url='http://wifibot:15030/Move/-440/440')
    req.get_method = lambda: 'PUT'
    jsonResp = urllib.request.urlopen(req)
    time.sleep(0.2)
```

Listing 4.1: Python code to instruct the Wifibot to do a 180° turn

As shown in Listing 4.1, the robot can be made to do a 180° turn in only seven lines of Python code. Issuing the Wifibot a move command is done easily by using the HTTP PUT method on the Move resource.

4.3.2 Retrieving the Status from the Wifibot

```python
import urllib.request
import json

req = urllib.request.Request(url='http://wifibot:15030/Sensors/Status')
jsonResp = urllib.request.urlopen(req).read().decode('utf-8')
statusObj = json.loads(jsonResp)
print('Battery: {0}
Left Wheel Speed: {1}
Right Wheel Speed: {2}'.format(statusObj['data']['battery'],
statusObj['data']['left']['speed'],
statusObj['data']['right']['speed']))
```

Listing 4.2: Python code to retrieve the Wifibot’s status

Retrieving the status from the Wifibot is simple using the GRF as it is encoded as a JSON object. In contrast, accessing the status via the Wifibot’s API requires decoding an array of 21 bytes into the separate status components [41].
4.3.3 Streaming LIDAR Data

```javascript
var http = require('http');
var req = http.request(
  {
    host: 'wifibot',
    port: '9090',
    method: 'GET',
    path: '/Sensors/LIDAR/Stream'
  });

req.on('response', function(resp) {
  var buff = null;
  resp.on('data', function(data) {
    console.log(data.toString());
  });
});

req.end();
```

Listing 4.3: Node.js code to stream LIDAR data

Streaming Wifibot data using the GRF is very simple, and can be made to be very efficient if event-based programming is used. In Listing 4.3, the function on line 10 will be called continuously with new LIDAR data until the request is terminated.

4.3.4 Comparison

In order to compare using the GRF against direct communication with the Wifibot API, an identical set of movement operations was executed using both methods. The test executed was to accelerate the robot from 0 to 440 in increments of twenty, increasing the speed every 20 ms.

4.3.4.1 Using the GRF

```python
import urllib.request
import time

def makeReq(path):
  req = urllib.request.Request(url='http://wifibot:15030/' + path)
```
4.3. USABILITY TESTS

Executing the movement instructions using the GRF was done easily by repeatedly issuing GET requests to the Wifibot’s Move resource with the desired speed as parameters.

4.3.4.2 Direct Communication

```python
import socket
import time
start = time.time()
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(('wifibot', 15020))
cmd = [0]*9

cmd[0] = 255
cmd[1] = 7

def crc16(cmd):
crc = 0xffff
polynome = 0xa001
cptbit = 0
parity = 0

for cptoctet in range(1, len(cmd)-2):
crc ^= cmd[cptoctet]
for cptbit in range(0, 8):
    parity = crc
    crc >>= 1
    if parity % 2 == 1:
        crc ^= polynome
    cmd[7] = crc & 0x00ff
    cmd[8] = crc >> 8

def changeSpeed(leftspeed, rightspeed, cmd, sock):
lsspeed = abs(leftspeed)
```
4.3. USABILITY TESTS

rspeed = abs(rightspeed)
cmd[2] = lspeed & 0x00FF
cmd[3] = lspeed >> 8
cmd[4] = rspeed & 0x00FF
cmd[5] = rspeed >> 8
cmd[6] = (((0x00 if leftspeed < 0 else 0x04) + (0x00 if rightspeed < 0 else 0x01)) << 4) + 0x08
crc16(cmd)
buf = bytearray(cmd)
sock.send(buf)
for i in range(0, 440, 20):
    changeSpeed(i, i, cmd, sock)
time.sleep(0.2)
diff = time.time() - start
print(diff)

Listing 4.5: Wifibot API acceleration test

Running this test using the existing Wifibot API required significantly more work. The command to move the robot needed to be encoded into a nine-byte array before sending. The desired speed for the left and right wheels of the robot had to be converted into two bytes each, and the direction of the acceleration for each wheel had to be specified by assigning specific bits in the seventh byte of the byte array. Lastly, the CRC of the first seven bytes of the message needed to be calculated and stored in bytes eight and nine.

4.3.4.3 Analysis

Running the acceleration test using the GRF required 13 lines of Python code, as opposed to the 43 lines of code (an increase of 231%) required when using the existing Wifibot API. The changeSpeed and crc16 functions in the Wifibot API test are however reusable, meaning adding further movement logic would not require any more code than adding the same extra logic to the GRF test. When considering simplicity and understandability, it can be seen that the code that uses the GRF is very simple and easy to understand, while the code that uses the existing Wifibot API requires an understanding of how to encode a movement command into a byte array according to the Wifibot protocol specification. Thus, using the GRF has greatly simplified movement of the Wifibot.
4.4 Performance Tests

4.4.1 Methodology

The speed at which the GRF processes requests was benchmarked by recording the time it took to execute various Python scripts. These Python scripts used the GRF’s RESTful API to execute movement instructions. A second set of Python scripts was then created to request an identical set of movement instructions to the first set of scripts, but sent them directly to the Wifibot API instead of using the GRF. The execution time of the scripts that used the GRF and the scripts that communicated directly with the Wifibot API were then compared.

4.4.2 Test 1: Repeatedly Executing a Command

The purpose of this test was to measure the throughput of the GRF compared to sending commands directly to the existing Wifibot API.

4.4.2.1 Using the GRF

```python
import sys
import urllib.request
import time

numReqsStart = int(sys.argv[1])
numReqsEnd = int(sys.argv[2])
step = int(sys.argv[3])
results = []
repeatNum = 5

for i in range(numReqsStart, numReqsEnd, step):
    totTime = 0
    for j in range(repeatNum):
        print('Starting repeat {0} for {1} requests'.format(j, i))
        start = time.time()
        for k in range(i):
            req = urllib.request.Request(url='http://wifibot:15030/Move/0/0')
            req.get_method = lambda: 'PUT'
            urllib.request.urlopen(req).read()
        end = time.time()
        totTime += end - start
    averageTime = totTime / repeatNum
    results.append(averageTime)
```


4.4. PERFORMANCE TESTS

Listing 4.6: Python code to perform Test 1 using the GRF

4.4.2.2 Using the Robot Server

```python
import sys
import socket
import time

numReqsStart = int(sys.argv[1])
numReqsEnd = int(sys.argv[2])
step = int(sys.argv[3])
results = []
repeatNum = 5

for i in range(numReqsStart, numReqsEnd, step):
    totTime = 0
    for j in range(repeatNum):
        print(f'Starting repeat {j} for {i} requests')
        start = time.time()
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.connect(('wifibot', 15020))
        cmd = [0]*9
        cmd[0] = 255
        cmd[1] = 7
        for k in range(i):
            cmd[2] = 0
            cmd[3] = 0
            cmd[4] = 0
            cmd[5] = 0
            cmd[6] = 10
            cmd[7] = 129
            cmd[8] = 171
            buff = bytearray(cmd)
            sock.send(buff)
            data = sock.recv(21)
        totTime += time.time() - start
        time.sleep(1)
    results.append((i, totTime/repeatNum))
print(results)
```

4.4. PERFORMANCE TESTS
4.4. PERFORMANCE TESTS

Listing 4.7: Python code to perform Test 1 using Wifibot API

```python
        totTime += time.time() - start
        time.sleep(1)
        results.append((i, totTime/repeatNum))

        print(results)
```

4.4.2.3 Results

In addition to testing the GRF running on the Wifibot itself, we also tested both the GRF deployed on a more powerful server, and the GRF components split between a more powerful server and the Wifibot (with the Robot Instance executing on the Wifibot).

The average response time per request for the tests can be seen in Figure 4.1. As would be expected, direct communication with the Wifibot API was the fastest and had the most consistent average response times. Running the GRF entirely on the Wifibot performed poorly and erratically in comparison. The tests revealed that on average, using the existing Wifibot API is 484.43% faster than using the GRF when it is running entirely on the Wifibot.
Figure 4.1: Throughput graph for Test 1
However, when splitting the GRF so that the Robot Instance is run on the Wifibot and the GRF is run on a more powerful server, the percentage difference drops to 301.65%. When both the GRF and the Robot Instance are run entirely on a more powerful server, the difference further drops to 153.72%. Considering that running the GRF entirely on a different host to the Wifibot means that an extra network round-trip is needed when the GRF sends a command to the Wifibot, the actual overhead added by the GRF is less than indicated by the results for running the GRF on a server.

Given the results, it is clear that the GRF performs very well when given a large amount of processing power. The GRF may have performed significantly better when run entirely on the Wifibot if it was built in a fast compiled language such as C or C++.

### 4.4.3 Test 2: Moving in a Square

The purpose of this test was to measure the difference in speed between the GRF and direct communication with the Wifibot API when simulating a more realistic application of the Wifibot. In this test, the Wifibot is instructed to move in a square, observing its environment with the camera before turning.

#### 4.4.3.1 Using the GRF

```python
import urllib.request
import time

def readCamera():
    req = urllib.request.Request(url='http://wifibot:15030/Sensors/Camera')
    cameraData = urllib.request.urlopen(req).read(12000)
    # some image processing
    time.sleep(0.20)

def makeReq(path):
    req = urllib.request.Request(url='http://wifibot:15030/'+path)
    req.get_method = lambda: 'PUT'
    urllib.request.urlopen(req)

numRuns = 5
totalTime = 0
for iteration in range(numRuns):
    start = time.time()
```
for i in range(4):
    #Move forward
    for j in range(3):
        makeReq('Move/250/250')
        time.sleep(0.20)
makeReq('Move/0/0')
# Would do some environment processing here
#time.sleep(0.4)
readCamera()
#Turn 90 degrees left
for j in range(4):
    makeReq('Move/335/-335')
    time.sleep(0.20)
makeReq('Move/0/0')
    time.sleep(0.4)
totalTime += (time.time()-start)
time.sleep(1)
print(avgTime)

Listing 4.8: Python code to perform Test 2 using the GRF

4.4.3.2 Using the Robot Server

import socket
import urllib.request
import time
cmd = [0]*9
cmd[0] = 255
cmd[1] = 7
def crc16(cmd):
crc = 0xffff
polynome = 0xa001
cptbit = 0
parity = 0
for cptoctet in range(1, len(cmd)-2):
crc = cmd[cptoctet]
for cptbit in range(0, 8):
    parity = crc
    crc >>= 1
if parity % 2 == 1:
4.4. PERFORMANCE TESTS

```python
crc = polynome

20 cmd[7] = crc & 0x00FF
21 cmd[8] = crc >> 8

def changeSpeed(leftspeed, rightspeed, cmd, sock):
24   lspeed = abs(leftspeed)
25   rspeed = abs(rightspeed)
26   cmd[2] = lspeed & 0x00FF
27   cmd[3] = lspeed >> 8
28   cmd[4] = rspeed & 0x00FF
29   cmd[5] = rspeed >> 8
30   cmd[6] = ((0x00 if leftspeed < 0 else 0x04) + (0x00 if rightspeed < 0
31          else 0x01)) << 4) + 0x08
32   crc16(cmd)
33   buff = bytearray(cmd)
34   sock.send(buff)

36 def readCamera():
37   req = urllib.request.Request(url='http://wifibot:8080/?action=stream')
38   cameraData = urllib.request.urlopen(req).read(12000)
39   #some image processing
40   time.sleep(0.20)

42 numRuns = 5
43 totalTime = 0
44 for iteration in range(numRuns):
45   start = time.time()
46   sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
47   sock.connect(("wifibot", 15020))
48   for i in range(4):
49     for j in range(3):
50       changeSpeed(250, 250, cmd, sock)
51       time.sleep(0.20)
52       changeSpeed(0, 0, cmd, sock)
53       readCamera()
54     for j in range(4):
55       changeSpeed(335, -335, cmd, sock)
56       time.sleep(0.20)
57       changeSpeed(0, 0, cmd, sock)
58       time.sleep(0.4)
59   sock.close()
60   totalTime += (time.time() - start)
61   time.sleep(1)
62 avgTime = totalTime / numRuns
```

4.5. ASSESSMENT OF GENERALITY

4.5.1 Results

The difference in time between using the GRF and communicating directly with the Wifibot API is substantially smaller than that found in Test 1. Running the test through the GRF on the Wifibot was only 3.95% slower than the test using the inbuilt Wifibot API and camera stream directly. Splitting the GRF between a server and the Wifibot, and running the GRF entirely on a server were, respectively, 2.95% and 1.95% slower.

The reason for the large difference in results between Test 1 and Test 2 is because of how the Wifibot is used. It is not useful to send the Wifibot commands in quick succession since the Wifibot will not have time to complete each command before it receives the next one. A delay between multiple movement commands is therefore necessary, and this is the reason for the large difference in results between the two tests.

These results are very specific to the Wifibot and how it is used, but they are a good indication that the performance penalty for using the GRF is negligible for practical robot applications.

4.5 Assessment of Generality

To deploy the GRF on a robot other than the Wifibot, the only components that would need to be changed are the Wifibot module and the web interface files (robotUI.html and supporting files). A robot module that provides methods to access the new robot’s functionality would need to be developed and integrated into the GRF as described in Section 3.5.2. An optional web interface could then be created using robotUI.html as an entry point. If Node.js is not used to create the robot module, the GRFInterface module would also need to be re-written in the appropriate language.

Given that deploying the framework onto a new robot only requires one file to be replaced (if using Node.js) to expose the new robot’s functionality, and two files to be replaced to both expose its functionality and a web interface, we conclude that the framework is suitably generalised.
4.6 Summary

The case study, code comparisons, and performance tests in this chapter were done to determine the ease of implementation, usability, and performance of the GRF respectively. Implementing the GRF on the Wifibot was found to be simple once a module that could access Wifibot functionality had been written. Comparisons between using the GRF versus the existing Wifibot API to perform a robot activity revealed that the GRF required significantly less Python code, and the code it required was easier to read and understand. A throughput comparison was done to evaluate the speed of the GRF, and it was found that the existing Wifibot API was between 153.72% and 484.43% faster than the GRF depending on where the components of the GRF were deployed. However, the benchmarking of a realistic application of the Wifibot revealed the existing Wifibot API to be only between 1.95% and 3.95% faster than the GRF. Lastly, an evaluation of the GRF’s generality found that only a small portion of the framework needed to be replaced in order to deploy the GRF on a different robot. Thus, the GRF was found to be suitably generalised, easy to implement and use, with good performance for practical robot applications.
Chapter 5

Conclusions

5.1 Project Summary

The goal of this project is the creation of a generalised computer-to-robot communication framework that exposes robot functionality over HTTP. As part of this goal, the prototype framework (the GRF) should allow robot functionality to be easily accessible by smartphone and tablet devices. A literature survey was performed that evaluated different web service infrastructures, web servers, message serialisation formats, and socket types for use in the GRF. It further considered the mobile platform, and evaluated related work. Thereafter, the GRF was designed and implemented with what was chosen as the most applicable of the technologies and frameworks evaluated. Finally, the ease of implementation, usability, performance, and generality of the GRF was evaluated to determine whether the GRF met its objectives.

5.2 Revisiting the Objectives

The primary objective was to investigate the feasibility of creating a generalised computer-to-robot communication framework that exposes robot functionality over HTTP, and has a web interface that is usable by both smartphone and tablet devices. The results given in Chapter 4 show that the GRF has met this objective. The implementation of the GRF on the Wifibot showed that it is feasible to expose robot functionality through the framework using HTTP, and access it simply with a mobile device. The secondary
objective of ensuring that the latency introduced by the GRF does not adversely affect robot functionality has also been achieved since the GRF adds negligible latency to robot usage in a practical scenario.

5.3 Possible Extensions

This section lists the possible extensions that can be made to the GRF prototype.

- The GRF could be extended to support multiple Robot Instances, making the Web Interface the central point of control for multiple robots.
- An authentication system could be added to only allow authorised access to all, or part of, robot functionality.
Bibliography


[24] Crockford, D. The application/json Media Type for JavaScript Object Notation.


Appendix A

Code Listings

A.1 Protocol Buffer Protocol

```protobuf
package ri;

message RoboRequest {
  required int32 id = 1;
  optional string operation = 2;
  optional string parameters = 3;
}

message RoboResponse {
  required int32 id = 1;
  optional string response = 2;
  optional bytes binary = 3;
}
```

Listing A.1: Protocol Buffer message structures

A.2 Simple Robot Instance

```javascript
var GRFInterface = require('GRFInterface.js');

var framework = GRFInterface.create();

var TheRobot = function () {
```

65
TheRobot.prototype.Resource1 = function(robot, id, parameters, cb, closed) {
  console.log('Got a persistent request for Resource1');
  closed(function() {
    console.log('The client closed the persistent connection to Resource1');
  });
  cb({id: id, responseObject: {result: 'success', data: 'You requested Resource1! Your connection will remain open.'}});
}

TheRobot.prototype.Resource2 = function(robot, id, parameters, cb) {
  console.log('Got request for Resource2 with parameters:');
  console.log('param1: ' + parameters.param1);
  console.log('param2: ' + parameters.param2);
  cb({id: id, responseObject: {result: 'success', data: 'You requested Resource2 with parameters: ' + parameters.param1 + ' and ' + parameters.param2}});
}

TheRobot.prototype.Resource3 = function(robot, id, parameters, cb) {
  console.log('Got request for Resource3');
  cb({id: id, responseObject: {result: 'failed', error: 'Your request for Resource3 has failed (not really)'}});
}

TheRobot.prototype.Resource4POST = function(robot, id, parameters, cb) {
  console.log('Got a POST request for Resource4');
  cb({id: id, responseObject: {result: 'success', data: 'This was a post request for Resource4'}});
}

TheRobot.prototype.Resource4PUT = function(robot, id, parameters, cb) {
  console.log('Got PUT request for Resource4');
  cb({id: id, responseObject: {result: 'success'}});
}

TheRobot.prototype.Resource4DELETE = function(robot, id, parameters, cb) {
  console.log('Got a DELETE request for Resource4');
  cb({id: id, responseObject: {result: 'success', data: 'This was a DELETE request for Resource4'}});
}
43 var robot = new TheRobot();
44 framework.RobotObj = robot;
45 framework.ResourcesList = {
46   '/Section1/Resource1' : { GET: { persistent: true, contentType: 'multipart/x-mixed-replace; boundary=boundarydonotcross', fn: 'Resource1', help: 'Example of a streaming resource' }},
47   '/Section2/Resource2/:param1/:param2' : { GET: { persistent: false, fn: 'Resource2', help: 'Example of a resource that takes parameters' }},
50     PUT: { persistent: true, fn: 'Resource4PUT', help: 'Example of a PUT resource with the same URL as a POST resource' },
51     DELETE: { persistent: false, fn: 'Resource4DELETE', help: 'Example of a DELETE resource with the same URL as a POST and PUT resource' }
52   }
53 },
54 framework.InstanceInfo = {
55   'robotName' : 'Dummy Robot',
56   'version' : '1.0',
57   'author' : 'Dale Tristram',
58   'contact' : 'D.Tristram@ru.ac.za',
59 },
60 framework.start();

Listing A.2: A simple Robot Instance implementation