University of Glasgow | School of Computing Science

Evaluating the Scalability of ROS in Multi-Robot Systems

Isaac Jordan

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — March 18, 2017

Abstract

Autonomous robots are being created in an increasingly modular way. This is leading to the creation of many types of robot middleware software frameworks, however much of the existing knowledge about the performance of these middlewares is informal and annecdotal. A single system comprising multiple individual autonomous reobots forms a multi-robot system. These systems have added complexity over single robot systems as inter-robot communication can be orders of magnitude slower than intra-robot communication.

This paper presents an overview of existing robot middlewares, before focussing on the ROS (Robot Operating System) middleware. An evaluation of the limitations of vertical communication scalability and high frequency message tranmission performance is conducted. Important factors affecting high frequency message transmission were found to be the processing power of the host machine - at high message frequencies a 50% reduction in CPU clock speed doubled the message latencies - and the network medium used: switching to Wi-Fi from Ethernet caused an order-of-magnitude decrease in the maximum supported message frequency. A Communication Scaling Limit Volume (CSLV) formula is proposed as a mechanism to predict whether a particular combination of node count, message frequency, and message size is feasible for a particular vertically scaled system. A series of experiments were conducted which appeared to establish the CSLV hypothesis as a valid means to predict performance for ROS nodes, by showing the predictions were accurate at previously untested message frequencies. Furthermore, a method for a horizontal scaling experiment is proposed.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: ____________________ Signature: ______________________

Contents

Chapter 1

Introduction

1.1 Context

With the increase in availability of commodity computing hardware it is becoming more feasible to experiment with robotics at home. This increase in hobbyist roboticists is one plausible reason for the increase in open source robotics development. These community driven development projects are generally very flexible and extensible, given the large number of types of users wishing to use them in different ways: students, hobbyists, researchers, and enterprises. One such project is ROS (Robot Operating System). ROS describes itself as "a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot[s]"[17]. Much of the knowledge on ROS and it's performance is anecdotal acquired and shared on sites such as StackOverflow, this project aims to present a more formal analysis of the performance and scalability of ROS both in an isolated system, and in a real robot car kit platform.

A system created using ROS comprises a number of nodes, where each node is conceptually a self-contained module with the goal of completing a single task. The behaviour of a robot which is built using ROS comes as an emergent property of the collection of nodes forming the robot. ROS nodes need not be colocated on a single machine (called a host). This allows for nodes to be distributed over a number of hosts; in the case that each host is an indepedent robot this creates a multi-robot system. The robot car kit platform used in this project comprises a number of Sunfounder Smart Video Car Kits[22], with each car being driven by a Raspberry Pi 3 Model B. The high availability of cheap self-contained computing devices such as Raspberry Pis means that these cat kits are representative of what many roboticists might use to drive their robots.

Scalability is a term widely used to refer to many different ideas, however this project uses the term to refer to three different types of scalability. The first, called frequency scaling, involves increasing the message frequency from a particular ROS node - until the system can no longer process the messages in a timely manner. The second, vertical scaling, refers to adding multiple ROS nodes to the same host. The third, horizontal scaling, involves increasing the number of hosts in the system while possibly increasing the number of nodes (depending on whether per-host node count or overall node count should be kept constant).

1.2 Aims and Objectives

This project aims to provide an evaluation of ROS in a multi-robot situation, by first evaluating the simplest multi-robot case (2 robots/hosts communicating with each other, 1 ROS node each) and looking at the limits of communication in this scenario - and attempt to identify possible causes for the limits. The next aim is to vertically scale these hosts to run many more nodes each - in order to explore the upper bound of how much communication a single ROS host can sustain (whether in terms of messages-per-second, bytes-per-second, or number of sending/receiving processes).

The intention of the research is to build a foundation of knowledge upon which further research in to communication systems of ROS can be conducted, so that future researchers need not depend on anecdotal performance estimations.

1.3 Achievements

This project makes several key contributions:

- **A comprehensive review of existing robotic middlewares, including an in-depth analysis of ROS** A survey of a wide range of software packages that compete with ROS as robot middleware was conducted, providing an understanding of where ROS is placed in the market. A number of common themes are identified across the variety of middlewares, broken down in terms of communication, computation, configuration, and coordination. This is covered in Chapter 2.
- **A systematic evaluation of high frequency communication performance with ROS in a multi-host network** A sequence of experiments were conducted investigating ROS' ability to scale to high frequency messages with inter-host communication. Frequencies ranging from 1Hz to 1MHz were considered with emphasis in the 200Hz to 2KHz range, with message sizes ranging from 11 bytes (a simple string), to 4KB sensor readings (recorded data provided by the MIT Stata Center dataset[8]) as well as 300KB video image messages (from the same dataset). Key findings are that high frequency ROS communication was constrained by processing power of the test system (a 50% reduction in CPU clock speed doubles message latency), however this reduces in importance as message sizes increase. Usage of a Wi-Fi communication channel was found to introduce significant latency compared to Ethernet (5ms latency at 2KHz with Ethernet, and 1800ms when using Wi-Fi), and reduce the maximum frequency that ROS could send messages at. These findings were confirmed to hold true on the Sunfounder robot kit car, indicating the results are applicable to real robot platforms. These experiments are covered in Chapter 3.
- **Proposal of a Communication Scaling Limit Volume (CSLV) which predicts frequency and vertical scalability limits** A trend in communication performance led to the proposal of a CSLV formula, allowing for the performance of a particular ROS host to be calculated given the total number of nodes on the host, their message frequencies, and the individual message size. To verify the predictive capabilities of the CSLV, two series of vertical scaling experiments were conducted on a pair of hosts. The first set of experiments used a range of high message frequencies (100Hz to 300Hz) to calculate an approximate CSLV value for the Raspberry Pi 3 Model 3 system. The next series of experiments were conducting using a lower range of frequencies (1Hz to 20Hz), and the performance results were correctly predicted by the CSLV formula - establishing the CSLV hypothesis. These experiments were conducted using 4KB per message sensor data acquired from the MIT Stata Center dataset[8], and were also repeated on a pair of Sunfounder robot car kit hosts. An experimental set-up is also proposed to conduct horizontal scaling experiments. These results are covered in Chapter 4.

Chapter 2

Background

2.1 Robotics Overview

Robotics is the study of autonomous computer systems, usually involving physical interaction with it's environment. Robotics lies at the intersection between computing science, eletrical and mechanical engineering. Robotics has been common in popular culture since the 1940s[47], but due to the speed of technological growth, has quickly become widely ingrained in many industries. The first industrial use of robotics was in manufacturing, with General Motors putting the first robot into service in 1961[44]. Nowadays, with great increases in computing power, sensor accuracy, research investment, and software algorithms, example applications of robotics include manufacturing, agriculture, construction, mining, disaster recovery, medicine, health care, and surveillance[45].

Today's robotic systems generally consist of many small autonomous systems working together to form a coherent whole[31]. For example, a particular sensor (say a camera) may be constantly recording data and storing it in some buffer (erasing the oldest when full). This subsystem does not depend on any others to complete its task (recording the environment), but other subsystems may rely on its output (such as a computer vision package which needs video frame inputs). This style of robot design is becoming increasingly prevelant, with many robotic middlewares adopting this distributed approach.

Robotics is related to a field called cyber-physical systems (CPS) that concerns integrating computer systems with the physical world[56]. These can include Internet Of Things (IoT) networks[30] which can (for example) be used to control things around the household, such as light switches, central heating systems, and hoovering. This is distinct from robotics as CPS generally embody a 'think globally, act locally' approach[42] (such as using data from many sources to improve the CPS' performance in each), compared to robotics which utilises a more 'think locally, act locally' strategy - meaning that each instance of a robot generally makes it's own decisions, and acts upon them solely.

This chapter aims to provide the background knowledge required to understand the significance of the results presented in this paper. Section 2.2 discusses the concept of multi-robot systems, as well as their application in society. Section 2.3 outlines what scaling refers to in the context of a robotic system. Next, Section 2.4 introduces robotic middleware, and why it is useful when constructing a robotic system. This section also presents a tabular overview of a range of common robot middlewares, and discusses the main concepts seen in many middlewares. Section 2.5 then outlines the architecture of ROS, which is the middleware of particular study in this paper. Finally, Section 2.6 discusses the configuration of the robot car kit platform used in later experiments.

2.2 Multi-Robot Systems

Multi-robot systems are specific instances of mult-agent systems. They represent a joint problem space between robotics, artificial intelligence, and distributed systems. Multi-robot systems as a formal concept is a recent development with a IEEE technical committee only being formed in 2014[23].

Multi-robot systems can consist of many intelligent agents (each of which may be comprised of many small autonomous systems) working to solve a task that any one system may not be able to solve alone. These multi-robot are distinct from a multi-agent system in which individual nodes are generally stationary, as each agent in a multi-robot system is mobile[67]. Mobile robotics has been made more possible recently by advances in battery[39] and wireless communication[37] technologies. One such application of a multi-robot (and mobile) system is warehouse automation. Hamberg (2012)[46] provides a comprehensive overview of such a system, but a brief description is provided below.

In a warehouse, millions of items can be spread throughout miles of shelving and the requirement is that a random subset of items (those that have been bought) must arrive at a specific point (the delivery pick-up point) at a specific time (when the van is there). This task previously could be solved by having human pickers wander the isles searching for items - however given recent increases in the size of online shopping this is no longer feasible. Now, online retailers (such as Amazon) are using hundreds of individual robots to intelligently move the shelving around and bring the correct items to stationary human pickers[66]. This system requires the coordination of the individual robots, but they must all act independently in order to efficiently keep pace with the items ordered. However, this system still requires human pickers to lift items off the shelves, and place them in boxes. Amazon is using competitions such as the Amazon Picking Challenge to encourage researchers to develop reliable methods to automate this picking task[38], although we have still not reached the level of full automation in a production environment.

2.3 Scalability in Robotics

When creating multi-robot systems, scalability becomes an important concern[54]. It is a crucial decision for a system architect to choose between using a small number of expensive highly-powered robot systems and using a larger number of simpler, cheaper robot systems to improve overall system performance. There are several factors to consider include the processing power of each robot, the communication framework required to organise them, the cost of each robot, and the suitability of the problem to a multi-robot system.

Scalability can refer to many different concepts. Some define it as the ability to handle more code, others as more processes, and yet others as more hardware[32]. This paper will consider the latter two definitions, and will refer to increasing the number of processes running on the same amount of hardware as 'vertical' scaling, and increasing the amount of hardware (e.g. number of host machines) as 'horizontal' scaling. There are a number of advantages and disadvantages to both.

Vertical scaling is useful if a single instance of the process (or node) does not saturate any of the resources of the host machine - for example, a single threaded computation-heavy application running a multi-core processor architecture. There are still other CPU cores available to run more instances of the application, with the benefit of gaining higher throughput. However, if a single instance does saturate a resource (e.g. high RAM usage, full disk usage, or network saturation) then vertical scaling will introduce resource contention problems, such as two disk IO (input-output) heavy processes competing for disk access and thus slowing both down. Extreme vertical scaling will also cause problems with context switching in the processor. If thousands of processes are competing to get time on the processor, then the process scheduler will only schedule each process for a very short timeslice before switching in a new process. This can introduce an overhead due to the time taken to switch out a running process, and switch in a new process.

Horizontal scaling alleviates some of the problems with vertical scaling. By scaling nodes over multiple hosts, each host has access to a completely independent set of resources (e.g. it's own CPU, RAM, disk, and GPU) - this reduces resource contention issues. However, horizontal scaling introduces a number of complex issues, such as the requirement for managing an increasing amount of hardware - all disks fail eventually and must be replaced, and increasing the total number of disks in the system results in an increased number of failures, requiring replacements to be fitted more often. For this reason, horizontally scaled (or distributed) systems must be more fault-tolerant than single host systems. Increasing the number of hosts also has implications for the monetary cost of the system, as duplicating hardware also duplicates costs (such as initial purchase, maintenance, and running costs). Swarm robotics is the extreme approach of horizontal scaling, involving creating many very simple robots which individually could not solve tasks or survive environments - however when they work together as a form of society they can work efficiently[63].

2.4 Robotic Middleware

As mentioned previously in Section 2.1, software for robots is generally written as a collection of autonomous subsystems, often called modules. Robotic middleware is a software infrastructure that is intended to provide convenient abstraction and communication paradigms for facilitating this multi-subsystem approach. In general, a robotics middleware would provide interfaces for defining each subsystem, and defining how each subsystem communicates with others. A specific example is the Player Project. Player provides an abstraction layer for robotic coding, which lets developers focus more on their specific application logic, rather than boilerplate communication code[65]. The different approaches of specific middlewares is discussed later in Section 2.4.1.

A typical robot may consist of many individual sensors, for example a camera that can record 720p (a resolution of 1280x720) RGB video at 30fps (frames-per-second), a LIDAR range sensor that can measure distances of up to 30m at a frequency of 1-500Hz, a tri-axis gyroscope and accelerometer capable of measuring up to 16g of force with a measurement frequency of 40Hz. These devices use a range of sophisticated technologies to measure and analyse the physical world, each recording a different data format at a different data rate. Figure 2.1 shows the sensors available on Willow Garage's PR2 robot. These sensor modules are all independently designed, resulting in a wide array of different software and hardware interfaces - meaning that each robot software developer that wishes to use these sensors must create the software to communicate with these sensors, consume their data, and then write the robotic software that makes use of it. This results in a wide variety of implementations for manipulating the same data on each sensor, increasing the likelihood of implementation mistakes, misunderstandings, and wasting researchers' time.

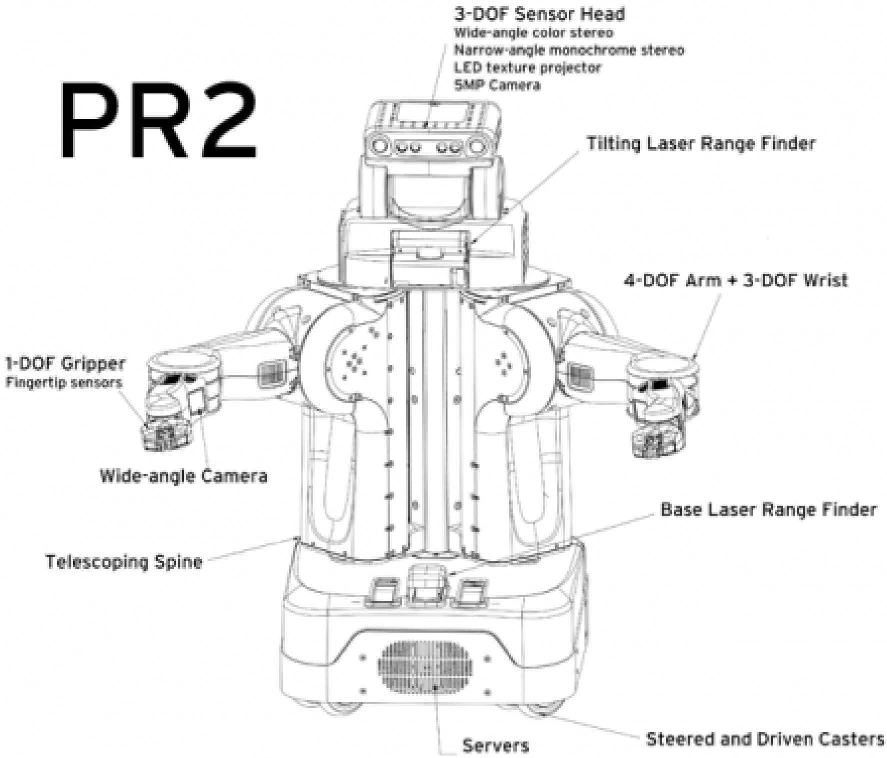

Figure 2.1: Diagram outlining the major sensors present on Willow Garage's ROS-powered PR2 robot[62]

Robotic middleware is the solution to this problem. A middleware provides a common interface design so that no matter what hardware is producing the data, the results are distributed in a consistent manner. This means that hardware manufacturers (or users) need only to implement one software system for each sensor that can then be distributed and reused amongst all users of that sensor, as long as those users are using that middleware. This means that creators of a system utilising the same middleware have easy access to the data created by a particular sensor as they know the software they create will be able to easily consume the sensor's data stream.

Another benefit of middleware is that this communication interface can be reused within the robotic system for communicating between distinct modules within the system. For example, a robot software developer may create a computer vision module that processes a video stream and returns a data stream containing a description of the objects in the video stream at each frame. When utilising a middleware, the researcher can make the result of their computer vision module available in a similar fashion to the sensor's video stream - meaning that high level software can utilise the results as if there was an 'object-detecting hardware sensor'. These levels of abstractions make software much more maintainable, and reusable.

2.4.1 An Overview Of Robotic Middleware

There is a wide variety of robotic middlewares in use currently. Many employ a free (libre) approach to software by making the source code available online, whereas others are created for commercial licensing using propriatary source. This overview predominantely covers open source middlewares, as there is a greater amount of information available on the design and implementation of these projects. Several of the covered middlewares have fallen out of use and/or development, and this has been noted where applicable. A discussion of the major aspects of the covered middlewares is presented after the table. *Note that several more middlewares were reviewed than are presented in this table, the other middlewares are presented in Appendix A.*

Name	Objective	Support	Capabilities	Supported Languages
ROS (Robot Operating System) [19]	• The goal of ROS is not to be a framework with the most features. Instead, the primary goal of ROS is to support code reuse in robotics research and development • Keep libraries ROS-agnostic • Easy to test • Scalable; appropriate for large runtime systems, and large development processes	Large, active open source development	• Can be used in conjunction with other robot frameworks • Distributed framework of processes allows for executables to be individually designed, and loosely coupled at runtime • Encourages collaboration by easy package sharing • Not a realtime framework, although can work with realtime code	Python, C++, and Lisp Experimental: Java, and Lua
MOOS (Mission Oriented Operating Suite) [9]	• Designed to facilitate research in the mobile robotic domain • Constitute a resilient, distributed and coordinated suite of software suitable for in-the-field deployment of sub-sea and land research robots • Process communcation should be utterly robust and tolerant of the repeated stop/start of any process	Not widely used (judging by GitHub popularity) Development is possibly stagnating (no GitHub commits since 26th May 2016), core not updated since 11th May 2016	• Platform independent, inter-process communication API • Sensor management • Navigation • Concurrent mission task execution • Vehicle safety management • Mission logging and replay • No P2P communication (client/server only)	C++ (appears to have Python bindings)
Player [15]	• Provides a clean and simple interface to the robot's sensors and actuators over the IP network	Discontinued No commits since May 2016 No releases since 2012	• Supports multiple concurrent connections between devices • Supports flexible network structure (including P2P)	Clients in C++, Tcl, Java, and Python

ROS2 [20]	• Target new use cases, such as multi-robot systems (providing a standard approach), embedded systems, real-time systems, non-ideal networks, and production environments [41] • Recreate ROS using existing new tech (such as Redis, WebSockets, DDS) • Overhaul of API (create consistent API without the 7+ years of backward compatibility that ROS1 has)	Pre-release, but active daily development. Unstable but good future prospects given popularity of ROS1	• Improved communication resilience on poor networks utilising DDS [29] [59] • Communication overhead of DDS shown to be non-trivial for local connection. For remote, overhead is trivial but throughput depends on DDS library used [59]	C99, C++11, Python3 Speculative: JavaScript
OpenRDK [11]	• Modular framework for distributed robotic systems • Communication achieved by a central 'repository' into which individual agents publish variables (and can store queues) • Uses URL-like addressing scheme • Focuses on mobile robots [35]	Open source, no news since 2010, created for a single research group	• Created with an eye on the competition (compares it's feature set with ORCA, OROCOS, and Player/Stage. • Has been used in multiple environments (single rescue robotic system, assistive robots) [35] • Has useful tools such as a graphical tool for remote inspection and management of modules, and also modules for logging and replaying [35] • No real-time support [10]	C++

CORBA (Common Object Request Broker Architecture) [2]	• Much more general than a 'robotic middleware', but often compared as it's communications interface is similar to many middleware's. • Software-based communications interface through which objects are located and accessed • OO abstractions utilising request-response in the library (via the Object Request Broker) • Uses Interface Definition Language (IDL) to define object interfaces	Active, open source and proprietary implementations	• Criticised for poor implementations of the standard • Good language and OS independence • 'Freedom from technologies', meaning that (for example) C++ code can talk to Fortran legacy code and Java database code (and each can be changed independently without having to update the other code bases) • Strong typing of messages, reducing human error • Small overhead to adding to system (but dependent on implementation) • Has real-time implementations of related standard (realtime CORBA)	Ada, C++, Java, COBOL, Lisp, Python, Ruby, Smalltalk Non-standard mappings exist for C#, Erlang, Perl, Tcl, Visual Basic

Middleware designs can be broken down in to four groups of concepts: Communication, Computation, Configuration, and Coordination [34]. The majority of robotic middleware differences can be demonstrated as a difference in one of these groups. The following sections provide an overview of the approaches that the middlewares summarised above use to tackle each of these areas.

2.4.2 Communication

All modern robotic middlewares are comprised of multiple modules. In a non-trivial robotic system these distinct modules must exchange a variety of information in a complex web. These information channels usually have some desired bounds or characteristics, such as reliability (guarantees on information delivery), performance (general low latency, or some guarantees on delivery times), and overhead (is the communication significantly more expensive than building one monolithic module). The middlewares presented in Section 2.4.1 have used a variety of approaches to inter-component communication.

CORBA, and those middlewares built on top of CORBA utilise a remote object abstraction. This allows for inter-component communication to appear consistent whether the method caller and callee exists in the same address space, or in a remote address. The only explicit step to enable remote object communication is to share the object reference with the remote process (or component). This is achieved via an Object Request Broker (ORB), which objects can be registered with, and references retrieved from. This form of communication provides very neat code abstractions (as there is no need to modify how the object is used, only how the reference is acquired).

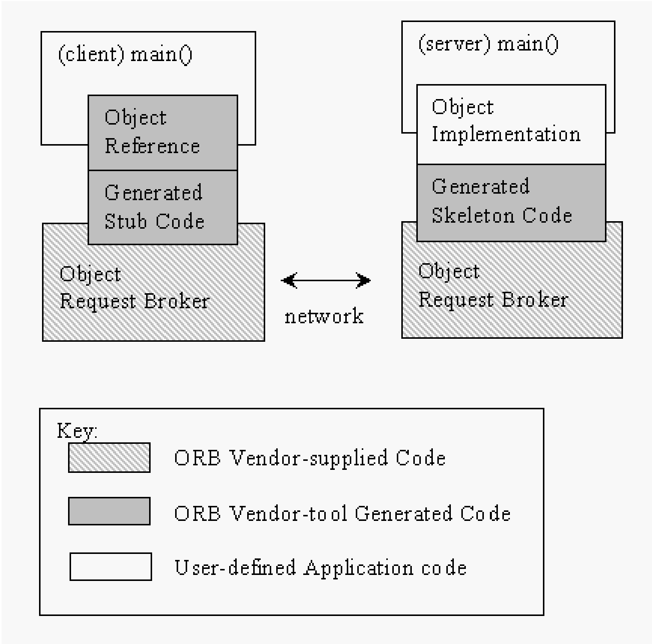

Figure 2.2: Illustration of the communication infrastructure when using CORBA[61].

Other middlewares such as ROS have a more explicit communication paradigm. ROS uses a network of software nodes which can create message queues (known as a topic), which they can publish data of a specific type to [18]. Other nodes can subscribe to topics, generally registering a callback function which is called when some new data has been published to the topic. These mechanisms are described in detail in Section 2.5. This method of communication also allows for code to be identical whether or not the two communicating nodes are in the same address space or are remote, but the communication code itself is explicit. The programmer must define when the topics are created, when data is published, and what happens in a subscriber when the data is published.

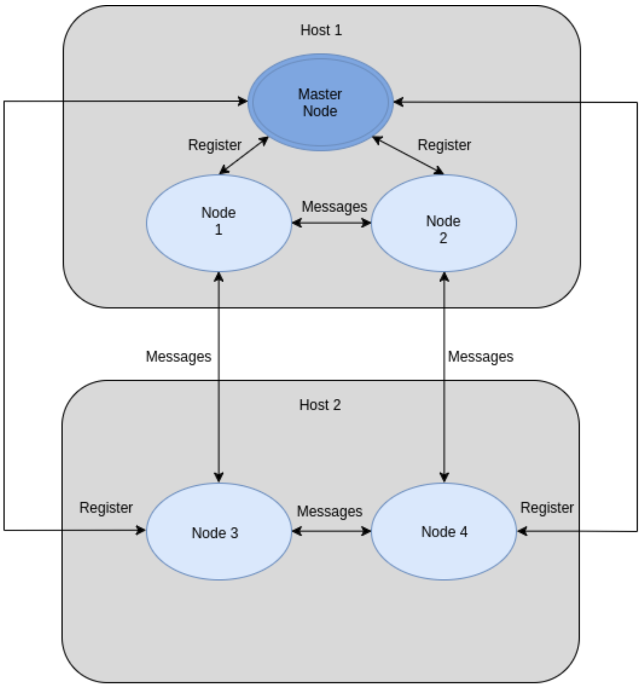

Figure 2.3: ROS node communication

OpenRDK utilises a blackboard model. The analogy refers to the idea of many people stood around a blackboard and communicating only via writing things in different areas of the blackboard, without direct communication between the individuals. This is a form of centralised communication. In OpenRDK the blackboard is called a repository, and each module can communicate by publishing values (called properties) to the repository [10]. Properties can be simple values, or a queue object. The properties are addressed via a global hierarchical URL-like addressing scheme, similar to ROS topic addressing.

Player consists of a centralised server which connects to control clients via a standard TCP socket [16]. The client and server communicate via a set of simple messages. This is a lower-level communication paradigm which is very explicit in code, and requires careful control of shared resources as very little is provided by the library.

2.4.3 Computation

Each module created using a robotic middleware is generally tasked with some computation. That computation could be as simple as processing a value read from a sensor, and publishing it to some communication channel, or as complex as a multi-layered object classifier for images requiring large amounts of computation. Middlewares need to support these use-cases and everything in between, with a coherent, consistent infrastructure model. Middlewares generally provide a way of decoupling distinct computational tasks so that they can be individually designed, implemented, and tested - and then coupled together using the middleware's communication and coordination framework.

OpenRDK models computation inside modules [10]. Each module runs on a single thread, and multiple modules are grouped together to form an agent (a single process).

ROS has a less layered system. Computations are performed by nodes and these nodes directly communicate with each other. Each node generally performs a single focused task. A node usually consists of a single thread, but the programmer is free to spawn new threads as they require. See Section 2.5 for a more complete description of ROS.

All of the presented middlewares present a similar computation structure as the previous two examples, if they prescribe a computation structure at all. For example, CORBA says nothing about how the robotic application is structured, merely that object references must be shared via the Object Request Broker.

2.4.4 Configuration

The configuration of a robotic system using a middleware is often specified using some mechanic of the middleware. There are several distinct stages at which configuration is important, such as compile-time, deployment-time, and run-time. Compile-time configuration involves specifying what compiler settings are required, what libraries should be linked to (and their versions), and what structures and metadata should be created. Configuration at deployment-time involves setting up the system the robotic software is running on - such as installing libraries via a package manager, copying software to specific locations, and setting environment variables. Run-time configuration has a wide variety of potential uses, but can consist of specifying API keys, database connection details, the number of threads that should be created for each process, the particular modules that should start (and when they should start), how components should interconnect and communicate with each other, and how exceptions should be handled.

ROS uses XML files at compile-time to resolve dependencies, export version numbers, and other miscellaneous meta information such as software license and author details. The ability to define the launch of ROS nodes is provided by the 'roslaunch' package. This package parses an XML file which defines which nodes should run, and also sets any required parameters on the ROS Parameter Server. It also provides some reliability functionality, such as respawning processes that have died [21]. ROS includes another way for new agents (nodes) to configure themselves - in the form of a parameter server. This parameter server runs on the ROS master node, providing a central repository of settings. New non-master nodes can request specific parameters from this server at deployment-time (and run-time). This means that configuration files need not be modified in all running instances of the nodes, merely the information stored by the master node need be modified - providing a more centralised configuration than simple local files.

MOOS also utilises a text file for runtime configuration called a 'Mission file' [60]. This mission file provides the necessary runtime parameters for components to set themselves up in a system. However, this configuration is simpler than ROS' systems - primarily it functions as a key-value store for MOOS processes.

OpenRDK utilises an XML configuration file to do similar tasks. However it is closer to ROS in terms of flexibility than MOOS, providing advanced key-value store mechanics, as well as a 'yellow pages' - a lookup table to contact the various agents running in the system.

2.4.5 Coordination

Most middlewares have explicit mechanisms for creating multiple concurrently running software components. These components need to exhibit some overall system behaviour, usually by working together. It is not enough for each component to independently run and share data, there needs to be some coordination to the system to provide controlled, reliable, behaviour. For this reason, some middlewares provide concurrency libraries, either explicitly, or implicitly such as by the computation and communication model or by language features.

In ROS, this concurrency is managed by running all nodes in separate threads, and designing the nodes in a reactive model. The node's computation only occurs when there is data to process, or some task to complete. In this model, subscribers to a particular data stream are immediately aware of when new data has been published and can begin processing immediately - reducing processing latencies.

OpenRDK agents communicate with each other primarily via the repository (a blackboard-type object). This simplifies the communication model of OpenRDK processes, but limits the behaviour of each client to only a pull-type of communication, meaning that each client must individually check the repository to see if there's new data to process. Each producer of data must wait for it's clients to check the blackboard, as it cannot directly tell the client to process new data.

Interconnections between components in many middlewares are achieved over TCP connections with the middleware libraries handling (de)serialization of the messages. Some middlewares such as Player are designed for a client/server architecture, with no/little intercommunication between client components, whereas ROS is designed entirely as a direct P2P network topology (with the master node mainly providing address look-up).

2.5 ROS (Robot Operating System)

The specific focal point of this investigation is ROS (Robot Operating System). ROS is one of the most popular robotic middlewares, with a large active community of users, and ongoing development. However, much of the existing knowledge about ROS' communication performance is anecdotal and scattered across posts on websites such as StackOverflow or individual blogs.

ROS's primary goal is one of sharing and collaboration. Robotic systems often use custom-created software such as driver software and higher-level algorithms like pathing. ROS creators want to make this custom created software reusable across a wide variety of platforms, reducing the amount of repeated independent development, and allowing for faster creation of useful robots. On top of this primary goal, ROS also aims to be very thin, allow libraries to be ROS-agnostic, be language independent, allow easy testing (unit and integration), and easily scale to large systems. ROS has two implementations: 'roscpp' using C++, and 'rospy' using Python. These two implementations generally mimick each other in as many ways as possible, however sometimes the behaviours of these implementations differ due to language differences.

ROS achieves it's primary goal via the use of packages. A ROS package contains all the information and files needed to perform one task. This can include code, datasets, and configuration files. ROS's computation and communication units are nodes. A ROS node represents a process that performs a particular computation. A package generally contains one or more nodes. Nodes can communcate between each other directly with the use of messages, or invoke services. The ROS Master node is a particular node which must run on every ROS system. The Master provides look-up services for nodes (so that they can find each other) with a URL-like system. The Master also provides the Parameter Server. The parameter server allows for nodes to store and retrieve data at runtime from a centralised, shared dictionary.

The majority of inter-node communication is achieved using topics. A topic represents a strongly typed message bus to which one or more nodes publish messages, and zero or more nodes subscribe to receive published messages. There are no access permissions to a topic, any node can publish or subscribe as long as they use the correct data type. Publishing to a ROS topic involves the following steps:

1. A message object is constructed containing the data to be sent
2. The message is placed in a queue of messages to be sent from that node
3. A message is pulled off the queue, and serialized
4. The serialized message is written in to a buffer
5. The buffer is written to the transport of every current subscriber

On the subscriber's end, the following occurs:

1. An incoming stream is written to an internal buffer
2. The buffer is split in to serialized messages
3. A message is deserialized to an object, and placed in to the subscriber's queue
4. The subscriber pulls message objects off the queue

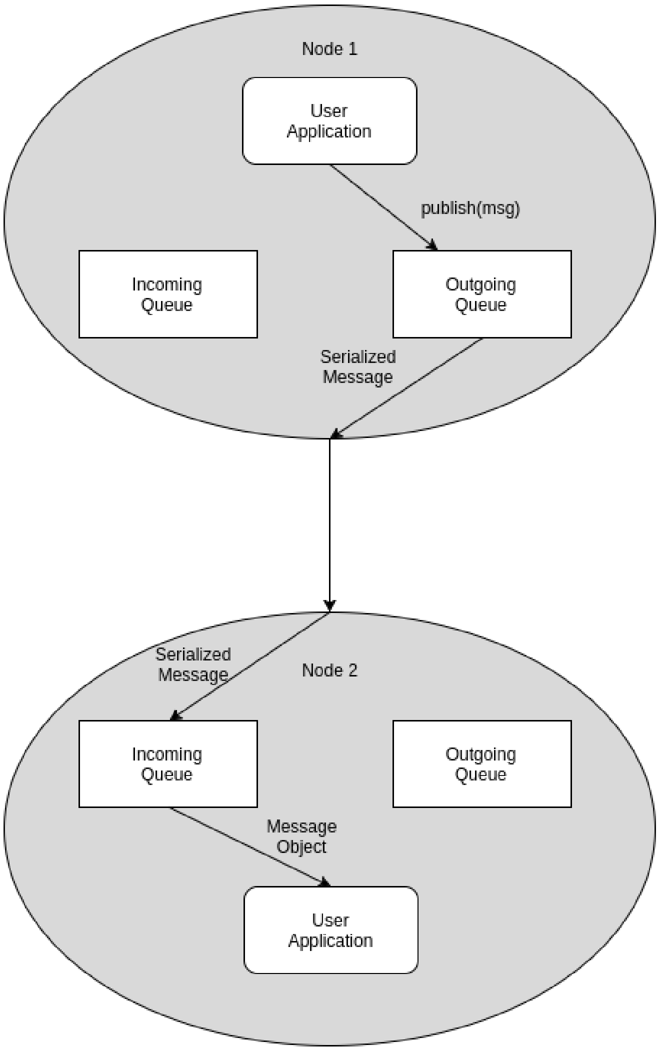

Figure 2.4: ROS process of sending a message from Node 1 to Node 2. Note that the internal buffers are ommitted from this figure.

Another option for inter-node communication is to use services. A service represents a restricted version of publisher/subscriber which implements a request/response interaction. When a node invokes a service, it sends a request (a message of a specific type) to the node implementing the service, and waits for the response. The response is sent back as another type (although possibly the same type).

The ROS community highly favours open source and sharing, as it aligns with the primary goal of ROS.

2.6 Configuration of Robots

This project involved the use of 9 identical robot cars with front-wheel steering. The cars were previously built from a Sunfounder Smart Video Car Kit for Raspberry Pi[22].

The car kit includes the physical pieces required to construct a robot car, such as a frame, gears, wheels, motors, step-down converters, and wires. The kit also includes a USB camera, and Wi-Fi adapter. It also includes a space for a Raspberry Pi (B+/2/3) to be seated. The robot cars were fitted with one Raspberry Pi 3 Model B each.

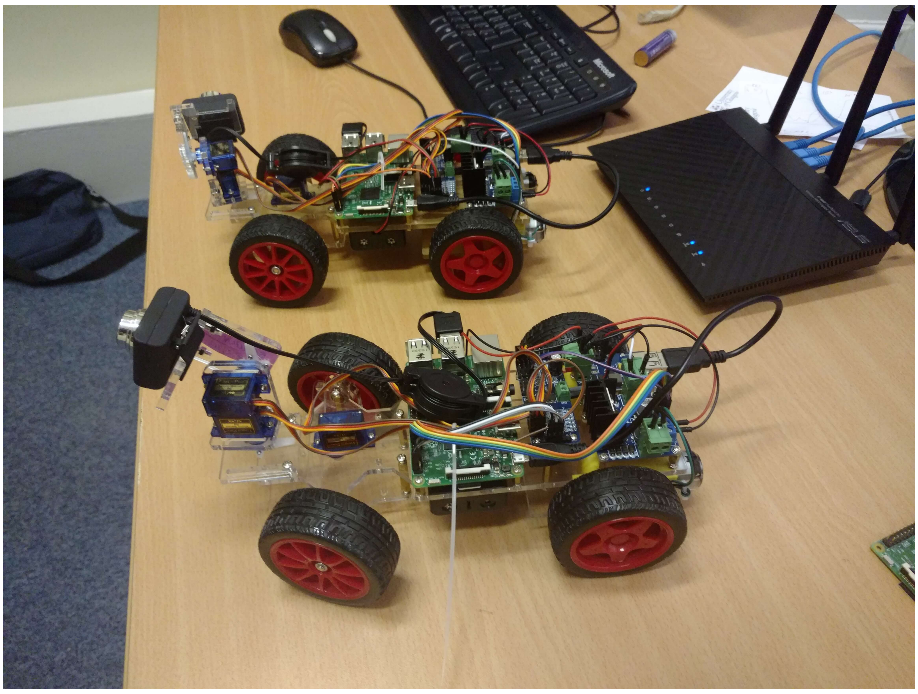

Figure 2.5: Photo of 2 of the Sunfounder Robot Cars

Chapter 3

Factors Affecting High Frequency Message Transmission

Multi-robot systems are types of distributed systems. The main concern in moving from a single-host distributed system to a multi-host distributed system is the introduction of more complex communication network. As the overall system is trying to complete a task, the individual nodes in the network must communicate in order to share things like the status of the node (CPU load, disk usage, etc), progress through the current task, and results of a task. The exact nature and direction of this communication is dependent on the architecture of the distributed system.

In ROS, the Master node monitors the progress and status of the other nodes, but does not involve itself in the application logic of the system, meaning that it does not process or handle the results of nodes. This is handled in a peer-to-peer (P2P) fashion dictated by the application developer - each node is responsible for choosing what information to request from other nodes, and what information to emit from itself using the Publisher/Subscriber model described in Section 2.5. The consequence of this architecture is that overall system performance can be dramatically affected by the communication performance between nodes.

In order to systematically evaluate how ROS' communication scales, one must first aim to understand which aspects of the system that are of greatest importance to performance - both in terms of latency of messages, and total throughput of communication. Section 3.1 aims to analyse the performance bottlenecks of a simple multi-robot system when sending very controlled and artificial data. Section 3.2 will then aim to validate the results of Section 3.1 using previously recorded data streams that are representative of a typical ROS system, with the goal that these conclusions will then be directly applicable to real ROS multi-robot systems. This chapter aims to provide an understanding of the fundamental communication performance characteristics of a multi-robot ROS system, so that vertical and horizontal scaling experiments can be contextualised.

Table 3.1: Chapter 3 Experiment Overview

Experiment	Description	Section
Revising Existing Code	Building upon prior work to obtain baseline expectations of ROS communication performance	3.1.1
Impact of Rebooting	Identifying whether rebooting the host machines affects performance of repeated message streams	3.1.2
Impact of CPU Clock Speed	Identifying whether performance of the message streams is limited by processing power	3.1.3
Impact of Wi-Fi Connection	Investigating the impact of performing ROS communication over a Wi-Fi based communication channel	3.1.4
Impact of CPU Clock Speed when using Representative Data	Aiming to validate the results of the CPU Clock Speed scoping experiment on data more typical of a ROS system	3.2.1
Impact of Wi-Fi Connection when using Representative Data	Aiming to validate the results of the Wi-Fi Connection scoping experiment using data more typical of a ROS system	3.2.2

3.1 Scoping Experiments

In order to acquire a detailed understanding of the performance characteristics of ROS' communication channels, a sequence of scoping experiments was designed. The experiments begin very simply - trying to gain baseline expectations of how well ROS will perform, and where problems may occur. Further experiments then investigate individual experimental parameters to identify problem areas for ROS. These experiments were performed using highly controlled data - specifically a single string consisting of 'Hello World'. This allows for rapid development and iteration of experiments, while keep the data sent constant. Later experiments (in Section 3.2) then explore how varying the data sent affects the conclusions of the scoping experiments.

3.1.1 Experiment 1 - Revising Existing Code

Introduction To begin, initial testing of ROS' communication ability must be carried out - meaning that before further investigation can be done, one must understand what baseline performance one can expect from ROS on the hardware platform used. A useful example is the need to identify an order of magnitude estimate for the message frequency (messages per second) we can expect from ROS. A network architecture must also be worked out that allows for repeatable and accurate measurements of inter-robot communication performance.

A prior investigator, Andreea Lutac[57], had began investigation of building a simple experimental environment - however the results were not as expected. The architecture created by Andreea involved three robots (or hosts), the master host would simply run the ROS Master node providing name resolution services to the other two nodes as described in Section 2.5. The sender host would contain a single ROS node that performs two tasks: sending a message containing a unique message identifier (ID) and a timestamp indicating the system time on the sender host the message was sent at, the second task would be to listen to responses from the third node. The third node is an echoer node which listens for messages sent from the sender, and simply sends them back. When the sender receives a message from the echoer, it again notes it's system time, and then writes message ID, sent time, and received time to disk - for later analysis. However, as mentioned earlier, the results were unexpected and unexplainable - the exact nature of which will be seen in the results section.

Objective The aim of the experiment was to analyse the transfer time of messages between two machines at varying message frequencies, and identify whether there was some limit as to how often ROS could send and receive messages on it's topics. The experiment therefore had several objectives. First, repeat Andreea's experiment using the same code[48], and verify the results were not due to a configuration error of the system. Second, to provide familiarity with ROS and grasp an understanding of how ROS communication works. And third, identify the underlying cause of the unexpected results and either analyse why ROS exhibits this behaviour or identify possible errors in the experiment code that would cause counter-intuitive results.

Hypothesis The expected result of the experiment was that message latency would be the same across all lower frequencies - until some bottleneck in the system was reached. It is expected that once a bottleneck is reached that message latencies would increase exponentially due to congestion (i.e. messages would be being generated faster than the system could process them, causing an increasing number to be left waiting in message queues).

Materials and Methodology The hardware set-up of this experiment was designed to isolate the system being tested from as many external variables as possible. The three host machines used were all Raspberry Pi 3 Model Bs freshly installed with the latest available stable version of the Raspbian OS (the official operating system for Raspberry Pis), and the latest available stable version of ROS (Kinetic). The Raspberry Pis were powered with a stable 5V power supply connected to the building's mains power. Each Pi was connected to a gigabit-capable Asus router via an Ethernet cable. The router was dedicated to the experiment, meaning that it had no other hosts connected to it other than the three Raspberry Pis and a Ubuntu desktop machine for controlling and monitoring of the Raspberry Pis. The Raspberry Pis had no peripherals (e.g. mouse, keyboard, camera, etc) connected, and were controlled via SSH connections.

As mentioned in the experiment introduction, the code base utilised to begin this experiment was previously developed[48], and involved three Raspberry Pis: a master, a sender, and an echoer. The master simply allows the sender and receiver to contact each other. The sender generates messages containing a message ID, a sent timestamp, and a payload - in this case, a simple 'Hello World' string - and sends them to the echoer. The echoer repeats any received message back to the sender exactly. Upon receiving the echo the sender then notes down message ID, sent time, and received time in a log file. This architecture is outlined in Figure 3.1, although the master node is omitted for clarity.

Figure 3.1: Experiment 1 - Communication Architecture

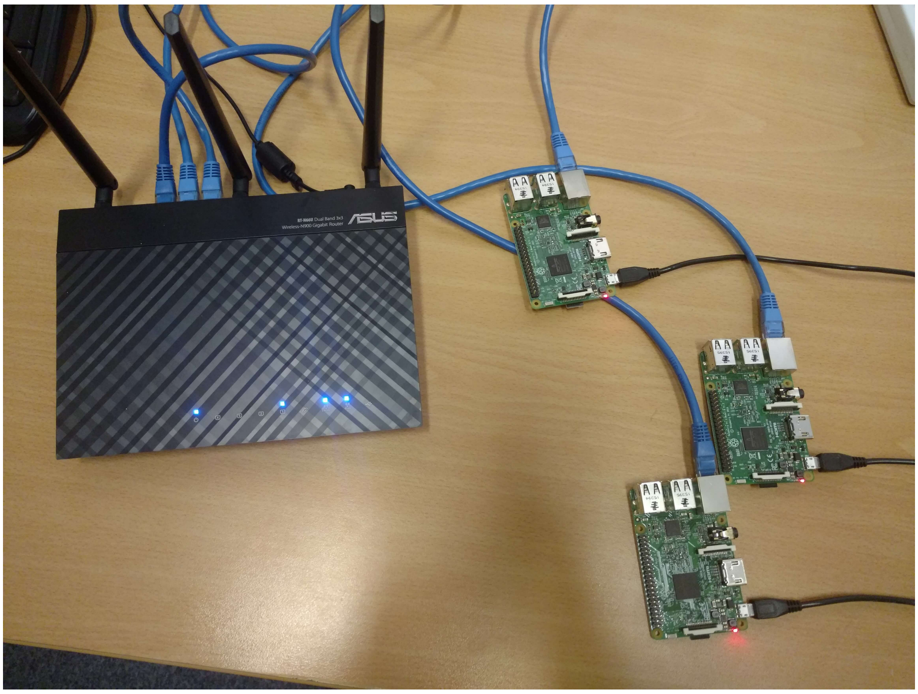

Figure 3.2: Photo of the three Raspberry Pi Model Bs and Asus router utilised in Experiment 1

Results and Discussion This set-up was coded and ran, however as mentioned, the results were unexpected - as Figure 3.3, the set-up gave results that were contrary to intuition. As the sending message frequency increased, better performance (lower message latency) was seen - although at very high frequencies a number of messages were completely dropped at the start. These correlations can be seen in Figure 3.3. *The results matched what was previously seen by Andreea - which directly conflicts with the hypothesis that increasing message frequency would eventually increase message latency.* Another interesting observation is that for 10KHz the first 169 messages were completely dropped (never received back from the echoer), and for 1MHz the first 312 messages were dropped. Figure 3.4 shows the total number of messages received for each frequency. *These dropped messages indicate that the publisher message queues for either the sender or the echoer reached max capacity (causing some messages to be overwritten before they are sent).*

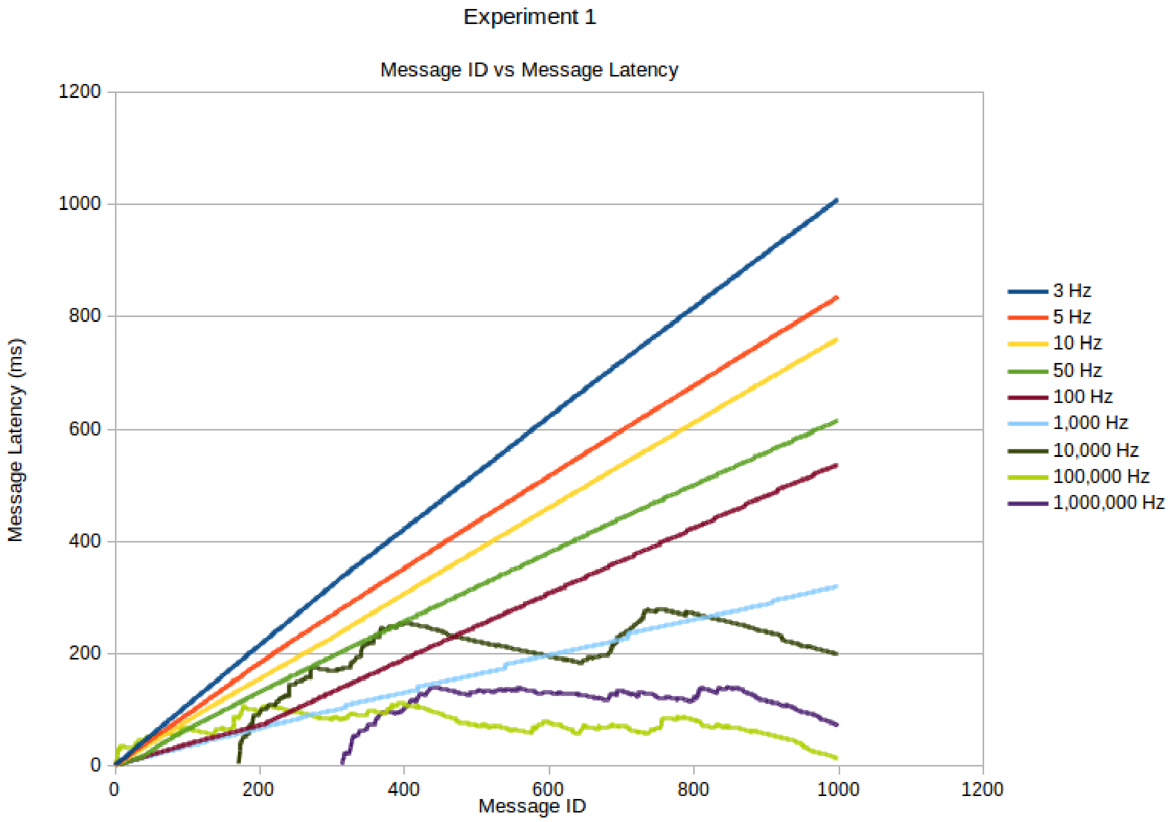

Figure 3.3: Experiment 1 - Message Latency by Message Frequency

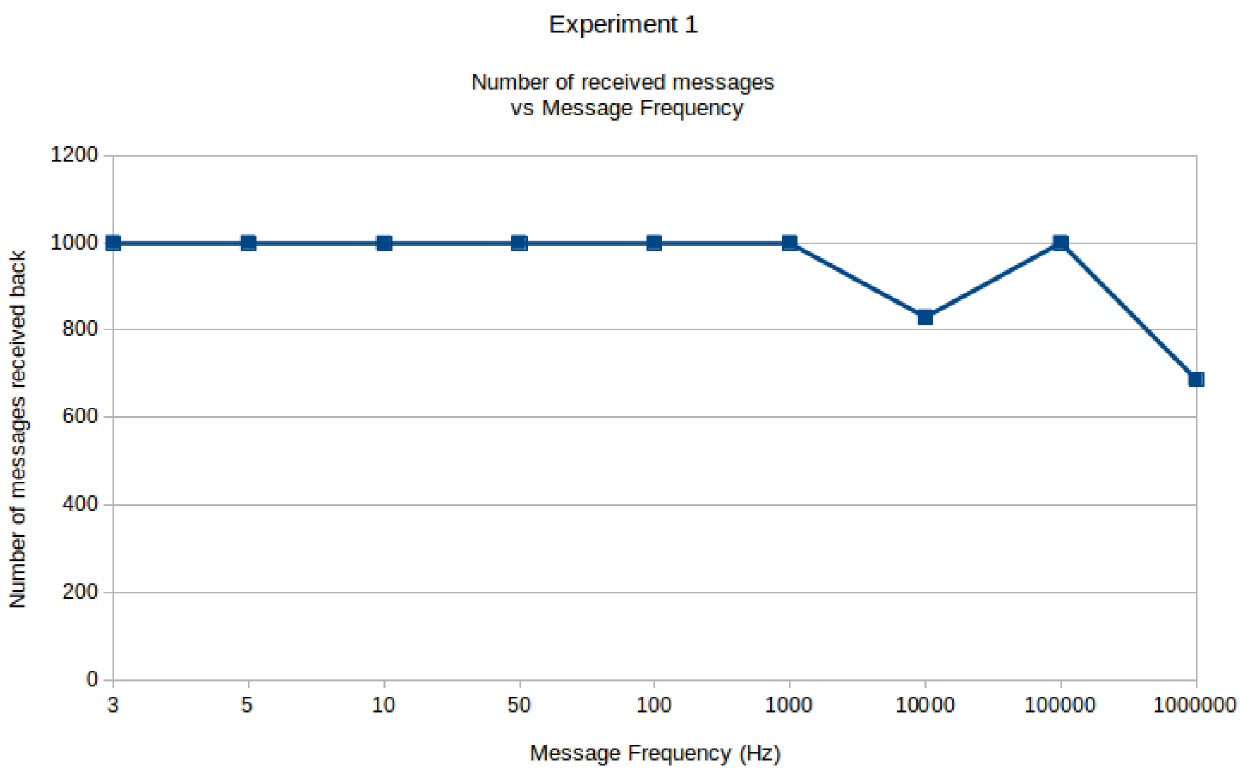

Figure 3.4: Experiment 1 - Number of Messages Received by Message Frequency

The next objective was to identify why these results were not as expected. The first step was to review the code being used to run the experiment, and try to identify any code that could cause anomalous behaviour. While conducting this review, two major issues were identified. The first was the echoer code had a delay similar to the sender when the experiment design mandated that the echoer always respond as fast as it can - so that only time taken to send the messages back-and-forth is measured. The second issue was the the maximum message queue size in ROS (how many messages can be buffered at once to compensate for a slow subscriber) was set equal to the message frequency of that run.

Listing 3.1: Echoer Original

```
import rospy
from rosberry_experiments.msg import StampedMessage
import time
import sys

RATE = None

def listener(msg, args):
    rate = rospy.Rate(RATE)
    pub = args[0]
    pub.publish(msg)
    rate.sleep()

def main():
    global RATE
    RATE = int(sys.argv[1])
    try:
        rospy.init_node('talker1', anonymous=True)
        pub = rospy.Publisher('chatter_s', StampedMessage, queue_size=RATE
            )
        sub = rospy.Subscriber("chatter_m", StampedMessage, listener,
            callback_args=[pub])
        rospy.spin()
    except rospy.ROSInterruptException:
        pass

if __name__ == '__main__':
    main()
```

These issues were resolved by removing the code that executed the delay in the echoer, and by setting the maximum queue size to be equal to 1000 in every experiment (the number of messages expected to be sent). These modifications can be seen in Listing 3.2. At this point a Bash script was created to automate this and future experiments. The script runs through a specific range of message frequency values multiple times (depending on the number of experimental runs specified). This allowed for reliable, repeatable experiments by reducing the margin for human error - and reducing the work required to repeat experiments.

Listing 3.2: Echoer Modified

```
import rospy
from rosberry_experiments.msg import StampedMessage
import time
import sys

def listener(msg, args):
```

```
    # No longer waits after receiving a message
    pub = args[0]
    pub.publish(msg)

def main():
    N = int(sys.argv[2])  # Total number of messages expected (1000)
    rospy.init_node('talker1', anonymous=True)
    pub = rospy.Publisher('chatter_s', StampedMessage, queue_size=N)
    sub = rospy.Subscriber("chatter_m", StampedMessage, listener,
        callback_args=[pub])
    try:
        # Wait until interrupted
        rospy.spin()
    except rospy.ROSInterruptException:
        print "Exception: ROSInterruptException"

if __name__ == '__main__':
    main()
```

The experiment was then repeated using this new echoer code[49]. Figure 3.5 shows the message latency for each message at all frequencies. All frequencies less than 10KHz had concistent message latencies around 1.5ms, but 10KHz, 100Khz, and 1MHz exhibited very erratic performance - indicating that beginning around 10KHz the system begins experiencing a bottleneck. Figure 3.6 shows mean latencies across the entire message streams for each frequency - *clearly demonstrating that performance is very consistent up until 10KHz for this set-up*.

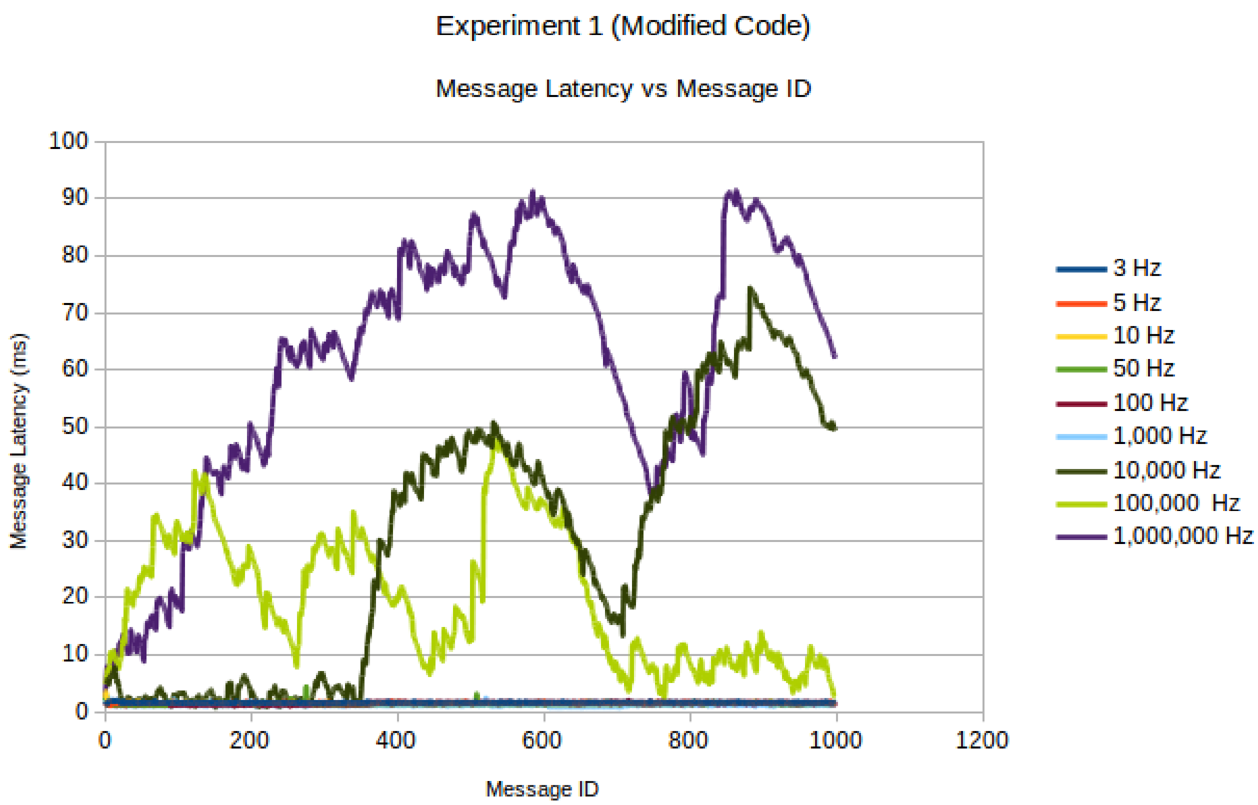

Figure 3.5: Experiment 1 (Modified Code) - Message Latency by Message Frequency

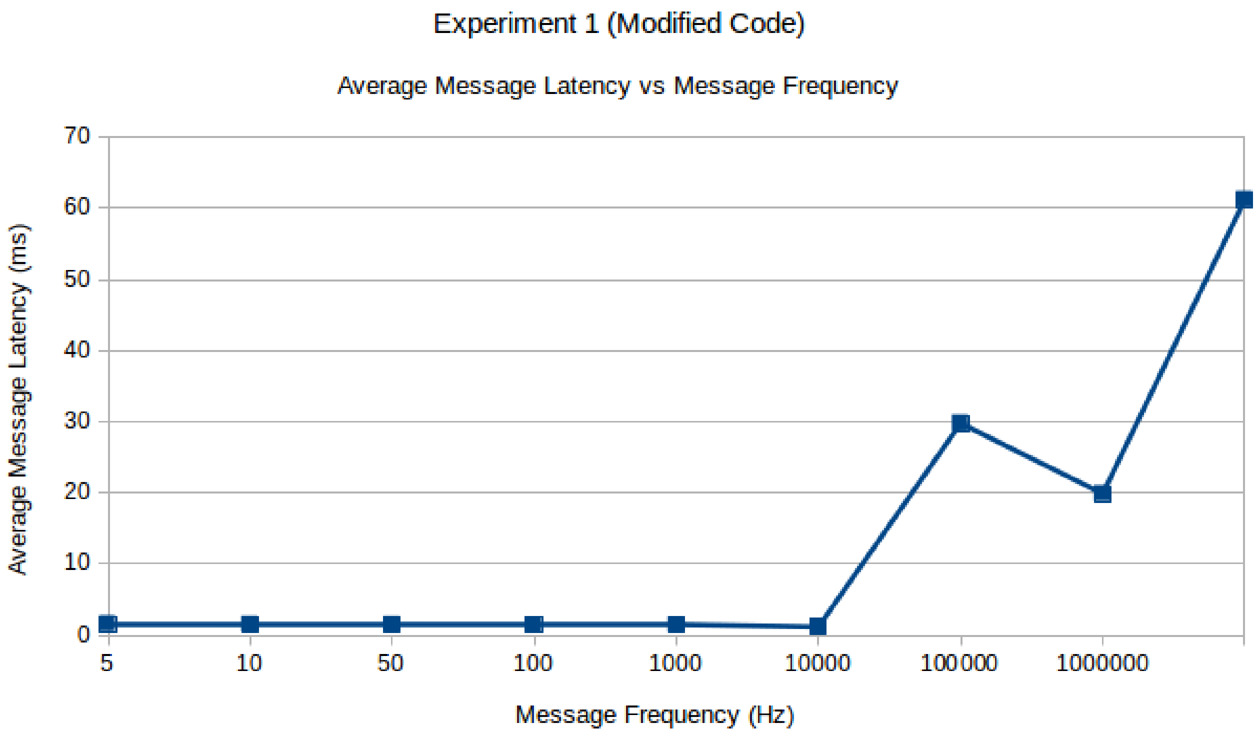

Figure 3.6: Experiment 1 (Modified Code) - Mean Message Latency by Message Frequency

Disk Writing

At this point it was theorized that writing the results to disk during the experiment (as the sender receives messages back from the echoer) could be adversely affecting the performance at high message frequencies. Thus, an informal experiment was conducted to investigate if it was feasible to postpone writing the records to disk until the end of the message stream. This was coded[50], however unexpectedly gave rise to significantly higher message latencies. It is presumed this was an issue with the implementation - but it was then noted that Python uses the operating system's buffering mechanic to efficiently write to disk in blocks. *Thus this idea was discarded, as it not expected that saving writing to disk until the end would be significantly faster than the buffering provided by Linux.*

3.1.2 Experiment 2 - Impact of Rebooting

Introduction It was seen while running Experiment 1 that at higher frequencies running the same experiment multiple times in a row would give erratic results. The first run might have reasonable performance, then repeating it would give poorer results, and the third run would be reasonable again. It was proposed that the behaviour could be caused by operating system buffers filling up and then not being purged until either full, or being cleared by some other process. Rebooting the systems has the opportunity to affect performance by stopping any background processes, interrupting slow processing messages from previous runs, and resetting any message caches and buffers in memory.

Objective Experiment 2's primary objective was to evaluate whether the performance of ROS' inter-robot communication was being affected by not fully rebooting the sending machine and the echoer machine in between experimental runs - in other words, whether the performance of ROS messages was affected by previous messages sent on the system. Experiment 2 also had the secondary objective of providing greater insight in to where the exact barrier between ‘good, consistent performance’ and ‘poor, erratic performance’ is - as in Experiment 1 there was a jump in latency between 1KHz and 10KHz message frequencies.

Hypothesis The result of the experiment was hypothesised to demonstrate no significant difference between rebooting and not-rebooting at any message frequency. It is expected that the erratic performance between runs is more likely to be caused by interference from uncontrolled background processes, or a resource bottleneck such as CPU speed.

Materials and Methodology The hardware set-up is identical as Experiment 1 (see Section 3.1.1), however the methodology is different. This experiment compares two set-ups: the first, ‘Full Reboot’ involves executed a full system reboot of all involved Raspberry Pis between each experiment run, and the second, ‘No Reboot’ involves simply waiting a few seconds before executing the next run. Both configurations are to be repeated 5 times in total, and latencies averaged across the 5 datasets. The messages being sent are also identical as Experiment 1. The software used in these experiments was the same as the revised software[49] in Experiment 1, although a different range of message frequencies was specified in the Bash script used to automate the experiment.

Results and Discussion Figure 3.7 demonstrates that for relatively low message frequencies the mean message latency was consistently 1 - 1.5ms (the peak around message 500 in the full reboot data was due to a single erroneous run, possibly indicating a system update or some other scheduled background process took place). Figure 3.8 is characteristic of the higher frequency runs - the no reboot runs generally gave equal or better performance compared to the full reboot runs. See Appendix B for other mean graphs, and individual run graphs.

We can conclude from this that *rebooting the host systems between experimental runs would result in worse communication performance for ROS*, thus in future experiments the host systems will not be rebooted between runs. Secondly, we can also conclude that *the exact frequency that message latencies begin increasing above nominal values is somewhere between 1KHz and 4KHz.*

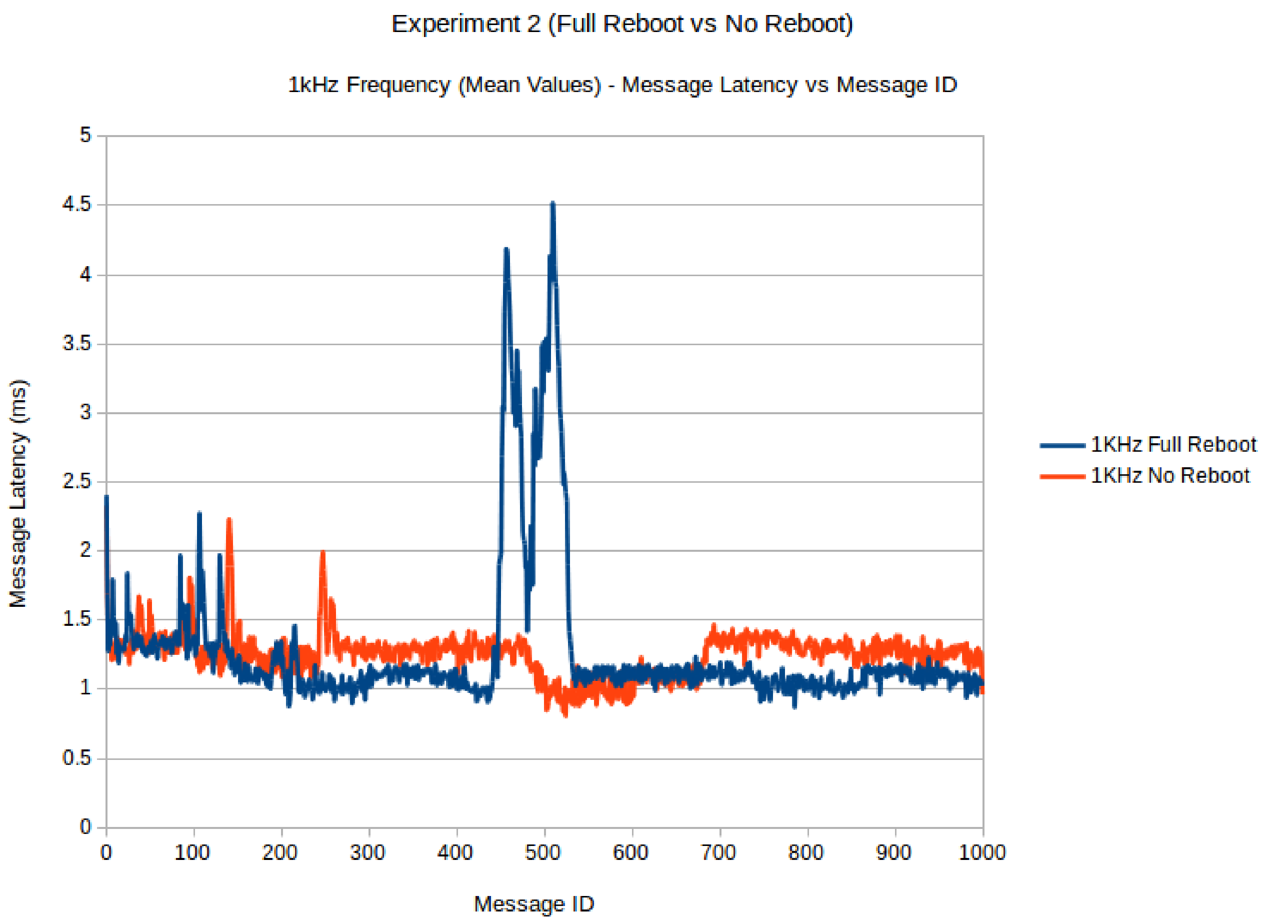

Figure 3.7: Experiment 2 - Mean Message Latency 1KHz Message Frequency

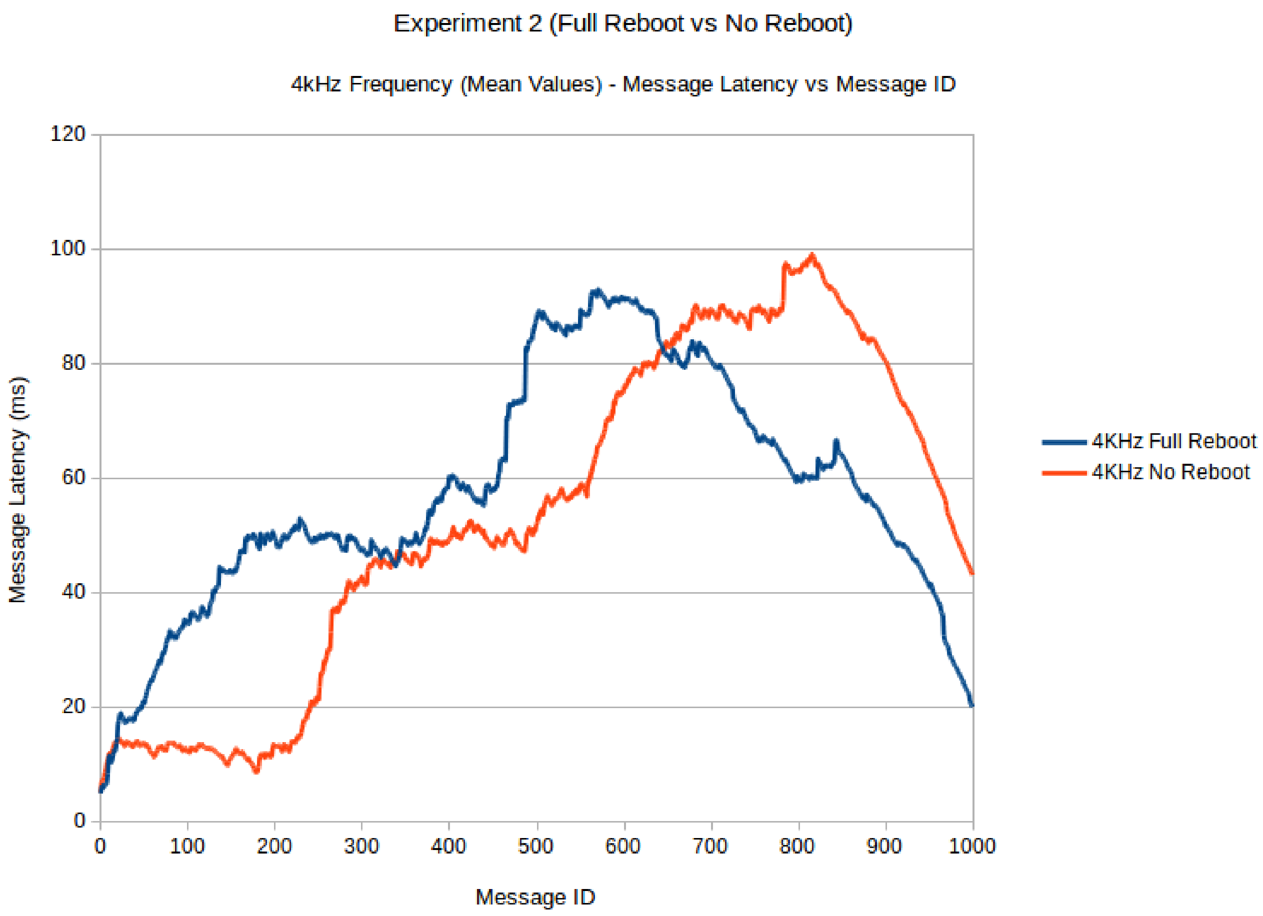

Figure 3.8: Experiment 2 - Mean Message Latency 4KHz Message Frequency

3.1.3 Experiment 3 - Impact of CPU Clock Speed

Introduction During execution of Experiment 1, it was noted that higher message frequencies generally used a greater percentage of CPU time (using the Linux 'top' utility). This gives rise to the possibilty that higher message frequencies cause higher latencies due to being CPU bound (meaning that the available processing power is not sufficient to process messages in a timely manner).

Objective The purpose of this experiment was to identify whether the limited CPU power of the Raspberry Pi system was playing a significant role in the bottlenecking of message communication at higher frequencies. As in Experiment 2, this experiment has a secondary objective of further narrowing down the precise message frequency boundary that message latencies begin reducing - Experiment 2 identified this to be somewhere in the range between 1KHz and 4KHz.

Hypothesis The hypothesis for the experiment was that frequencies which we had previously seen high message latency for would result in even higher latency, and the maximum 'low latency' frequency would be lower as the core clock speed reduces.

Materials and Methodology As in Experiment 2, the hardware platform is the same as presented in Experiment 1 (Section 3.1.1), as is the software[49]. This experiment requires modifying the CPU speed of the host machines. This is achieved by underclocking the Raspberry Pi 3 Model B CPU by 25%, and 50%. The reason to do this instead of changing the hardware (for example changing to desktop machines) is that this method keeps variables such as CPU architecture, disk speed, and software stack the same across the experiment. It is also preferable to underclock rather than to overclock the Raspberry Pis (say to 125%) as this can introduce a number of system instabilities (e.g. corrupted memory, or crashing) which can be difficult to detect. One potential issue is that modern CPUs often automatically underclock (reduce the clock speed and voltage) themselves, thus it can be hard to know exactly what CPU speed is being used throughout the experiment. Thus, several times throughout the experiment the current CPU speed was checked, and verified to be running at the specified maximum frequency. The exact method utilised to underclock the Raspberry Pis was to set the 'arm_freq' variable in the '/boot/config.txt' configuration file on the Raspberry Pis - the new clock speed then takes effect on the next reboot.

In order to give fine-grained results, many more message frequencies were tested than in previous tests. The experiment cycles through a range of message frequencies from 200Hz to 2KHz in 200Hz steps (e.g. 200Hz, 400Hz, ..., 1800Hz, 2000Hz). Three runs were conducted at each CPU speed, cycling through the entire frequency range each time. There was a 15 second waiting period between each message frequency run to allow for slow messages to clear the network before the next message frequency was run.

Results and Discussion The results agreed with the hypothesis. At 100% CPU clock speed, only the highest 3 frequencies showed sustained degradation of performance, however at 75% the top 5 frequencies had increased message latencies, and at 50% the top 7 had increased message latencies. Overall message latency was also increased as CPU core clock speed reduced. Even at the lowest frequency of 200Hz, 100% CPU had an average message latency of 1.188ms, 75%'s was 1.399, and 50%'s was 1.609ms. Higher message frequencies demonstrated greater differences as shown in Figure 3.9.

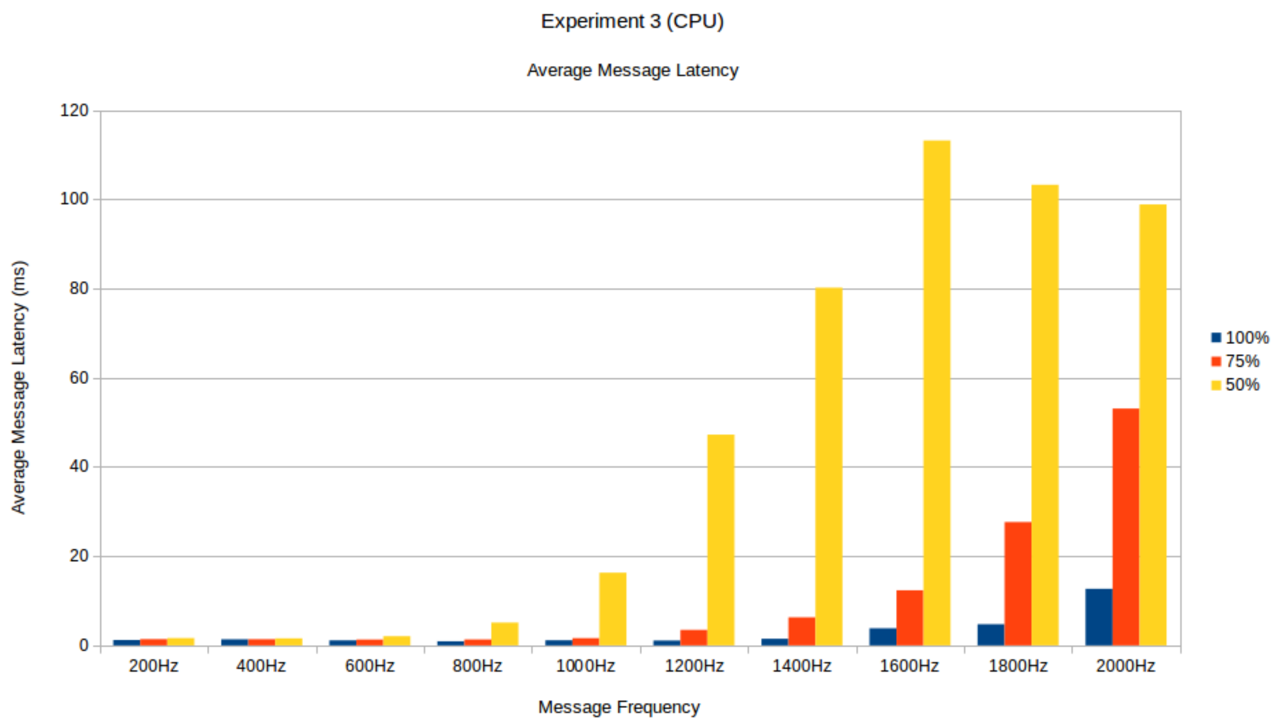

Figure 3.9: Experiment 3 - All CPU Speeds, All Frequencies

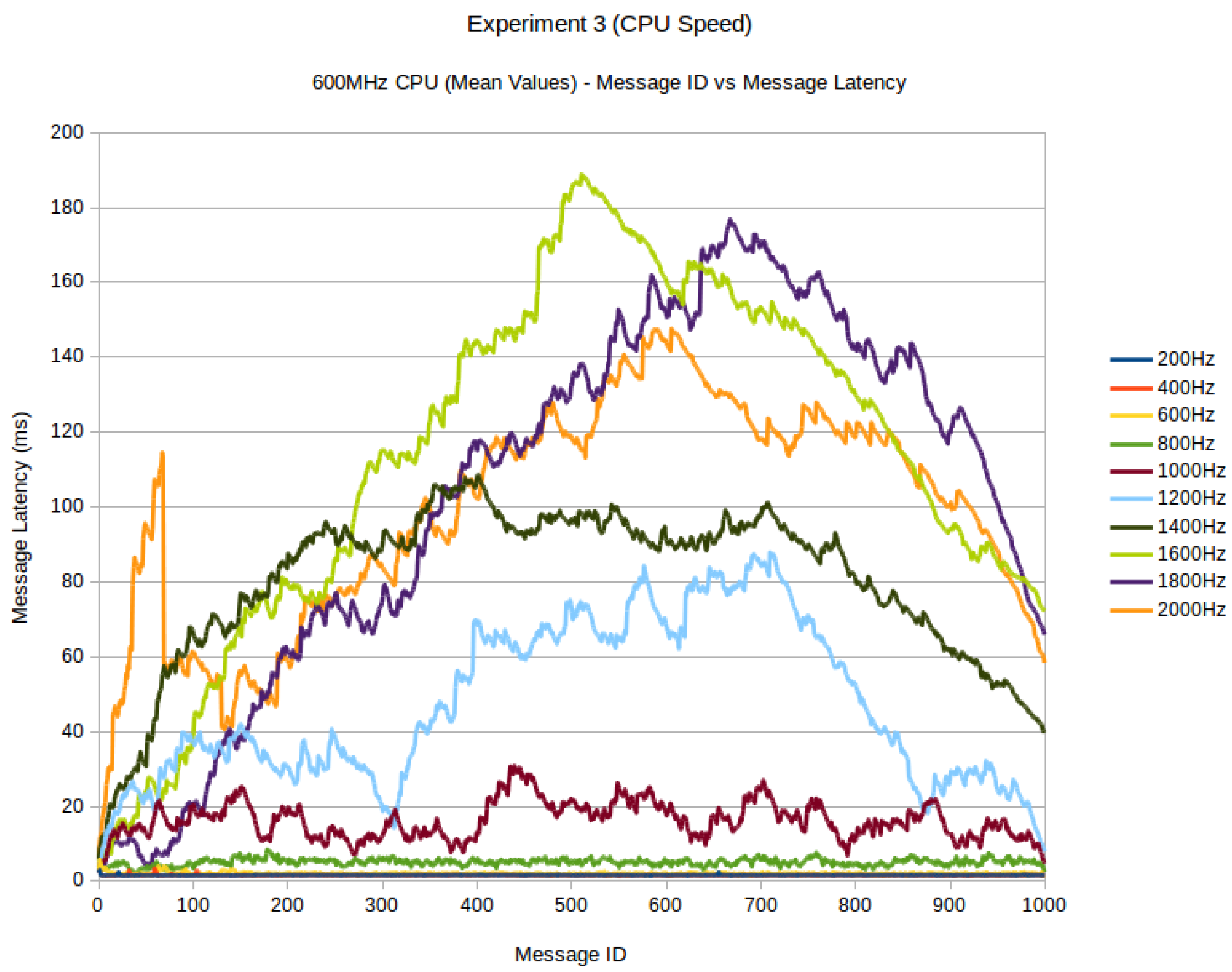

Figure 3.10: Experiment 3 - 50% CPU Speed, All Frequencies

We can conclude from these results that the *CPU performance of the host machines can be a limiting factor for high frequency ROS communication, and thus CPU utilisation is an important metric when evaluating why a ROS system is experiencing poor performance.* Secondly, we can conclude that *for this test set-up and message size, at 100% CPU speed the maximum sustainable message frequency is approximately 1400Hz.*

3.1.4 Experiment 4 - Impact of Wi-Fi Connection

Introduction Experiments 1, 2, and 3 were all conducted using a wired (Ethernet) connection from each of the Raspberry Pis to the router. This is optimal for communication performance as Ethernet is generally significantly faster than a wireless (Wi-Fi) connection, and is also significantly stabler - meaning that less packets become corrupted during sending, and therefore don't need to be sent multiple times. Wi-Fi is also much more susceptible to interference from other devices. However Wi-Fi does have the advantage of physical freedom of movement. In a multi-robot system it is reasonable that it would be advantageous (or even a requirement) for each robot to be physically mobile, thus in some cases it may be unavoidable to use Wi-Fi.

Given that it is plausible for a multi-robot system to use Wi-Fi as it's communication channel (the robot car platform described in Section 2.6 only supports Wi-Fi as physical access to the Ethernet ports is not possible), it is important to understand how Wi-Fi communication will affect performance compared to the ideal wired situation.

Objective Experiment 4 investigates the impact of using a Wi-Fi network connection for message passing experiments. The target robot car platform described in Section 2.6 utilise a Wi-Fi connection, thus understanding the effect this will have on the communication performance is required to understand the final performance of the system.

Hypothesis The hypothesis was that switching to Wi-Fi would increase message latencies across all frequencies, and also reduce the maximum frequency that message latency remains consistent across the entire message stream (as in the previous experiment described in Section 3.1.3).

Materials and Methodology The hardware platform (Raspberry Pi 3 Model B) utilised in these experiments contains Broadcom 802.11n WiFi chips which support a 2.4GHz network connection. This should give the Raspberry Pis an average data throughput of 25 Mbps, according to Cisco data[40].

The experiment consists of running the same code as created in Experiment 1[49], in both wired (Ethernet) and wireless (Wi-Fi) settings. As before, the code will send timestamped messages from the sender host to the echoer host, which will send it back to the sender. The sender then records the message ID and sent and received times to file. The code is run 3 times with a range of message frequencies from 200Hz to 2000Hz in 200Hz steps (200Hz, 400Hz, ..., 1800Hz, 2000Hz) to obtain averaged results for each message frequency.

Results and Discussion The experimental results after 3 runs in each setup appeared to mostly agree with the hypothesis. Overall message latencies were higher (as expected) across all frequencies using WiFi except 2000Hz - as shown in Figure 3.11.

Figures 3.12 and 3.13 demonstrate averages across 3 runs for all frequencies of the message streams, for Ethernet and WiFi respectively. It is clear from these figures that *WiFi gives more erratic results than Ethernet, as well as overall lower performance, and thus it's impact must be carefully considered when using WiFi in future experiments.*

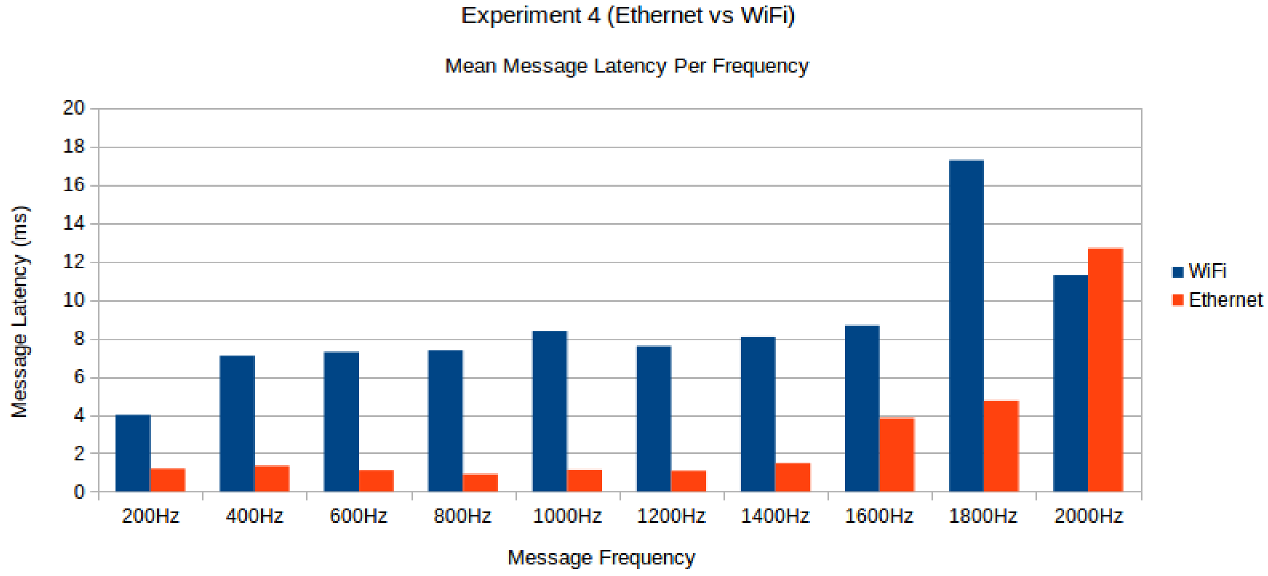

Figure 3.11: Experiment 4 - Ethernet, All Frequencies

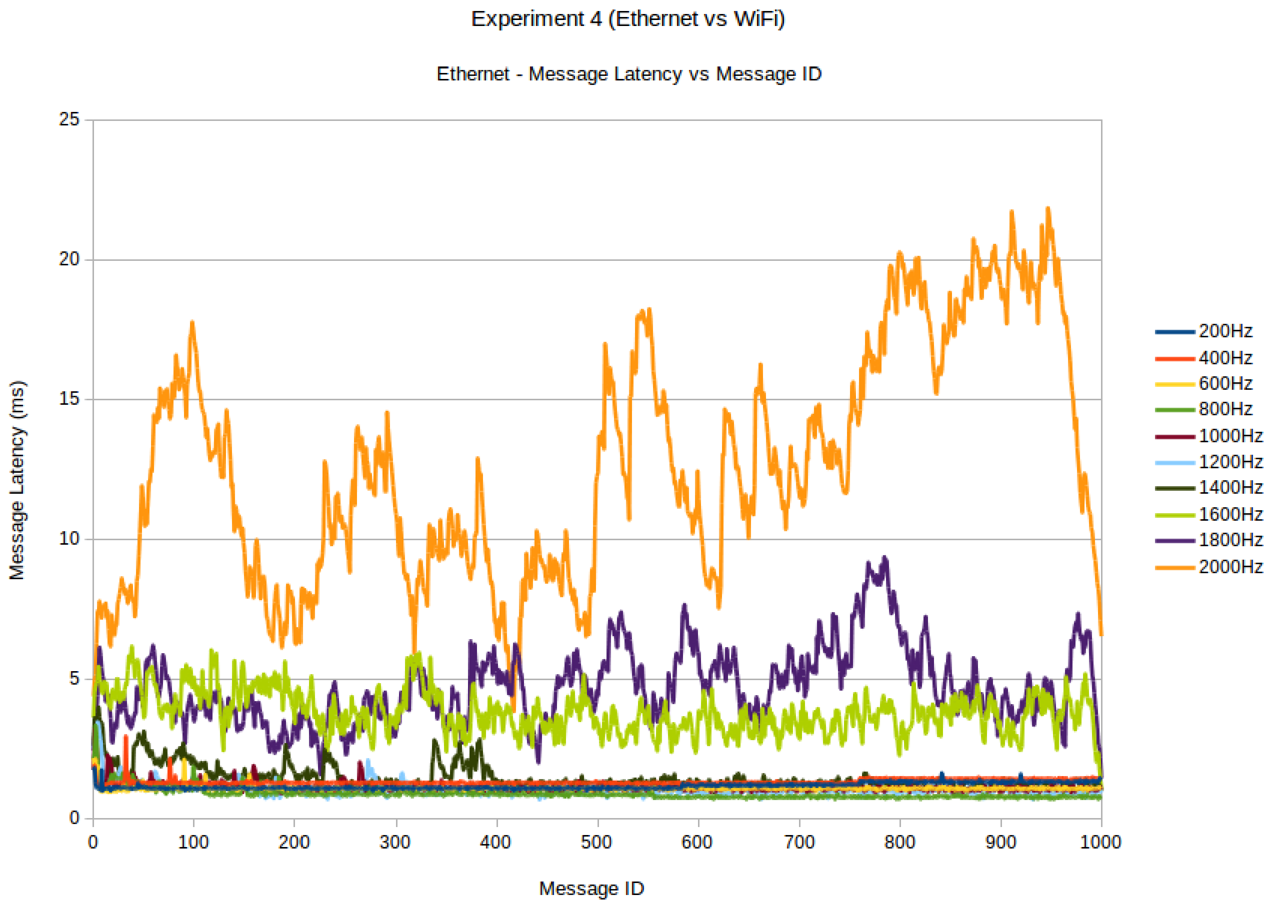

Figure 3.12: Experiment 4 - Ethernet, All Frequencies

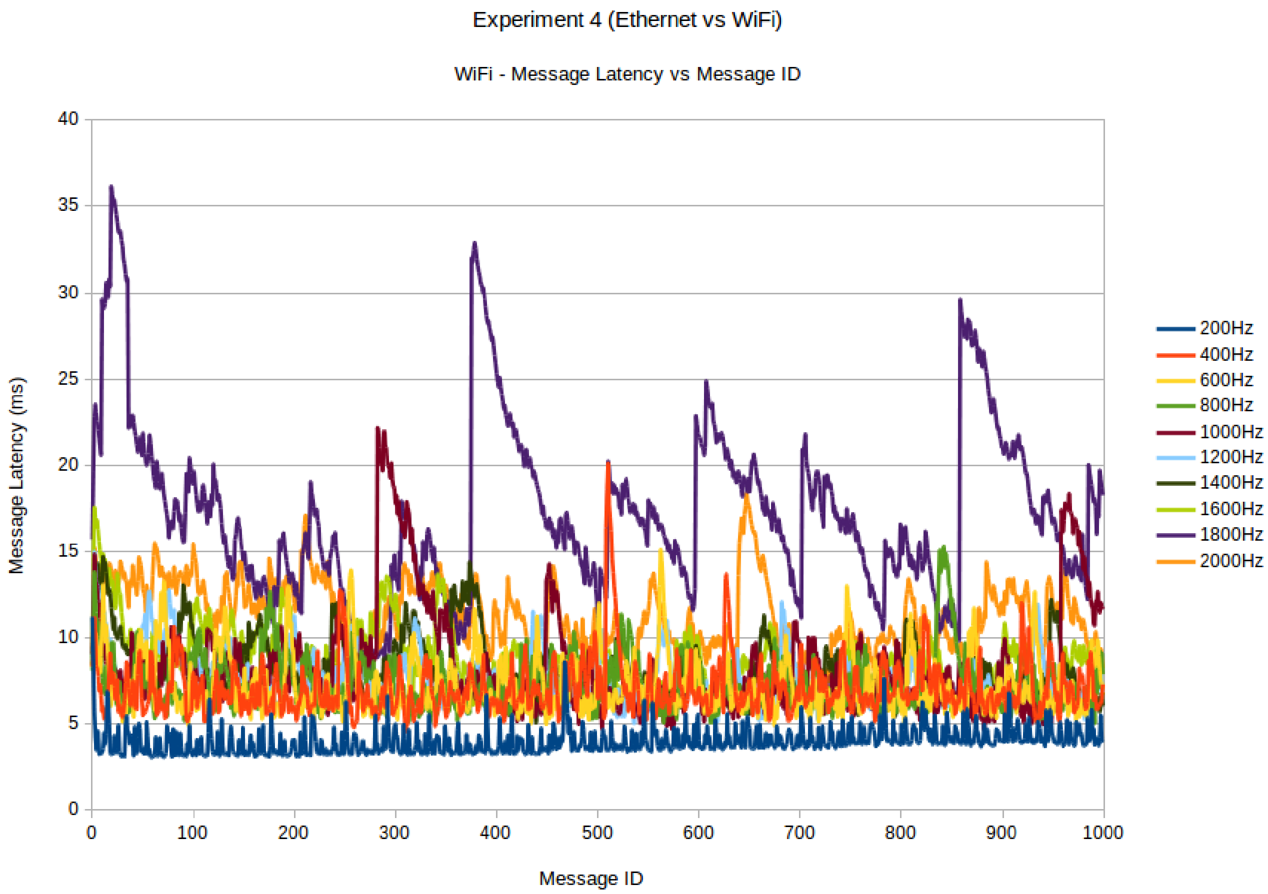

Figure 3.13: Experiment 4 - Wi-Fi, All Frequencies

3.2 Representative Data Experiments

Previous experiments (in Section 3.1) were presented as scoping experiments. These were designed to systematically evaluate and gain understanding of which variables were of concern when running tests on ROS' communication performance. However, it was not certain that these prior conclusions would translate well to using ROS in a realistic scenario since throughout the scoping experiments, the same message of 'hello world' was used as dummy data. This is a very small message payload of only 11 bytes.

In order to verify the conclusions were correct, samples of realistic data were explored. Two data types were settled on, 'sensor data' and 'video data'. Sensor data is the type of data likely the come from a physical sensor on the robot. This data is characterised by small message sizes, such as 4KB. Video data is the type of data recorded from a video camera peripheral, such as a webcam. This type of data generally has larger message sizes in range of hundreds of kilobytes, up to several megabytes per message depending on the resolution and bitrate of the camera.

The exact data sets used in the coming experiments were acquired from the MIT Stata Center dataset [8]. The dataset is described as follows by the producers of the data: "The MIT Stata Center Data Set is a vast scale data set collected over a multi-year period in a 10 storey academic building. It contains sensor data collected since January 2011. As of September 2012 the data set comprises over 2.3TB, 38 hours and 42 kilometres (the length of a marathon)"[8]. A dataset from 2012 may appear outdated; however it suits the goal of these experiments perfectly, as only data types and message sizes, which haven't changed since, are of importance. The data is recorded from sensors on the PR2 (a two-armed wheeled robot), which was one of the first robots to ever be designed using ROS as a middleware. The exact values of the data are not important for this evaluation of ROS, it is only the size and frequency of the messages that are important as there is no analysis or computation being executed on the data being transmitted.

This dataset contains both sensor feeds (from a laser sensor), and video feeds (from a Kinect RGB + depth camera). The sensor data was a 20Hz message stream, which was measured to use 85kB/s of bandwidth, giving an individual message size of 4.25kB. The video data used was a 30Hz (30 frames-per-second) RGB (colour) video stream, with a resolution of 640 x 480 pixels. This message stream was measured to use 9.25MB/s of bandwidth, implying a message size of 308KB. These data sets are very large (20 - 50GB) which would require modification to the test system (Raspberry Pi 3s). Thus these datasets have been filtered down to 60 seconds of recording, resulting in 1791 camera images, and 1194 LaserScan readings. This number of messages was chosen at it would give a good number of messages to average latency over (over 1000 for each data type), but would also easily fit on to the SD-card storage of the Raspberry Pis, which is only 8GB in size. The total size of the 60 seconds of data was approximately 1GB.

3.2.1 Experiment 5 - Impact of CPU Clock Speed when using Representative Data

Introduction In Section 3.1.3 it was concluded that the upper limit of message frequency (in terms of reliable performance) was likely caused by a limitation of the processing power of the host system when using Ethernet. This indicated that ROS developers must be aware of the processing power of their system when creating communication-oriented systems. However this was concluded when using very small message payloads (only 11 bytes), and a variation of message size may introduce other more important factors.

Objective This experiment aims to verify the results of Experiment 3 in Section 3.1.3 when using data representative of a real ROS system, using an Ethernet connection as before. This is achieved by repeating the experimental set-up and procedure, but replacing the sent messages with a pre-recorded message stream.

Hypothesis The expectation was that sensor data (with it's relatively small message sizes) would give similar results to the dummy data used previously (a string consisting of 'hello world'), and that video data would demonstrate different performance characteristics due to the significantly larger message sizes.

Materials and Methodology The experiment is almost identical to what was presented in Section 3.1.3 for Experiment 3, with the single difference being the data sent as a payload is switched out for pre-recorded data. Changing the data types being sent, and reading data from a ROSbag required some implementation changes[51]. The exact data used is a LaserScan sensor stream, and a video stream - as described in Section 3.2. This is achieved using a ROS bag provided as part of the MIT Stata Center Dataset. A ROS bag is a datastructure which allows replaying ROS topics. The topic can be replayed at the same rate it was recorded at, or any other desired rate. This allows for a variety of message frequencies to be used, as in previous experiments.

Results and Discussion *The experiment confirms the results of Experiment 3.* For sensor data, 100% CPU speed (1.2GHz) was the lowest latency at 7 out of 10 message frequencies, as shown in Figure 3.14, and 50% CPU speed (600MHz) was slowest at 9 out 10 frequencies.

For video data, the trend held at the lower experimental frequencies (see Figure 3.15) with low message latencies, which stepped up slightly when CPU speed was decreased. However, the change in performance was significantly less than for the smaller message sizes of the sensor data - for example, at 20Hz the top and bottom values were only 4.9% different, whereas for sensor data at 400Hz the top and bottom values were 54% different. This leads to the conclusion that *CPU speed becomes a less important consideration as message sizes increase.* CPU speed became entirely insignificant at higher message frequencies for video data, as it is thought the network interface is the bottleneck due to the larger message size. Message frequencies at larger than 30Hz gave much higher average latencies (around 4 seconds at 40Hz, compared to 60 milliseconds at 20Hz), and showed no correlation with CPU speed (see Figure 3.16).

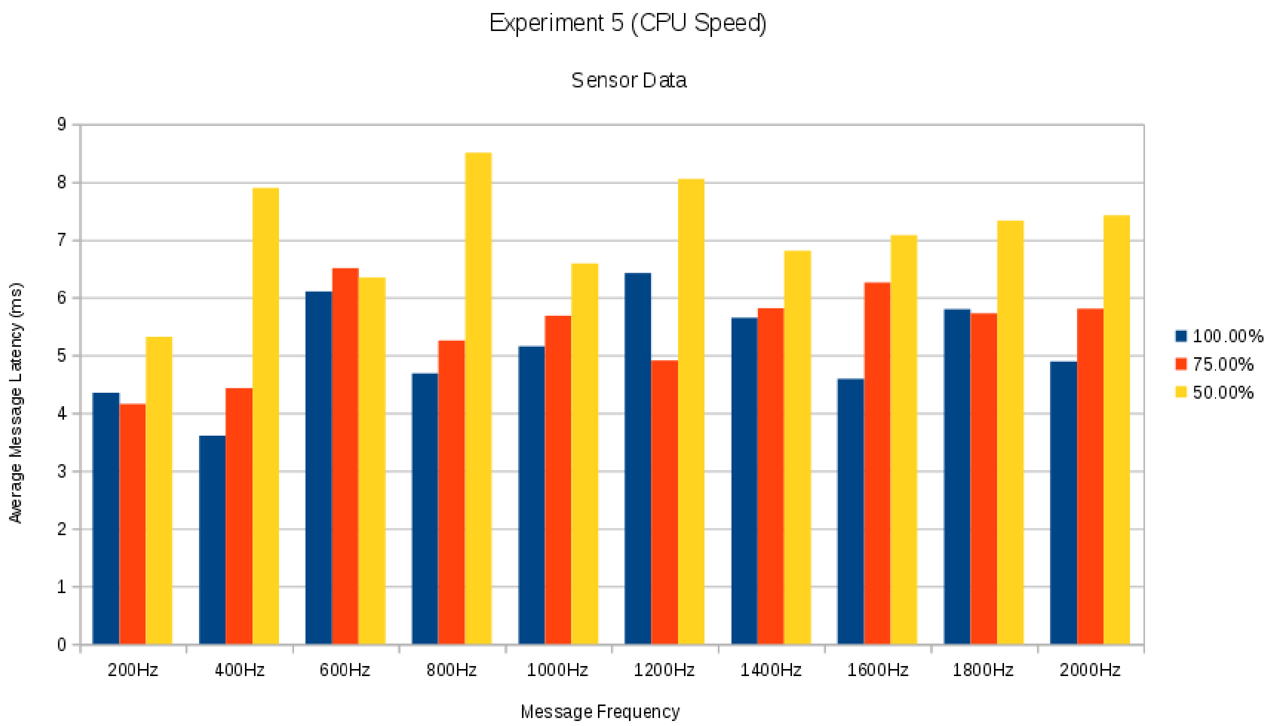

Figure 3.14: Experiment 5 - Sensor Data, All Frequencies

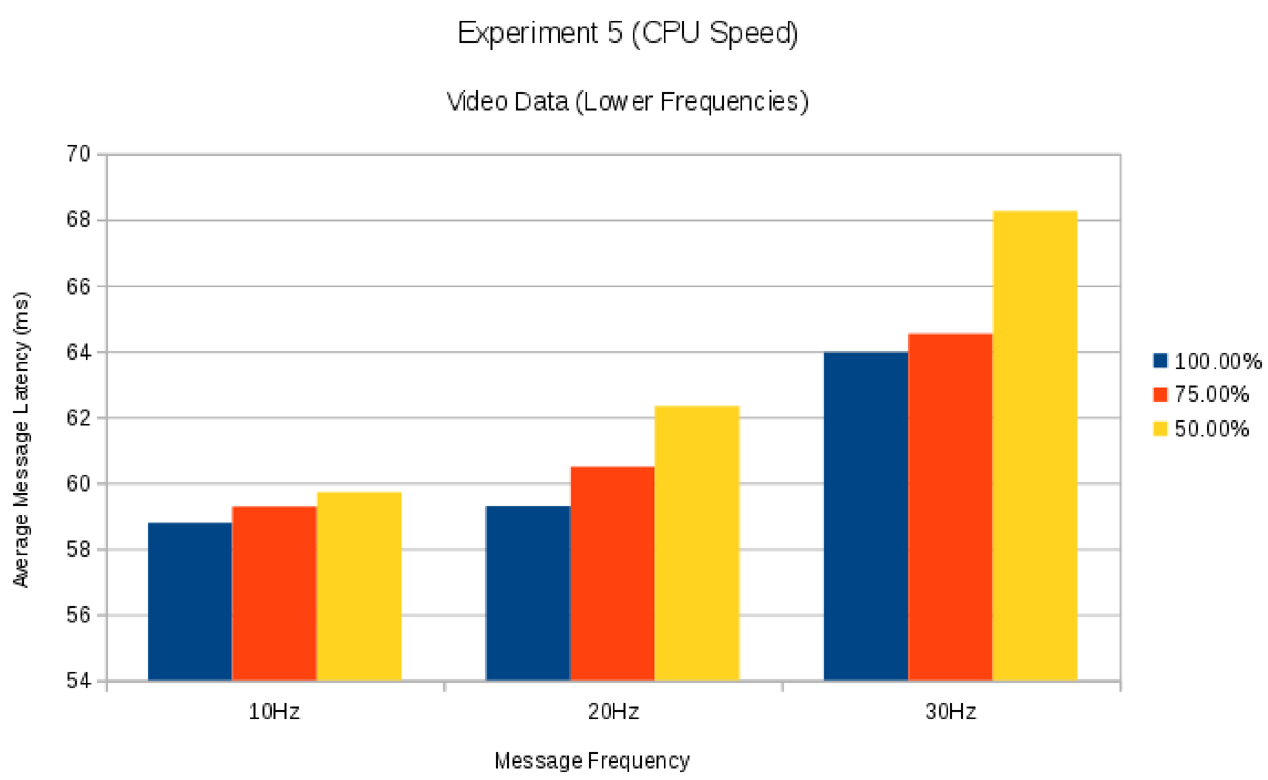

Figure 3.15: Experiment 5 - Video Data, Low Frequencies

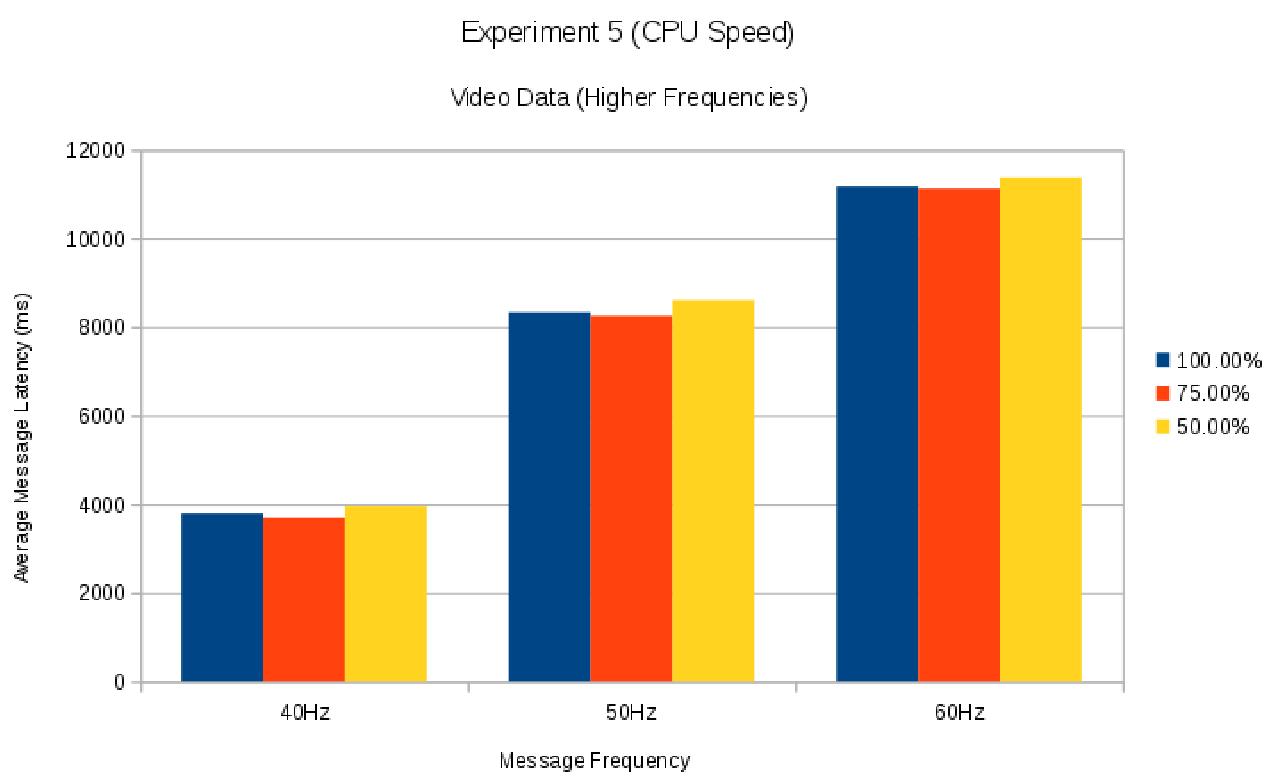

Figure 3.16: Experiment 5 - Video Data, High Frequencies

3.2.2 Experiment 6 - Impact of Wi-Fi Connection when using Representative Data

Introduction In Section 3.1.4 it was concluded that using a Wi-Fi connection between ROS hosts resulted in less predictable message latencies, as well as higher message latencies at all message frequencies due to the slower communication channel.

Objective The aim of this experiment was to verify the results of Experiment 4 were valid when using more common message types. The two message types to be used are sensor data, and video data, as discussed previously in Section 3.2.1.

Hypothesis Expectations from this experiment were that the results of Experiment 4 would be exacerbated, meaning that overall latencies would be even higher than seen in Section 3.1.4 and the frequency at which performance starts to become worse would be lower. The higher latencies of Wi-Fi seen in Section 3.1.4 would be even higher by using larger message sizes, and performance would degrade even faster than in Experiment 4.

Materials and Methodology This experiment was a repeat of Experiment 4 (described in Section 3.1.4), except using data representative of what might be used in a typical ROS system. The data used was the same MIT Stata Center dataset used previously in Experiment 5, described in detail in Sections 3.2 and 3.2.1 - however only sensor data was evaluated as video data was too large to work with on the Raspberry Pi platform since message latencies were frequently several seconds even at very low frequencies. The implementation of this experiment required changing the message data types from what was used in Experiment 4, to what was used in Experiment 5 - thus the code was the same as used in Experiment 5[51].

Results and Discussion Results showed what has been classified as 'good' performance at only 200Hz (see Figure 3.17). Raising the message frequency to 400Hz (and higher) introduced a significant drop in performance - from an average message latency of 23.1ms at 200Hz to 726.4ms at 400Hz, and peaking with a latency of 1721.4ms at 2000Hz (see Figure 3.18).

This leads to the conclusion that *WiFi is a suitable choice of communication medium, as long as suitable low message frequency is chosen so as to ensure consistent performance for the message size being used.* There was notable difference in performance at 200Hz from ethernet (23.1ms for WiFi, compared to 4.3ms for ethernet), however as the difference was consistent throughout the message stream (see Figure 3.17) this difference can be taken in to account for future experiments.

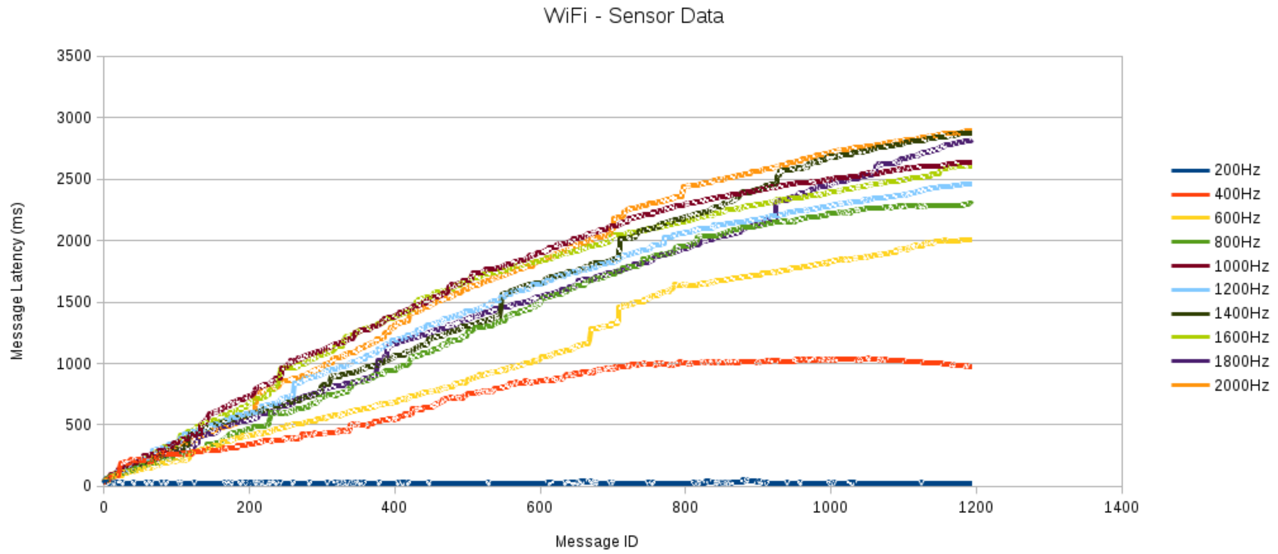

Figure 3.17: Experiment 6 - Sensor Data WiFi, All Frequencies

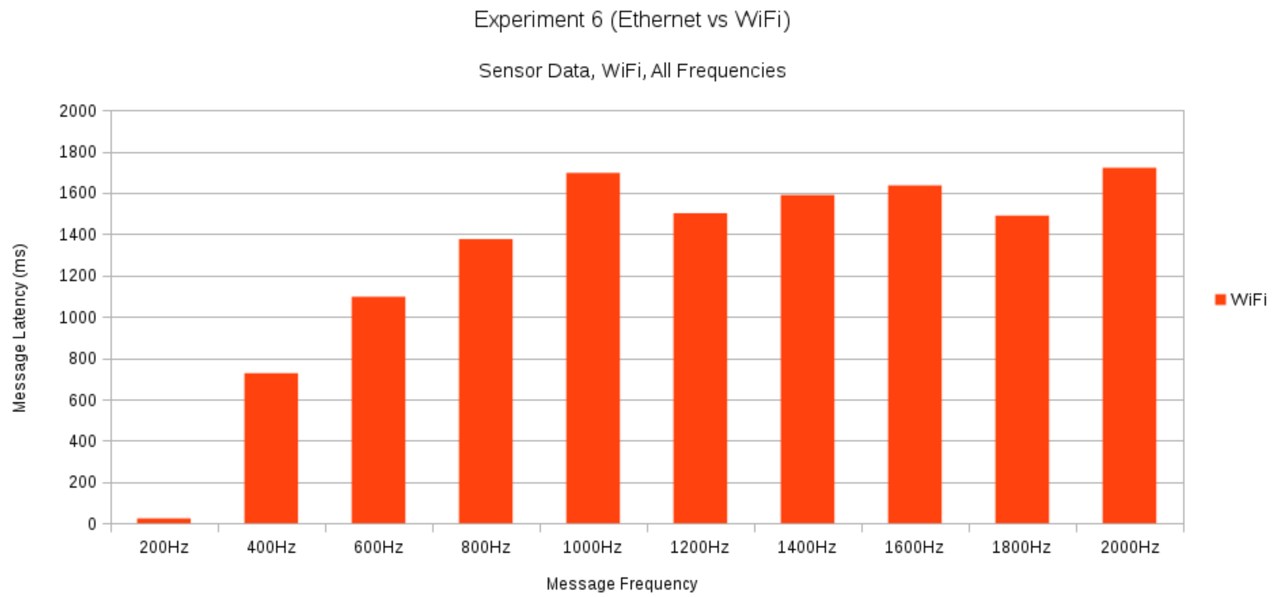

Figure 3.18: Experiment 6 - Sensor Data WiFi, All Frequencies

Further Investigation After the results shown in Figure 3.18 demonstrated a significant difference between message frequencies of 200Hz and 400Hz, it was decided that further runs should be conducted between these frequencies to investigate the point at which performance degrades. The extra runs conducted started at 100Hz (to gain information about performance below 200Hz), and increased in 50Hz increments up to 600Hz. The main areas of interest were 250Hz, 300Hz, and 350Hz.

As Figure 3.19 shows, message stream performance was similar as seen in Figure 3.17. Certain streams exhibited consistent low latency throughout their streams, but above a certain frequency (350Hz and up in this case) performance begins to degrade. Figure 3.20 more clearly shows the jump in average message latency between 300Hz and 350Hz. This leads to the conclusion that *for the sensor data message size (around 108kB),* *a suitable maximum frequency for publishing to ROS topics would be 300Hz for this message size and hardware platform.*

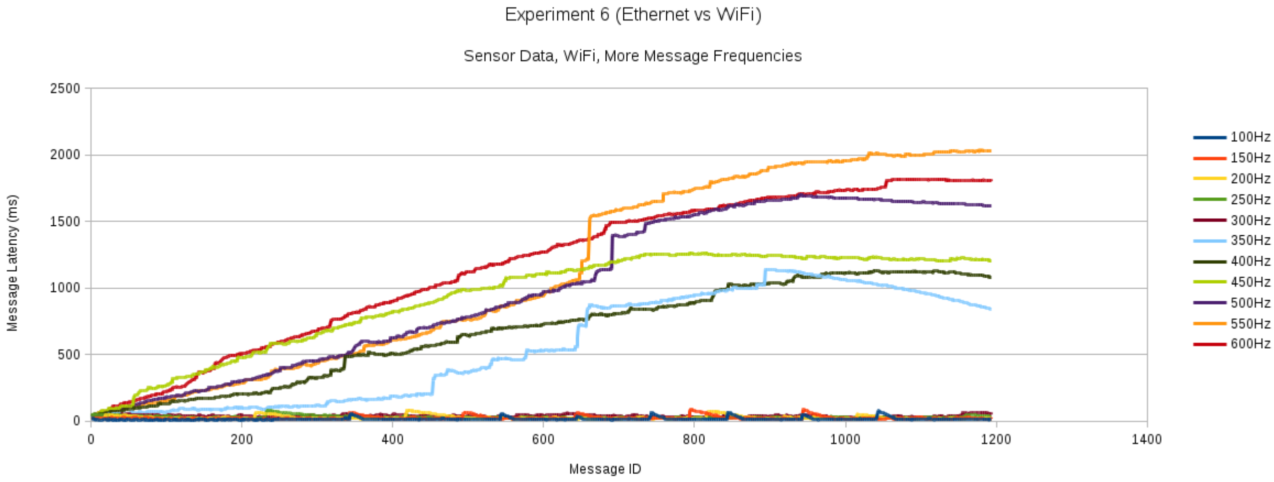

Figure 3.19: Experiment 6 - Sensor Data WiFi, More Frequencies

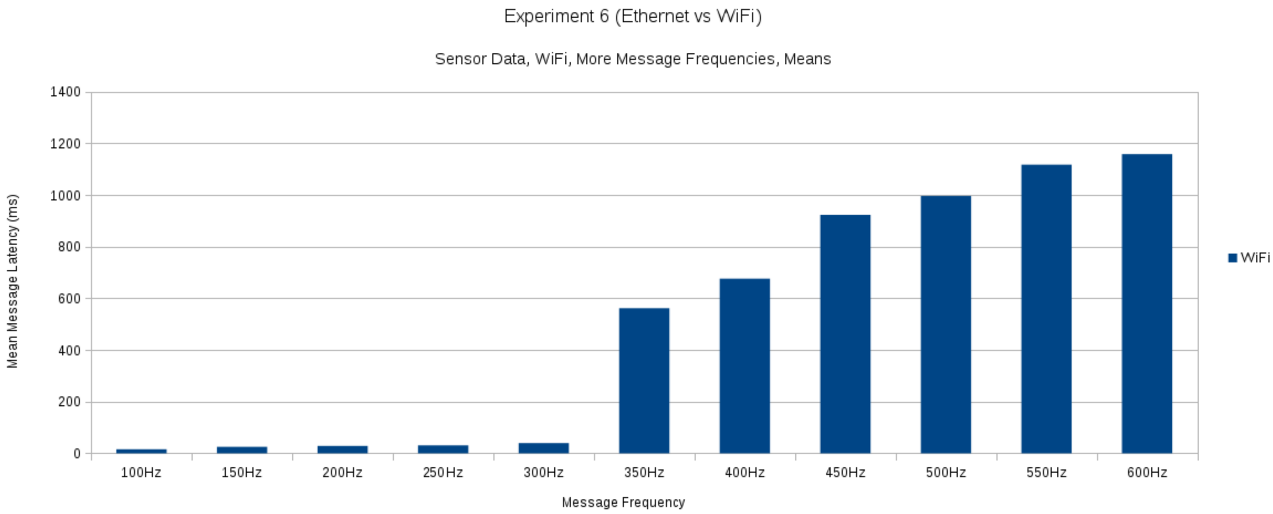

Figure 3.20: Experiment 6 - Sensor Data WiFi, More Frequencies

Chapter 4

Distributed Scalability with ROS

Chapter 3 provided grounding work to build further experiments upon, with an understanding gained of ROS' communication performance at a variety of message frequencies, and with a variety of hardware set-ups such as reduced processing power and Wi-Fi connections.

Chapter 4 now aims to investigate how well a multi-robot system can scale with ROS. As mentioned in Section 2.3, scalability can refer to many different concepts - in this chapter scalabiltiy will be considered in terms of vertical and horizontal scaling. Vertical scaling refers to running multiple identical processes on the same host machines. This has the opportunity to allow for more CPU cores to be utilised (in the case of a multi-core host), however can introduce issues if other system resources are a bottleneck (e.g. if there is not enough RAM to support the new processes, or the new processes are storage-heavy tasks causing resource contention). Horizontal scaling refers to distributing workloads across increasing numbers of host machines. This has the benefit that each host will generally have an entire set of resources (CPU, RAM, disk, GPU, etc) to itself, circumventing the issue of resource contention seen in vertical scaling. There are significant downsides to horizontal scaling though. Horizontal scaling introduces a need for complex communication between the hosts to organise and distribute tasks, or to communicate results. Horizontal scaling also has an economic impact, as it requires new hardware to be purchased and managed.

Table 4.1: Chapter 4 Experiment Overview

Experiment	**Description**	**Section**
Vertical Scaling	Evaluating the ability of ROS to scale communication-heavy applications vertically (more nodes on the same hosts)	4.1
Vertical Scaling On Car Platform	Confirming the results of the previous experiment are applicable to the robot car kit platform	4.2
Horizontal Scaling On Car Platform	A proposal to evaluate the ability of ROS to scale communication-heavy applications horizontally (adding more hosts)	4.3

4.1 Experiment 7 - Vertical Scaling

Introduction As mentioned at the start of the chapter, prior experiments have been setting the groundwork for these scaling experiments. The experiments in Chapter 3 provided understanding of how a communication intensive ROS application behaves in the simple case (2 hosts, 1 node per host), and how switching to a wireless network affects the performance. This experiment now aims to understand how adding more communication nodes will affect individual message latency, and total system throughput in terms of messages sent.

Objective This experiment will involve adding more nodes to both sender, and echoer. Each sender node will send messages to 1 echoer node, who will echo the message back to that same sender node, as was done in experiments in Chapter 3. This allows the sender to record both sent and receive times consistently.

Hypothesis Expectations are for Wi-Fi network performance to bottleneck the performance of communicating nodes. It is expected that the maximum combined frequency (all sending frequencies summed) will be similar to the maximum good performing frequency in a single node scenario. For example, if a single node could sustain 400Hz, then two nodes sending at 200Hz each might see similar performance. As the number of nodes gets higher, it is expected to see the same kind of performance degradation as seen in the high message frequency single node scenario.

Materials and Methodology Increasing numbers of nodes will be added to each host, and message times recorded. The number of hosts will be a constant (two) as this experiment investigates the impact of increased ROS node counts only. The data used will be the sensor data stream used in experiments 5 and 6, in Sections 3.2.1 and 3.2.1.

The implementation of this experiment is considerably more complex than Experiments 1 to 6, as this experiment requires the coordinated starting and stopping of numerous ROS nodes on multiple hosts. For example when there are 64 nodes on the sender, there also need to be 64 nodes on the echoer. There exists a ROS software package to control the starting and stopping of (possibly remote) ROS nodes, called 'roslaunch'. However, roslaunch is not designed for programmatic control (such as via a script), it is designed to be used via configuration files and thus does not expose a public API. Due to the large number of configurations to be tested in this experiment, manual creation of configuration files is not an option. Therefore, a combination of the developer documentation and source code for roslaunch was used to reverse-engineer the creation of a Python script that would directly use the roslaunch internal API to launch and monitor the correct number of ROS nodes on both the sender and echoer hosts[52]. This was in combination with code similar to that which was used in Experiments 5 and 6 to create the source code used in this experiment[53].

Results and Discussion Figure 4.1 displays the averaged (across 3 runs) data for the 100Hz message frequency. This figure shows that performance degrades very rapidly for high node counts. At 100Hz using 8 total nodes (4 senders) results in poor performance with a message latency of 5 seconds by the end of the stream.

Figure 4.2 shows only the 'good' performing node counts (1 and 2 senders) for a message frequency of 100Hz. There are apparent spikes in the message latency at regular intervals. One hypothesis for these spikes is that the senders are buffering messages, perhaps to allow other system processes to use the network interface. However, these spikes do not significantly affect the wider experiment, since they are only short lived spikes.

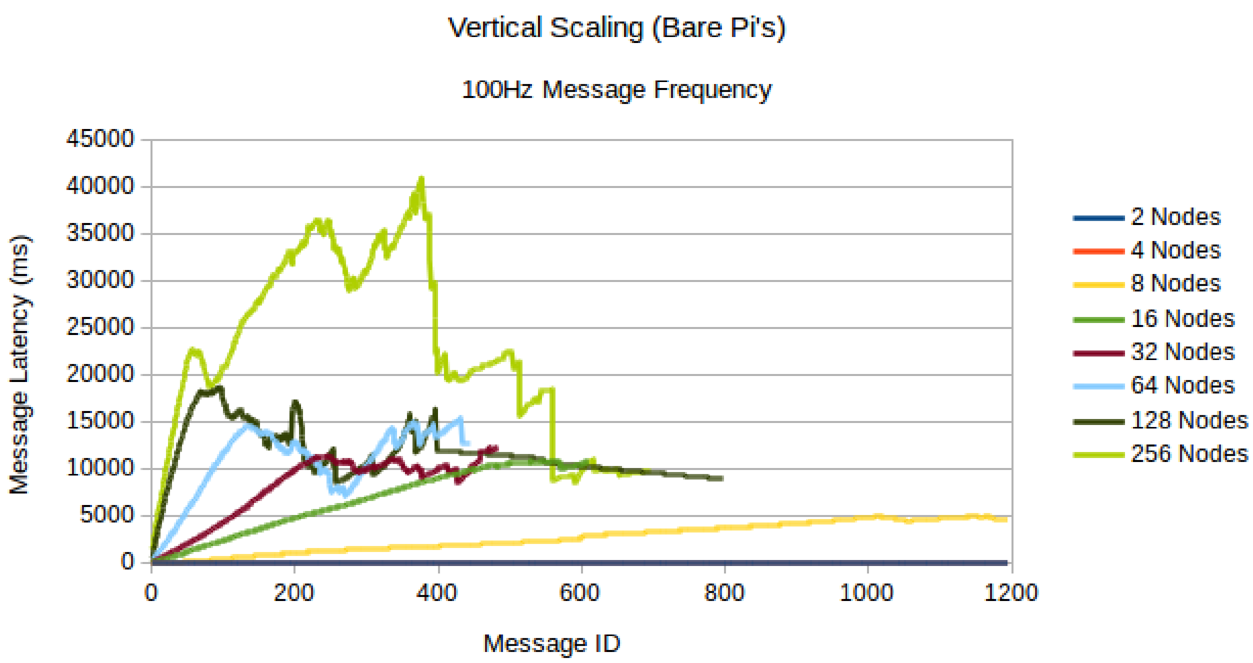

Figure 4.1: Experiment 7 - 100Hz Message Frequency, All Node Counts

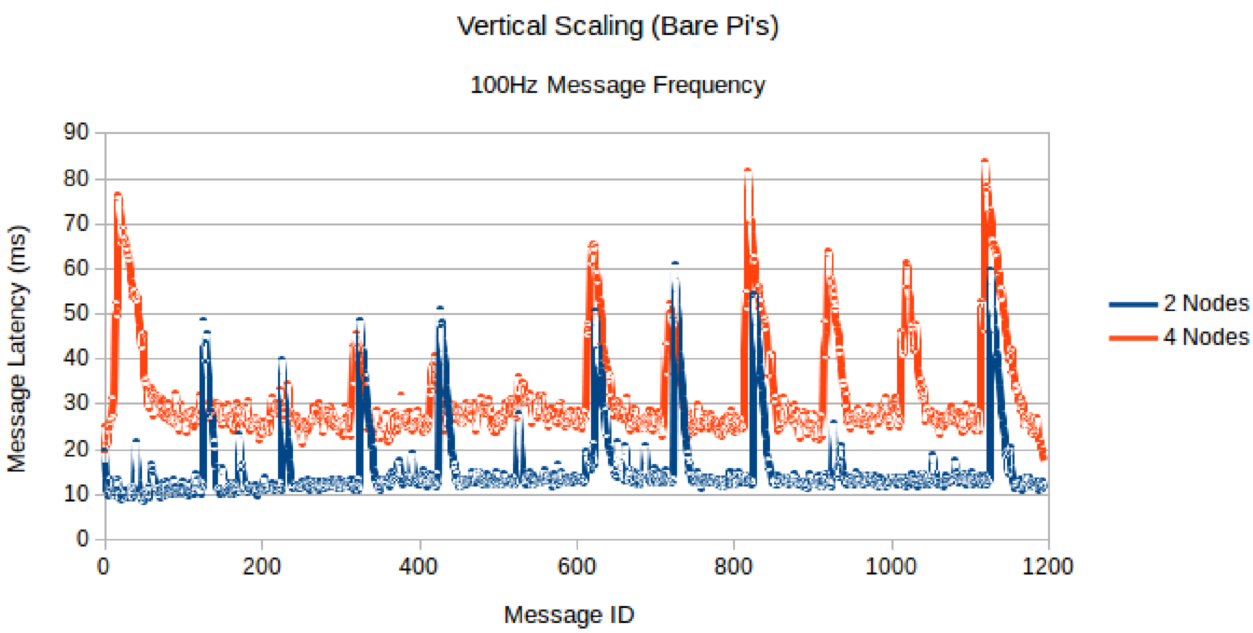

Figure 4.2: Experiment 7 - 100Hz Message Frequency, Low Node Counts

Figure 4.3 concerns a message frequency of 200Hz, and all node counts. This shows a similar pattern as Figure 4.1, with a couple notable differences.

Perhaps most interesting is the behaviour of the 256 node (128 sender) case. As expected, performance is significantly worse at the start, peaking at a message latency of 50 seconds, however after this peak the performance begins to recover - finishing at similar performance levels as the 4 node (2 sender) case. The theorised cause behind this behaviour is that at a very high node count (256) the host system locks up and ROS nodes begin dying, as the master node can not get a health-check response. In this setup, there is no restarting-mechanism for the nodes so they are not restarted. Thus, if enough nodes die, the rest can continue functioning. This, however, is not a useful behaviour in practise, as the system must halt for a long-enough time for the ROS Master's SSH connections to time out.

Another notable difference in the 200Hz case is that the 4 node (2 sender) case begins exhibiting degraded performance. This harkens back to results of Experiment 6 in Section 3.2.2, where a maximum frequency of greater than 300Hz began to show unsustainable performance.

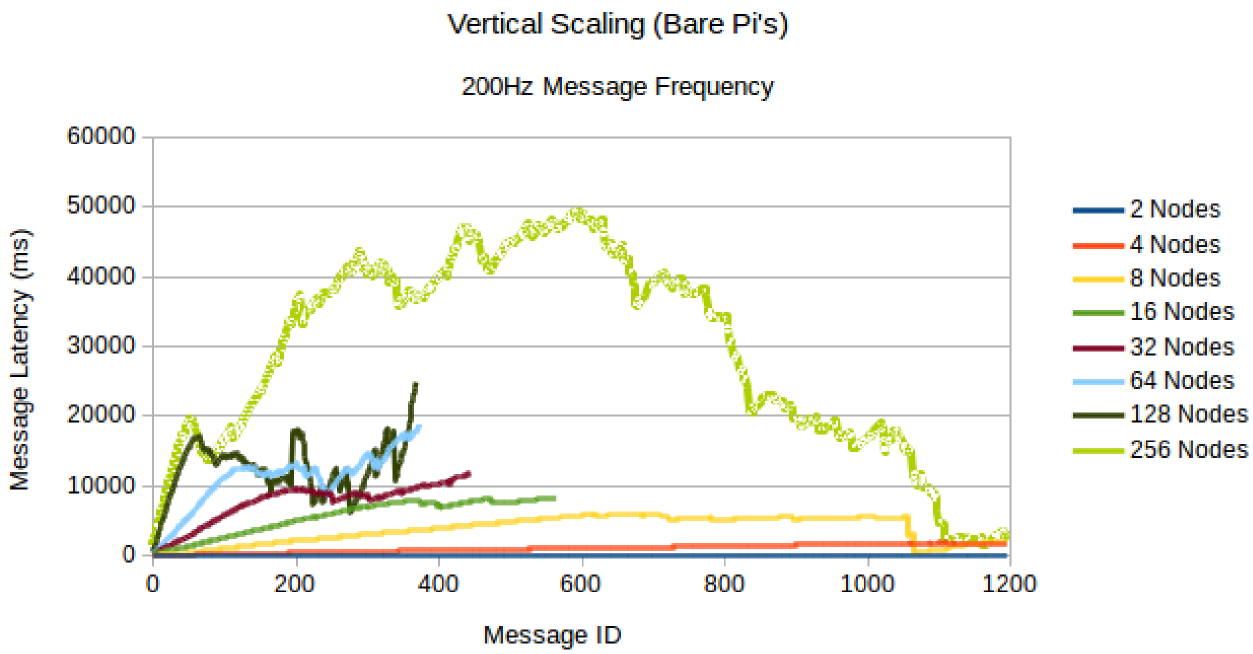

Figure 4.3: Experiment 7 - 200Hz Message Frequency, All Node Counts

Both of these observations hold true for the 300Hz message frequency scenario, with more pronounced effects in both cases. From this data, we can see the system achieves good, sustainable performance with the following set-ups: 100Hz * 1 (sending) node, 100Hz * 2 nodes, 200Hz * 1 node, and 300Hz * 1 node.

From these observations we can conclude that the results of Section 3.2.2 hold true in a more general sense, these results show that each host appears to have a communication scaling limit. This can be represented as the product of three quantites: the number of sending nodes on the host (N), the message frequency (f_m), and the message size (S_m) - giving the following equation for a Communication Scaling Limit Volume (CSLV):

$$CSLV = N \cdot f_m \cdot S_m$$

Using this equation, we can calculate the CSLV for our particular platform (Raspberry Pi 3 Model B, running ROS Kinetic) using experimental data - with the data acquired in this experiment we estimate the CSLV of this platform to be approximately 300. Now, with an estimated CSLV value for the platform it is possible to theoretically evaluate the performance of a new configuration, without requiring experimentation. This allows for questions such as “Given a message size of X bytes, and a desired sending frequency of Y, what's the maximum number of nodes this host can support?” to be answered. In order to demonstrate the robustness of this hypothesis, further experiments - involving a variety of message sizes (S_m), frequencies (f_m), and node counts (N) - however, a detailed investigation is outside the scope of this project.

4.1.1 Further Investigation

In order to get a clearer idea of how adding more ROS nodes affects the system, it is clear that we must use a lower message frequency which allows for sustainable performance at higher node counts. Thus the set-up presented in Section 4.1 was repeated, but with lower message frequencies. If the CSLV hypothesis were to hold true, we should be able to calculate which node counts will begin to show issues at each message frequency.

In this experiment, message frequencies of 1Hz, 10Hz, and 20Hz were used. Thus, at 1Hz we can expect greater than approximately (300 / 1Hz) 300 nodes to cause issues (we can ignore message size as we are using the same message size as previously). Respectively, we can expect the limit to lie around 30 nodes at 10Hz, and 15 nodes at 20Hz.

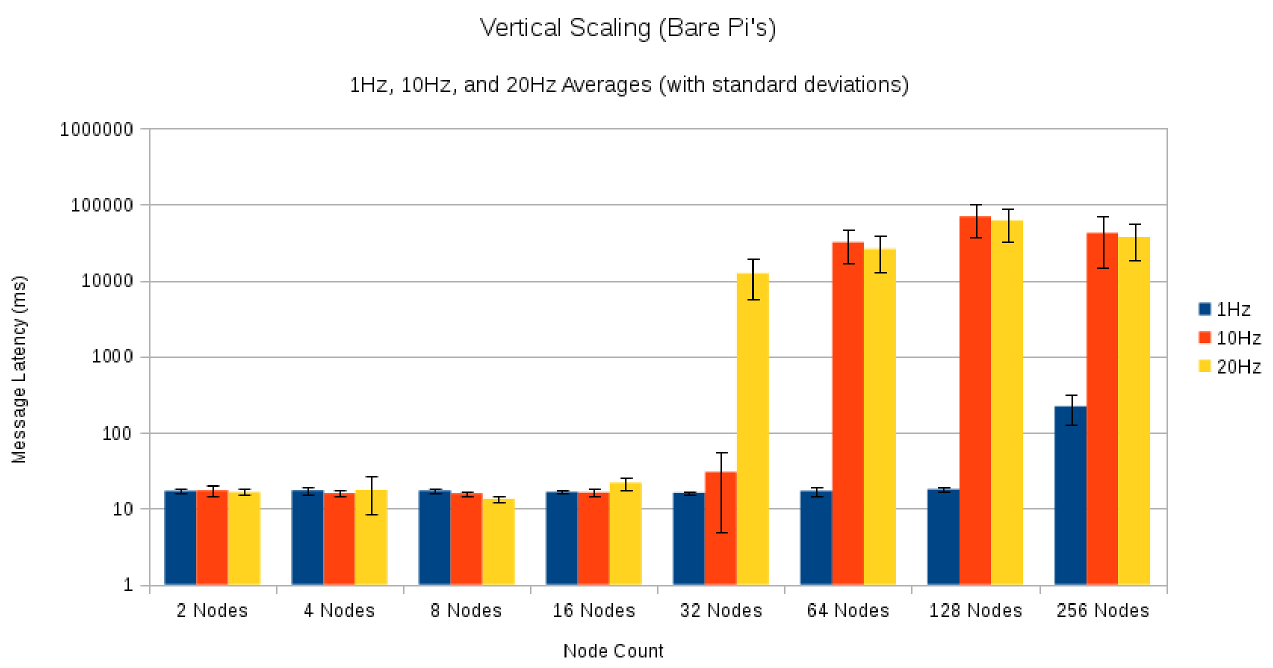

Figure 4.4: Experiment 7 - Logarithmic Average Message Latency (with Standard Deviation), All Frequencies, All Node Counts

As Figure 4.4 shows (note the logarithmic latency scale), *the expected supported node calculations were approximately correct, but somewhat underestimated the actual limits.* This under-estimation is expected, as the resolution of Experiment 6 was only 50Hz (300Hz was acceptable, 350Hz had poor performance), thus the true maximum is somewhere between 300Hz and 350Hz. The figure shows that performance was acceptable at $\leq$ 16 nodes at 20Hz, $\leq$ 32 nodes at 10Hz, and $\leq$ 256 nodes at 1Hz.

A data series of interest is that of 256 nodes at 1Hz message frequency. We can see in Figure 4.5 the performance at this configuration is significantly worse than that at 128 nodes and less, but the performance was not as catastrophic as seen at all other frequencies (1Hz mean is 221ms, vs 10Hz mean of 41,895ms). Performance at this level was also somewhat consistent throughout the message stream, with most messages having 200ms RTT (Round-Trip Time) - again, unlike other message frequencies at 256 nodes.This indicates that *a different bottleneck is possibly coming in to play.* For example, perhaps the time spent switching processes contexts (between 256 different Python processes) is causing a consistent delay. This reduced performance at 256 nodes was unexpected, according to the CSLV equation. Further investigation would be required to identify the exact cause behind this new bottleneck, however that investigation is beyond the scope of this project.

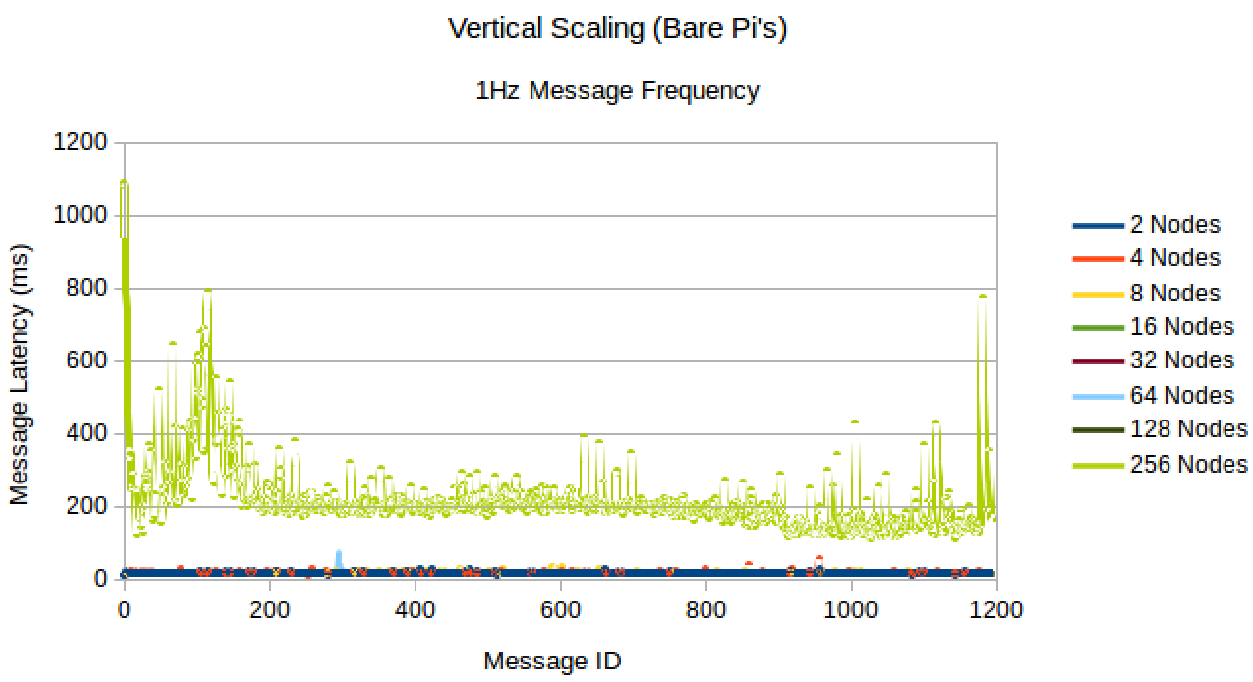

Figure 4.5: Experiment 7 - 1Hz Message Frequency, All Node Counts

4.2 Experiment 8 - Vertical Scaling On Car Platform

Introduction Prior experiments have been completed using what was called 'bare' Raspberry Pi hosts. This refers to the fact that they are off-the-shelf Raspberry Pis with no peripherals or modifications, using standard mains power. However, the test platform described in Section 2.6 represents a more realistic scenario, utilising a more unreliable power source (a battery), and several peripherals (such as a video camera). These differences in the host configuration are not expected to affect performance of the vertical scaling limits in the host (since the car platform utilises the same model of Raspberry Pi) - however, this must be experimentally confirmed.

Objective This experiment aims to verify the results of experiment 7 (Section 4.1) on a real robot platform - namely the robot kit cars described in Section 2.6.

Hypothesis The expected outcome of the experiment is that the conclusions of Section 2.6 will hold true, as those prior experiments were run a test platform that is fundamentally similar to the robot car platform. If a performance drop were to be seen, it is expected that this would be due to the different power supply mechanisms between the bare Raspberry Pis and the robot car kit platform (mains power and batteries, respectively).

Materials and Methodology As this experiment is effectively a re-run of Experiment 7 (as presented in Section 4.1), this experiment has the same methodology as Experiment 7, but conducted on the robot car platform. This presents some new challenges in the execution of the experiment. Due to the battery power supply, the entire experiment (3 runs at 3 message frequencies with 8 different node counts) can not be run in a single battery lifetime. A single charge lasts long enough for 1.5 runs, however in order to reduce likelihood of inconsistent data it was decided to conduct the experiment 1 run at a time, with a battery charge conducted between runs - thus the experiment had to be executed over a number of days.

Results and Discussion Figure 4.6 shows a similar graph as Figure 4.4, but with data acquired from 3 runs on the robot car kit platform. It shows that the results gathered in Experiment 7 were valid on the robot car kit platform, as we see spikes in the message latency at the same node counts as in the previous experiment on the bare Raspberry Pis. Note that 128 node, and 256 node counts were skipped for 10Hz, and 64, 128, and 256 were skipped for 20Hz due to the extremely long time it takes those setups to conclude their runs (on the scale of several hours) - and the robot cars would run out of battery before all sending/receiving nodes can finish. However, the node counts that have been run demonstrate that the performance bottlenecks occur at the same node counts.

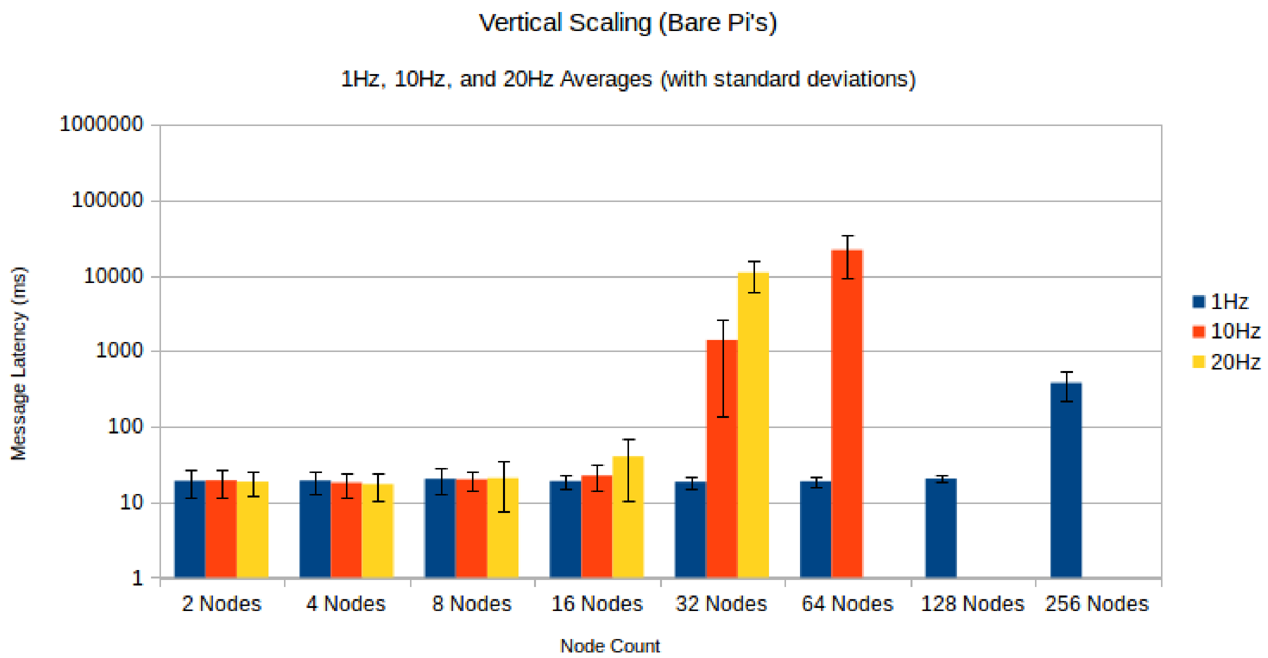

Figure 4.6: Experiment 8 - Logarithmic Average Message Latency (with Standard Deviation), All Frequencies, All Node Counts

Therefore we can conclude that *CSLV as presented in Section 4.1 is a concern for real ROS systems, and that for this specific set-up (utilising Raspberry Pi 3 Model Bs on WiFi) the limit is approximately between (4.25 * 10 * 16 =) 680kB/s and (4.25 * 20 * 16 =) 1360kB/s depending on exact message latency tolerance.*

4.3 Experiment 9 - Horizontal Scaling On Car Platform

4.3.1 Proposal

Up until this point experiments have utilised the simplest multi-robot system (a 2-host system). Using this system, experiments have identified the scaling limits achievable when vertically scaling (adding more nodes) a 2-host system with ROS, however it remains to be seen how ROS will deal with horizontal scaling. This brings extra challenges for each host, as it may be required to write to numerous transport buffers (one for each subscriber). This might be a very minimal overhead, thus allowing ROS systems to scale very well horizontally (with the implication that the new nodes would be well connected) - however, it may be the case that this buffer overhead is significant. In this case, scaling nodes horizontally which are already at their vertical limit may just further decrease the performance of each host, resulting in poor horizontal scaling potential for communication-intensive ROS applications.

In order to experimentally evaluate the horizontal scaling performance of ROS the following experiment is proposed. For N hosts in the range (2, 4, 6, 8) assign 50% to be ‘sending/receiving’ hosts, and 50% to be ‘echoing’ hosts. Each ‘sending/receiving’ host would contain one or more ROS nodes which sends messages to EVERY echoer node, each echoer node echoes the message back to the ‘sending/receiving’ node it received the message from (allowing for a Round-Trip Time (RTT) calculation to be made). This is a well-connected network graph, and would represent an almost worst-case scenario for horizontal scaling as the number of connections increases on the order of 2^{N-1}. For thoroughness, running the previously described set up with multiple nodes per host (M) should be conducted, as well as at several message frequencies. With M nodes per host, this would result in M^{N-1} connections between the two host partitions (‘sending/receiving’ and ‘echoing’).

With this experiment, if the average RTT per message across the system increases as N increases, then this may indicate that ROS does not scale horizontally well with a communication-intensive user application. Due to time constraints with this project, Experiment 9 was not executed - as it would require significant implementation work, as well as time to run the various experiment set ups required.

Chapter 5

Conclusion

5.1 Summary

This project began with a comprehensive review of a large range of robotic middlewares in Section 2.4. This review covered a wide range of current middlewares, rarely seen in the literature of the field. An overview of the communication and computation architecture in ROS was also presented in Sections 2.4.2 through 2.4.5.

The next phase of the project investigated what factors had an impact on the performance of high frequency communication between ROS nodes in a multi-robot system. Through a series of experiments in Chapter 3, it was found the processing power (Sections 3.1.3, and 3.2.1) and connection type (Sections 3.1.4, and 3.2.2) of the ROS hosts were the largest factors when trying to attain reliable communication at a high message frequency. These results were verified by sending message payloads containing data of a variety of sizes (11 bytes up to 308kB), including data recorded from a real ROS system (the PR2 robot).

Finally, research was conducted in Chapter 4 into how ROS scales vertically and horizontally. Vertical scaling tests resulted in the proposal of a mechanism to calculate the maximum supported combination of the number of nodes per host, the message frequency from each host, and the message size (Section 4.1). This proposed formula was called the Communication Scaling Limit Volume (CSLV). The CSLV value for the test platform was estimated using experimental data, used to estimate the maximum number of nodes at a new range of message frequencies, and then verified experimentally at that new range. These new frequencies performed as expected, confirming the accuracy of the CSLV calculation - except in the 256 node/1Hz frequency case which unexpectedly had reduced performance. An explanation for this result was proposed in Section 4.1.1. An experimental set-up was proposed to evaluate the horizontal scaling capabilities of ROS, however the experimented was not conducted due to time constraints in the project.

Finally, research was conducted in Chapter 4 into how ROS scales vertically and horizontally. Section 4.1 proposed the hypothesis that there exists a calculable Communication Scaling Limit Volume (CSLV) which could be used to predict the performance of a particular ROS node configuration on a given host. In order to calculate an initial CSLV value for the Raspberry Pi 3 Model B platform, several node counts were tested at 100Hz, 200Hz, and 300Hz and the configurations that caused low performance were noted. Using this information to calculate an approximate CSLV value would then allow performance predictions to be made on new (untested) configurations. This was tested by calculating the maximum number of nodes that could be sustainably run for a set of new message frequencies: 1Hz, 10Hz, and 20Hz (Figure 4.4). Section 4.1.1 tested these predictions, and found the results generally agreed with the predictions - establishing the CSLV hypothesis.

5.2 Future Work

While conducting the research in this project a number of investigatory avenues were proposed which could not be explored due to time constraints:

- **Conduct the horizontal scaling experiment** Section 4.3 proposes an experimental set-up to evaluate the ability of ROS to sustain many hosts all communicating at high message frequencies. This experiment was not conducted due to time constraints, however it would be required to paint a complete picture of the scalability of ROS.
- **Investigate the effect of other network topologies** The vertical scaling experiments covered in Sections 4.1 and 4.2 utilise a pairwise network topology (each sender node is paired with a single echoer node). It was planned to experiment with a fully-interconnected network (each sender node sends to every echoer node), however this was not conducted due to time constraints.
- **Investigate primary cause of performance discrepency between Ethernet and Wi-Fi** Using real sensor data (a message size of 4.25kB), Section 3.2.2 saw latencies of only 5ms at 2KHz using Ethernet, and latencies of almost 1800ms with WiFi at the same frequency. It is possible that this discrepency could be mitigated (or further understood) by investigating what aspects of Wi-Fi is causing higher latencies (e.g. high numbers of packets being dropped, or low bandwidth).
- **Optimisations to reduce message-sending overhead** Sections 3.1.3 and 3.2.1 found that high frequency message transmission was limited by the CPU power of the test platform (Raspberry Pi 3 Model B). It is possible that this bottleneck could be reduced by investigating what function calls are consuming the majority of CPU time while sending at a high frequency. If these CPU-heavy function calls could be reduced in call frequency, or time-per-call then it is possible higher message frequencies could be sustained. One method of investigating this would be to replace ROS' communication libraries with higher performance libraries (for example, written in a faster language than Python).
- **Comprehensive investigation of the proposed Communication Scaling Limit Volume (CSLV)** Section 4.1 proposed the CSLV, a formula which proposes there is an constant upper limit to the product of the number of nodes on the host, the message frequency of those nodes, and the message size, for a given system. A detailed investigation to validate the accuracy of this model would require a experiment which varies all three variables - however this was out of the scope of this project.
- **Investigate new bottleneck seen at high node counts with low message frequency** Section 4.1 saw that message latencies were unexpectedly high with 128 senders on a single node, each sending at frequency of 1Hz. This result was contrary to the prediction provided by the CSLV calculation (which estimated good performance up to approximately 300 nodes). This result could indicate that a new factor is bottlenecking the communication performance of the system at high node counts. It was suggested that it could be the effect of the Raspberry Pis processor context switching between the 128 different processes, however the investigation to confirm this was outside the scope of the project.

Appendices

Appendix A

Continued Middlewares Overview

Name	Objective	Support	Capabilities	Supported Languages
KERL (Kent Erlang Robotic Library) [4]	• Created as a practical way of teaching Erlang. Simple API designed to let students learn Erlang, rather than learn KERL[43]. • Builds upon Player and Stage as a platform. Providing a simplified Erlang interface on the top[43]. • Contains simple single robot API for initial learning, and multi-robot APIs for advanced uses[43]. • Contains APIs for common tasks such as leader election, and broadcasting data to groups of processes[43].	Open Source, but not widely used. Development has been halted. No commits/updates since 2009 [5].	• Has provided a good starting point for robotics in Erlang[58]. • No full evaluation of suitability for production uses exists, however it is primarily an Erlang wrapper around Player (which is known to be suitable for production), thus may has a solid foundation to build upon.	Erlang, and C

YARP (Yet Another Robot Platform) [28]	• Supports collection of programs communicating P2P • Extensible family of connection types (tcp, udp, multicast, local, MPI, XMLRPC,) • Flexible interfacing with hardware devices • Goal to increase the longevity of robot software projects	Active, open source effort	• Data carrier method seems more flexible than ROS • General network set up seems similar to ROS. Many processes across one or more machines communicating P2P using Observer design pattern. [3] • Supports more operating systems than ROS	SWIG (binding auto-generator)
Orocos (Open RObot Control Software) [14]	• Component based system design • Multi vendor (doesnt aim to solve every problem, but facilitate use of many projects) • Focus (aims to be the best free software framework for realtime control of robots and machine tools, nothing more, nothing less)	Non-active. Still in use, but by very few people (judging by forum activity, and documentation errors (listed source host has gone down). No ‘news’ since 2013.	• Provides toolchain to create realtime robotics applications using modular, run-time configurable software components • Provides Kinematics and Dynamics Library for modelling and computation of kinematic chains, their motion specification, and interpolation (basically controlling things like robot arms)	C++
CARMEN (Carnegie Mellon Robot Navigation Toolkit) [1]	• Open source collection of software for mobile robot control • Modular software to provide basic navigation functionalities, such as base and sensor control, logging, obstacle avoidance, localization, path planning, and mapping	Discontinued, no new releases since 2008.	• Uses inter-process communication platform IPC • Centralised parameter server • Only supports a limited number of specific mobile robot bases.	C and Java

Orca [13]	• Open source framework for developing component-based robotic systems • Goal to enable software reuse by defining commonly used interfaces • Aims to be as flexible as possible by not applying architectural constraints	Discontinued, no new releases since 2009	• Provides some interfaces and implementations of commonly used components • Primarily client/server architecture, but allows for creation of distributed systems due to flexibility (not provided out of the box though)	C++, examples in Java, Python, and PHP. Interfaces can be compiled to C++, Java, Python, PHP, C#, Visual Basic, Ruby, and Obj C.
Microsoft Robotics Developer Studio (v4) [6]	• Goal to make creating robotics applications very accessible • Supports visual programming (drag and drop components) • Supports simple Hello Robot to complex applications in mutli-robot scenarios	No release since 2012	• RESTful (Representational state transfer) communication, services oriented runtime • Supports centralised, and decentralised communcation	C#, and Microsoft Visual Programming Language (VPL)
OpenRTM-aist [12]	• Open source platform to develop component oriented robotic systems.	Seems moderately active (last release May 2016). Moderately active community (more popular in Japan)	• Supports communication based on Publisher/Subscriber model • Has a number of tools for robot system development	C++, Python, Java
Miro [7]	• Builds upon other-widely used middlewares (ACE, TAO CORBA, Qt) to provide object-oriented abstractions [64] • Split in to 3 layers: Device, Service, and Framework • Communication achieved using CORBA client/server	Last release was 2014	• Provides same capabilities as CORBA (type-safe and network-transparent interfaces) [64] • Demonstrated capabilities in multirobot environment • Does not have true OS independence (all robots used Linux), but shown that this can be ported to Solaris in 1 day [64]	Any that have CORBA implementations

Xenomai [26]	• Real-time development framework (can be used to create any kind of real-time interface) • Important goals are extensibility, portability, and maintainability • Uses a dual-kernel approach to hard realtime [36]	Sustained, active, open source development [25]	• Poor availability of detailed documentation and a lack of technical support [55] • Runs on top of an OS (most commonly the Linux kernel) • Shown to be suitable for 100% hard real-time applications [33] • No communication abstractions	Preferred C [27] Possible: C++
Urbi [24]	• Urbiscript aims to provide a programming experience tailored towards robotics (parallel, event-based, functional, OO, client/server, distributed) • Consists of defining modules called 'UObject's which are shells around regular components • These UObjects are then naturally supported by urbiscript which allows easier communication and orchestration	Doesnt appear to be widely used, but is open source with many commits	• Interoperable with CORBA, RT-Middleware, openHRP (among others), thus URBI can act as a central platform to integrate other technologies • Brings many useful abstractions over other middlewares such as Player/Stage, Microsoft Robotics Studio, RT-Middleware, and CORBA. • Can move UObjects after compile-time, e.g. compile once and copy result to many hosts	C++, Java Custom urbiscript scripting language for orchestration

Appendix B

Experiment 2 Other Graphs

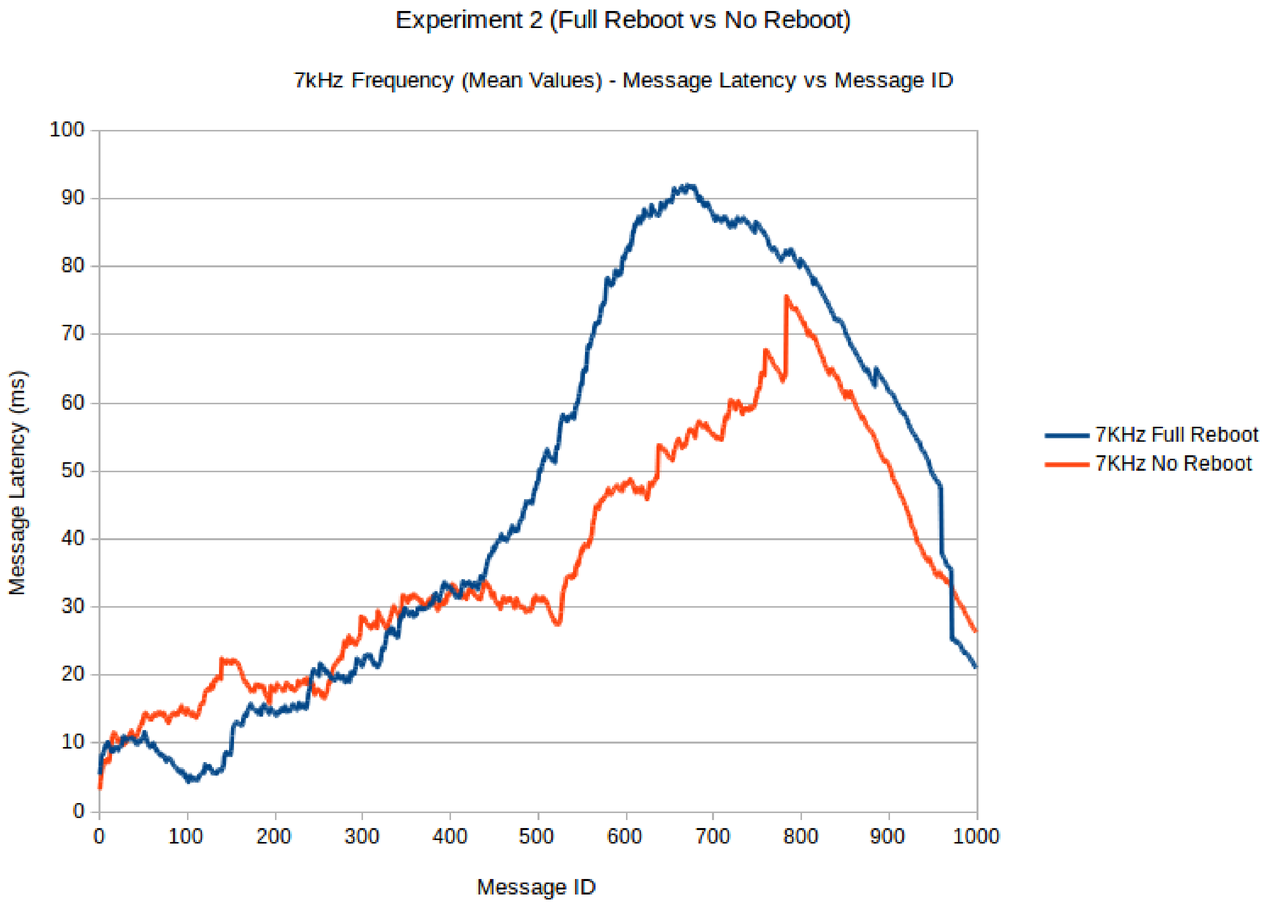

Figure B.1: Experiment 2 - 7KHz Message Frequency

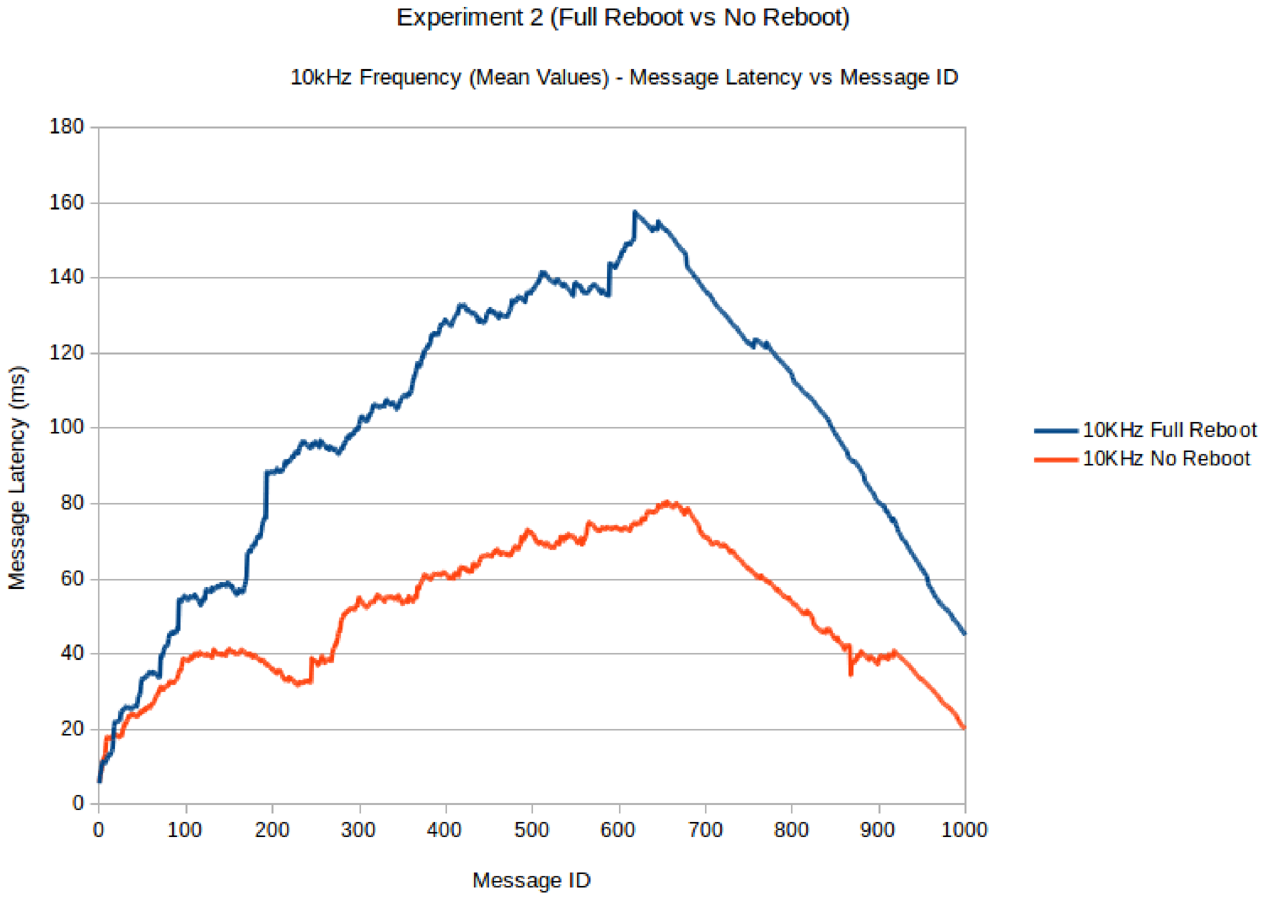

Figure B.2: Experiment 2 - 10KHz Message Frequency

Bibliography

[1] Carmen (Carnegie Mellon Robot Navigation Toolkit) Project Homepage. http://carmen.sourceforge.net/. Accessed: 2017-01-27.

[2] CORBA (Common Object Request Broker Architecture) Project Homepage. http://www.corba.org/. Accessed: 2017-01-27.

[3] Getting Started with YARP Ports. http://www.yarp.it/note_ports.html. Accessed: 2017-01-27.

[4] KERL (Kent Erlang Robotics Library) Project Homepage. http://kerl.sourceforge.net/. Accessed: 2017-01-27.

[5] KERL SVN Repository. https://sourceforge.net/p/kerl/code/HEAD/tree/. Accessed: 2017-01-27.

[6] Microsoft Robotics Developer Studio 4 Project Homepage. https://www.microsoft.com/en-gb/download/details.aspx?id=29081. Accessed: 2017-01-27.

[7] Miro Project Homepage. http://users.isr.ist.utl.pt/~jseq/ResearchAtelier/misc/Miro%20-%20Middleware%20for%20Robots. Accessed: 2017-01-27.

[8] MIT Stata Center Dataset. https://projects.csail.mit.edu/stata/downloads.php. Accessed: 2017-01-27.

[9] MOOS (Mission Oriented Operating Suite) Project Homepage. http://www.robots.ox.ac.uk/~mobile/MOOS/wiki/pmwiki.php. Accessed: 2017-01-27.

[10] OpenRDK Introduction. http://openrdk.sourceforge.net/index.php?n=Main.Introduction. Accessed: 2017-01-27.

[11] OpenRDK Project Homepage. http://openrdk.sourceforge.net/. Accessed: 2017-01-27.

[12] OpenRTM-aist Project Homepage. http://www.openrtm.org/. Accessed: 2017-01-27.

[13] Orca Project Homepage. http://orca-robotics.sourceforge.net/. Accessed: 2017-01-27.

[14] Orocos (Open Robot Control Software) Project Homepage. http://www.orocos.org/. Accessed: 2017-01-27.

[15] Player Project Homepage. http://playerstage.sourceforge.net/. Accessed: 2017-01-27.

[16] Player Server Manual. http://playerstage.sourceforge.net/doc/Player-1.6.5/player-html/. Accessed: 2017-01-27.

[17] ROS About Page. http://www.ros.org/about-ros/. Accessed: 2017-03-03.

[18] ROS Concepts. http://wiki.ros.org/ROS/Concepts. Accessed: 2017-02-05.

[19] ROS (Robot Operating System) Project Homepage. `http://www.ros.org/`. Accessed: 2017-01-27.

[20] ROS2 (Robot Operating System) Project Homepage. `http://design.ros2.org/`. Accessed: 2017-01-27.

[21] ‘roslaunch’ ROS Package. `http://wiki.ros.org/roslaunch`. Accessed: 2017-01-27.

[22] Sunfounder Robot Car Kit. `https://www.sunfounder.com/rpi-car.html`. Accessed: 2017-01-27.

[23] Technical Committee on Multi-Robot Systems (TC MRS) of the IEEE Robotics and Automation Society. `http://multirobotsystems.org/`. Accessed: 2017-01-27.

[24] Urbi Project Homepage. `http://www.gostai.com/products/urbi/`. Accessed: 2017-01-27.

[25] Xenomai Git Repositories. `https://git.xenomai.org/`. Accessed: 2017-01-27.

[26] Xenomai Project Homepage. `https://xenomai.org/`. Accessed: 2017-01-27.

[27] Xenomai Tutorial. `http://www.cs.ru.nl/lab/xenomai/exercises/ex01/Exercise-1.html`. Accessed: 2017-01-27.

[28] YARP (Yet Another Robot Platform) Project Homepage. `http://yarp.it/`. Accessed: 2017-01-27.

[29] Mateusz Macia Adam Dbrowski, Rafa Kozik. Evaluation Of ROS2 Communication Layer. `http://roscon.ros.org/2016/presentations/rafal.kozik-ros2evaluation.pdf`. Accessed: 2017-01-27.

[30] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[31] B. Bauml and G. Hirzinger. Agile Robot Development (aRD): A Pragmatic Approach to Robotic Software. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3741–3748, Oct 2006.

[32] André B. Bondi. Characteristics of Scalability and Their Impact on Performance. In *Proceedings of the 2Nd International Workshop on Software and Performance*, WOSP ’00, pages 195–203, New York, NY, USA, 2000. ACM.

[33] Jeremy H Brown and Brad Martin. How fast is fast enough? Choosing between Xenomai and Linux for real-time applications.

[34] Davide Brugali and Azamat Shakhimardanov. Component-based robotic engineering (part ii). *IEEE Robotics & Automation Magazine*, 17(1):100–112, 2010.

[35] D. Calisi, A. Censi, L. Iocchi, and D. Nardi. OpenRDK: A modular framework for robotic software development. In *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1872–1877, Sept 2008.

[36] Byoung Wook Choi, Dong Gwan Shin, Jeong Ho Park, Soo Yeong Yi, and Seet Gerald. Real-time control architecture using Xenomai for intelligent service robots in USN environments. *Intelligent Service Robotics*, 2(3):139–151, 2009.

[37] Carlos Cordeiro, Dmitry Akhmetov, and Minyoung Park. IEEE 802.11 ad: Introduction and performance evaluation of the first multi-Gbps WiFi technology. In *Proceedings of the 2010 ACM international workshop on mmWave communications: from circuits to networks*, pages 3–8. ACM, 2010.

[38] Nikolaus Correll, Kostas E Bekris, Dmitry Berenson, Oliver Brock, Albert Causo, Kris Hauser, Kei Okada, Alberto Rodriguez, Joseph M Romano, and Peter R Wurman. Lessons from the Amazon picking challenge. *arXiv preprint arXiv:1601.05484*, 2016.

[39] Deborah Estrin, David Culler, Kris Pister, and Gaurav Sukhatme. Connecting the physical world with pervasive networks. *IEEE pervasive computing*, 1(1):62–63, 2002.

[40] Jim Florwick, Jim Whiteaker, Alan Cuellar Amrod, and Jake Woodhams. Wireless LAN design guide for high density client environments in higher education. *Design guide, Cisco*, 2013.

[41] Brain Gerkey. Why ROS 2.0? `http://design.ros2.org/articles/why_ros2.html`. Accessed: 2017-01-27.

[42] Geoff Gordon and CMU Machine Learning. Think Globally, Act Locally.

[43] Sten Grüner and Thomas Lorentsen. Teaching Erlang using robotics and Player/Stage. In *Proceedings of the 8th ACM SIGPLAN workshop on ERLANG*, pages 33–40. ACM, 2009.

[44] Martin Hägele, Klas Nilsson, J Norberto Pires, and Rainer Bischoff. A Short History of Industrial Robotics. In *Springer handbook of robotics*, pages 1386–1392. Springer, 2016.

[45] Martin Hägele, Klas Nilsson, J Norberto Pires, and Rainer Bischoff. Robots at Work. In *Springer handbook of robotics*, pages 1385–1781. Springer, 2016.

[46] Roelof Hamberg and Jacques Verriet. *Automation in warehouse development*. Springer, 2012.

[47] Neil G Hockstein, CG Gourin, RA Faust, and David J Terris. A history of robots: from science fiction to surgical robotics. *Journal of robotic surgery*, 1(2):113–118, 2007.

[48] Isaac Jordan. Experiment 1 - Message Latency Initial Code. `https://github.com/Sheepzez/ros-multirobot-evaluation/tree/master/experiments/message_latency/andreeas%20version`. Accessed: 2017-03-18.

[49] Isaac Jordan. Experiment 1 - Message Latency Revised Code. `https://github.com/Sheepzez/ros-multirobot-evaluation/tree/master/experiments/message_latency/no_echo_delay`. Accessed: 2017-03-18.

[50] Isaac Jordan. Experiment 1 - No Disk Writing Code. `https://github.com/Sheepzez/ros-multirobot-evaluation/tree/master/experiments/message_latency/no_echo_delay_no_disk_write`. Accessed: 2017-03-18.

[51] Isaac Jordan. Experiment 5 - ROSbag Code. `https://github.com/Sheepzez/ros-multirobot-evaluation/tree/master/experiments/message_latency/bag-sender-receiver`. Accessed: 2017-03-18.

[52] Isaac Jordan. Experiment 7 - Node Launching Script. `https://github.com/Sheepzez/ros-multirobot-evaluation/blob/master/experiments/message_latency/vertical_scaling/roslaunch_script.py`. Accessed: 2017-03-18.

[53] Isaac Jordan. Experiment 7 - Vertical Scaling Code. `https://github.com/Sheepzez/ros-multirobot-evaluation/tree/master/experiments/message_latency/vertical_scaling`. Accessed: 2017-03-18.

[54] Eric Klavins. Communication complexity of multi-robot systems. In *Algorithmic Foundations of Robotics V*, pages 275–291. Springer, 2004.

[55] Jae Hwan Koh and Byoung Wook Choi. Real-time performance of real-time mechanisms for RTAI and Xenomai in various running conditions.

[56] Edward A. Lee. Cyber Physical Systems: Design Challenges. Technical Report UCB/EECS-2008-8, EECS Department, University of California, Berkeley, Jan 2008.

[57] Andreea Lutac. Analysing ROS Distribution Capabilities (Blog). `http://www.dcs.gla.ac.uk/research/rosie/blog.html`. Accessed: 2017-03-16.

[58] Andreea Lutac, Natalia Chechina, Gerardo Aragon-Camarasa, and Phil Trinder. Towards reliable and scalable robot communication. In *Proceedings of the 15th International Workshop on Erlang*, pages 12–23. ACM, 2016.

[59] Yuya Maruyama, Shinpei Kato, and Takuya Azumi. Exploring the Performance of ROS2. In *Proceedings of the 13th International Conference on Embedded Software*, EMSOFT '16, pages 5:1–5:10, New York, NY, USA, 2016. ACM.

[60] Paul Newman. Introduction to Programming with MOOS. *Communications,(November)*, pages 1–34, 2009.

[61] Sean E. Parker. Diagram illustrates autogeneration of the infrastructure code from an interface defined using the CORBA IDL. `https://en.wikipedia.org/wiki/File:Orb.gif`. Accessed: 2017-03-05.

[62] Adam Pennington. Diagram illustratrating the major sensors present on Willow Garage's PR2 robot. `https://robotsfuture.blogspot.co.uk/2011_03_01_archive.html`. Accessed: 2017-03-15.

[63] Erol Şahin. Swarm robotics: From sources of inspiration to domains of application. In *International workshop on swarm robotics*, pages 10–20. Springer, 2004.

[64] H. Utz, S. Sablatnog, S. Enderle, and G. Kraetzschmar. Miro - middleware for mobile robot applications. *IEEE Transactions on Robotics and Automation*, 18(4):493–497, Aug 2002.

[65] Richard T Vaughan, Brian P Gerkey, and Andrew Howard. On device abstractions for portable, reusable robot code. In *Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, volume 3, pages 2421–2427. IEEE, 2003.

[66] Peter R Wurman, Raffaello D'Andrea, and Mick Mountz. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI magazine*, 29(1):9, 2008.

[67] Zhi Yan, Nicolas Jouandeau, and Arab Ali Cherif. A Survey and Analysis of Multi-Robot Coordination. *International Journal of Advanced Robotic Systems*, 10(12):399, 2013.