University of Glasgow | School of Computing Science

On The Scalability of ROS

Isaac Jordan

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

Level 4 Project — January 18, 2017

Abstract

Robots, distributed systems, and middleware.

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

Robotics is the field of creating physical systems to automate tasks. As a concept, automation dates back hundreds of years - however the idea of replicating biological systems seen in nature (for example, humans) was not first seen until the 20th century. Robotics was mainly considered a theoretical or fictitious fantasy, however the main discussers of the concept (such as Isaac Asimov) assumed technological capability would inevitably reach the level where robotics would either be inseperable from society, or be banned due to it's impact.

In the 19th century the beginnings of robotics could be seen in the creation of vehicles which could be remotely operated using electric signals. By the early 20th century, wireless radio guidance systems were advanced enough that remotely operated aircraft could be demonstrated in 1917 (by Archibald Low [citation needed]).

Technology advanced so rapidly during the 20th century that by the 1970s the Soviet Union could explore the surface of the moon using a remotely operated vehicle robot.

However, these uses of robotics displayed no 'intelligent behaviour' on behalf of the robotic system. Beginning in the 1980s, the growing field of Artificial Intelligence (A.I.) was revived and began creating systems which could intelligently answer domain-specific questions - called expert systems.

Robotics is a fast-progressing field which has seen major advances the past decades. From obvious examples such as Amazon's item pickers, to more integrated applications such as autonomous cars - the scale of robotics is increasing.

1.2 Aims and Objectives

If I had an aim it would go here.

1.3 Achievements

If I had achieved anything it would go here.

Chapter 2

Background

2.1 Robotics

Robots are the future. (Why robots are important)

Robotics is the field concerning 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, however the physical realisation of these systems did not begin to be practical until the 2000s when increases in computing power, sensor accuracy, research investment, and software algorithms allowed for useful robotics systems to be created.

Today's robotic systems generally consist of many small autonomous systems working together to form a coherent whole. 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 it's task (recording the envionrment), but other subsystems may rely on it's output (such as a computer vision package which needs video frame inputs).

Robotics is part of a field called cyber-physical systems (CPS) that concerns computer systems that interface with the physical world. These can include Internet Of Things (IOT) networks 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 (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.

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[8].

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. One such example is a warehouse management system.

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 are using thousands of individual robots to intelligently move the shelving around and bring the correct items to stationary human pickers. 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.

2.3 Scalability in Robotics

When creating multi-robot systems, scalability becomes an important concern. At what level does a system architect choose to switch from small highly-powered robot systems to a larger number of simpler robot systems. 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.

Swarm robotics is the extreme approach of 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.

2.4 Robotic Middleware

Robot designers want to work at a high level. Sensors are low level. (Why is middleware needed, whats available)

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 defning how each subsystem communicates with others.

Robots 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. These sensor modules are all independently designed, resulting in a wide array of different software and hardware interfaces - meaning that each robot system 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.

Software called middleware is the solution to this problem. A middleware provides a common interface design so that no matter what hardware is creating the data, the results are distributed in a consistent manner. This means that hardware manufacturers (or users) need only implement one software system for each sensor that can them be distributed and reused amongst all users of that sensor, as long as those users are using that middleware. This means that creators of systems utilising that middleware have easy access to the data created by a particular sensor as they know the software the 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 researcher 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 higher 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.

There are 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.

2.4.1 An Overview

Name	**Objective**	**Support**	**Capabilities**	**Supported Languages**
ROS (Robot Operating System)	• 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)	• 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 repeated stop/start cycling of any process	Not widely used (judging by GitHub stars at least) Development is 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)

KERL (Kent Erlang Robotic Library)	• Created as a practical way of teaching Erlang. Simple API designed to let students learn Erlang, rather than learn KERL[17]. • Builds upon Player and Stage as a platform. Providing a simplified Erlang interface on the top[17]. • Contains simple single robot API for initial learning, and multi-robot APIs for advanced uses[17]. • Contains APIs for common tasks such as leader election, and broadcasting data to groups of processes[17].	Open Source, but not widely used. Development has been halted. No commits/updates since 2009 [2].	• Has provided a good starting point of robotics in Erlang[19]. • No full evaluation of suitability for production uses exists, however is mainly an Erlang wrapper around Player, thus likely has a solid foundation.	Erlang, and C
YARP (Yet Another Robot Platform)	• Supports collection of programs communicating P2P • Extensible family of connection types (tcp, udp, multicast, local, MPI, XML, RPC,) • 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. [1] • Supports more operating systems than ROS	SWIG (binding auto-generator)

Orocos (Open RObot Control Software)	• 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)	• 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	• Open source framework for developing component-based robotic systems • Goal to enable software reuse by defining commonly used interfaces	Discontinued, no new releases since 2009	• Provides some interfaces and implementations of commonly used components • No particular distributed network things • Seems to use client/server architecture	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)	• 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	• REST-style, services oriented runtime • Supports centralised, and decentralised communcation	C#, and Microsoft Visual Programming Language (VPL)

OpenRTM-aist	• 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
Player	• 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	• Target new use cases, such as multi-robot systems (providing a standard approach), embedded systems, real-time systems, non-ideal networks, and production environments [16] • Recreate ROS using existing new tech (such as Redis, WebSockets, DDS) • Overhaul of API (create consistent API without ¿ 7 years of backward compatibility)	Pre-release, but active daily development. Unstable but good future prospects given popularity of ROS1	• Improved communication resilience on poor networks utilising DDS [11] [20] • Communication overhead of DDS shown to be non-trivial for local connection. For remote, overhead is trivial but throughput depends on DDS library used [20]	C99, C++11, Python3 Speculative: JavaScript

OpenRDK	• 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 [14]	Open source, no news since 2010, created for a single research group	• Created with an eye on the competition (not a copy of another framework) • Has been used in multiple environments (single rescue robotic system, assistive robots) [14] • Has useful tools such as a graphical tool for remote inspection and management of modules, and also modules for logging and replaying [14] • No real-time support [4]	C++
Miro	• Builds upon other-widely used middlewares (ACE, TAO CORBA, Qt) to provide object-oriented abstractions [22] • 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) [22] • 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	Any that have CORBA implementations
Xenomai	• 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 [15]	Sustained, active, open source development [9]	• Poor availability of detailed documentation and a lack of technical support [18] • Runs on top of an OS (most commonly the Linux kernel) • Shown to be suitable for 100% hard real-time applications [12] • No communication abstractions	Preferred C [10] Possible: C++

CORBA (Common Object Request Broker Architecture)	• 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
Urbi	• 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	C++, Java Custom urbiscript scripting language for orchestration

Middleware designs can be broken down in to four groups of concepts: Communication, Computation, Configuration, and Coordination [13]. 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 above have used a variety of approaches to inter-component communication.

CORBA, and those 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).

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. Other nodes can subscribe to topics, generally registering a callback function which is called when some new data has been published to the topic. 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.

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. In OpenRDK the blackboard is called a repository, and each module can communicate by publishing values (called properties) to the repository. 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 [5]. 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. 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.

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. At deployment time configuration 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 a uses, but can consist of specifying API keys, database connection details, how many threads should be created for each process, which modules should start (and when), which components should interconnect, 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. OpenRDK utilises an XML configuration file to do similar tasks. It also provides some reliability functionality, such as respawning processes that have died [6]. MOOS also utilises a text file for runtime configuration called a ‘Mission file’ [21]. This mission file provides the necessary runtime parameters for components to set themselves up in a system.

ROS includes another way for new agents 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 during set up (and execution). 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 files.

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 them all 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.

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. 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 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 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.

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

The robots used in this project are 9 identical robot cars with front-wheel steering. (What is their set up)

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[7].

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.

Chapter 3

Communication Scalability

The communication experiments look in to the scalability of communication in ROS. How well does it handle having to send high frequency work-loads of varying sizes.

3.1 Scoping Experiments

Scoping experiments were quick experiments using dummy data. Designed to identify useful areas to explore in more detail in realistic experiments later.

3.1.1 Experiment 1 - Revising Existing Code

The first experiment was designed to analyse the transfer time of messages between two machines at varying message frequencies. This would highlight whether there was some limit as to how often ROS could send and receive messages on it's topics.

These two machines were Raspberry Pi 3 Model Bs connected via ethernet to an Asus router.

The messages were sent and received using ROS Kinetic, running on the Raspbian OS.

The experimental setup was that one Raspberry Pi would run a ROS master node, another Raspberry Pi runs a sender program that notes down the message-sent time, and sends it to a 3rd Raspberry Pi which merely echoes the message back to the sender. Upon receiving the message, the sender/receiver notes the current time, and writes ‘message X which was sent at Y, was received at Z’ to a text file. The resulting Round Trip Time (RTT) for each message would therefore be the difference between the message-sent time and the message-received time.

The expected result of the experiment was that message latency (RTT) would be the same across all lower frequencies, until some bottleneck was reached that would then cause message latencies to exponentially increase due to congestion.

Code had been written prior to execute this experiment, so initially this was used B.1. However, this gave results that were contrary to the hypothesis. An increase in message frequency resulted in a reduction in message latency. A number of messages on higher frequencies were also dropped, and never received. As this was the opposite of the hypothesis, the first step was to critique the experiment code.

This review highlighted two major issues, the first was the echoing machine had a delay similar to the sender when the experiment design mandated that the echoer always respond as fast as it can. 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.

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).

The experiment was then repeated using this code, and the results from these runs agreed with the hypothesis.

3.1.2 Experiment 2 - Rebooting

Experiment 2 was an investigation in to whether the performance of ROS messages was affected by previous messages sent on the system. This was tested by comparing the performance of ROS messages of 5 consecutive message passing runs vs 5 message passing runs with system reboots between runs. This was repeated at 1KHz, 4KHz, 7KHz, and 10Khz frequencies for two reasons. The first was to investigate whether areas in which we observed consistent performance before would exhibit any difference between reboot and no reboot. Secondly, it would give further insight in to where the exact barrier between ‘good, consistent performance’ and ‘poor, erratic performance’ is.

Rebooting the systems involved 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.

The result of the experiment was hypothesised to demonstrate no significant difference between rebooting and not-rebooting at any message frequency.

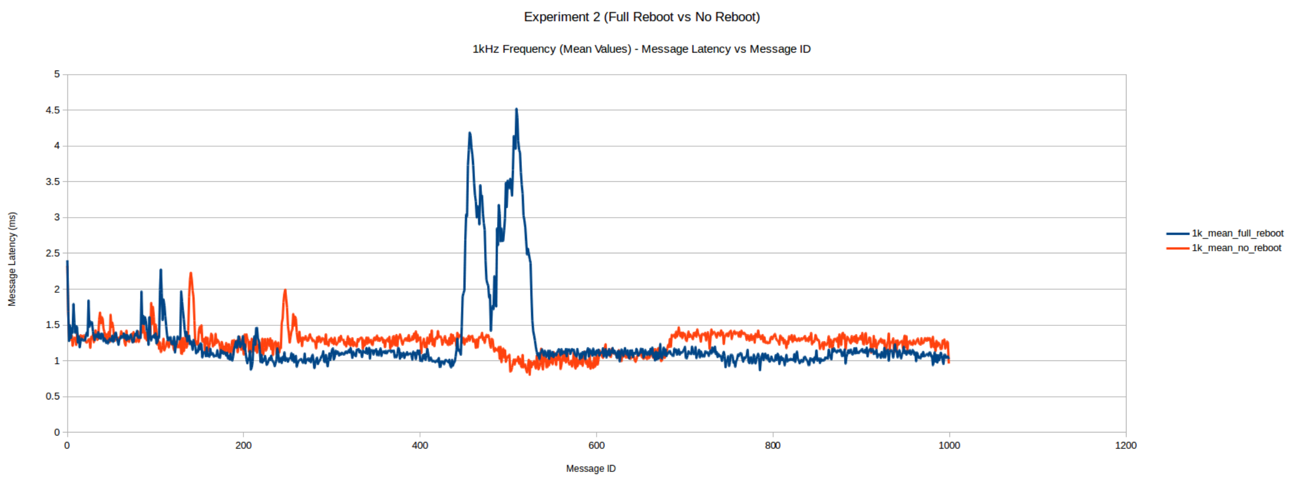

Figure 3.1: Experiment 2 - Mean Message Latency 1KHz Message Frequency

Figure 3.1 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 1 erroneous run at that data point). Figure 3.2 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 A for other mean graphs, and individual run graphs.

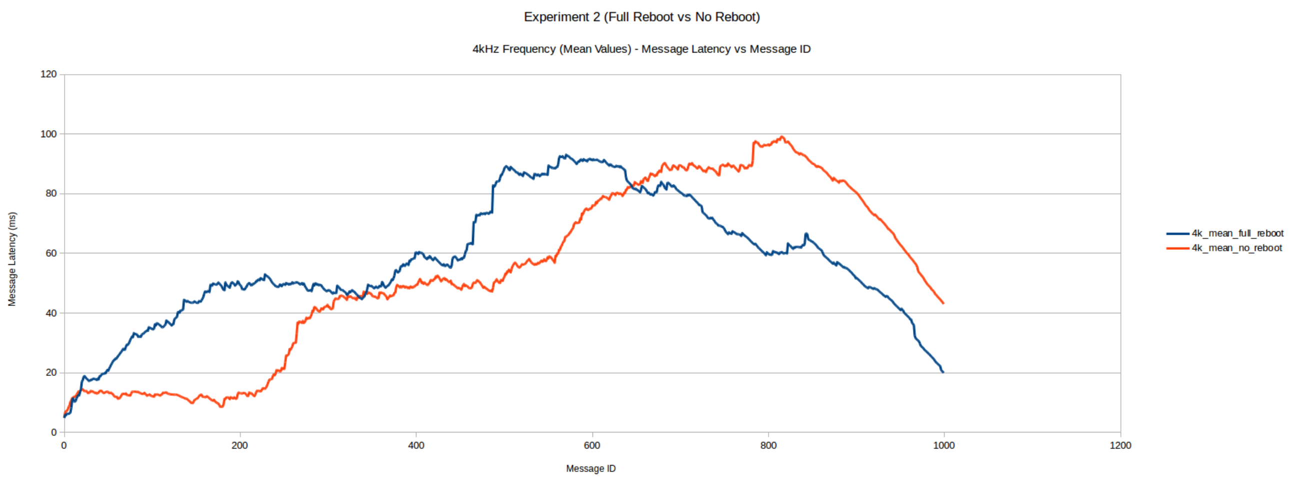

Figure 3.2: Experiment 2 - Mean Message Latency 4KHz Message Frequency

3.1.3 Experiment 3 - CPU Clock Speed

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.

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.

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.

One potential issue is that modern CPUs often 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.

Results

The results agreed with the hypothesis. At 100% CPU clock speed, only the highest 3 frequencies demonstrated sustained degradation of performance, however at 75% the top 5 frequencies showed, and at 50% the top 7 had reduced performance.

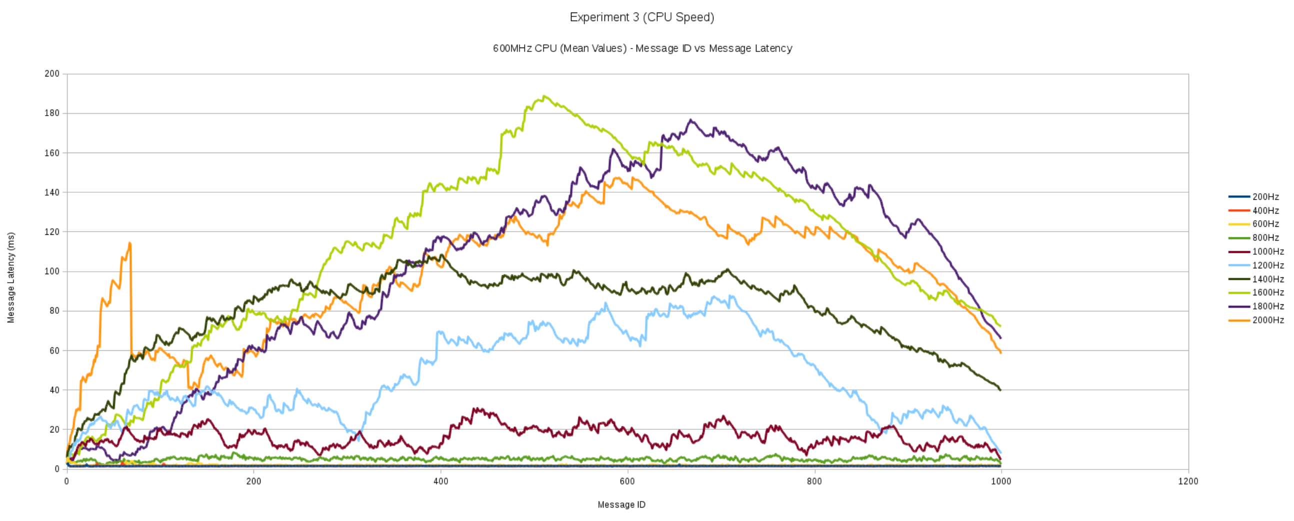

Figure 3.3: Experiment 3 - 100% CPU Speed, All Frequencies

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 3.4.

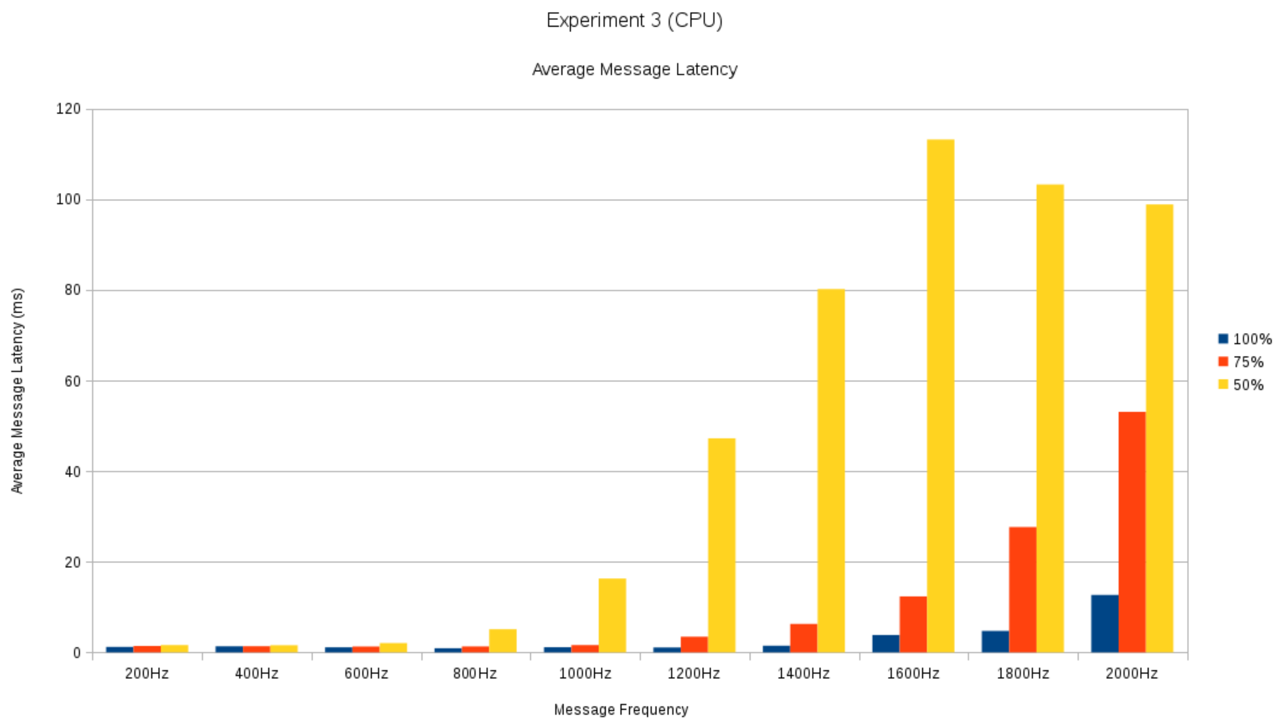

Figure 3.4: Experiment 3 - All CPU Speeds, All Frequencies

3.1.4 Experiment 4 - WiFi Connection

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

It is also reasonable that multi-robot systems may desire each robot to be physically mobile, thus a wired network connection may not be feasible in many set-ups. It is important to understand how this technology will affect performrmance compared to the ideal wired situation.

The Raspberry Pi 3 Model B's utilised in these experiments contain 802.11n WiFi chips which support a 2.4GHz network connection.

The experiment consists of running the same code in both a wired (Ethernet) situation and a wireless (WiFi) situation. 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 several times with a range of message frequencies from 200Hz to 2000Hz in 200Hz steps (200Hz, 400Hz, 600Hz, ...).

The hypothesis was that switching to WiFi 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).

Results

The experimental results after 3 runs in each setup appeared to somewhat agree with the hypothesis. Overall message latency was higher across all frequencies using WiFi except 2000Hz - as shown in Figure 3.5.

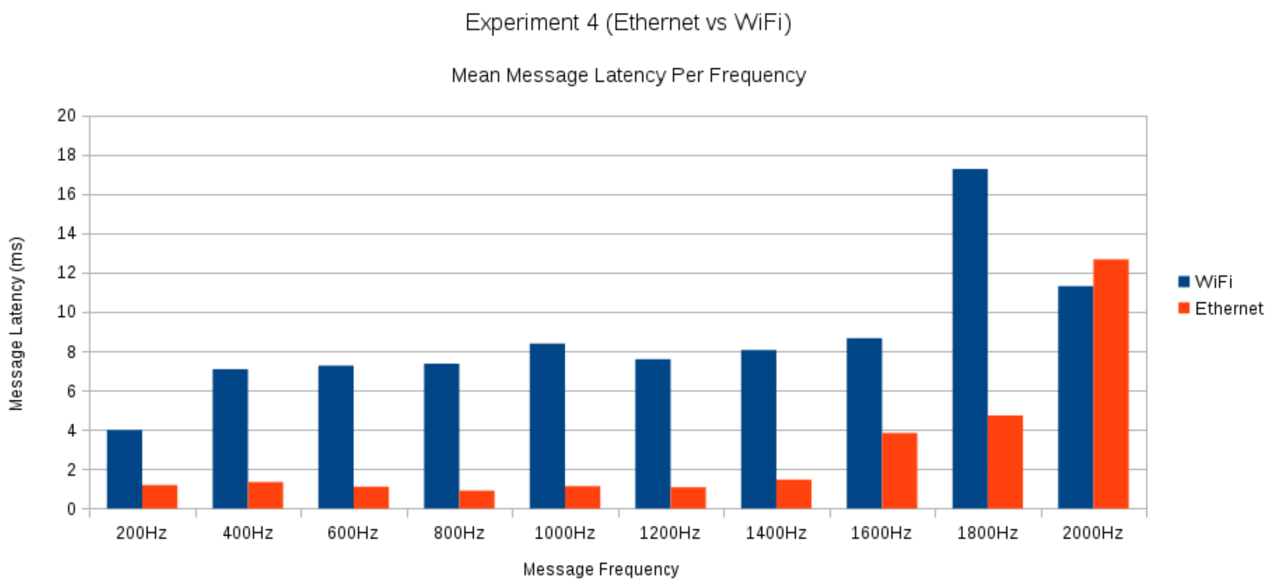

Figure 3.5: Experiment 4 - Ethernet, All Frequencies

Figures 3.6 and 3.7 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 even more erratic results than Ethernet, and thus it's effects must be carefully considered if using WiFi in future experiments.

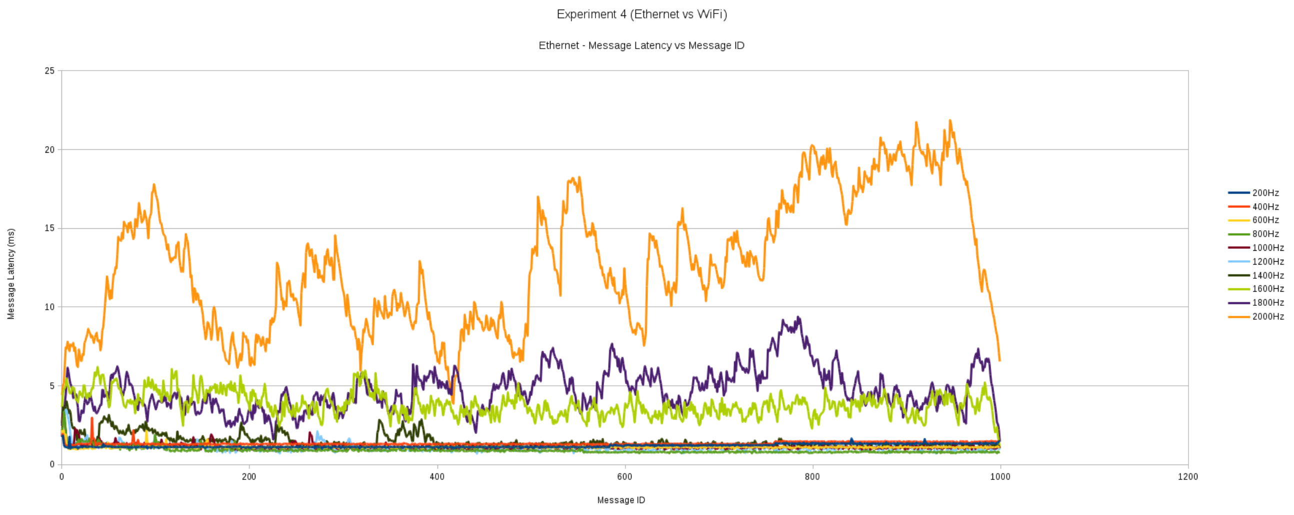

Figure 3.6: Experiment 4 - Ethernet, All Frequencies

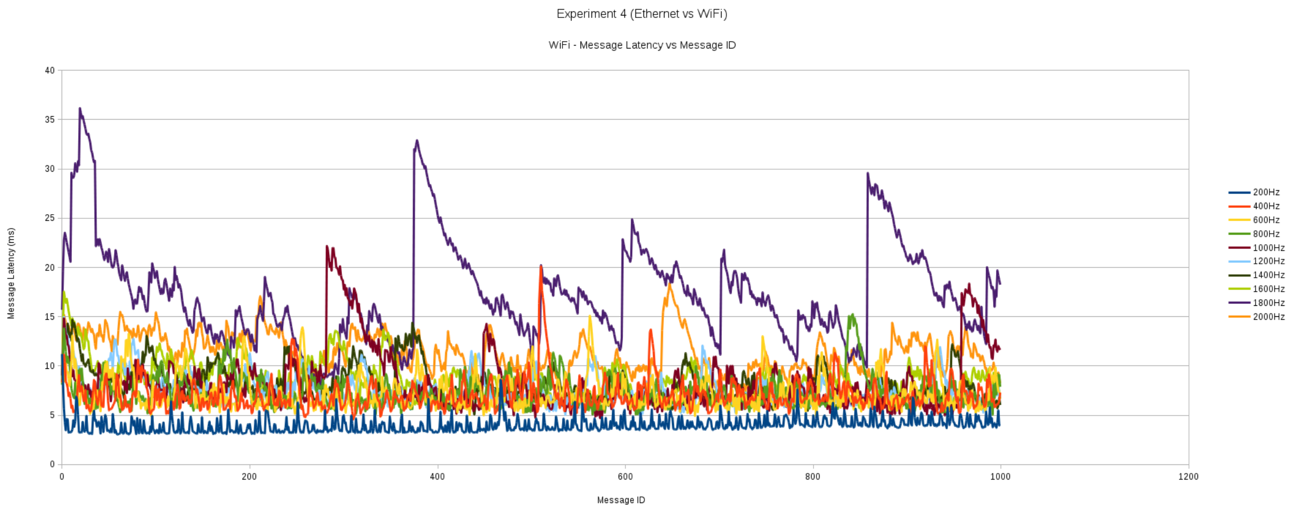

Figure 3.7: Experiment 4 - WiFi, All Frequencies

3.2 Realistic Data Experiments

Previous experiments were presented as scoping experiments. These were designed to systematically understand which variables were of concern when running tests on ROS's communication performance. However, it was not certain that these prior conclusions would translate well to using ROS in a realistic scenario. Throughout the scoping experiments, the same message of ‘hello world’ was used as dummy data.

In order to verify the conclusions were correct, samples of realistic data was explored. Two likely 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 used was a 30Hz (30 frames-per-second) RGB 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.

Both data sets were acquired from the MIT Stata Center dataset [3]. This dataset contains both sensor feeds (such as from a laser sensor), and video feeds (from a Kinect RGB + depth camera). 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.

3.2.1 Experiment 5 - CPU Clock Speed (Real Data)

This experiment aims to verify the results of Experiment 3 in Section 3.1.3 when using real data. This is achieved by repeating the experimental set-up and procedure, but replacing the sent message with a recorded payload.

The experiment was repeated using the previously mentioned sensor data, as well as video data.

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 different message sizes.

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 of 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

The experiment appeared to confirm the results of Experiment 4. For sensor data, 100% CPU speed (1.2GHz) was the lowest latency at 7 out of 10 message frequencies, as shown in Figure 3.8, and 50% CPU speed (600MHz) was slowest at 9 out 10 frequencies.

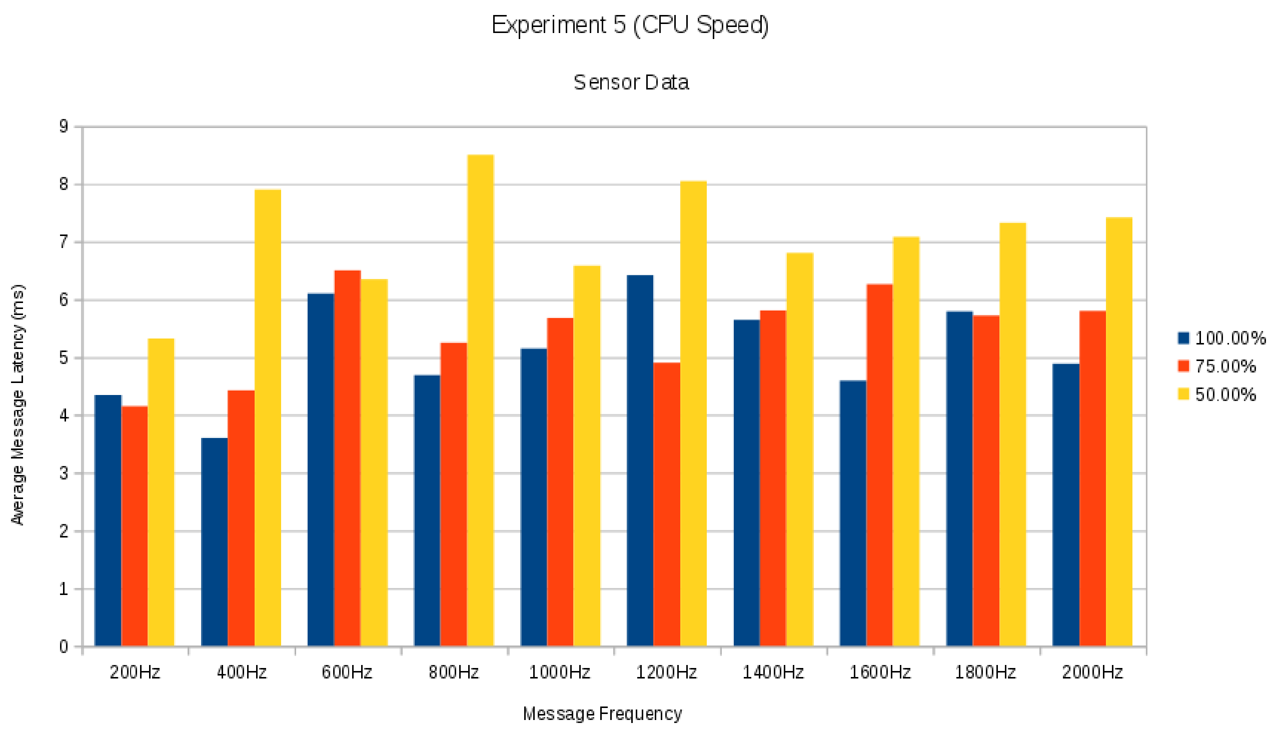

Figure 3.8: Experiment 5 - Sensor Data, All Frequencies

For video data, the trend held at the lower experimental frequencies (see Figure 3.9) 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 is less impactful as message sizes increase. CPU speed became entirely insignificant at higher message frequencies for video data, as it is thought the network interface becomes the bottleneck. 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 3.10.

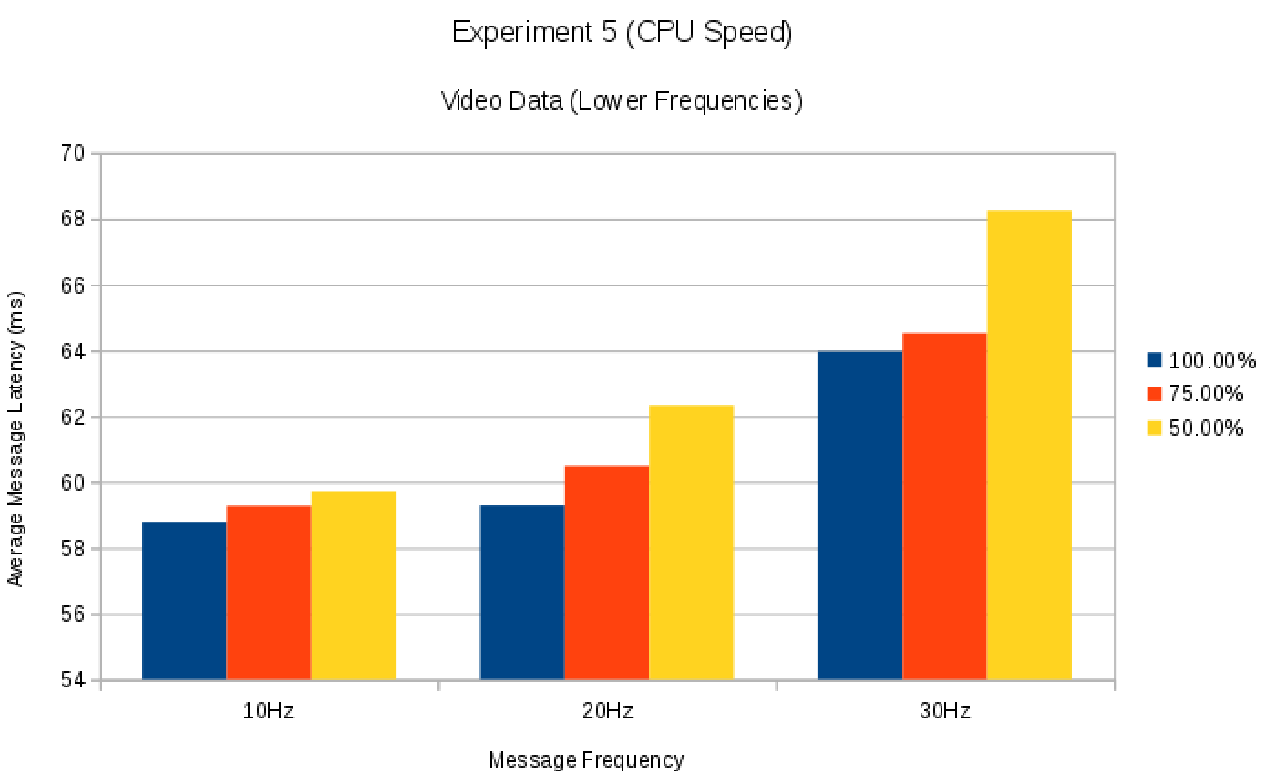

Figure 3.9: Experiment 5 - Video Data, Low Frequencies

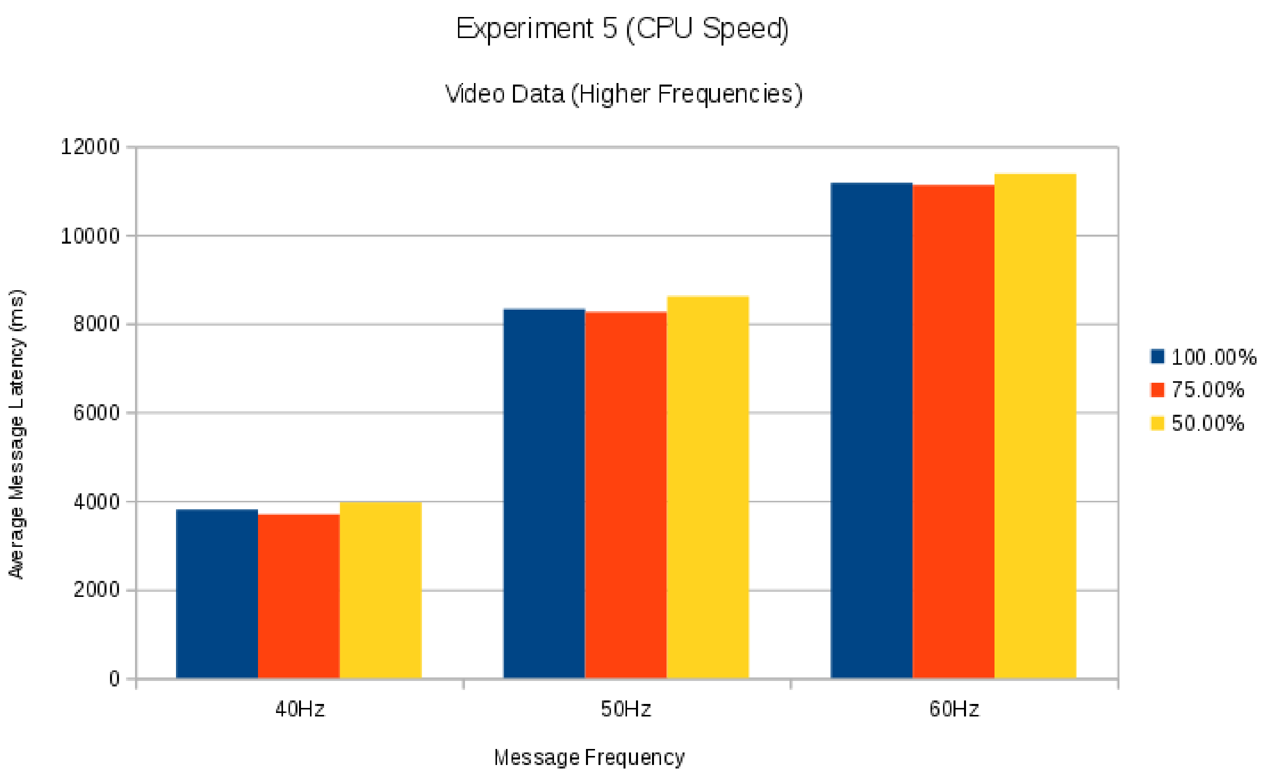

Figure 3.10: Experiment 5 - Video Data, High Frequencies

3.2.2 Experiment 6 - WiFi Connection (Real Data)

This experiment was a repeat of Experiment 4, except using realistic data. The data used was the same MIT Stata Center dataset used previously in Experiment 5.

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

Expectations from this experiment were that the results of Experiment 4 would be exacerbated. The higher latencies of wifi would be exaggerated by using larger message sizes, and performance would degrade even faster than in Experiment 4.

Results

Results showed what has been classified as 'good' performance at only 200Hz (see Figure 3.11). 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 3.12).

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. 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 3.11) this difference can be taken in to account for future experiments.

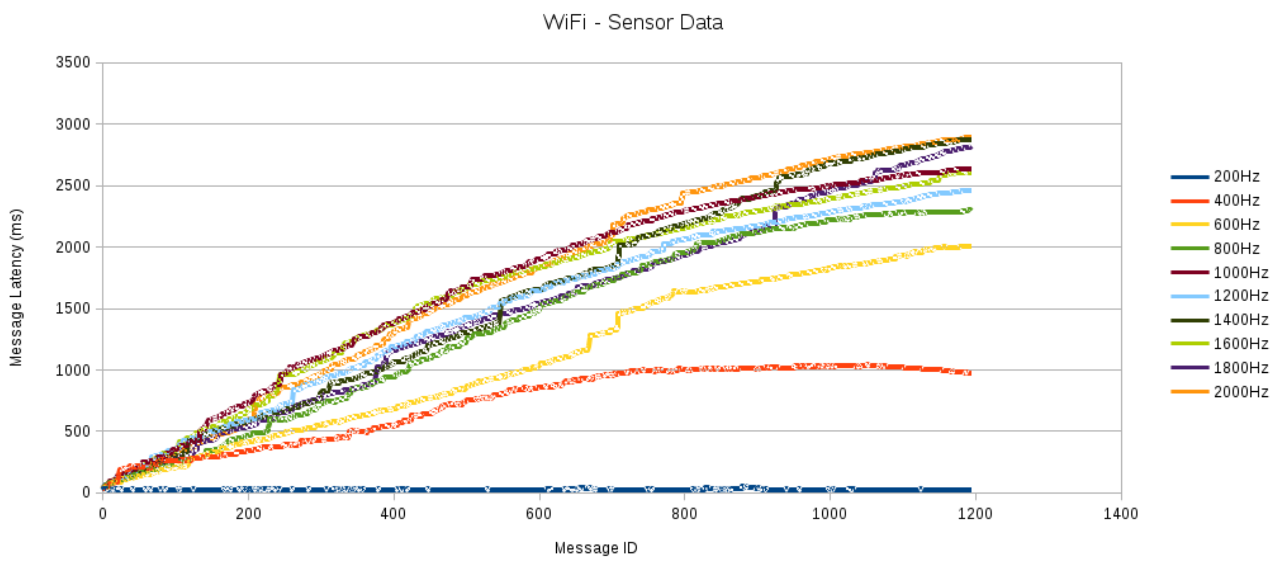

Figure 3.11: Experiment 6 - Sensor Data WiFi, All Frequencies

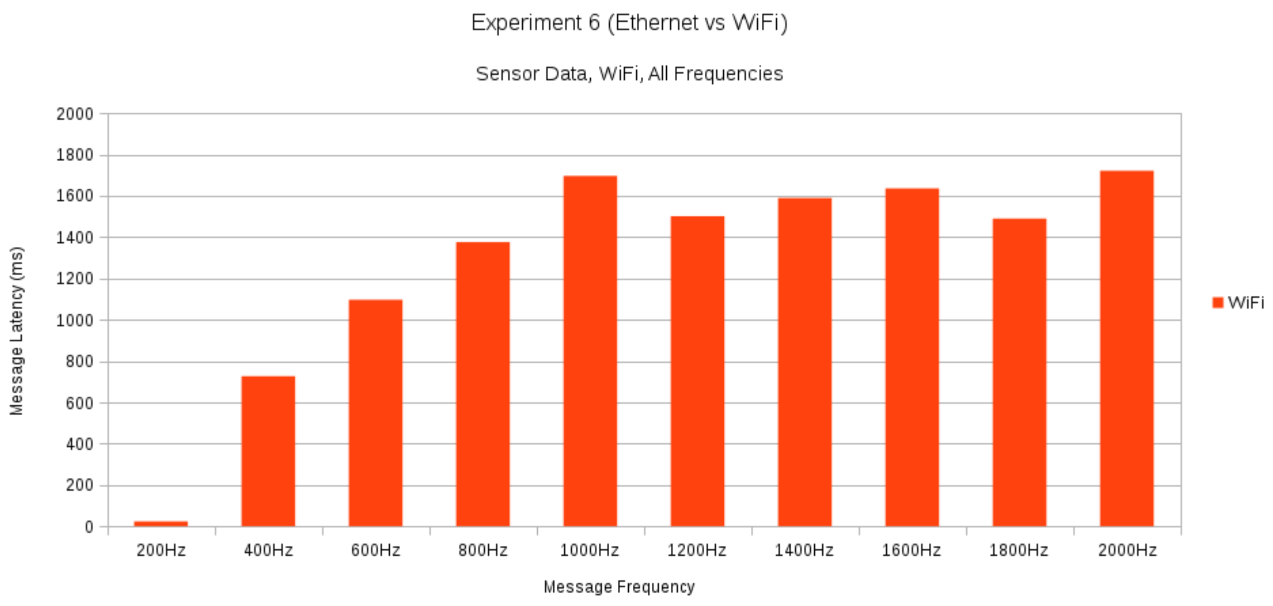

Figure 3.12: Experiment 6 - Sensor Data WiFi, All Frequencies

Further Investigation

After the results shown in Figure 3.12 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 3.13 shows, message stream performance was similar as seen before. 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 3.14 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.

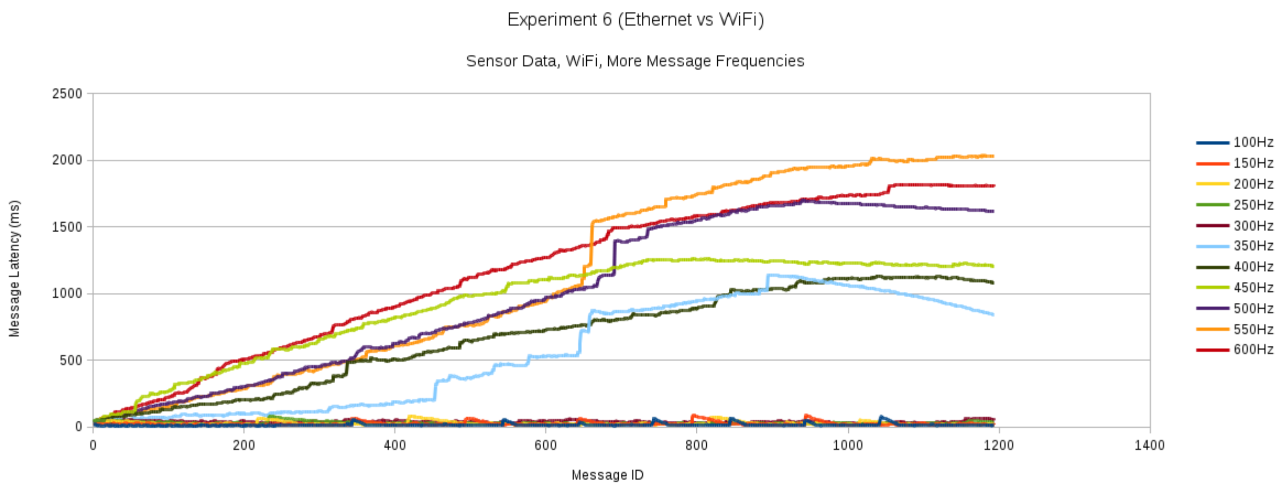

Figure 3.13: Experiment 6 - Sensor Data WiFi, More Frequencies

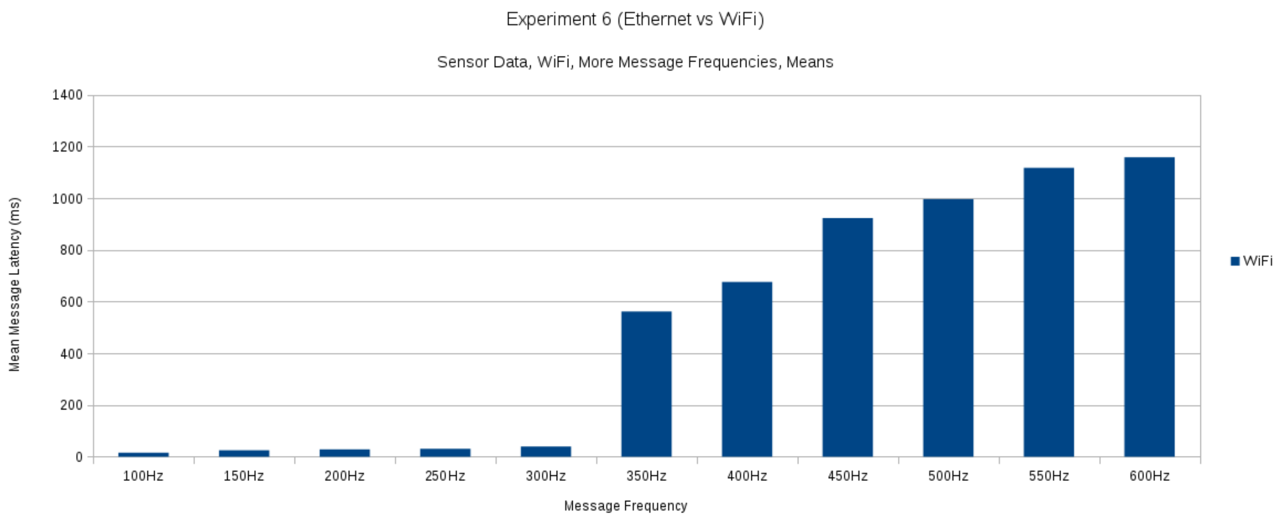

Figure 3.14: Experiment 6 - Sensor Data WiFi, More Frequencies

Chapter 4

Host Scalability

These scalability experiments look in to how well ROS scales when introduced to more nodes, or more hosts, or both.

4.0.3 Experiment 7 - Vertical Scaling

Chapter 5

Real World Scenario

This subproject is designed to demonstrate ROS's scalability in a real world scenario, using robot cars.

Appendices

Appendix A

Experiment 2 Other Graphs

Other graphs commented out for now.

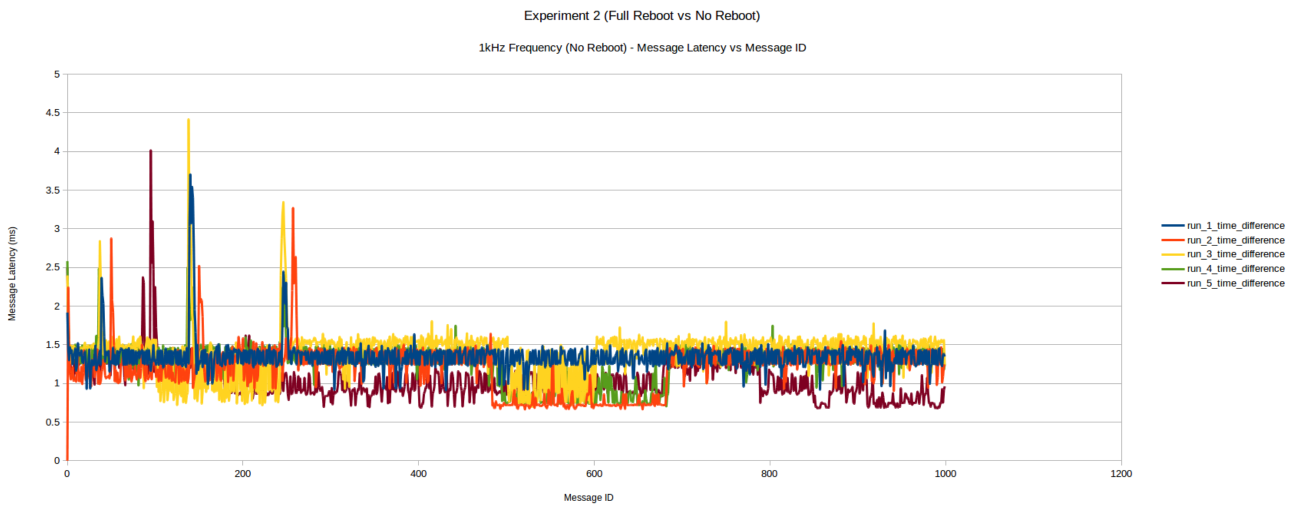

Figure A.1: Experiment 2 - No Reboot 1KHz Message Frequency

Appendix B

Code

B.1 Experiment 1 - Initial Code

B.1.1 Sender and Receiver

Code commented out.

Appendix C

Running the Programs

To compile this dissertation:

```
> pdflatex dissertation
> bibtex dissertation
> pdflatex dissertation
    > pdflatex dissertation
```

An example of running from the command line is as follows:

```
> rosrun rosberry_experiments run_experiment.py
```

Appendix D

Generating Random Graphs (Example Appendix)

```
> java RandomGraph 100 0.9 > 100-90-00.clq
```

Bibliography

[1] Getting Started with YARP Ports. http://www.yarp.it/note_ports.html.

[2] Kerl svn repository. https://sourceforge.net/p/kerl/code/HEAD/tree/.

[3] Mit stata center dataset. https://projects.csail.mit.edu/stata/downloads.php.

[4] OpenRDK Introduction. http://openrdk.sourceforge.net/index.php?n=Main.Introduction.

[5] Player server manual. http://playerstage.sourceforge.net/doc/Player-1.6.5/player-html/.

[6] roslaunch ros package. http://wiki.ros.org/roslaunch.

[7] Sunfounder robot car kit. https://www.sunfounder.com/rpi-car.html.

[8] Technical Committee on Multi-Robot Systems (TC MRS) of the IEEE Robotics and Automation Society. http://multirobotsystems.org/.

[9] Xenomai Git Repositories. https://git.xenomai.org/.

[10] Xenomai tutorial. http://www.cs.ru.nl/lab/xenomai/exercises/ex01/Exercise-1.html.

[11] Mateusz Macia Adam Dbrowski, Rafa Kozik. EVALUATION OF ROS2 COMMUNICATION LAYER. http://roscon.ros.org/2016/presentations/rafal.kozik-ros2evaluation.pdf.

[12] Jeremy H Brown and Brad Martin. How fast is fast enough? choosing between xenomai and linux for real-time applications.

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

[14] 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.

[15] 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.

[16] Brain Gerkey. Why ROS 2.0? http://design.ros2.org/articles/why_ros2.html.

[17] 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.

[18] Jae Hwan Koh and Byoung Wook Choi. Real-time performance of real-time mechanisms for rtai and xenomai in various running conditions.

[19] 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.

[20] 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.

[21] Paul Newman. Introduction to programming with moos. *Communications,(November)*, pages 1–34, 2009.

[22] 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.