JACK N. AVERITT COLLEGE of GRADUATE STUDIES GEORGIA SOUTHERN UNIVERSITY

Georgia Southern University

Digital Commons@Georgia Southern

Electronic Theses and Dissertations

Graduate Studies, Jack N. Averitt College of

Fall 2015

Networked Heterogeneous Systems in a ROS-Enabled Cloud Environment

Christopher Reid

Follow this and additional works at: https://digitalcommons.georgiasouthern.edu/etd

 Part of the Acoustics, Dynamics, and Controls Commons

Recommended Citation

Reid, Christopher, "Networked Heterogeneous Systems in a ROS-Enabled Cloud Environment" (2015). *Electronic Theses and Dissertations*. 1433.
https://digitalcommons.georgiasouthern.edu/etd/1433

This thesis (open access) is brought to you for free and open access by the Graduate Studies, Jack N. Averitt College of at Digital Commons@Georgia Southern. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of Digital Commons@Georgia Southern. For more information, please contact digitalcommons@georgiasouthern.edu.

NETWORKED HETEROGENEOUS SYSTEMS IN A ROS-ENABLED CLOUD ENVIRONMENT

by

CHRISTOPHER REID

(Under the Direction of Biswanath Samanta)

ABSTRACT

It is important in the development of cloud robotics that the challenges presented by transferring computational loads to networked resources are properly addressed. The challenges include network latency, data integrity, security, and privacy. The objective of the present work is to investigate the issues of latency and data integrity in a representative cloud robotics environment. The present work involves setting up a cloud robotics network in an open-source Robot Operating System (ROS) framework and carrying out investigations on the levels of latency and reduction in data integrity as utilization of the network increases. In this study, a virtual datacenter has been set up to provide the foundation on which to build software systems to provide cloud services. Robot Operating System (ROS) framework has been used to facilitate communication among heterogeneous systems in the network. Three types of robots, including the Parrot AR.Drone2.0, the Kobuki Turtlebot 2, and the LEGO EV3 have been implemented in the system. The system has been tested for baseline connectivity and under low- and high-bandwidth conditions to determine the latency and data integrity of the network connections. Additionally, a heterogeneous system consisting of sensor feedback from the AR.Drone2.0 and motor control of the Turtlebot 2 has been built to examine the connection between the devices themselves. Through this study, it has been demonstrated that under low-bandwidth conditions, the network performs reasonably well in the areas of latency and data integrity. However, for high-bandwidth conditions involving image transmission, the network performance deteriorates considerably, both in terms of latency and data integrity. One possible reason is the wireless router used in the current setup. It is also recommended that, especially under high-bandwidth loads, it is necessary for networked systems to perform some portion of their computations on-board and high-bandwidth wireless connectivity to the cloud is facilitated. Ongoing research and future directions are also outlined.

INDEX WORDS: Cloud robotics, Network, Heterogeneous, Robot Operating System

NETWORKED HETEROGENEOUS SYSTEMS IN A ROS-ENABLED CLOUD ENVIRONMENT

by

CHRISTOPHER REID

B.Sc., Georgia Southern University, 2013

A Thesis Submitted to the Graduate Faculty of Georgia Southern University in Partial

Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

STATESBORO, GEORGIA

©2015

CHRISTOPHER REID

All Rights Reserved

NETWORKED HETEROGENEOUS SYSTEMS IN A ROS-ENABLED CLOUD ENVIRONMENT

by

CHRISTOPHER REID

Major Professor: Biswanath Samanta

Committee: Christopher Kadlec

Minchul Shin

ACKNOWLEDGEMENTS

I would like to thank Dr. Samanta for his continuous support and advisement, and Dr. Kadlec for his patience with me and his dedication to this work.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

FPGA – Field Programmable Gate Array

IoT – Internet of Things

RAM – Random Access Memory

SDN – Software Defined Networking

UAV – Unmanned Aerial Vehicle

VM – Virtual Machine

CHAPTER 1: INTRODUCTION

1.1 NETWORKED SYSTEMS AND INTERNET OF THINGS

A network is a collection of devices linked together through cabling or wireless transmissions to share resources and information (Whitson 2015). These networks are an integral part of work in collaborative environments and are indeed necessary to make the internet possible. Computers are the primary devices featured on such networks, but other network devices such as wireless routers, Ethernet switches, and more are used as well. When other systems are connected to the network, such as a controller for a robot, the concept of *networked systems* is applied. These systems when networked have a much higher functionality than their isolated counterparts. For example, a small robot with limited computing power cannot perform complex calculations in a timely manner while exercising control over its sensors and actuators, but such a robot could provide inputs to a networked computer running a code that may involve even a neural network simulator, and receive its instructions from that machine.

There are two significant obstacles in implementing networked systems. First, many of these systems feature embedded software or firmware that are not easy to modify or replace. For these items, the implementation is limited to the capabilities of the devices as it was produced, and it cannot be tailored to fit the need of the system. Second, there is no standard for the formatting of the data these devices transmit to the network. The datasheet for each device must be checked to determine the format in which the device transmits its data to the network, which includes the sequence and size of the data packets, the encoding applied to the data, and the rate at which the data is sent, as well as the expected size and sequence of data received.

Additional obstacles to the use of networked systems include latency, data integrity, privacy, and security. Weir describes latency, performance, and reliability as "critical considerations in the design and deployment of cloud computing platforms" (Weir 2013). Willcocks et al. identified privacy and security as the largest risks for transitioning to using cloud computing, and explored potential legal issues created by the transfer of data across political borders. Additionally, they describe the risk of using cloud service providers that are potential targets of hacking through hosting controversial web sites (Willcocks, Venters and Whitley 2013).

When the set of networked devices expand beyond robots and control systems to include a plethora of things such as clothing, vehicles, lighting, appliances, and many more, the concept of the Internet of Things (IoT) is achieved. Challenges faced with IoT are similar to the challenges previously mentioned for networked systems, applied to a much larger scale. In near future there will be many more dissimilar (heterogeneous) devices in the environment competing for network connectivity, transmitting sensitive data, and providing their capabilities as services to be used by other connected devices. There is a need to get ready for this computing environment in the age of IoT (Pye 2014).

The Robot Operating System (ROS), developed by Quigley et al. (Quidley, et al. 2009), is an open-source software framework that provides the ability to develop code and applications across multiple types of (heterogeneous) robots and platforms (Doriya, Chakraborty and Nandi 2012). This work aims to evaluate the current state of wireless connectivity between heterogeneous devices and a network and anticipate problems that will arise as the number of devices present within an environment increases.

1.2 Scope of Present Work

With the concept of the Internet of Things fast approaching implementation, a study was needed to examine the effect of numerous active heterogeneous networked devices and cloud services on the performance and integrity of a local network. The term heterogeneous is used in the sense that, while multiples of the same type of device may be used, there will be different types of devices in use and communicating with the cloud. The main hypothesis of this work is that a network infrastructure can be built to support the communication of numerous heterogeneous robots and their utilization of cloud resources for remote processing without a significant decrease in network performance and integrity.

To test the hypothesis, the following objectives of this work were set:

- To build a datacenter capable of providing cloud resources to the robots,
- To design a network infrastructure to support heterogeneous system communication and the use of the cloud services.
- To implement various multi-robot systems using ROS, and
- To examine the effects of using multiple, heterogeneous systems on the ROS-enabled network simultaneously.

To achieve the stated objectives, a virtual datacenter was built to support the development of the software needed to interact with and control the networked robot agents. A network was designed and implemented to enable robot-to-robot and robot-to-cloud communication without the additional overhead of network traffic not related to the function of the robots or their utilization of the cloud. Several software systems were developed in machines in the datacenter to utilize the Robot Operating System (ROS) in establishing communication with the robot

agents and interacting with them. Finally, the impact of using these systems on the network (e.g. network latency, packets dropped) was examined and analyzed.

1.3 Organization of Thesis

The rest of the thesis is organized in the following order:

Chapter 2 is the literature review covering eight topics. First the background of and need for heterogeneous robot communication is covered. Next, the concept of cloud robotics is reviewed, followed by the challenges presented by cloud robotics. The fourth section examines the topic of virtualization and the benefits of virtual datacenters. Next, the Robot Operating System and its ability to facilitate heterogeneous robot communication is examined. Sections six, seven, and eight review the capabilities of the three types of robots used in this study: the Parrot AR.Drone2.0, the Kobuki Turtlebot 2, and the LEGO EV3.

Chapter 3 covers the research methodologies used in this study. First the overall system configuration is presented. Next, the design and implementation of the bio-Inspired Robotics and Intelligent Systems (bIRIS) virtual data center is detailed, followed by the components of the testbed network. The implementation of ROS within the cloud is presented next, followed by the configuration of each of the three types of robots to make them compatible with the cloud. Finally, section eight details the design of the experiments to test the robots' effect on latency and data integrity.

Chapter 4 presents the experimental results and related discussions, starting with establishing a baseline network bandwidth capacity for the system. The AR.Drone2.0 is the first robot examined, and baseline, low-bandwidth, and high-bandwidth tests are conducted for the UAV and the results presented. Likewise, the same experiments were conducted on the Turtlebot,

which is presented in section three. Section four details the LEGO EV3 experiments, which include baseline performance, low-bandwidth performance, and cloud-in-the-loop processing, which examines the performance of a system that uses the cloud to implement a controller in a time-sensitive control loop. Finally, a cloud-processed tilt-control experiment is performed which queries the pose of an AR.Drone2.0 to process and determine the desired state of a Turtlebot.

Chapter 5 presents a summary of the present work along with recommendations for future work.

CHAPTER 2: LITERATURE REVIEW

2.1 HETEROGENEOUS SYSTEMS

As the tasks which are delegated to machines grow more complex and more numerous, so too must the machines grow more complex and powerful to execute them. These machines, functioning alone in an environment and expected to perform all tasks to satisfaction, present numerous disadvantages. First, data travelling through the system must be processed by the machine, and thus exists a data chokepoint at the machine, reducing the reliability and response time of the system. Second, the system's sum of resources, both hardware and software, would exist at a single location, making any downtime the machine experiences catastrophic to the system. Finally, even with enormous complexity, a system of multiple, cooperative robots performing tasks simultaneously would outperform a single machine as the task complexity increases (Cao, Fukunaga and Kahng 1997).

Thus, it is easy to see why heterogeneous systems are preferable; heterogeneous systems vary in shape, size, and capabilities, complementing each other and increasing the capabilities of the system as a whole. Heterogeneous systems are more capable, efficient, fault tolerant, and expendable than their monolithic counterpart (Parker and Tang 2006). While these systems have been tested in isolated, experimental environments, there has yet to be a study on a fully-functioning, always-on environment to support heterogeneous system operation.

2.2 CLOUD ROBOTICS

First, to consider the concept of cloud robotics, the definition of a cloud and the components that constitute the cloud should be provided. NIST defines cloud computing as “a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources […] that can be rapidly provisioned and released with minimal management effort or service provider interaction.” (The NIST definition of cloud computing

2011) The systems that provide these capabilities, including the physical servers and network hardware along with the software systems that enable interaction with these resources, make up the cloud. While heterogeneous multi-agent systems offer tremendous advantages over homogeneous systems, when combined with the concept of cloud robotics, there are even more advantages to be had. Kehoe, Patil, Abbeel, and Goldberg define cloud robotics as "Any robot or automation system that relies on either data or code from a network to support its operation, i.e., where not all sensing, computation, and memory is integrated into a single standalone system." (Kehoe, et al. 2015). Unlike traditional systems that are generally self-contained, the cloud offers the ability to have a remote system perform computations and store data. This provides several abilities, from downsizing local resources, achieving dependability and robustness through backups and redundancies, and the ability to scale systems without having to similarly scale the local resources (Enabling a new future for cloud computing 2014). Through cloud robotics, remote agents can utilize the processing, storage, and connectivity capabilities of non-mobile computing systems. In particular, cloud robotics, and cloud computing in general, offers the concept of utility computing; instead of buying computational resources to own, these resources can be provided for use on a short-term, as-needed basis (Armbrust, et al. 2010).

2.3 Challenges in Cloud Robotics

2.3.1 Network Latency

Generally, the reaction time of some local-processed system's output to an input is very small, in the order of nanoseconds or less. When utilizing cloud services, however, the data spends much time in transportation through the network, resulting in noticeable latency. This latency can be very expensive to cloud service providers (Kelmendi 2013). Latency is unavoidable when data is being transferred over a network, though the acceptable amount of latency will differ based on the service being used (Sharp 2012).

2.3.2 Data Integrity

When transmitting data through a network, there is some risk of the packets of data becoming corrupted or lost. This is especially true for wireless connections, which compete with each other to transmit their data in the same frequencies. The quality of wireless connections is subject to the environment, which includes physical obstructions as well as electromagnetic noise (Gungor and Hancke 2009). Similar to latency, excessive packet loss leads to unreliable systems, where the level of acceptable packet loss depends on the services being provided by the cloud. To reduce packet loss in networked systems, some research conducted in this area examines the data packets themselves and reduces the size of the packets or number of packets to be sent as a method of reducing packet loss (An, et al. 2012). Other research proposes adding compensators to controllers or plants to generate replacement data for lost packets (Nguyen and Yoo 2015). Strategies for scheduling multisensory data retrieval are proposed to improve the performance of networked systems (Wang, Liu and Meng 2015). Regardless of the compensation method, data integrity must be accounted for when building networked systems, especially those that feature wireless connections.

2.4 VIRTUALIZATION

In this study, many of the cloud resources were provided using the resources and capabilities of a virtual datacenter. Virtualization is the concept of abstracting hardware resources such that multiple paths to hardware can be provided, allowing numerous users' access to the same hardware. A virtualized server, for example, provides its resources for multiple virtual machines, each running its own operating system, to operate with. Each virtual machine has access to some portion of the server's resources, which can be statically or dynamically allocated depending on the capabilities of the management software. The use of virtualization minimizes the number of server machines needed for the operation of the datacenter and allows the remaining servers to

run much more efficiently, which significantly reduces the energy consumption of the datacenter (Deka 2014).

2.5 Robot Operating System

The Robot Operating System (ROS) is an open-source framework that enables heterogeneous robot communication through defining standards in data packet construction and communication protocols (Quidley, et al. 2009). ROS is designed to be implemented in a modular fashion and spread out over many systems and devices. ROS terminology (ROS/Concepts 2014)is explained below, as it is used frequently throughout this thesis.

- **Master or Core**- The ROS master or core is a node that handles the routing of ROS traffic through the network. It is not a centralized hub; that is, data streams do not typically flow through the ROS master node. Instead, the ROS master node coordinates the overall flow of information and instructs individual nodes in where they are sending their data streams.
- **Nodes –**A ROS Node is an executable script or program that runs within the ROS framework. Nodes are designed to be modularized as much as possible, and a single robot or system may have many nodes running to fully interface the hardware and non-ROS software with ROS.
- **Messages –** A ROS message is a data structure made some number of primitive data types and arrays of primitive data types. ROS messages are strongly typed and can be custom built to meet the needs of the system.
- **Topics –** A ROS topic describes a particular data stream in which a single type of message is transferred from node to node. A node is said to *subscribe* to a topic when it is receiving the data stream from the topic. A node is said to *publish* to a topic when

the node is sending data to the topic. A topic can have arbitrarily many subscribers and publishers.

- **Services –** A ROS service functions much like a remote procedural call; a node may send data via a service to be processed in some fashion, and will expect a response in return, whether it is a return value from some computation or merely an affirmation that a procedure was performed.
- **Packages –** ROS packages are the base level bundles of code that are designed to be uploaded for sharing and downloaded for use. ROS packages can contain zero or more of nodes, messages, libraries, configuration files, build instructions, documentation, and more. Because of ROS's open source nature, packages are often free to download and use, and are developed by programmers across the world for a multitude of purposes that, due to the modular, standard-focused nature of ROS, can be applied to many different types of robots.

2.6 PARROT AR.DRONE2.0

The Parrot AR.Drone2.0 is a hobby unmanned aerial vehicle (UAV) with a quadrotor design. The AR.Drone2.0 features front- and down-facing cameras for visual navigation, stability, and ground velocity measurements, as well as a three-axis accelerometer, magnetometer, and gyroscope. Also included are a pressure sensor and ping sensor for altitude measurements (Parrot AR.Drone2.0 2015). The AR.Drone2.0 runs on Linux BusyBox, a stripped down operating system designed for embedded systems. The BusyBox system interfaces with the motors and actuators of the AR.Drone2.0 and provides the ability to make a wireless connection, but is unable to run ROS directly.

2.7 KOBUKI TURTLEBOT 2

The Kobuki Turtlebot is a two wheel drive robot base packaged with a Microsoft XBOX Kinect sensor and an Asus netbook. The netbook runs Linux Ubuntu and has ROS installed, allowing the robot to make use of ROS directly. The robot base has several sensors, including a bump sensor to detect frontal collisions, a battery sensor to detect the current battery level, infrared sensors for detecting and docking with the recharging base, and gyroscope to detect rotational movement, while the Kinect sensor provides laser scanning depth information along with camera images (Turtlebot 2 n.d.).

2.8 LEGO EV3

The LEGO EV3 is a newer version of the NXT with a major new feature: the ability to run a Linux based operating system hosted on an SD card. ROS can be run natively on this OS and applications developed directly on the controller. Furthermore, the EV3 has a USB port for a wireless USB adapter, making communication with the cloud much faster than the NXT's Bluetooth connection (LEGO Education EV3 2015).

CHAPTER 3: RESEARCH METHODOLOGY

3.1 System Configuration

To conduct experiments on the topics examined in this study, namely latency and data integrity, a complex, powerful cloud environment was designed to imitate the real-world implementation of networked systems utilizing a cloud for communication and remote processing. First, a virtual datacenter was designed to provide the foundation on which the cloud services, e.g., remote operation, image viewing, data acquisition, were supported. Next, multiple networks were designed to segregate the traffic generated by the networked systems away from the traffic generated through management of the datacenter and storage of the virtual machines. Then, ROS was implemented in the cloud to be used as the framework on which to provide cloud services to the networked systems.

Three types of robots were acquired to be used in the experiments conducted for this work: the Parrot AR.Drone2.0, the Kobuki Turtlebot 2, and the LEGO EV3. The network settings of each robot to be used in the experiments were configured to connect to the cloud network through their wireless adapters, using a passcode. To interface the robots with ROS, ROS drivers were either downloaded or built based on availability for each particular robot. Additional software was developed through the ROS framework for other tasks such as directing robot movement, acquiring and storing robot sensor data, and integrating multiple robots into a single control system.

For each of these tests, the latency and data integrity of the connection between the robot and the cloud was tested by streaming data packets to the target robot and measuring the time taken for the packet to return, or whether the packet returned at all. This data was plotted to visualize

the stability of the network connection over time. The tests were conducted each time an additional robot was connected to the network to observe the impact on the network connectivity of each robot.

3.2 bIRIS Virtual Datacenter

To provide the computational and storage resources necessary for the implementation of a cloud computing environment, a datacenter was designed that would host these resources as well as provide the ability to build software systems that would, when connected to the remote robot agents, perform calculations and communications as necessitated by the remote agents. A virtual datacenter was chosen to make efficient use of server hardware while providing a dynamic, reconfigurable environment in which services and control systems could be built, altered, and shut down as needed by the client robots or users. These services and control systems were implemented on virtual machines - software-defined computers that utilize some portion of its host server's resources to operate. Because these computers are software-defined, resources can be quickly allocated or deallocated as demand fluctuates, maximizing efficiency of the datacenter. Furthermore, the virtualized datacenter provides a high degree of fault tolerance for the virtual machines that isn't found in physical datacenters; virtual machines can be immediately rebooted upon crashing, or migrated to another host machine without an interruption in service should the host machine fail.

3.2.1 bIRIS Datacenter Design

Six physical servers were built in total to provide ample resources for the virtual datacenter and future expansion. Five of the machines functioned as host servers for virtual machines, while the sixth provided network-attached storage (NAS) for the storage of the virtual machine files. NAS was chosen as the virtual machine storage method to provide both a failsafe by separating

storage from computational resources to prevent the loss of both in the event of a failure of either, and to provide the ability for virtual machines to migrate from host to host to balance the load on each host machine, reducing thermal output and increasing efficiency.

3.2.2 Network Design

Due to the virtual nature of the datacenter, software-defined networking (SDN) was available to be used to define a number of networks, as opposed to a single network, to be used in the design of the cloud environment. Through SDN, the virtual machines and host machines did not have to use the same network interfaces for communicating with other devices, allowing for the segregation of network traffic based on the nature or purpose of the data being transferred. Three networks were defined to segment network traffic into three categories: management, storage, and testbed. Management traffic is data related to the operation of the host machines, including host performance data and communications between the hosts and the datacenter management software (Section 3.2.4). Management traffic was placed on its own network to prevent the disruption of the virtual datacenter operation by high network usage by client or storage devices. Likewise, storage network traffic was placed on its own network such that the link between a virtual machine and its storage would not be disrupted. Separating the management and storage networks from the testbed network also provided a layer of security against removing the channel for client devices or users to access or manipulate sensitive data, whether accidentally or purposefully. Finally, this leaves the testbed network, which contains all robots, virtual machines hosting control software, cloud resources outside of the virtual datacenter, and client PCs through which users interact with the cloud.

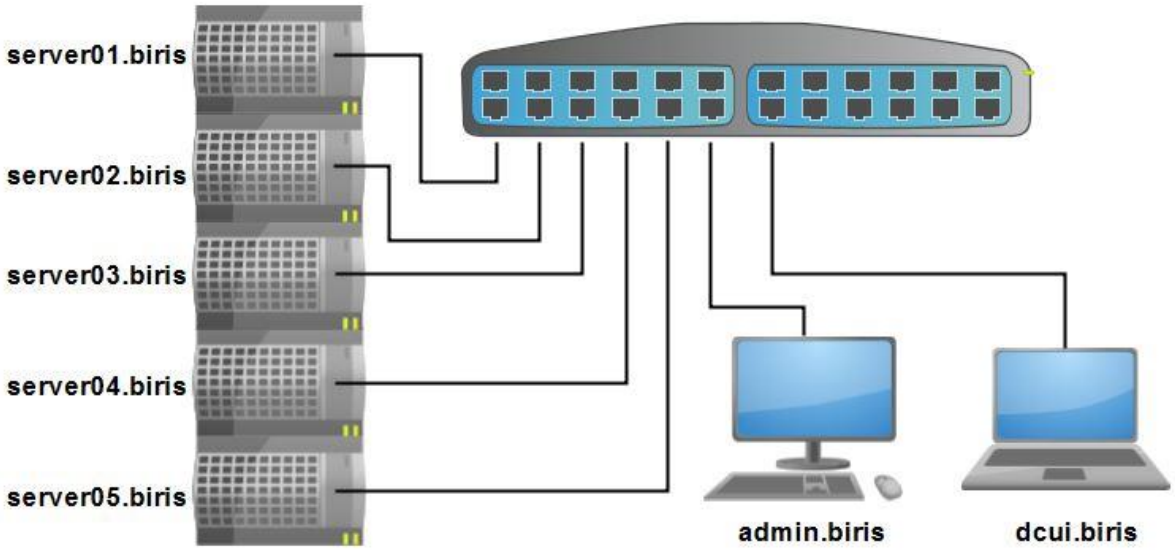

Figure 3.1 Management Network Physical Topology

Physically, the devices were connected to a Dell PowerConnect 6248 Ethernet switch via category 6 twisted pair cables. The PowerConnect switch was segmented into three virtual local access networks (VLANs) to create the three aforementioned networks (shown in Figure 3.1 through Figure 3.3) and prevent traffic from crossing over the networks. For wireless connectivity on the Testbed network, a Linksys E2700 wireless router was used to provide a Wi-Fi access point for remote systems.

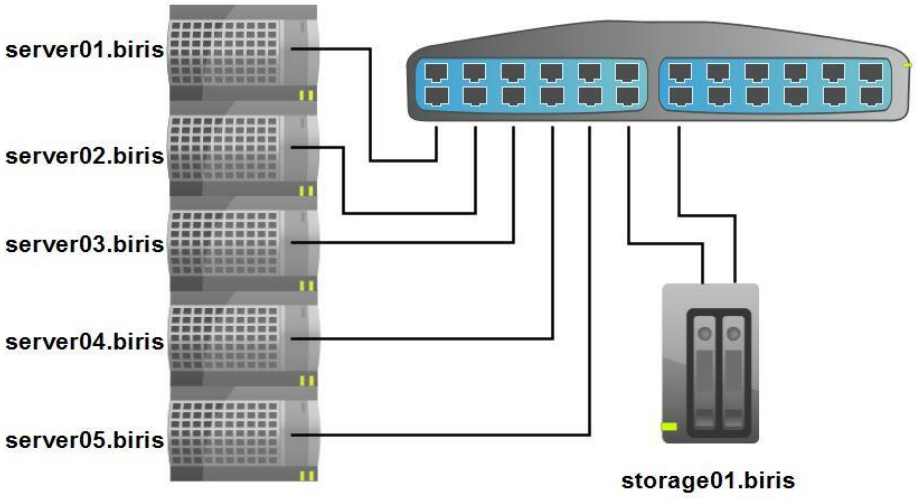

Figure 3.2 Storage Network Physical Topology

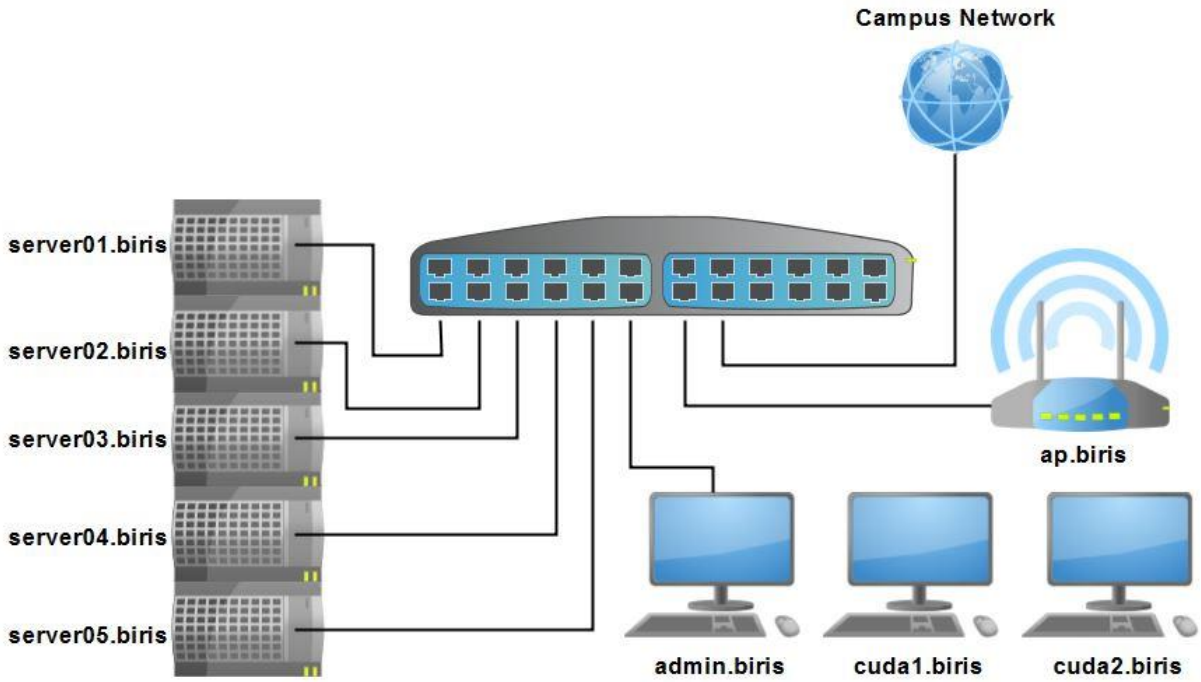

Figure 3.3 Testbed Network Physical Topology

IPv4 was used exclusively in addressing all devices on the network due to its universal compatibility among networkable devices. Each network was assigned its own unique network

prefix (the first 24 bits of the 32 bit IPv4 address) and, in the case of the testbed network, an IP scheme was established to organize devices on the network by their function. Because each device requires a unique IPv4 address to be identified, the last 8 bits of the 32 bit IPv4 address is referred to as the host identifier and each device that uses the same network prefix has a unique host identifier. Due to the minimal number of devices on the management and storage networks, no IP scheme was used on those networks. The scheme for assigning IPv4 addresses to devices on the Testbed Network is shown in Table 3.1 below.

Table 3.1 Testbed Network IP Address Morphology

IP Address Morphology	
Host Identifier	Designation
1-15	Network Services
16-45	Physical PCs
46-125	Virtual Machines
126-200	Robots
201-254	DHCP

3.2.3 Host Server Installation

Six computers were built to function as the host servers of the datacenter, though one was later designated to provide NAS for the datacenter. The six machines were built nearly identically, with the sole difference being the NAS machine contained larger hard drives for increased storage capacity. Each machine was built with a Supermicro X9SCL/X9SCM motherboard with four Intel Xeon E3-1245 V2 processors with a clock speed of 3.4GHz each, as well as 32GB of RAM each. Each machine also featured 1TB of hard drive storage, though this storage was not used for virtual machine files. In addition, each machine included a network interface card (NIC) which, in addition to the two integrated with the motherboard, allowed for a total of three physical network connections. These network interfaces were utilized in the

segmentation of the networks, described in section 3.2.2 . The sixth machine hosted five 1TB Western Digital hard drives for VM storage.

Five machines were provided with USB flash drives imaged with VMware's ESXi 5.5 Hypervisor operating system. The ESXi configuration screen is shown in Figure 3.4. USB flash drives were used as opposed to the local hard drives so that there would be no need to create a small partition on a hard drive in each machine for operating system storage; this way, the hard drives remain fully free for cloud storage use. Each machine was connected to an Avocent Autoview Digital KVM Switch, to which a console was attached that would provide a direct console user interface (DCUI) to each machine. Through the DCUI, each machine's operating system was configured to be used in the datacenter. The five ESXi machines were configured with static IPv4 addresses of the management network prefix, with host identifiers of 11 through 15. Each of these machines was also assigned a hostname, or computer name, of server01 to server05. Once the ESXi machines were configured, the FreeNAS machine was configured in a similar fashion, except for being placed on the storage network instead of the management network.

Figure 3.4 ESXi Host Configuration

3.2.4 Datacenter Management Software

Next, VMware's vSphere Client was installed on a computer temporarily located on the management network. Through the vSphere client, a local datastore was configured on which to store the datacenter management VM. *Server01*'s networking was further configured to designate the network interface connected to the management network to be used for management traffic. A network interface to the Testbed network was added for VMs to use to connect to robot agents.

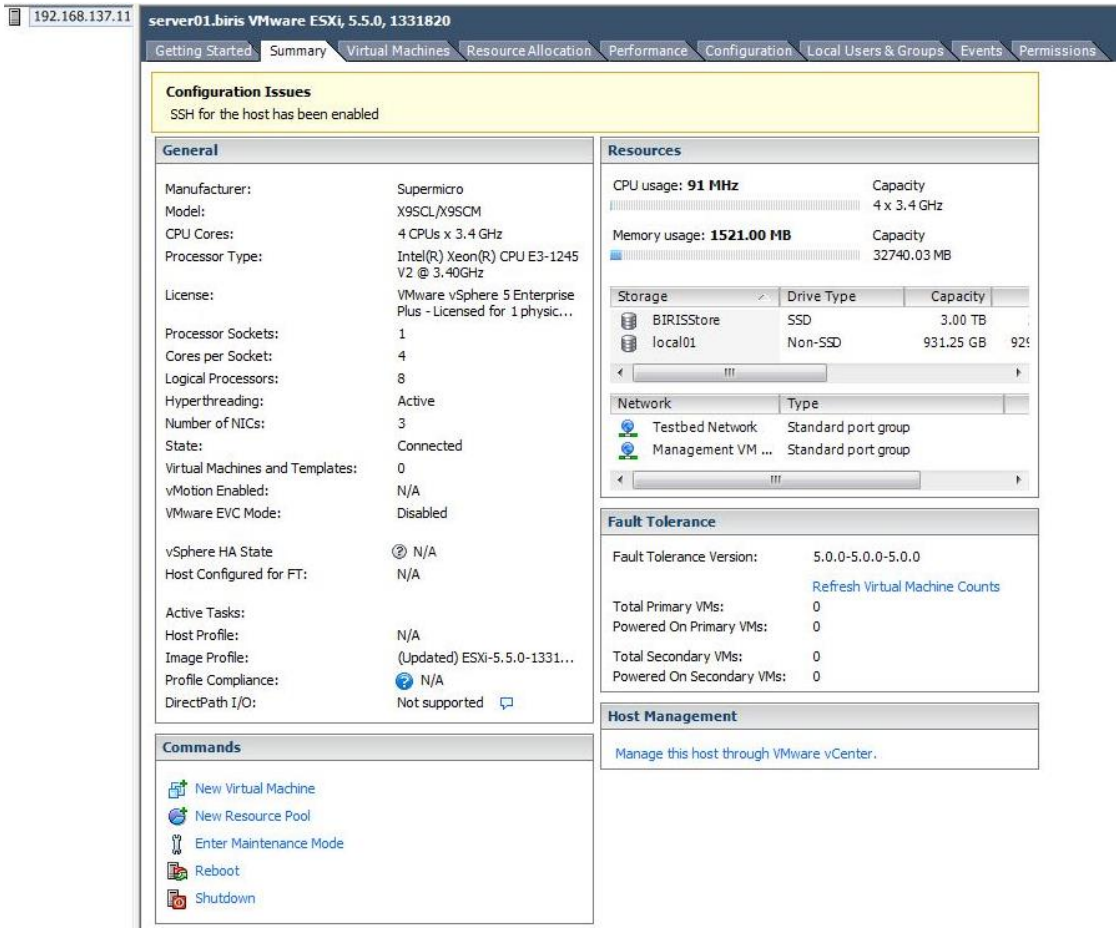

Figure 3.5 vSphere Web Client Interface to ESXi Machine

Through the vSphere Client interface shown in Figure 3.5, a virtual appliance (vApp) was deployed on *server01*. vApps are simply VMs with a preconfigured operating system and settings. This particular Linux-based vApp was loaded with the vCenter Server management software, which provides a management interface at the datacenter level, as opposed to the host level the ESXi operating system provides. The vCenter Server VM was provided a network interface and IP address for the testbed network, so that administrators of the datacenter could access and modify various components of the datacenter. Shown in Figure 3.6 is the web client for vCenter Server, with all five host servers configured and added to the datacenter. The

vCenter Server vApp is shown to be currently hosted by server01, though the management software allows VMs to be migrated to other hosts or datastores at will.

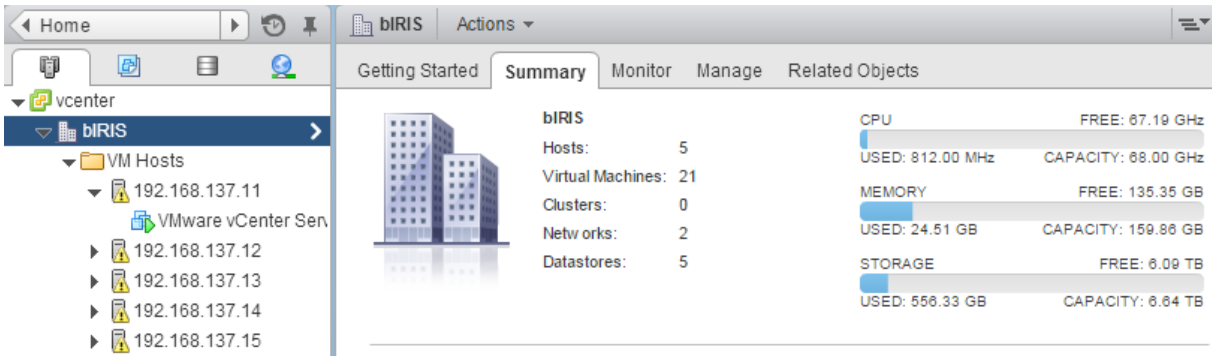

Figure 3.6 vSphere Web Client Interface to vCenter Server

An additional VM with hostname *localdns* was built to offer DHCP and DNS services to the testbed network. Through the DHCP server, the IP address of every Testbed network interface (whether a robot, PC, or virtual machine) was defined in a single location, as opposed to each robot, PC, and virtual machine deciding its own IP address. This method of reserving IP addresses for particular devices is called address reservation, and this ensured the IP address of each device would remain constant unless changed by the administrator. The DHCP server also provided dynamically generated IP addresses for those devices which were not recorded in the reserved address list, so that new devices could be added to the network without altering the DHCP server files.

Additionally, a DNS server, running on the same machine, translated the static IP addresses to names, and vice versa. By using DNS, devices could be referred to by name as opposed to IP, which is much harder to remember. Furthermore, when referring to a device by DNS name, the IP address of a device could be changed, and as long as the DNS entry was updated, there would be no lack of connectivity due to the address change. The Testbed network was defined to have a domain name of *biris*. The addition of a domain name allowed for the easy logical segregation of

outgoing connections, searching for addresses ending in *biris* back at the local networks Testbed, Storage, and Management hosts from those on the internet. Thus, the fully qualified domain name (FQDN) of a bIRIS networked host would look like *server01.biris*.

3.3 bIRIS Testbed Network

The Testbed network consisted of many different systems and devices, including robots, cloud resources not managed by the datacenter, control systems and the VMs they were hosted on, and the client computers through which users could access the Testbed network. The physical resources for the datacenter, as well as a sample of the robots used for this study, are shown in Figure 3.7.

3.3.1 Non-Virtualized Resources

For intense calculations, specifically for neural network control systems, two physical computers were provided with Compute Unified Device Architecture (CUDA)-Enabled graphics processing units (GPU). These NVIDIA Tesla K20c GPUs provided the computational resources for running neural network simulations, which run far more efficiently on GPUs than CPUs. These two machines were assigned hostnames of *cuda1* and *cuda2* (FQDN *cuda1.biris* and *cuda2.biris*) and were loaded with the Ubuntu operating system along with the ROS software, like the VMs. CUDA software was also installed on *cuda.biris* and *cuda2.biris* to provide local access to the GPUs' computational ability.

3.3.2 Robots

Three types of robots were used in the experiments conducted as part of this work: the Parrot AR.Drone2.0 UAV, the Kobuki Turtlebot 2, and the LEGO EV3. These robots were used to acquire data and interact with the environment using the mobility that mounted sensors and

actuators would not offer. The configuration of each type of robot is detailed in Sections 2.6 through 2.8 .

Figure 3.7 The bIRIS Datacenter and Target Robots

3.3.3 Virtual Machines

Many VMs were built in the datacenter to host software dedicated to interfacing the robots with the ROS framework or providing cloud services to the robots. These VMs had a network interface on the Testbed Network and were statically assigned IPs through the DHCP server, as well as domain names through the DNS server. Furthermore, a remote desktop protocol (RDP) application was installed on these VMs to provide users – through the physical computers on the Testbed Network – access to the VMs to develop and test software.

3.3.4 Client Computers

Several physical computers were placed on the Testbed Network to allow users' access into the network to configure robots and run software located on VMs. RDP software was installed on these computers to be used to access the VMs. These client computers were outfitted with a NIC such that the computers would maintain their connection to the campus network while also maintaining a connection to the bIRIS Testbed Network.

3.4 ROBOT OPERATING SYSTEM (ROS)

ROS was implemented in the cloud to provide the framework on which robot communication was built. To implement ROS in the cloud, first a Linux Ubuntu VM was built to run the core ROS functionality. This VM was named *roscorevm.biris* and was installed with a minimal version of ROS that did not include any GUI tools but did include the base communication libraries. This VM would not be used for developing software, but only for directing ROS traffic throughout the network.

Next, a VM was built for the purpose of becoming the template from which figure ROS VMs would be cloned. This VM was installed with Linux Ubuntu and the full desktop version of ROS, which included many ROS packages such as simulators, navigation, and visualization tools. The ROS development workspace was set up to fast-track development of ROS software. Source

control software was installed to enable easily distributable code and tracking of code changed. RDP software was installed to provide access to the VM from client computers on the network. Finally, the VM was converted to a template.

The vCenter Server software allows us to make use of VM templates. These templates are VMs that cannot be turned on, but can quickly be cloned into operational VMs. VM Templates can have additional software installed and files stored on the computer, as well has have a custom hardware configuration. The previously created template was used as the foundation of all VMs that provided cloud robotics services through ROS.

3.5 AR.DRONE2.0 SETUP

3.5.1 AR.Drone2.0 Wireless Configuration

The networking configuration of the AR.Drone2.0s (shown in Figure 3.8) required considerable modification. First, the default state of the UAV after booting is to create a wireless ad-hoc network, such that a single device can connect to the drone to control it. For this study, the UAVs had to be reconfigured to connect to the bIRIS Testbed Network. Furthermore, software had to be pushed to the UAV to enable it to connect to secure wireless networks via the Wi-Fi Protected Access II (WPA2) protocol (Araos 2013). Once this was complete, the UAV was ready to connect to the secure bIRIS Testbed Network.

Figure 3.8 AR.Drone2.0

To connect each UAV to the bIRIS Testbed Network, a virtual machine with a wireless adapter connected to the ad-hoc network each AR.Drone2.0 creates at boot up. A script is then executed in the virtual machine that interfaces with the connected UAV via the telnet protocol. The script shuts down the AR.Drone2.0's DHCP server, turns off the ad-hoc networking mode, and provides the SSID and passphrase to connect the UAV to the network. Once connected to the network, the DHCP server on *localdns.biris* recognizes the UAVs hardware address and assigns the preallocated IPv4 address to it, completing the process of connecting the AR.Drone2.0 to the network.

3.5.2 AR.Drone2.0 ROS Driver

As the case is with many robots, a ROS driver for the AR.Drone2.0 has been developed and is available for general use under the Berkeley Software Distribution (BSD) license. The *ardrone_autonomy* (Monajjemi 2015) package provides an extensive ROS driver for the AR.Drone2.0 that publishes both raw and estimated sensor data and allows for remote configuration of the UAV, including motor speed, camera selection, LED animation, and other settings. This package was used to interface the AR.Drone2.0s with the ROS framework in this study.

3.6 TURTLEBOT SETUP

3.6.1 Turtlebot Netbook Installation

The Turtlebot's netbook was loaded with Ubuntu Linux and each turtlebot was provided with the *turtlebot* hostname, numbered sequentially, i.e., *turtlebot01*, *turtlebot02*, *turtlebot03* and *turtlebot04*. A Turtlebot is shown in Figure 3.9.

Figure 3.9 Turtlebot

3.6.2 Turtlebot Wireless Configuration

Because the Turtlebot shipped with a netbook, integrating the Turtlebot with the network was a much simpler matter than the AR.Drone2.0. The Turtlebot's native wireless interface was connected to the bIRIS Testbed Network's using the network's SSID and each Turtlebot was provided the password.

3.6.3 Turtlebot ROS Driver

The full ROS software package was installed on the Turtlebot, along with the Turtlebot-specific ROS packages necessary for interfacing with the robotic base. Additionally, ROS-OpenNI software was installed for interfacing with the Turtlebot's Kinect camera.

3.7 LEGO EV3 SETUP

3.7.1 LEGO EV3 Installation

The EV3 (shown in Figure 3.10) has its own firmware but with the addition of a MicroSD card could support a Linux Debian operating system called *ev3dev*. A virtual machine was built to modify the ev3dev operating system from within the Linux Ubuntu environment. This was done so that software could be installed on the EV3 brick, which runs on an ARM 9 processor and requires software to be built from source to run on that processor. Through virtualizing the ev3dev OS before implementing it on the brick, the source code for the desired software could be stored outside of the MicroSD card, which had limited space. Once the software was built, the source code was no longer necessary to keep on the local storage.

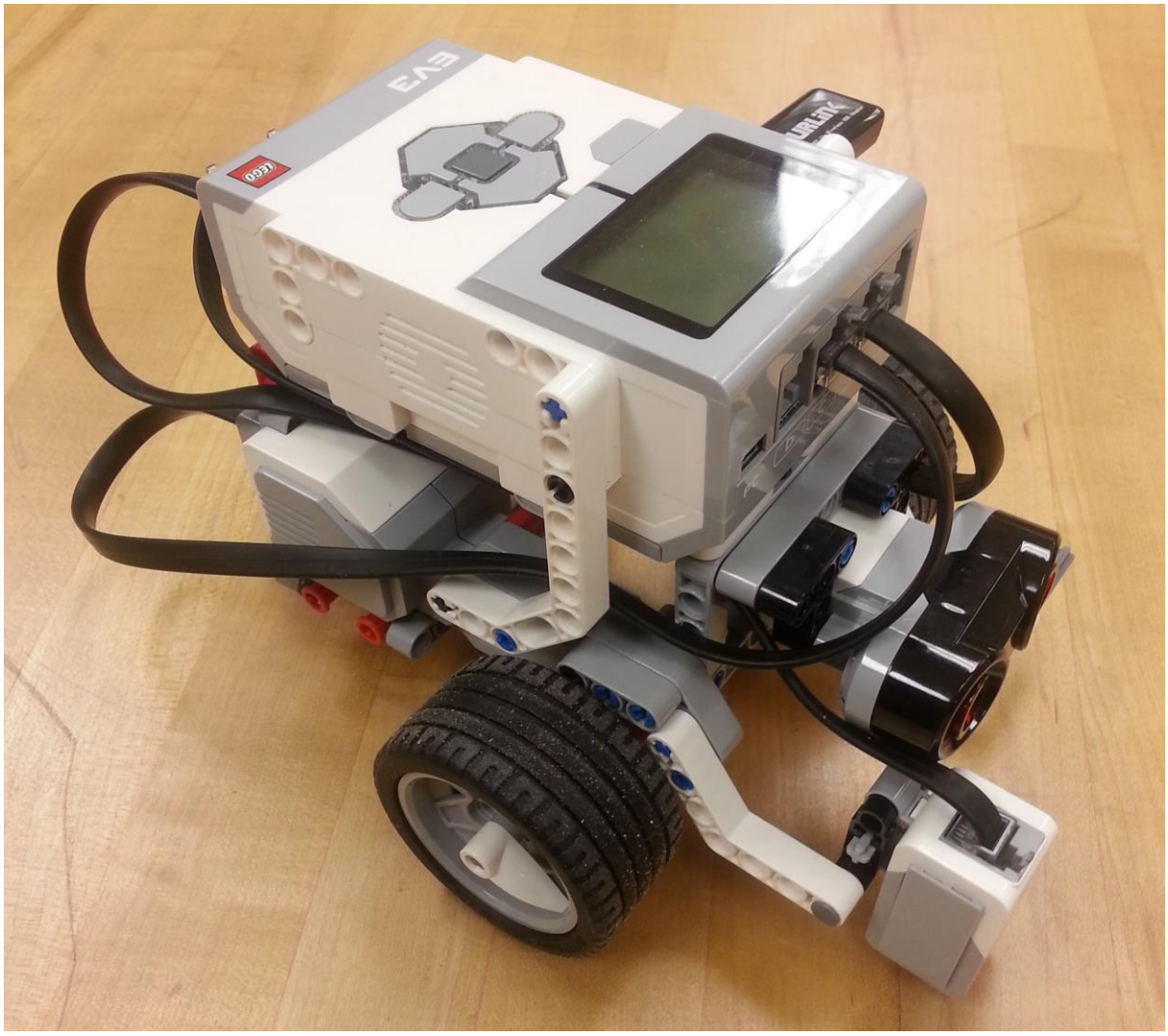

Figure 3.10 LEGO EV3

A minimal version of ROS was built from source and installed on the ev3dev system, which included the standard ROS communication packages as well as the *geometry_msgs* package necessary for describing the velocity of the EV3. Once the software was built, the virtual operating system was converted to an image and flashed to the MicroSD card, which was inserted into the EV3 brick.

3.7.2 LEGO EV3 Wireless Configuration

A wireless USB adapter was connected to the EV3 brick to provide the capability to connect to wireless networks. Like the Turtlebot, the software for connected to wireless networks was already provided through the ev3dev operating system, and so the SSID and password was simply entered through the brick's interface, and the brick was connected to the bIRIS Testbed Network.

3.7.3 LEGO EV3 ROS Driver

No existing ROS driver for the EV3 could be found, despite instructions existing for installing ROS on the EV3. Thus, a simple ROS driver was built using the python-ev3 project on GitHub. This project provided Python libraries for interfacing with the EV3 brick, which in turn interfaced with the various sensors and motors LEGO provides. As one of ROS's compatible languages is Python, integrating the two was a simple matter. A simple ROS driver was built that would acquire data from the LEGO Color Sensor and publish it to the ROS topic of *ev3/light*, while the driver listened for messages published to the *ev3/cmd_vel* messages and converted them into motor commands to move the EV3.

3.8 EXPERIMENTAL DESIGN

Multiple control and data acquisition systems utilizing multiple types of robots were designed and implemented within the test environment. Experiments were conducted for each type of robot under three conditions:

- **Baseline:** A connection was established between the target robot and the Testbed Network. The only network communication occurring between the robot and network was due to the network services offered (i.e. DHCP and DNS information).

- **Low Bandwidth:** A ROS *geometry_msgs/Twist* message was published to the target robot at a rate of 10Hz to control the robot's velocity, while low-bandwidth feedback (e.g. light or ultrasonic sensor readings) was transmitted back to the cloud.
- **High Bandwidth:** Again, the *geometry_msgs/Twist* control message was published to the target at 10Hz, while high-bandwidth feedback (e.g. color and depth images) was transmitted back to the cloud.

For each of these experiments, latency and packet loss tests were conducted as additional robots were added to the system to measure the network integrity and the viability of robot operation utilizing cloud resources. Tests were run on weekdays during business hours to measure performance when the ambient wireless interference levels were high due to the high number of wireless devices in use. Tests were also conducted on weekends to measure performance during low wireless usage periods.

Experiments were conducted as field tests; that is, the testing environment conformed to the manner in which an actual heterogeneous networked system would be implemented in a real world environment. To measure the performance of the network as various systems were added, a network performance baseline had to be established, featuring the minimal amount of devices necessary to define the environment. Network performance was evaluated based on three measures:

- Network throughput, the speed at which data is transferred between devices,
- Network latency, the amount of time a data packet is in transition, and
- Dropped packets, the percentage of data packets that do not arrive at their destination and must be retransmitted.

Network throughput was tested by transferring a large file between a Linux Ubuntu VM built for the network tests, and the target device. Size of the file divided by the time required to complete the transfer provided the throughput value for that connection. The network latency and dropped packets were tested utilizing the *ping* program on the same VM. A total of 600 pings were sent to the indicated target at one second intervals, which made for a ten minute test of the target device's network performance. The average latency experienced by a sample robot is plotted for each test case and the statistical parameters of the experiment are reported in a following table. For the calculation of the statistical parameters, periods of abnormally high latency were ignored as they represented environmental interference and were not representative of the robot's effect on the network. These periods of interference were not removed from the presented graphs, however.

CHAPTER 4: EXPERIMENTAL RESULTS AND DISCUSSIONS

4.1 BASELINE NETWORK TESTS

A test of the wired bandwidth of the network was conducted to determine the maximum bandwidth the network was capable of supporting on a single interface. The wired network bandwidth far exceeds the wireless network bandwidth, but it is useful to know the theoretical limitations of the wired components of the system. The maximum bandwidth data is shown below in Figure 4.1.

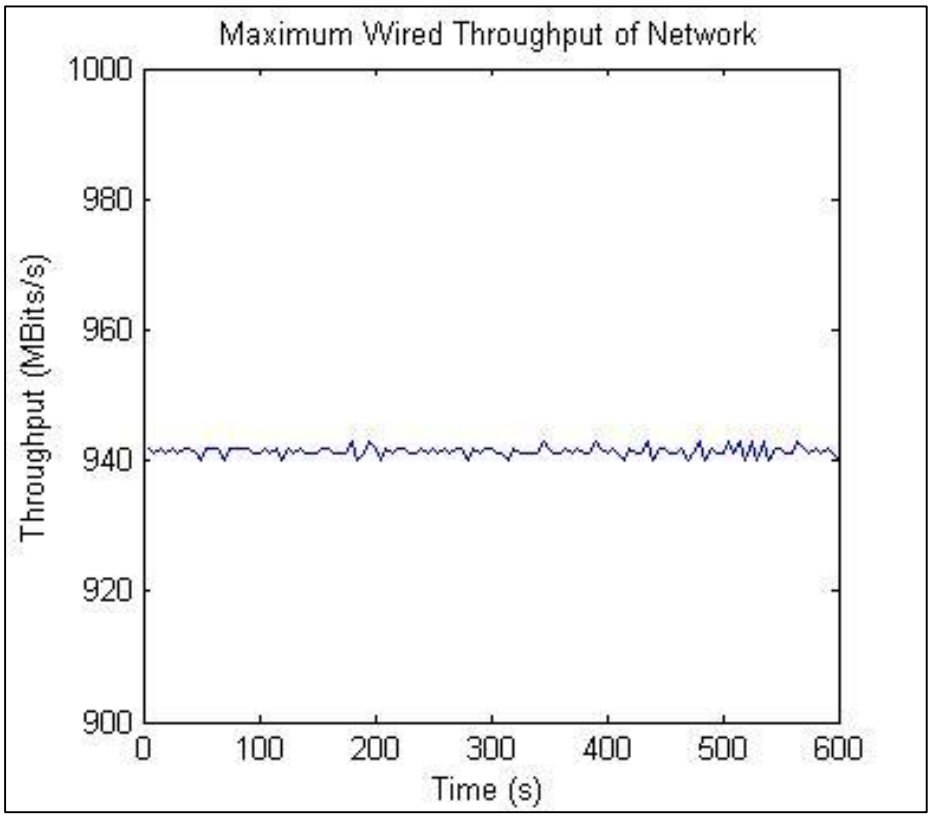

Figure 4.1 Maximum Wired Throughput of Network

Maximum wired throughput of the wired network was found to be 941.4 Mbits/second. Given the 1Gbit/s specification of the hardware used (including the Ethernet switch and Ethernet ports of the devices used), this maximum throughput was expected and deviates from the theoretical capacity of 1Gbit/s by 5.86%. This is the maximum speed the wired network will

achieve under unloaded conditions. Baseline wireless tests were conducted for individual robots instead of the whole system as wireless connectivity is more dependent on the quality of the wireless interface and would vary from robot to robot even under ideal experimental conditions.

4.2 AR.DRONE2.0

First, a baseline test (shown below in Figure 4.2) was conducted by testing the latency between the network and an AR.Drone2.0 connected to the network, but without any system sending to or receiving data from it.

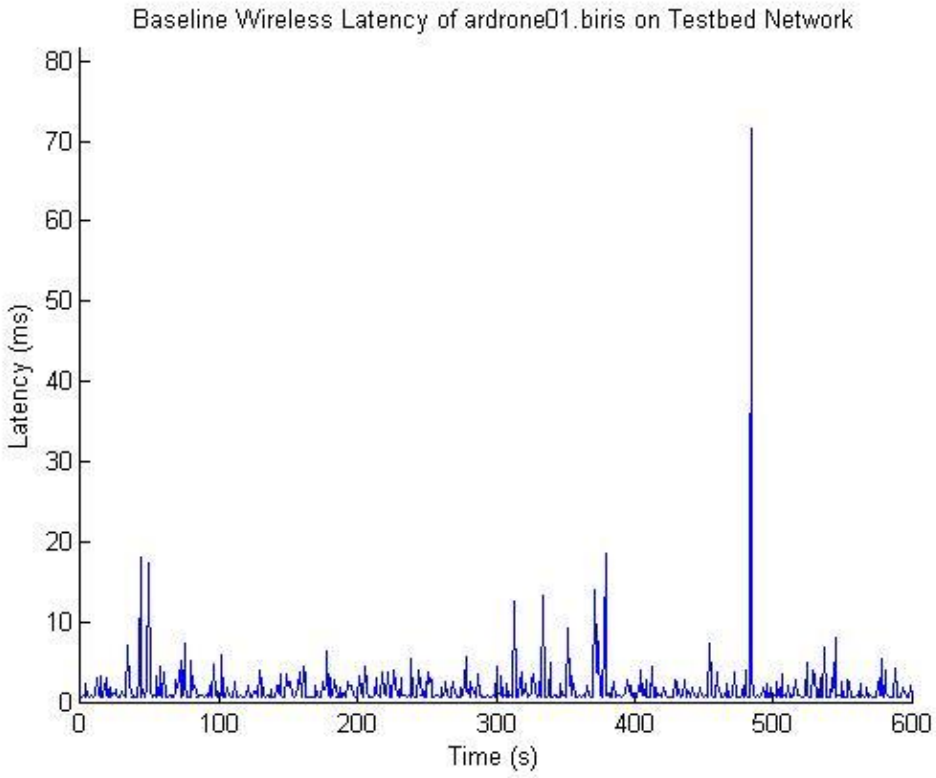

Figure 4.2 Baseline AR.Drone2.0-to-Cloud Latency

Baseline wireless latency tests produced an average of 1.40ms latency with a standard deviation (s) of 1.32ms and zero lost packets for the duration of the ten minute test. Thin spikes in latency are evident in the graph, which result from the wireless interference of the many campus wireless access points and their client devices present in the building.

4.2.1 AR.Drone2.0 ROS Driver

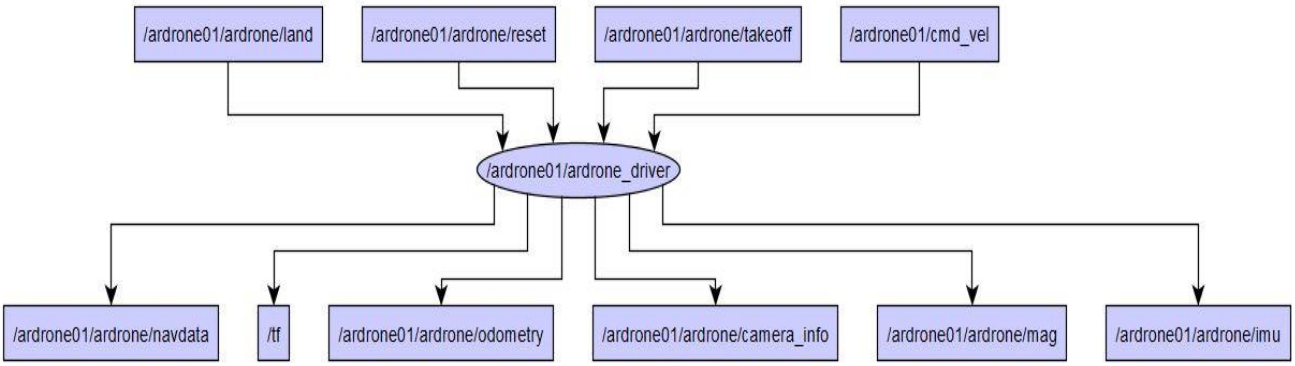

Figure 4.3 AR.Drone2.0 Driver ROS Computational Graph

Next, the driver was launched in the console with command line arguments to specify the IP address of the target AR.Drone2.0 and to push all the ROS computations into a specified namespace. The latter step is necessary to properly segregate the data transfers of this ROS node from the data transfers of other ROS nodes of the same name, i.e. when multiple *ardrone_driver* nodes are running other AR.Drone2.0s simultaneously. The ROS computational graph for the *ardrone_driver* node is shown in Figure 4.3, though many of the 43 topics published or subscribed to by the *ardrone_driver* node have been hidden for clarity. The ellipses are ROS nodes, the executables that perform computations. The quadrilaterals are topics, and the lines between the nodes and topics indicate the flow of data between the two.

Five *ardrone_driver* nodes in total were launched to interface five AR.Drone2.0s with ROS. Though for this test, there was no data being sent through the ROS side of the interface, due to the AR.Drone2.0s firmware a constant stream of data was moving between each drone and its respective *ardrone_driver* node. This network traffic significantly impacted the network performance and led to image decoding failures as the fourth and fifth AR.Drone2.0s were added to the system.

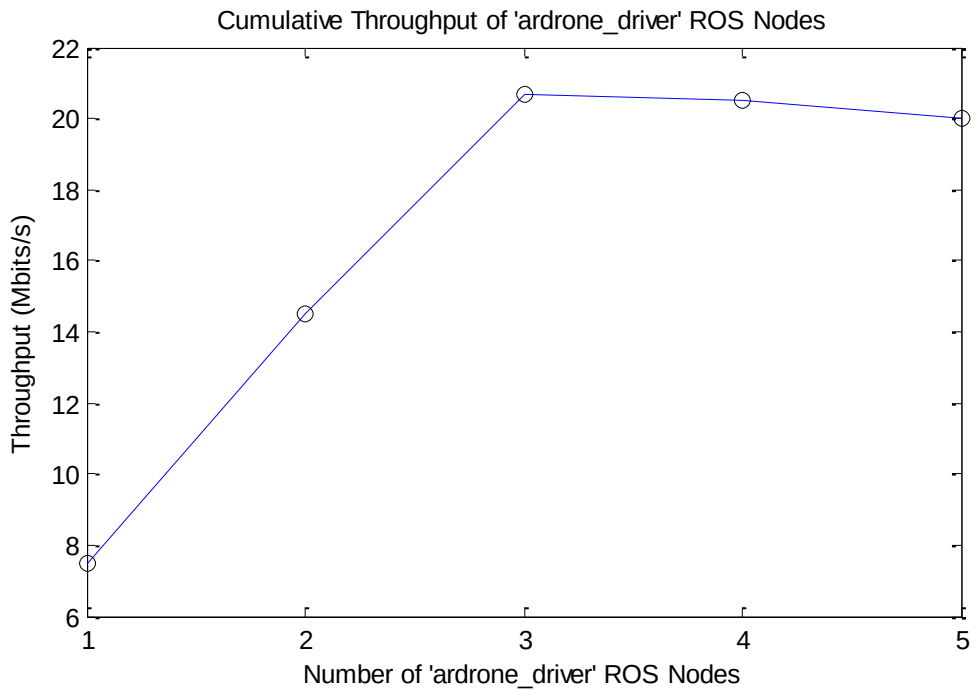

Figure 4.4 Cumulative Throughput of *ardrone_driver* ROS Nodes

Shown in Figure 4.4 is the cumulative throughput used by the AR.Drone2.0s as more are added to the system. Throughput rises linearly as expected as up to three AR.Drone2.0s were added to the system but saturates immediately afterwards as the fourth and fifth AR.Drone2.0s were added. As mentioned previously, a disruption in connection became apparent as AR.Drone2.0 four and five were added, resulting in occasional failures to decode the image streams being received from the UAVs.

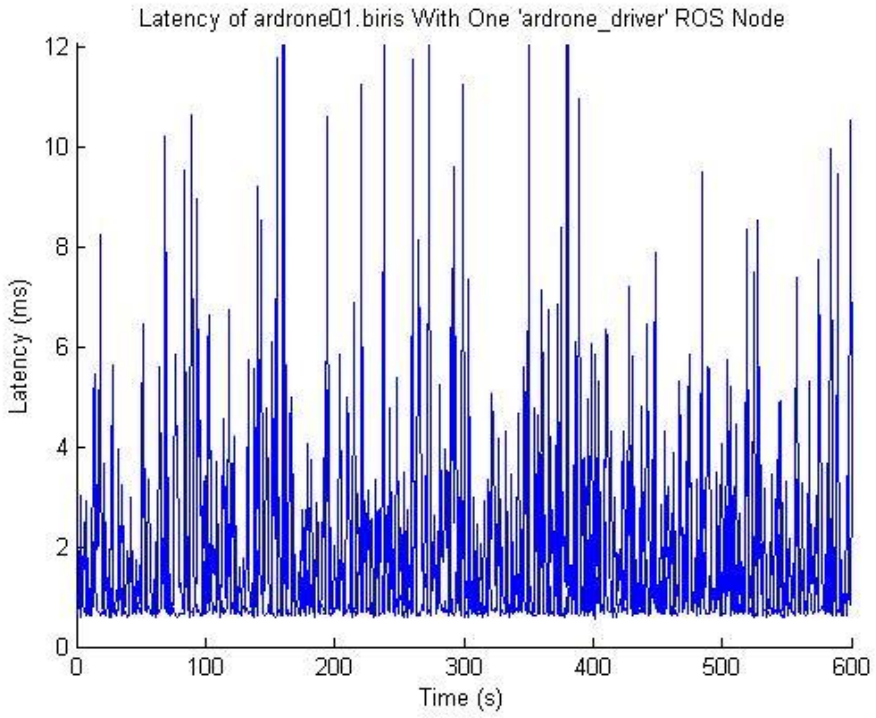

Figure 4.5 Latency of ardrone01.biris With One *ardrone_driver* ROS Node

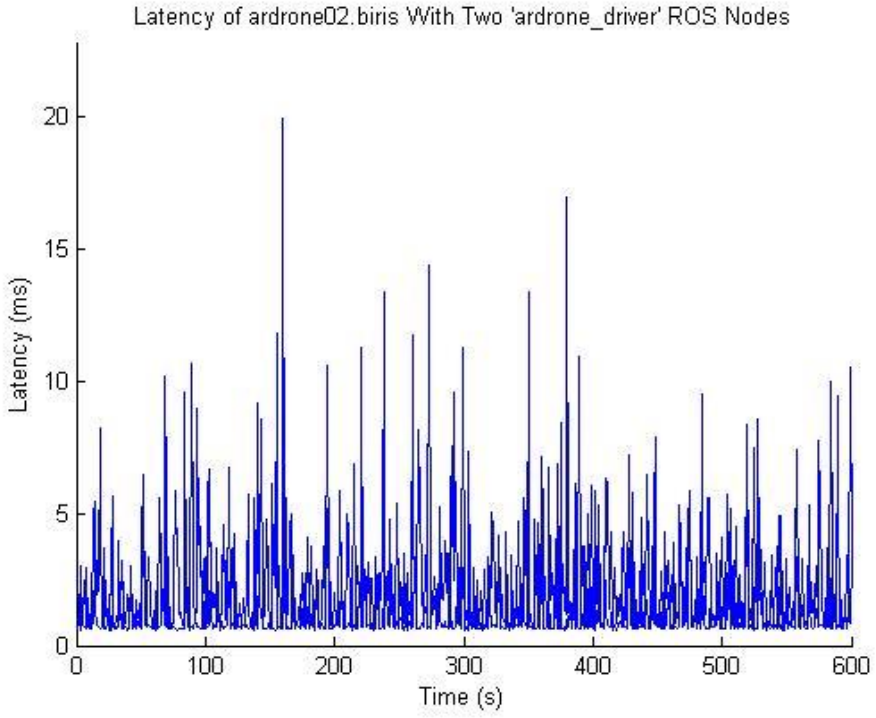

Figure 4.6 Latency of ardrone02.biris With Two *ardrone_driver* ROS Nodes

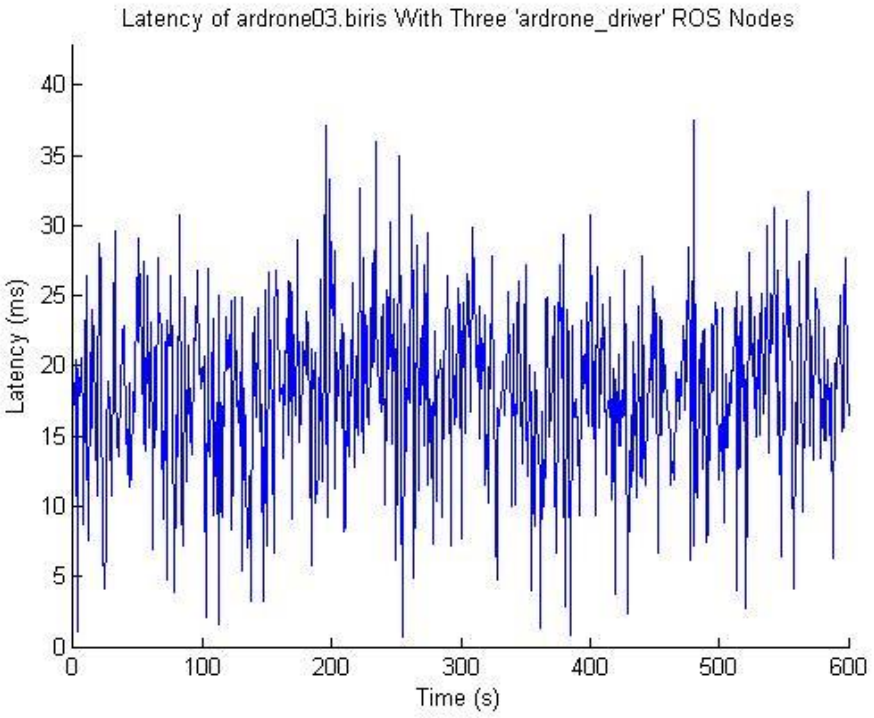

Figure 4.7 Latency of ardrone03.biris With Three *ardrone_driver* ROS Nodes

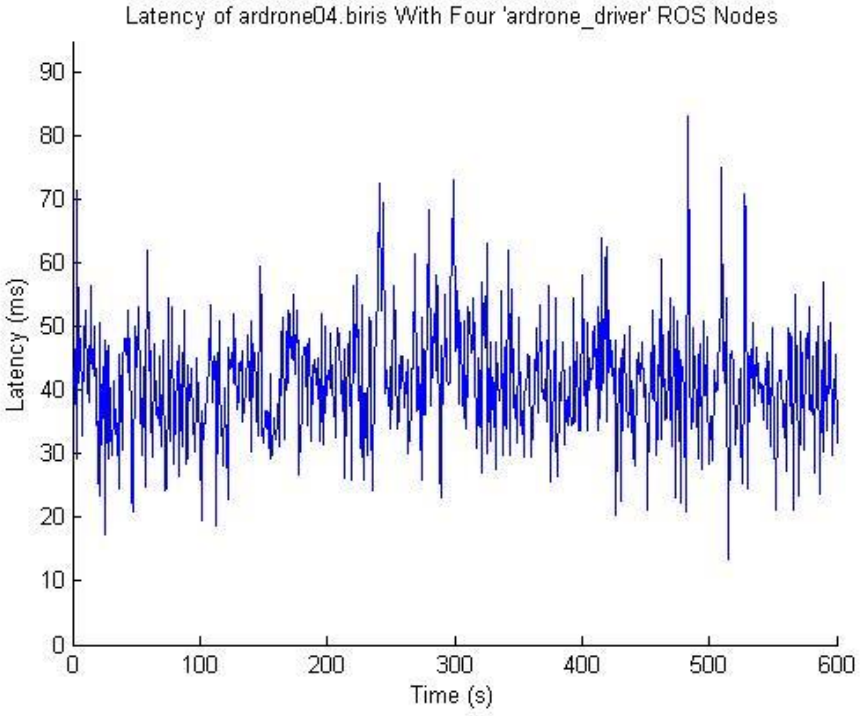

Figure 4.8 Latency of ardrone04.biris With Four *ardrone_driver* ROS Nodes

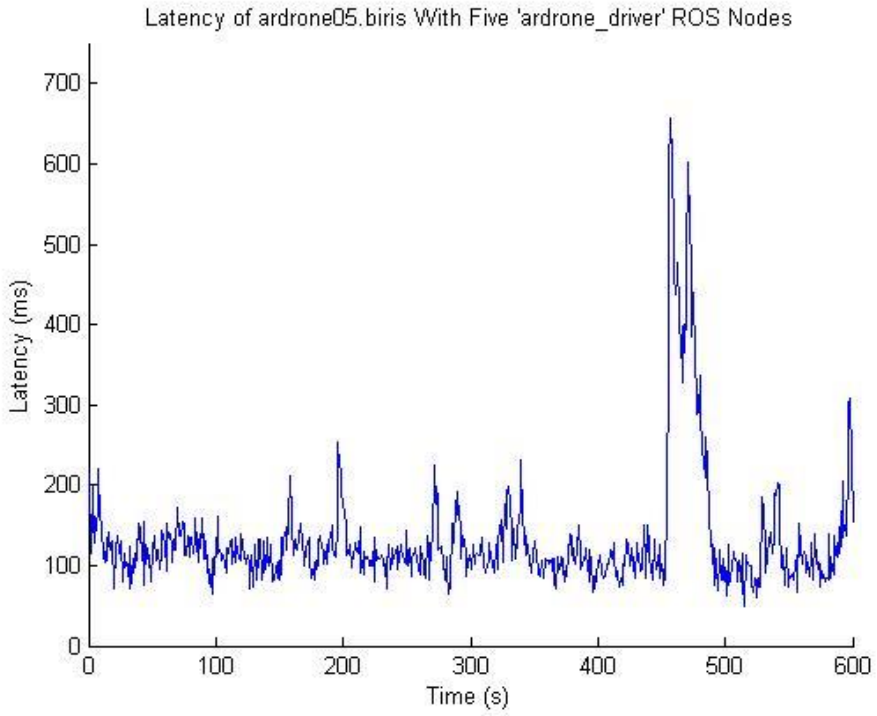

Figure 4.9 Latency of ardrone05.biris with Five *ardrone_driver* ROS Nodes

A sample of the results of the latency and packet loss tests for connecting a number of AR.Drone2.0s to the cloud and running the *ardrone_driver* ROS node for each is shown in Figure 4.5 through Figure 4.9, with the results and statistical data also recorded in Table 4.1. Each subsequent test, which features an additional AR.Drone2.0 added to the system, indicated that both the average value and scatter (standard deviation) of the latency increased between the UAVs and the cloud as more UAVs streamed data to the cloud. Latency data for each AR.Drone2.0 in each test was recorded and is presented in the table below.

Table 4.1 ROS Node *ardrone_driver* Latency Results

Number of *ardrone_driver* Nodes	Average Latency (ms)	Maximum Latency (ms)	Minimum Latency (ms)	Standard Deviation (ms)	Packets Dropped (%)
1	1.23	10.6	0.575	1.24	0.00%
2	2.29	20.0	0.565	2.27	0.00%
3	18.2	50.5	0.635	6.61	0.00%
4	41.7	83	8.85	9.95	0.00%
5	123	387	45.9	39.3	0.03%

A graph of the average latency of the group of robots as more AR.Drone2.0s were added to the system is shown below in .

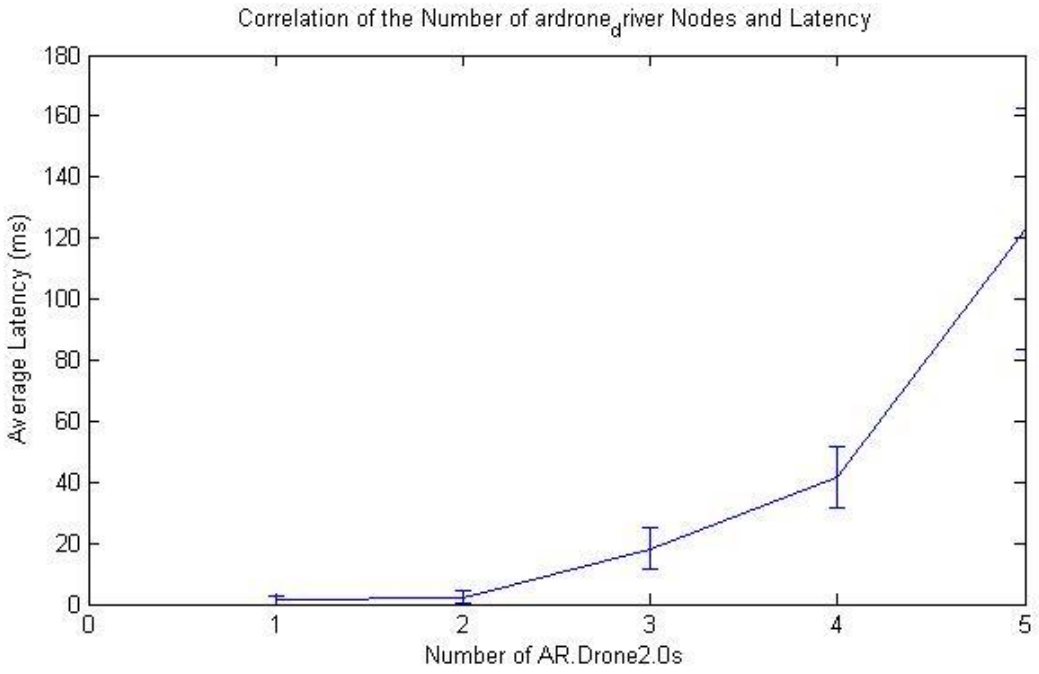

Figure 4.10 Correlation of Number of ardrone_driver Nodes and Network Latency

4.2.2 AR.Drone2.0 Low-Bandwidth Data Acquisition and Control

Next, sensor data was acquired from each AR.Drone2.0 to verify the connectivity of the UAVs with the cloud and determine how the integrity of ROS data streams was affected by network load. For each test, an *ardrone_driver* node was initialized to interface each AR.Drone2.0 with ROS. A *ros_parse* node which used ROS's command-line topic-reading functionality recorded the data stream and stored it in a file for later use. A *ros_plot* node which used ROS's graphical plotting tool *rqt_plot* plotted the data on screen as it was acquired. The simplified ROS computational graph of the setup is shown in Figure 4.11.

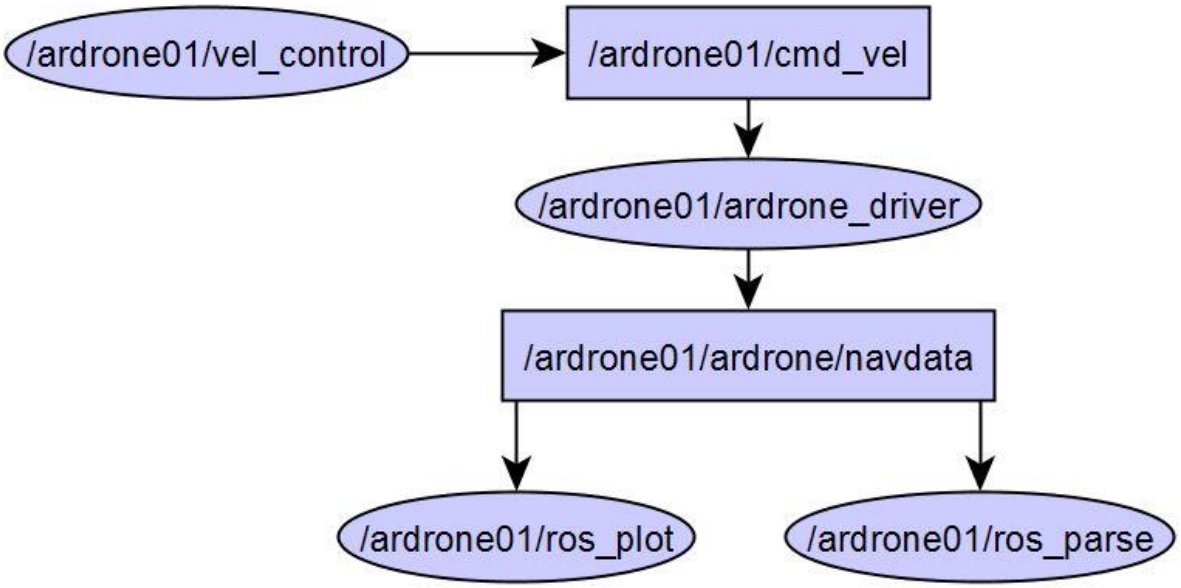

Figure 4.11 ROS Computation Graph for AR.Drone2.0 Sensor Data Acquisition and Control

The node *ardrone_driver* publishes data to the topic *ardrone/navdata*, which includes accelerometer, magnetometer, rotation, and gyroscope readings, as well as battery level, estimated wind speed, pressure, altitude, and estimated ground velocity, among other things. The *ros_parse* node read the x-axis accelerometer data published to the *navdata* topic, and parsed the output into a comma-separated-value (csv) file for later plotting. The ROS plotting tool *rqt_plot*

graphed the data onscreen in real time, which is shown in Figure 4.12. It is worth noting the absence of axis labels in Figure 4.12, as that functionality is not supported by the *rqt_plot* tool.

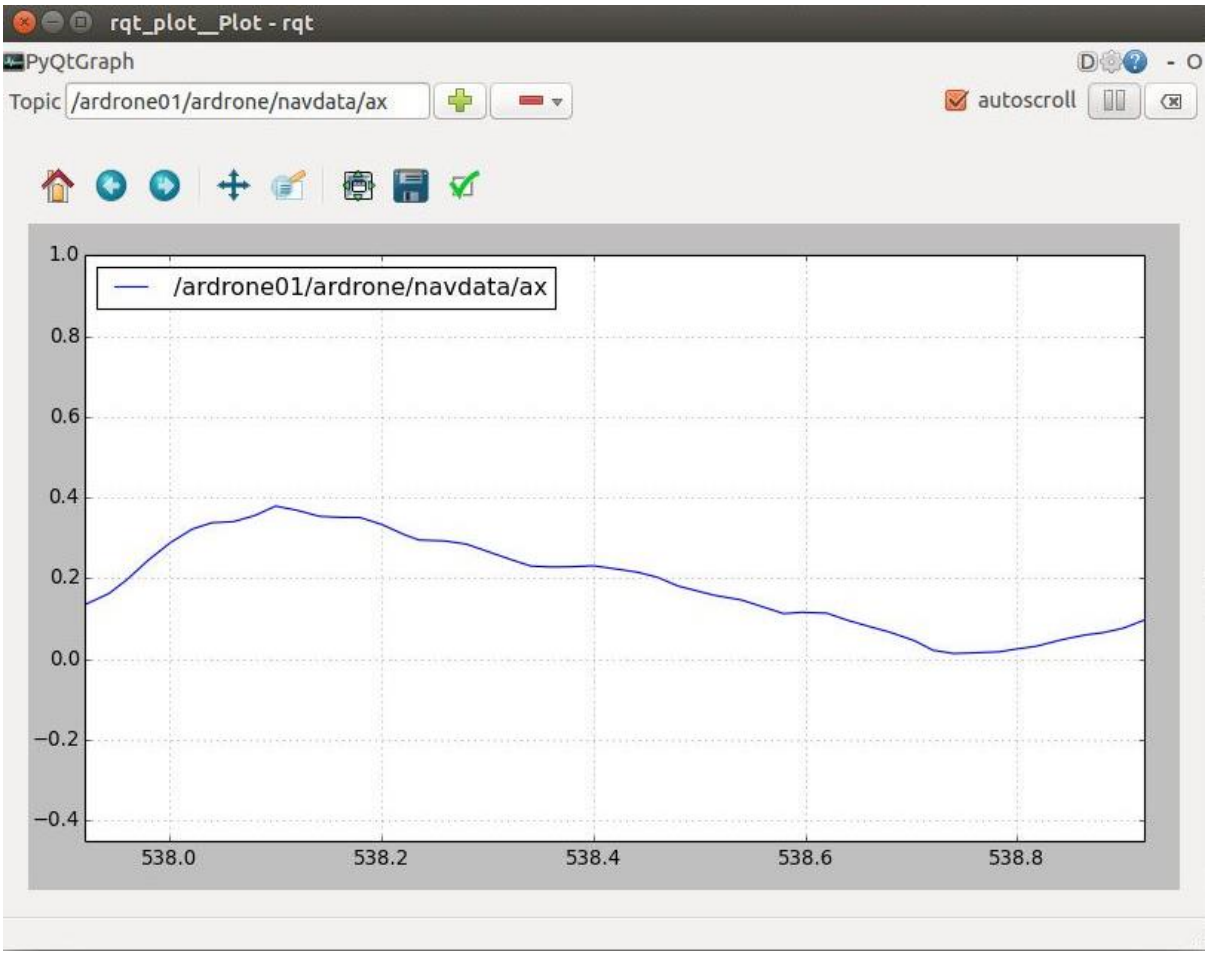

Figure 4.12 ROS *rqt_plot* plotting accelerometer data in real-time

The data stored in the csv file was imported and plotted using MATLAB; the results are shown in Figure 4.13.

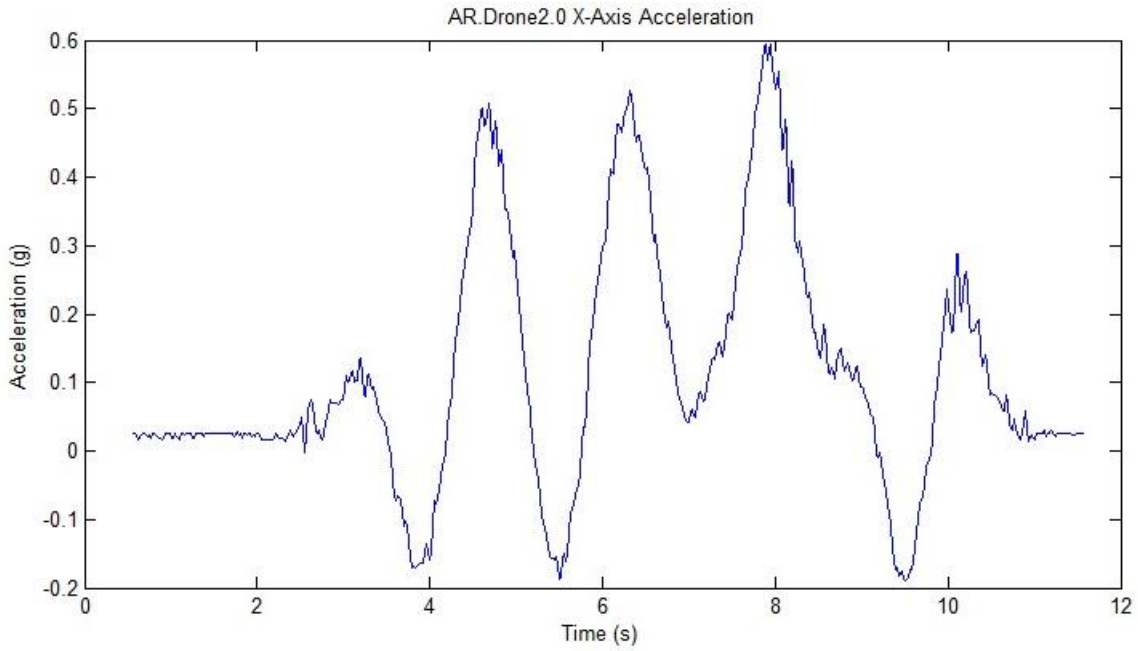

Figure 4.13 MATLAB plot of parsed accelerometer data

The results of the latency tests for adding subsequent AR.Drone2.0s to the system and acquiring accelerometer data are shown in Figure 4.14 through Figure 4.18.

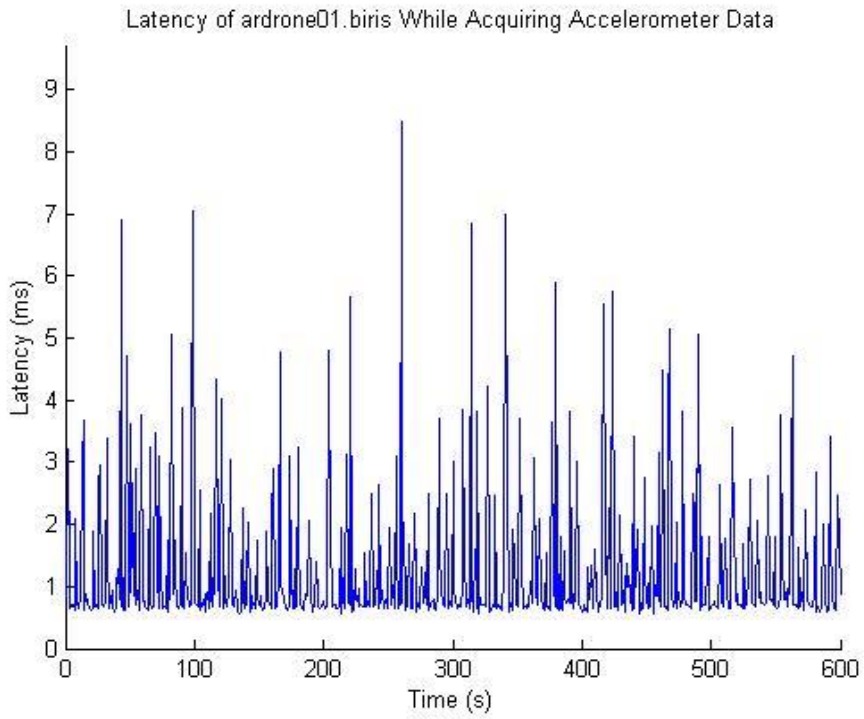

Figure 4.14 Latency of ardrone01.biris While Acquiring Data from One AR.Drone2.0

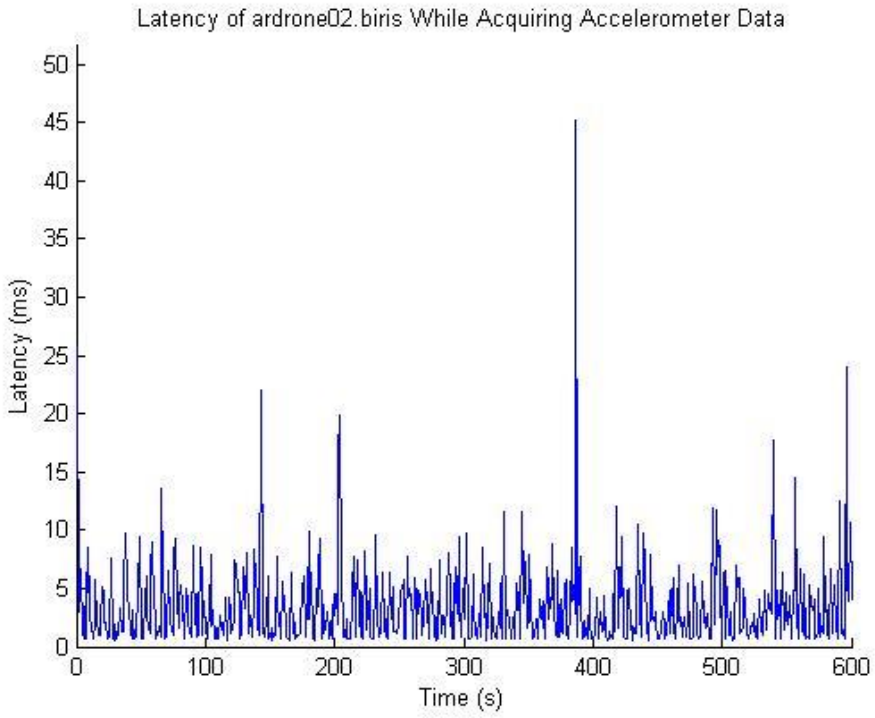

Figure 4.15 Latency of ardrone02.biris While Acquiring Accelerometer Data from Two AR.Drone2.0s

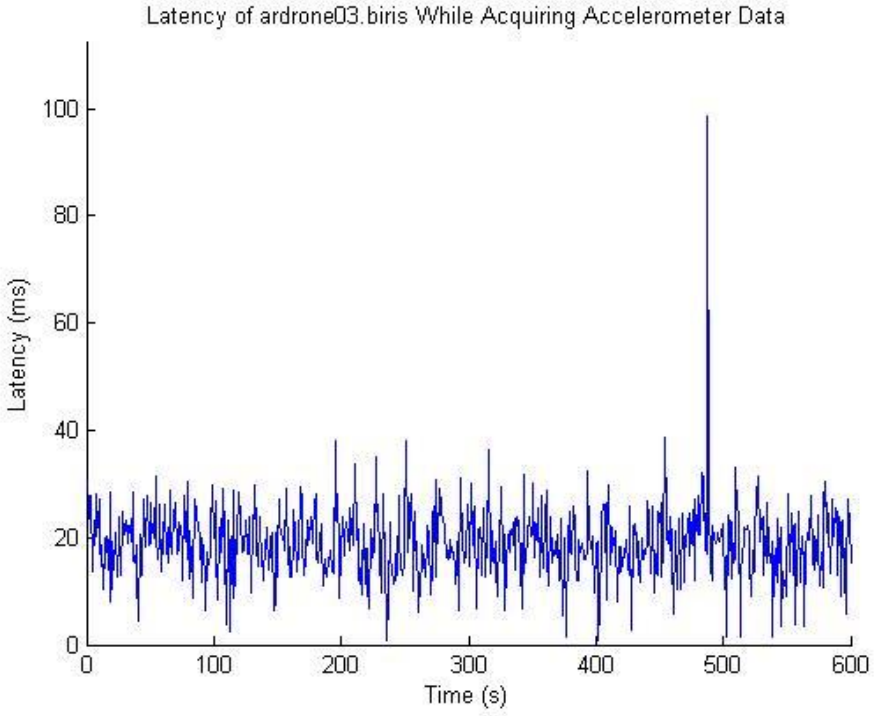

Figure 4.16 Latency of ardrone03.biris While Acquiring Accelerometer Data from Three AR.Drone2.0s

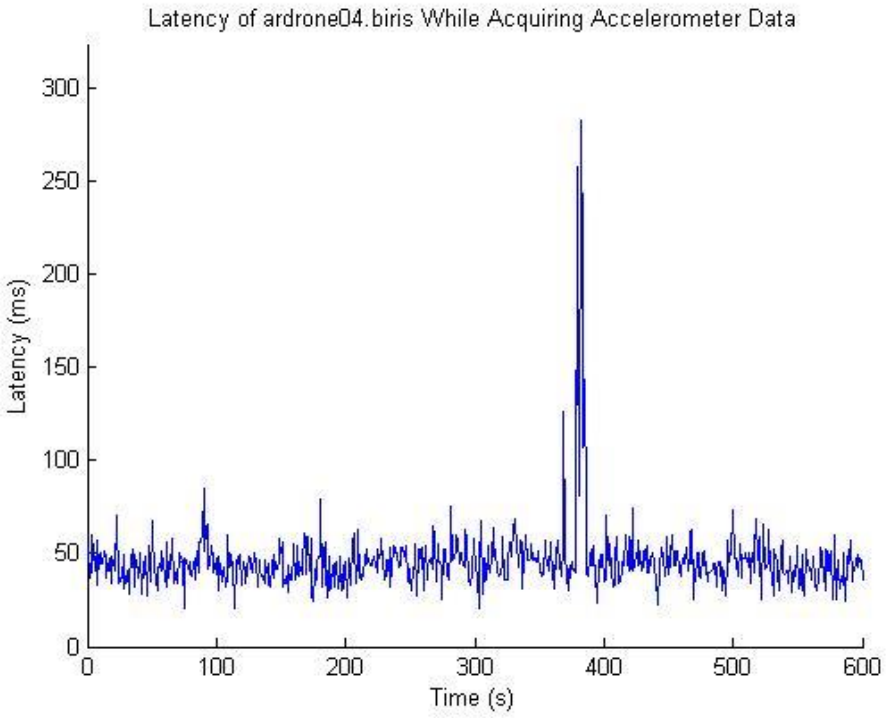

Figure 4.17 Latency of ardrone04.biris While Acquiring Accelerometer Data from Four AR.Drone2.0s

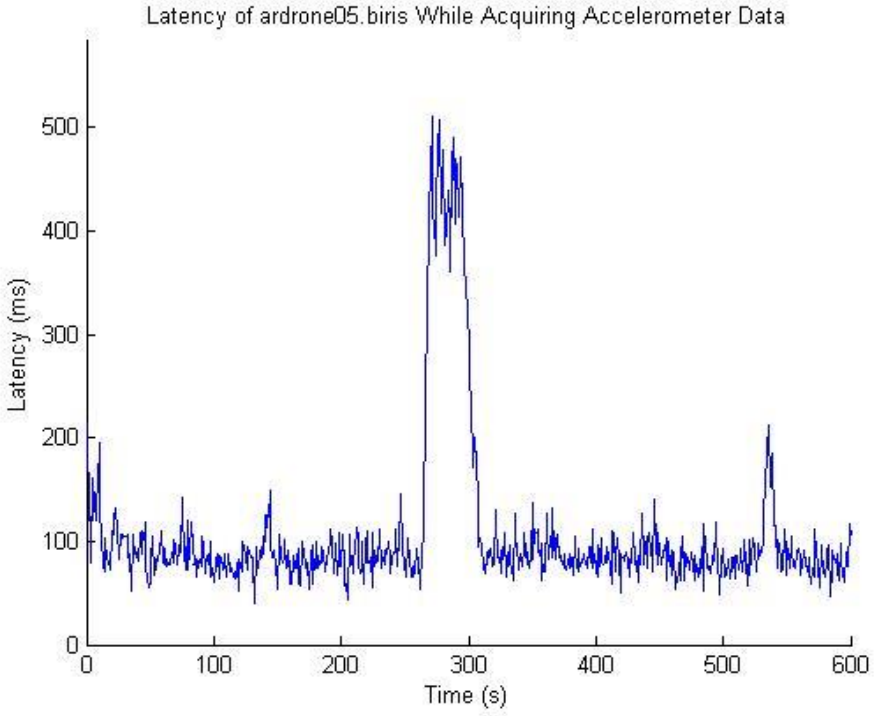

Figure 4.18 Latency of ardrone05.biris While Acquiring Accelerometer Data from Five AR.Drone2.0s

The data acquisition statistics shown in Table 4.2 indicate an exponential increase in network degradation as more acquisition nodes were run, similar to what occurred as more *ardrone_driver* nodes were started up.

Table 4.2 Network Connectivity Statistics for Low-Bandwidth AR.Drone2.0 Data Acquisition and Control

Number of Acquisition Nodes	Average Latency	Maximum Latency	Minimum Latency	Standard Deviation	Packets Dropped
	(ms)	(ms)	(ms)	(ms)	(%)
1	1.24	8.50	0.570	1.11	0.00%
2	2.82	25.9	0.555	2.86	0.00%
3	18.5	56.5	0.670	6.5	0.00%
4	43.6	90	18.5	9.91	0.00%
5	92.1	365	40.000	29.1	0.03%

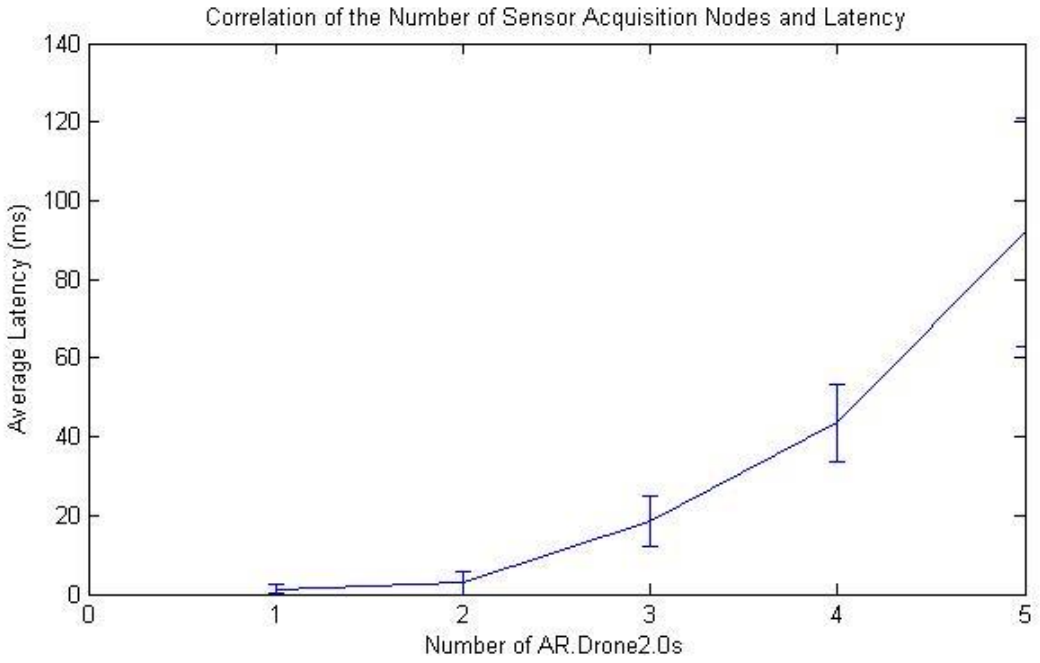

Figure 4.19 Correlation of Number of Sensor Acquisition Nodes and Network Latency

4.2.3 AR.Drone2.0 High-Bandwidth Data Acquisition and Control

For this test, image data from the front-facing camera on the AR.Drone2.0 was acquired and displayed onscreen through a virtual machine. For each AR.Drone2.0 that was added to the system, a new *ardrone_driver* and *image_view* node was initiated to acquire and display the images received from the UAV.

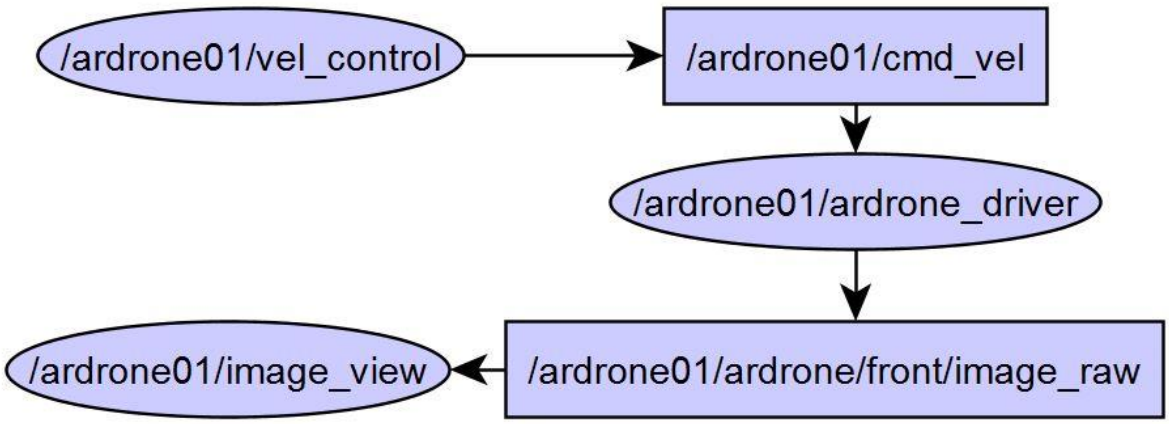

Figure 4.20 ROS Computation Graph of AR.Drone2.0 Image Acquisition

The *vel_control* node of the system in Figure 4.11 was reinitiated for these experiments. Instead of reading accelerometer data from the *ardrone_driver* node though, image data was acquired through the */ardrone01/ardrone/front/image_raw* topic and visualized with the */ardrone01/image_view* ROS node, all of which is displayed in Figure 4.20. Latency tests were run on the system as additional AR.Drone2.0s were added, the results of which are displayed in Figure 4.21 through Figure 4.25 below.

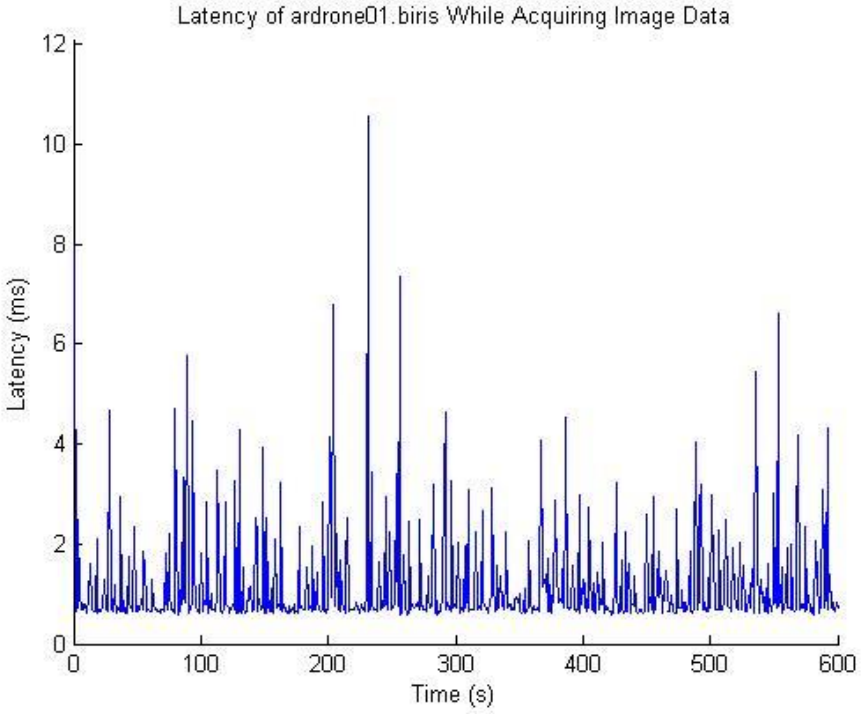

Figure 4.21 Latency of ardrone01.biris While Acquiring Images from One AR.Drone2.0

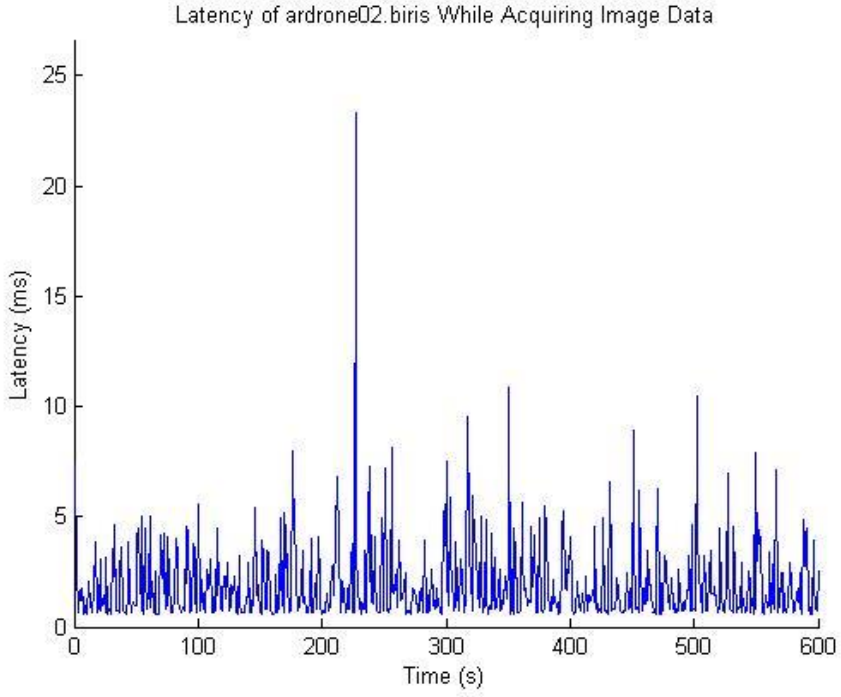

Figure 4.22 Latency of ardrone02.biris While Acquiting Images from Two AR.Drone2.0s

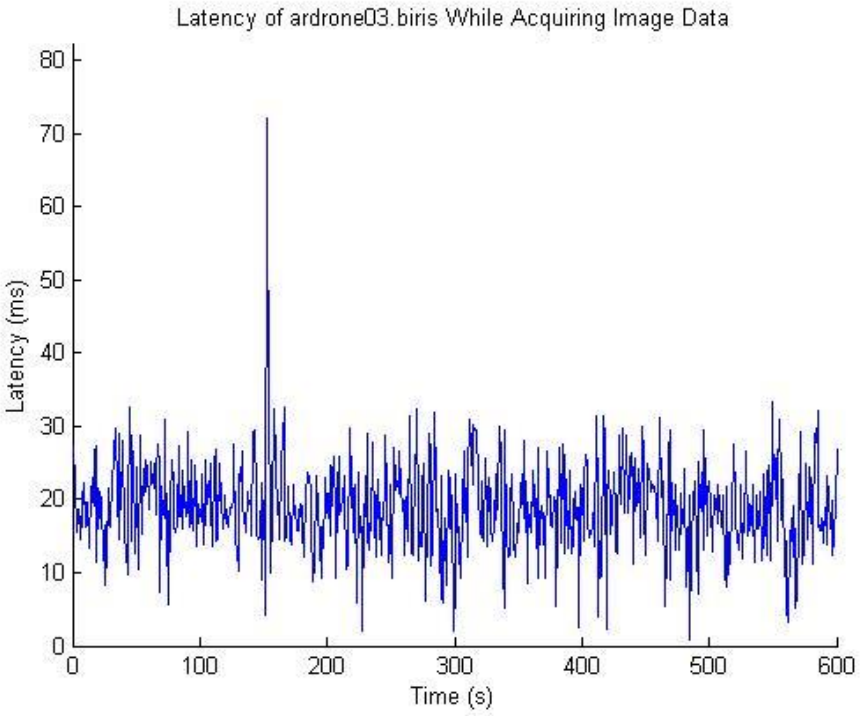

Figure 4.23 Latency of ardrone03.biris While Acquiring Images from Three AR.Drone2.0s

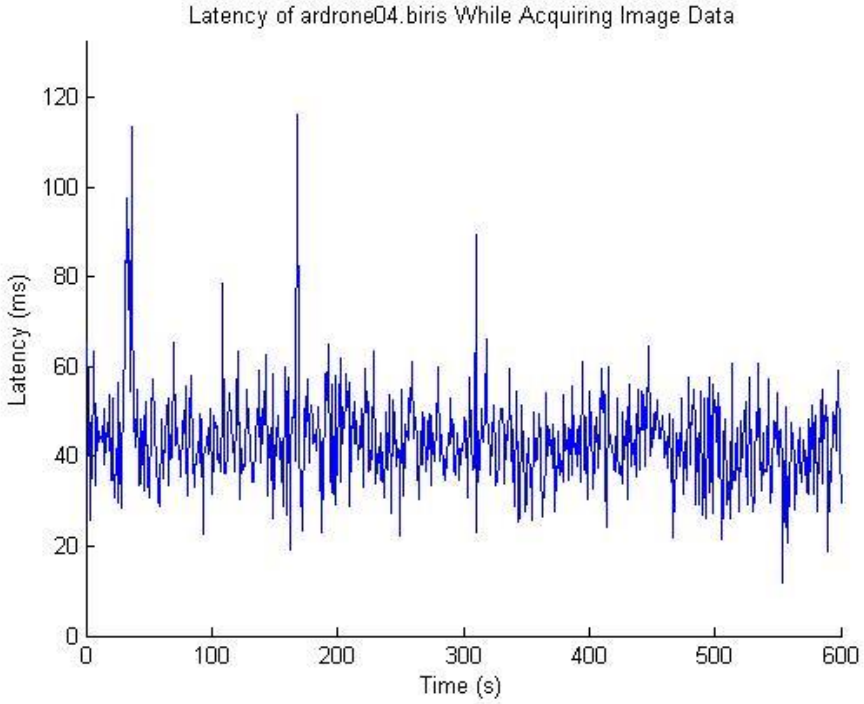

Figure 4.24 Latency of ardrone04.biris While Acquiring Images from Four AR.Drone2.0s

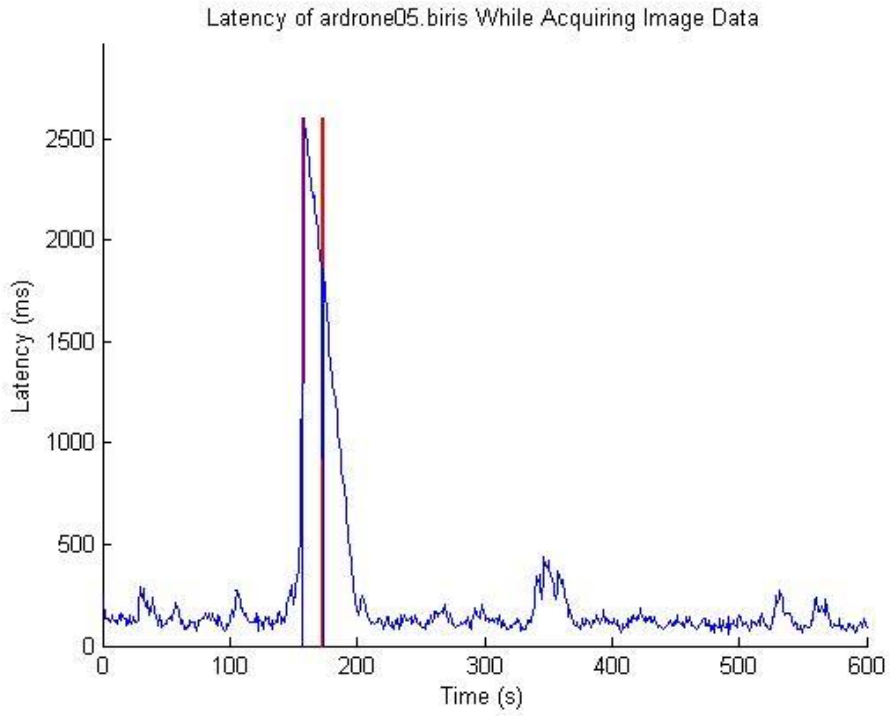

Figure 4.25 Latency of ardrone05.biris While Acquiring Images from Five AR.Drone2.0s

The average and scatter of robot-to-cloud latency increased as the additional network load of image acquisition was applied to the system. The first instance of packet loss occurred as the fifth AR.Drone2.0 was added to the system and the image acquisition nodes initialized and is shown in Figure 4.25. The thin red bars indicate the time of transmission of data packets that did not reach their destination. A total of 7 out of 600 data packets were lost, resulting in a 1.17% packet loss for this experiment. The remainder of the connectivity statistics is shown in Table 4.3, while the correlation between the number of image acquisition nodes and the network latency is shown in Figure 4.26.

Table 4.3 Network Connectivity Statistics for High-Bandwidth AR.Drone2.0 Image Acquisition and Control

Number of Acquisition Nodes	Average Latency (ms)	Maximum Latency (ms)	Minimum Latency (ms)	Standard Deviation (ms)	Packets Dropped (%)
1	1.16	10.6	0.570	1.04	0.00%
2	2.04	20.1	0.535	1.92	0.00%
3	18.7	39.6	0.630	6.26	0.00%
4	42.5	117	12.0	10.2	0.00%
5	139	478	48.8	58.3	0.23%

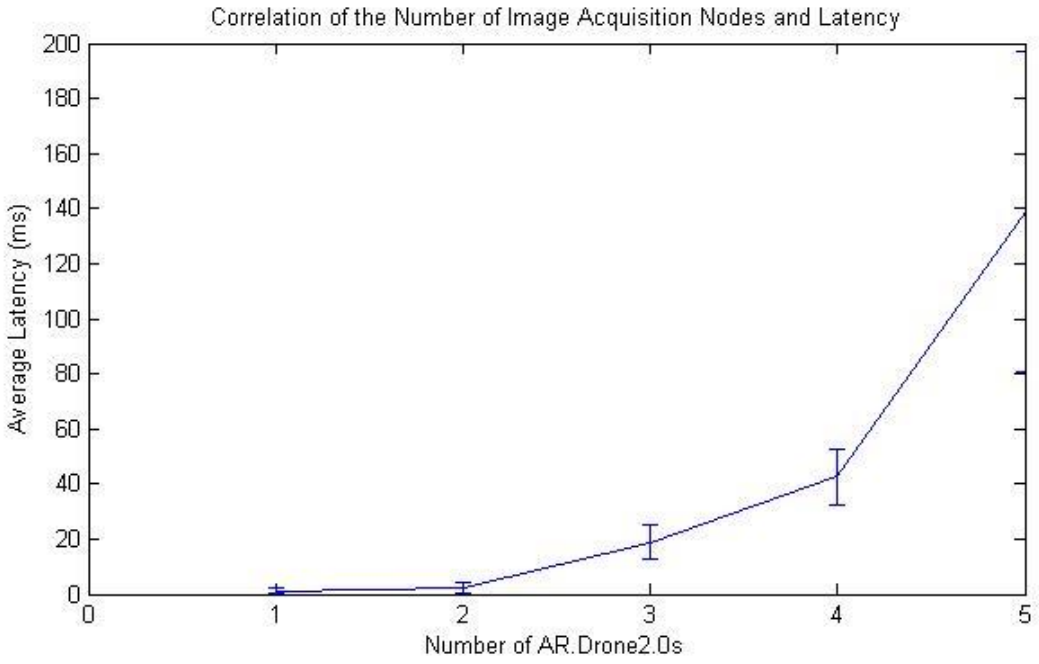

Figure 4.26 Correlation of Number of Image Acquisition Nodes and Network Latency

Figure 4.10, Figure 4.19, and Figure 4.26 were combined in Figure 4.27 below to compare the average latencies of each test and analyze the performance of the network connections as data from the robots was used to a higher capacity. Because the AR.Drone2.0 cannot host a local ROS installation, however, data fully describing the robot's state is continually transmitted to the cloud, regardless of if it is needed. Therefore, little difference is seen in the network performance

of the connection as the size of the data being processed increases, as this has little effect on the data being transmitted from the robot over the network.

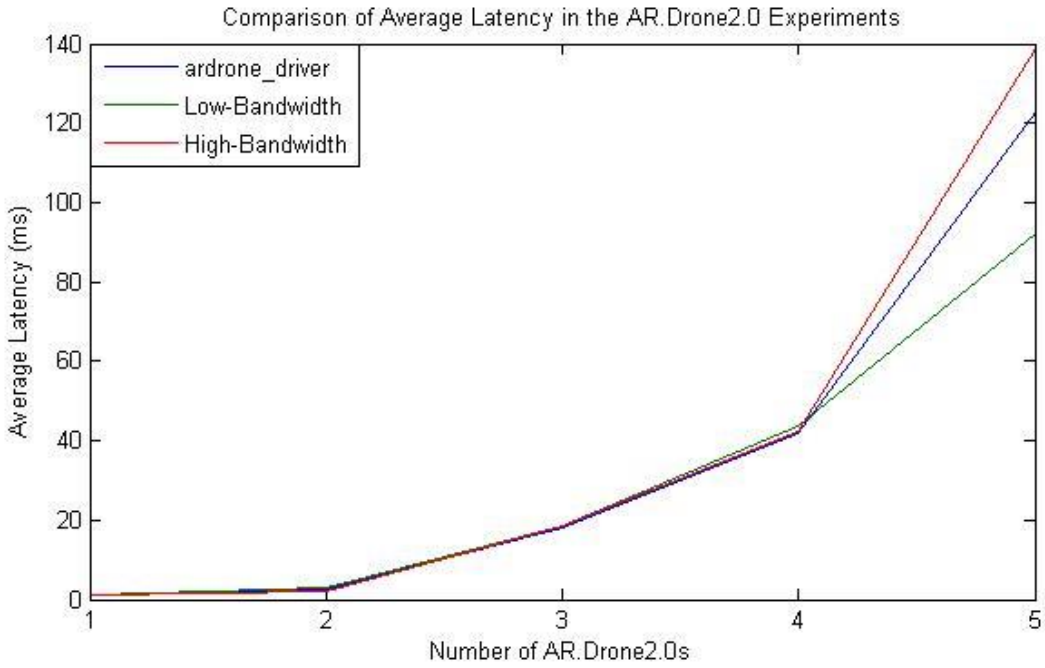

Figure 4.27 Comparison of Average Latency in the AR.Drone2.0 Experiments

4.3 TURTLEBOT

The Kobuki Turtlebot 2 was supplied from the factory with its own netbook capable of running Linux Ubuntu, and thus was able to support a full ROS installation natively, reducing the network bandwidth necessary for running the robot to zero. Baseline latency tests were run by connecting the netbook to the wireless access point and observing the latency between the Turtlebot and the VMs on the network. The baseline latency data is plotted in Figure 4.28.

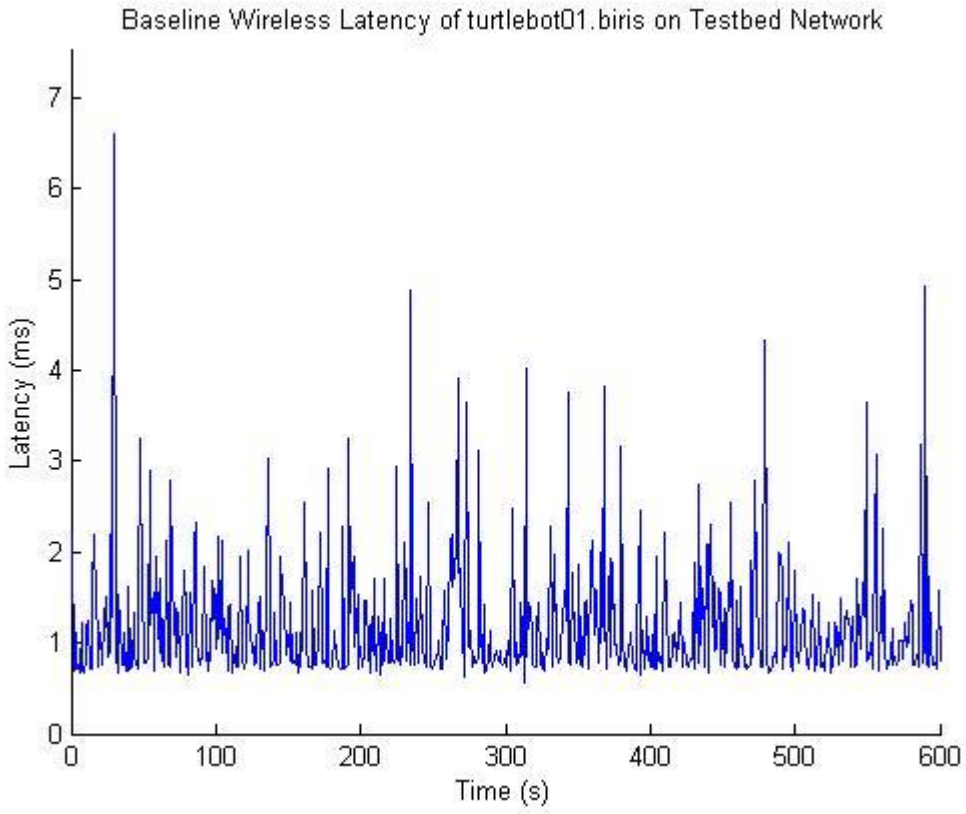

Figure 4.28 Baseline Turtlebot-to-Cloud Latency

The Turtlebot's baseline latency is much improved over the AR.Drone2.0s baseline latency due to the much reduced baseline network bandwidth necessitated by the Turtlebot as compared to the AR.Drone2.0. The Turtlebot's average baseline latency of 1.18 (s = 0.666) was a 19% improvement over the AR.Drone2.0's average baseline latency of 1.43 (s = 7.41).

4.3.1 Turtlebot Low-Bandwidth Data Acquisition

The Turtlebot's ROS drivers (*minimal.launch* for the Kobuki base and netbook, and *openni.launch* for the Microsoft Kinect sensor) were initialized to interface the Turtlebot and Kinect with ROS. Unlike the AR.Drone2.0, both of these launch files (which are used to initialize a collection of nodes from one file) launched ROS nodes on the Turtlebot platform, and a cloud-hosted ROS driver was not needed. Once the ROS drivers were initialized, velocity

commands were sent from the cloud to the Turtlebot while low-bandwidth sensor data was acquired. For the Turtlebot, readings from the local odometry calculations were taken to estimate pose and velocity values. The ROS computational graph for this setup is shown in Figure 4.29.

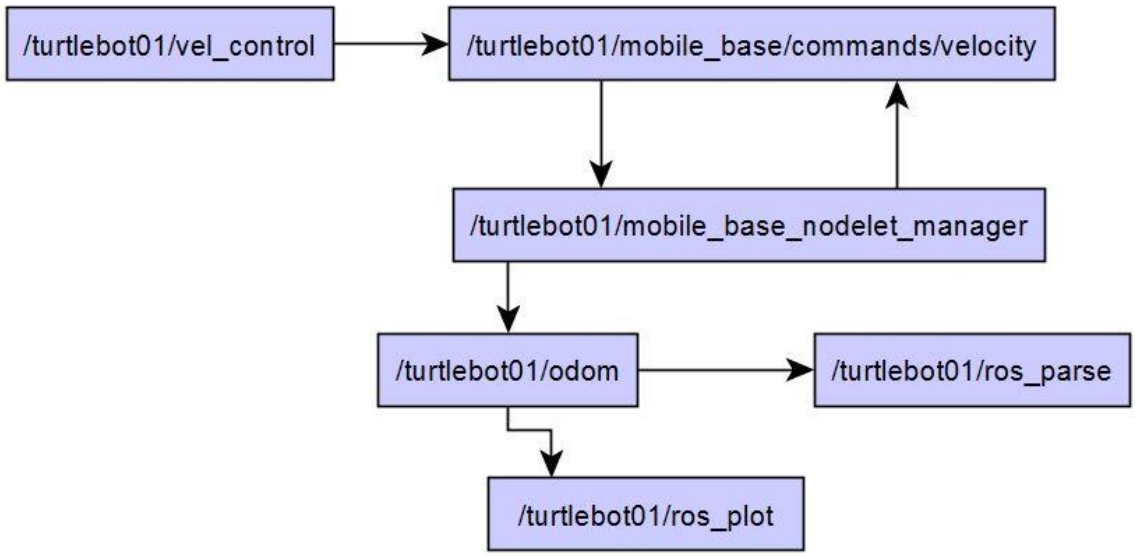

Figure 4.29 ROS Computational Graph for Turtlebot Sensor Data Acquisition and Control

The latency results for the low-bandwidth tests are shown in Figure 4.30 through Figure 4.33.

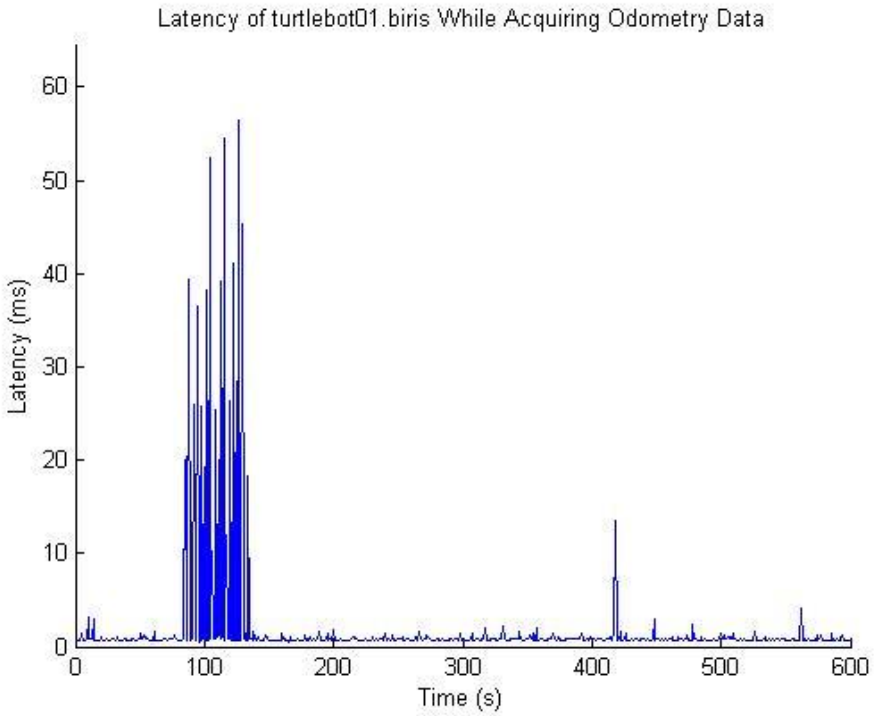

Figure 4.30 Latency of turtlebot01.biris While Acquiring Odometry Data from One Turtlebot

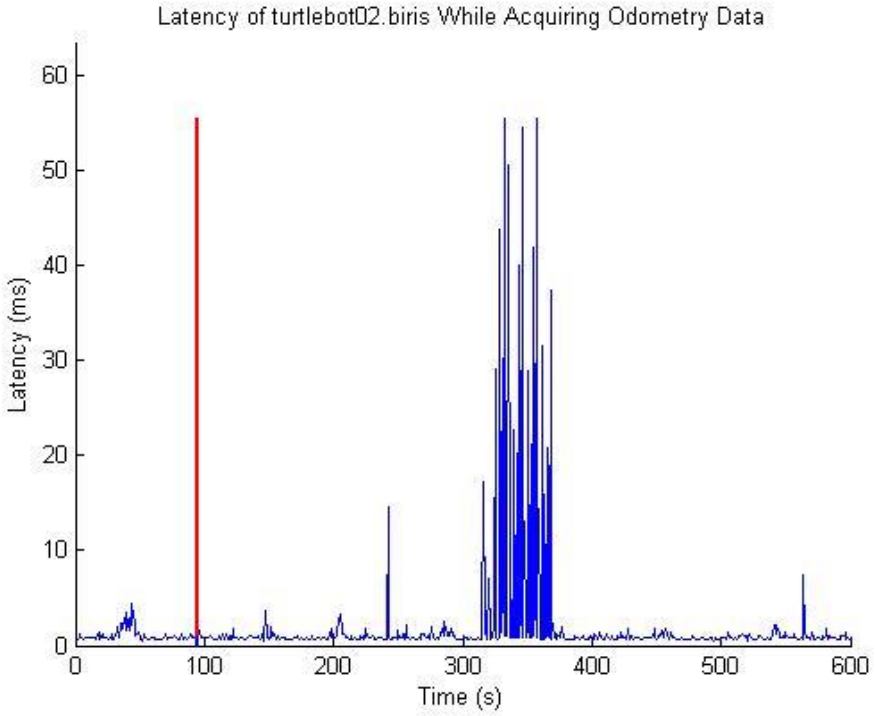

Figure 4.31 Latency of turtlebot02.biris While Acquiring Odometry Data from Two Turtlebots

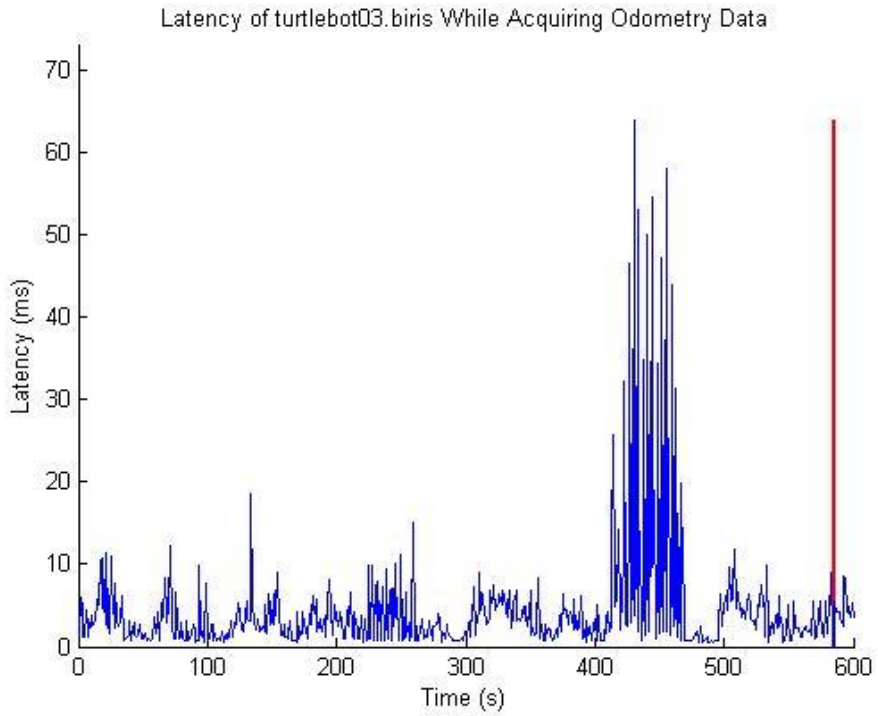

Figure 4.32 Latency of turtlebot03.biris While Acquiring Odometry Data from Three Turtlebots

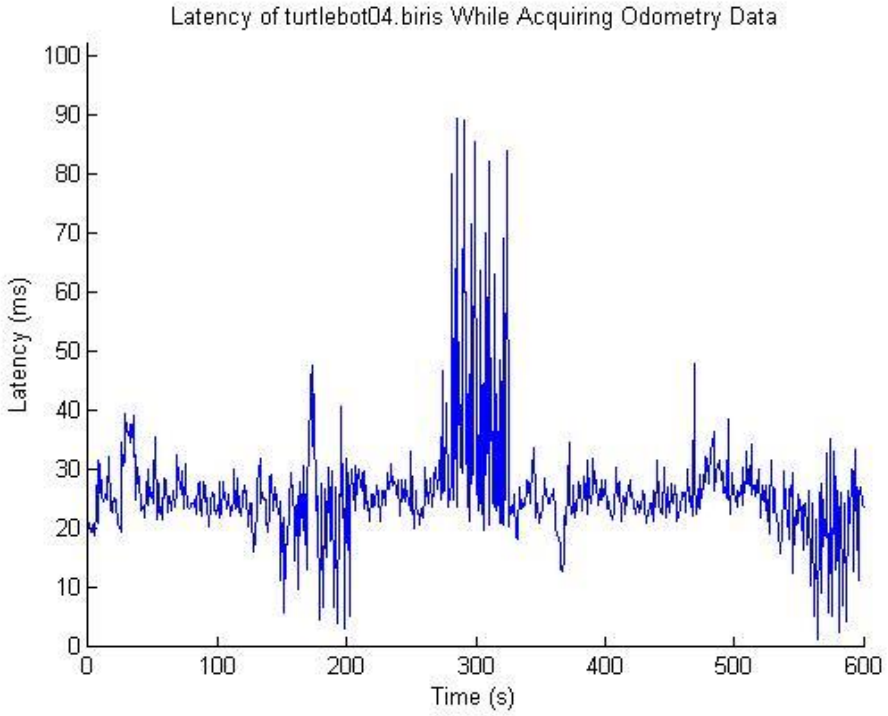

Figure 4.33 Latency of turtlebot04.biris While Acquiring Odometry Data from Four Turtlebots

The statistical parameters of the results of the Turtlebot low-bandwidth test were calculated and are reported in the table below.

Table 4.4 Turtlebot Low-Bandwidth Acquisition Results

Number of Acquisition Nodes	Average Latency (ms)	Maximum Latency (ms)	Minimum Latency (ms)	Standard Deviation (ms)	Packets Dropped (%)
1	0.846	4.1	0.460	0.301	0.00%
2	0.948	8.5	0.416	0.556	0.08%
3	2.96	16.5	0.437	2.31	0.33%
4	24.8	46.4	0.73	5.89	0.21%

The average latency and standard deviation of latency of the system as more Turtlebots were added was graphed and is reported in Figure 4.34.

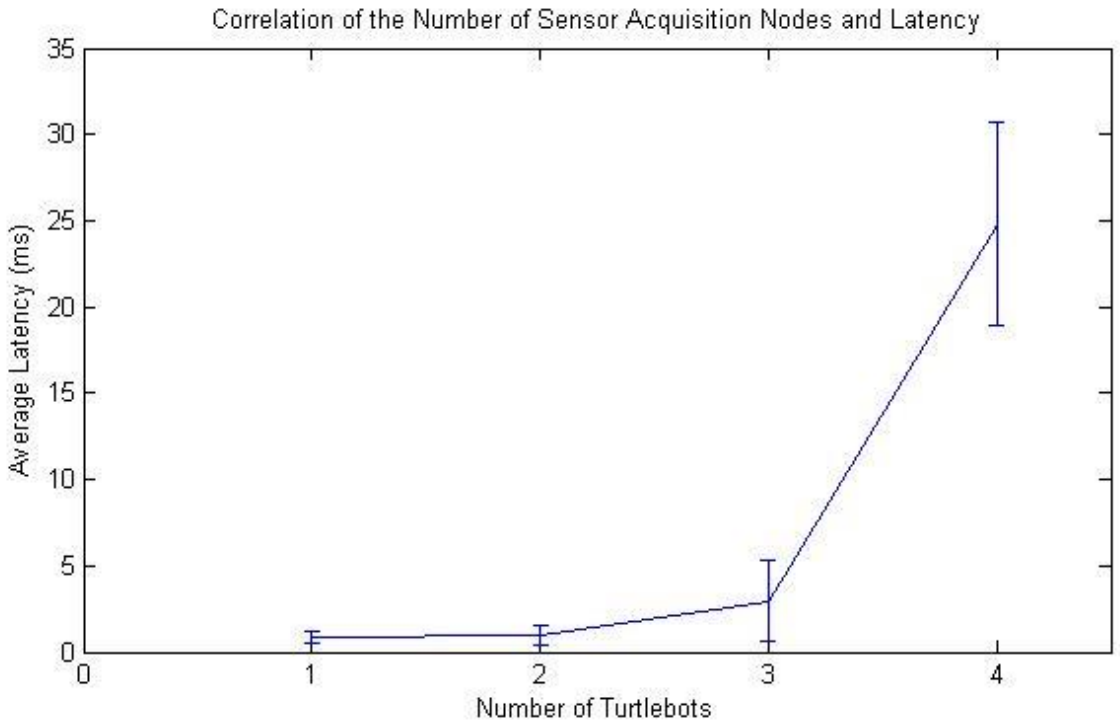

Figure 4.34 Correlation of Number of Sensor Acquisition Nodes and Network Latency

4.3.2 Turtlebot High-Bandwidth Data Acquisition and Control

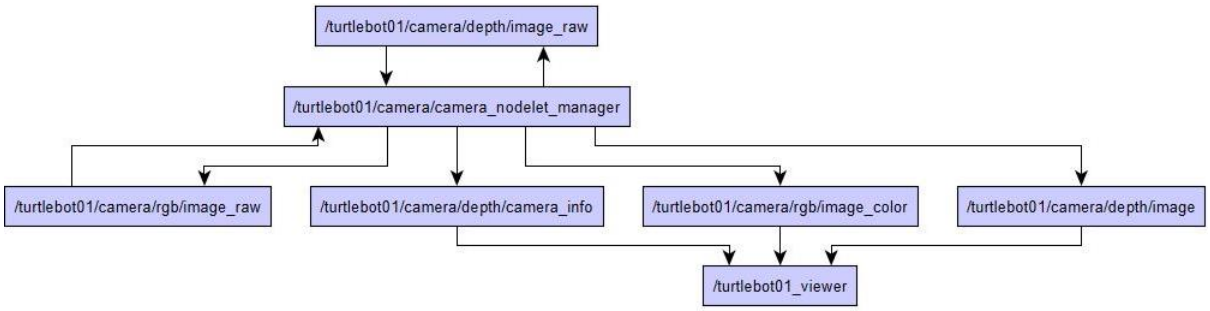

Figure 4.35 ROS Computational Graph of Turtlebot Depth Image Acquisition and Display

The ROS computational graph for the high-bandwidth data acquisition test for the Turtlebots is shown in Figure 4.35. For the high-bandwidth experiments for the Turtlebots, the color and depth images from the onboard Microsoft Kinect were acquired and displayed in a ROS 3D visualization node called RViz shown in Figure 4.36. The latency data for these experiments follows in Figure 4.37 through Figure 4.39.

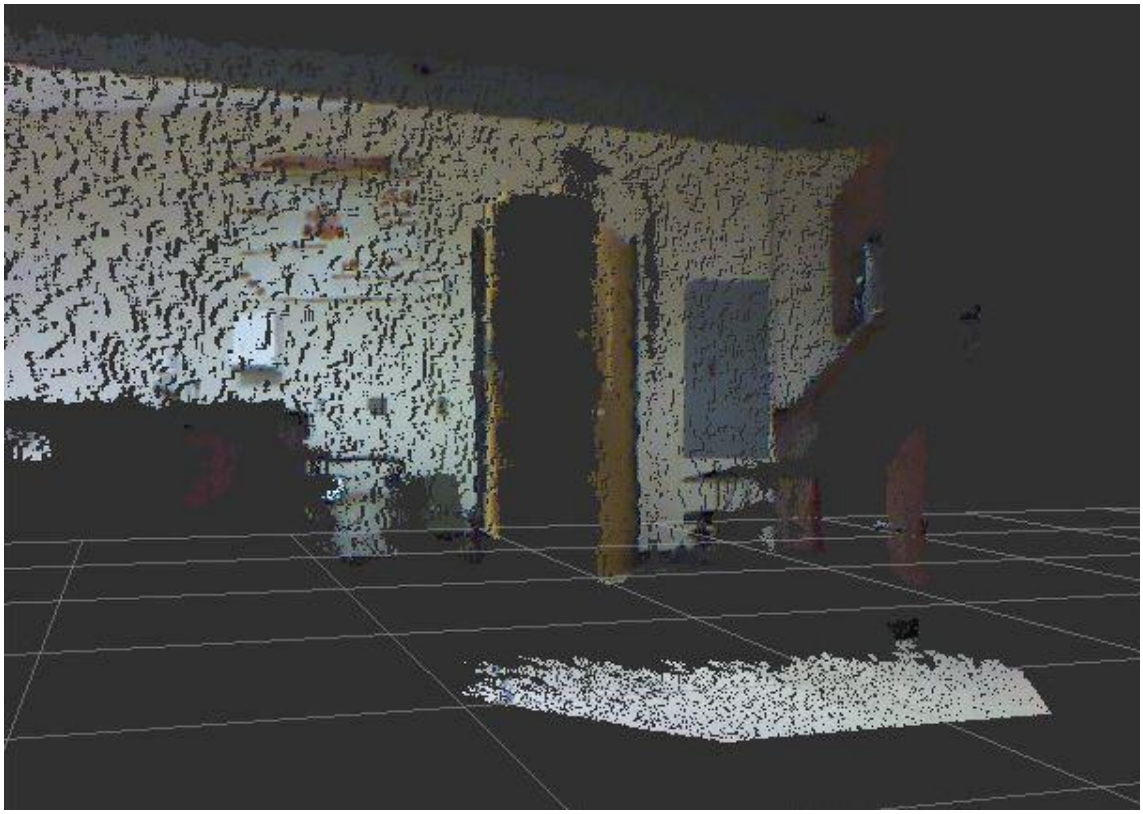

Figure 4.36 Turtlebot Depth and Color Image in RViz, the ROS 3d Visualization Tool

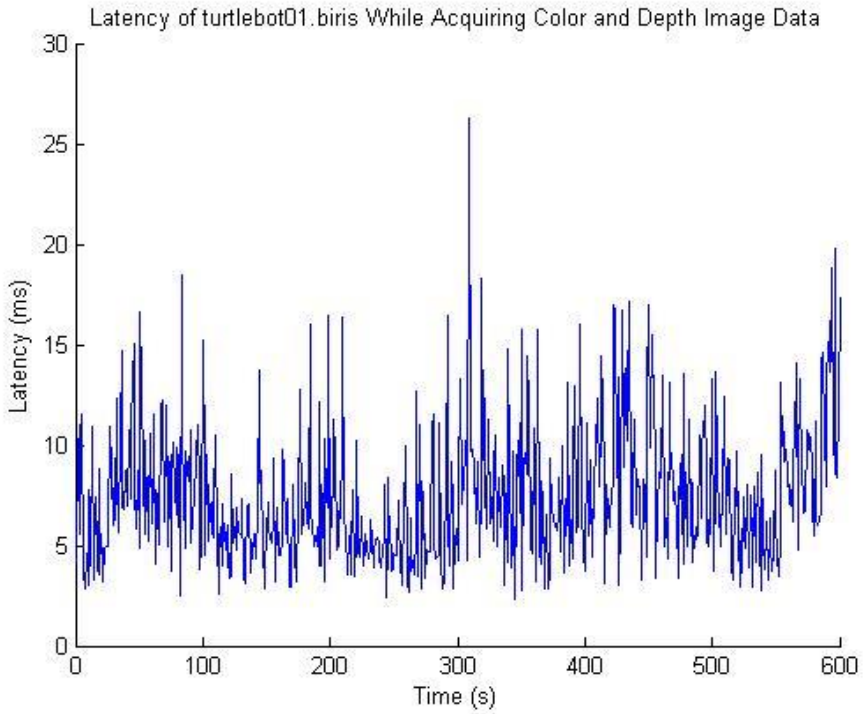

Figure 4.37 Latency of turtlebot01.biris While Acquiring Depth and Color Images from One Turtlebot

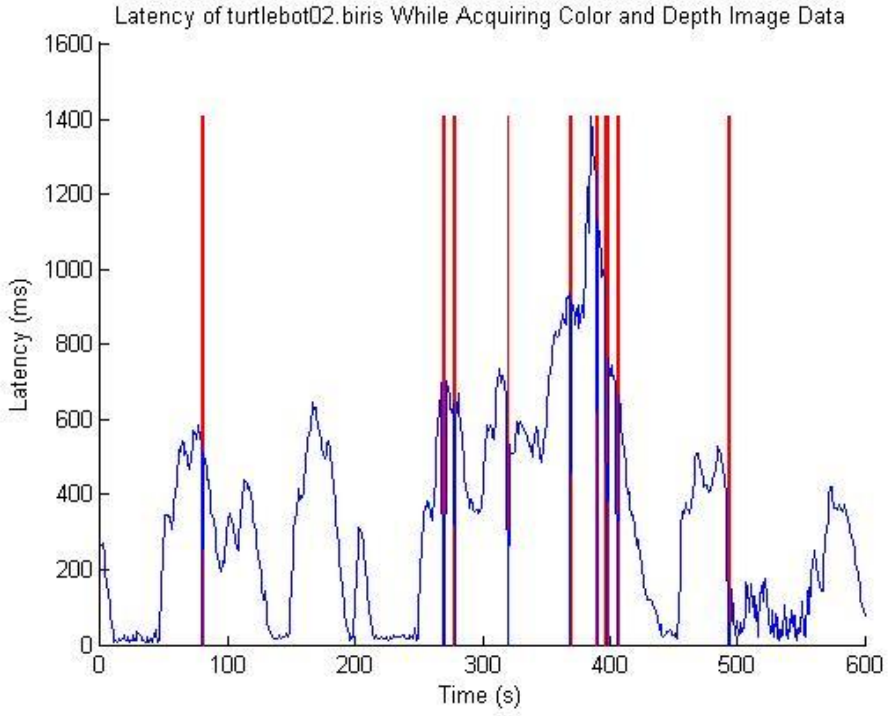

Figure 4.38 Latency of turtlebot02.biris While Acquiring Depth and Color Images from Two Turtlebots

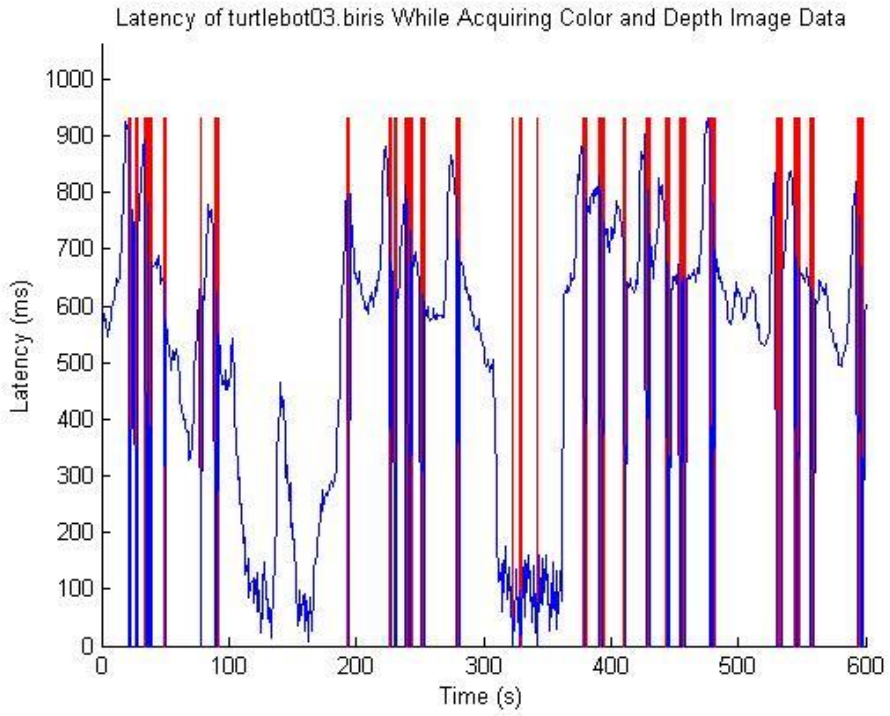

Figure 4.39 Latency of turtlebot03.biris While Acquiring Depth and Color Images from Three Turtlebots

Turtlebot latency and packet loss increased drastically as more Turtlebots were added to the system to stream their depth and color camera data. Network integrity suffered tremendously under this load, and in fact while the third Turtlebot did maintain a connection to the cloud, the bandwidth was already so saturated with the use of turtlebot01.biris and turtlebot02.biris that turtlebot03.biris could not send enough useful data to interpret the images it was attempting to transmit. Even through image data from turtlebot03.biris could not be deciphered and displayed, images from the other two Turtlebots was still decipherable and usable, though at a much lower quality. The lack of latency data for a fourth Turtlebot is an extension of this; the addition of turtlebot04.biris's data stream completely disrupted the wireless network, and no images from any of the Turtlebots were usable. The statistical results of the Turtlebot high-bandwidth test are reported in the table below.

Table 4.5 Turtlebot High-Bandwidth Test Results

Number of Acquisition Nodes	Average Latency	Maximum Latency	Minimum Latency	Standard Deviation	Packets Dropped
	(ms)	(ms)	(ms)	(ms)	(%)
1	7.41	26.3	2.370	3.41	0.00%
2	325	1410	5.40	275	2.50%
3	579	944	7.200	219	10.83%
4	Loss of Network Integrity				

The average latency and standard deviation of the latency was graphed and is reported in Figure 4.40 below.

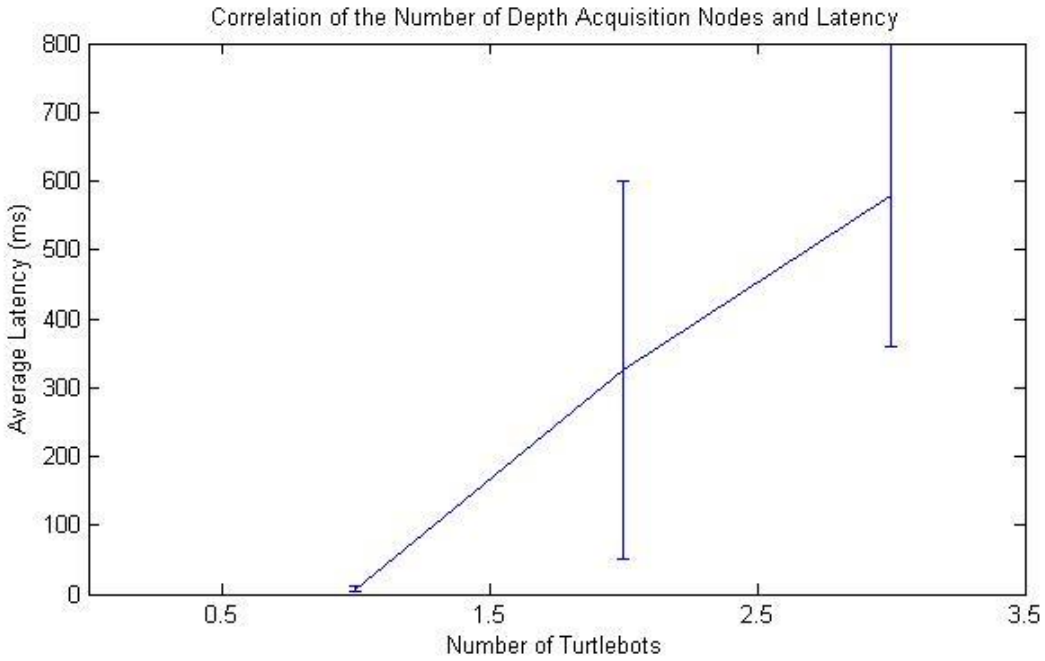

Figure 4.40 Correlation of Number of Depth Acquisition Nodes and Network Latency

The bandwidth data gathered during the experiment actually indicated the available bandwidth was used when only the first Turtlebot was being tested, though at the addition of the second Turtlebot, both robots shared the bandwidth almost equally. The third Turtlebot, however, was able to use only minimal bandwidth and thus the data received from turtlebot03.biris was not usable in any way. The bandwidth data is shown in Figure 4.41.

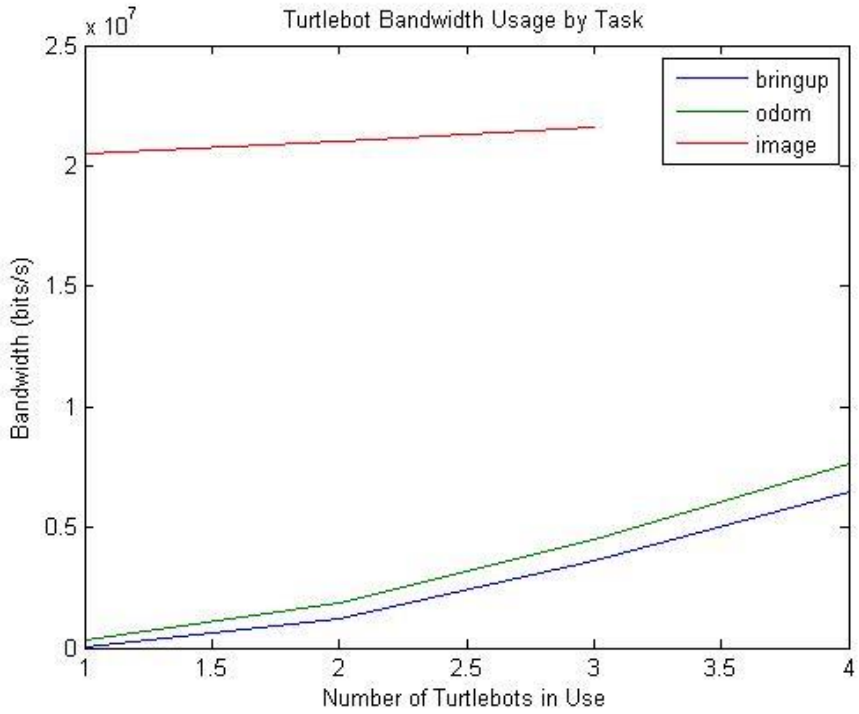

Figure 4.41 Turtlebot Bandwidth Usage by Task

The average latencies of each Turtlebot test are plotted together in Figure 4.42 below for comparison. Compared to the AR.Drone2.0 comparison graph in Figure 4.27, the Turtlebot average latencies significantly increased from the Low-Bandwidth test to the High-Bandwidth test, and adding additional Turtlebots to the Low-Bandwidth test did not drastically degrade the network performance as it did with the AR.Drone2.0s. Again, the advantages of local processing and packet scheduling are clearly evident, and will be necessary to implement in order to maintain network performance as the number of devices increases.

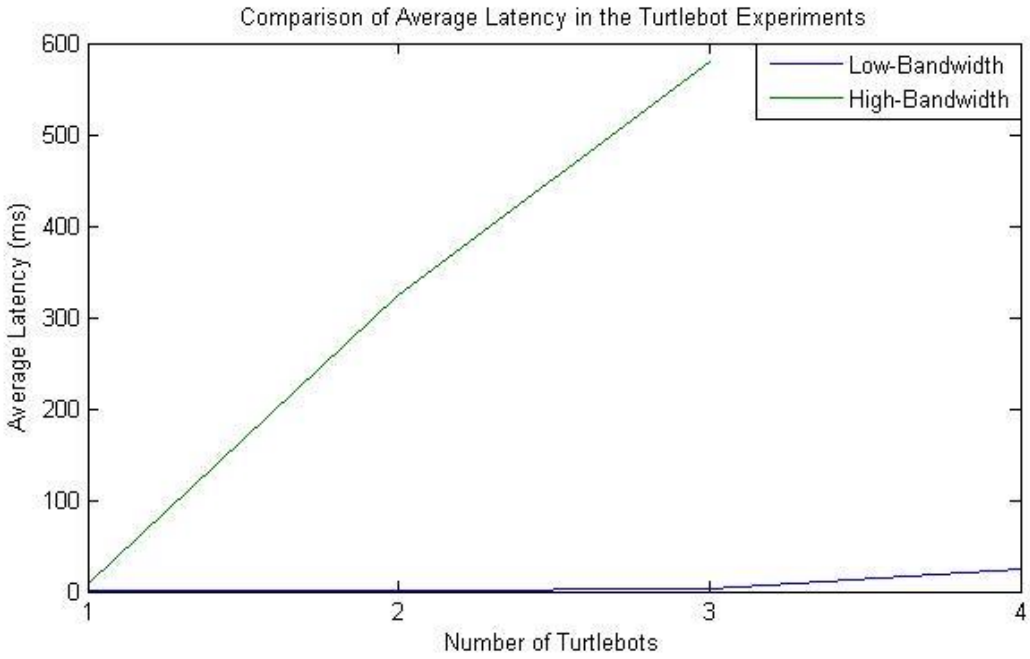

Figure 4.42 Comparison of Average Latency in the Turtlebot Experiments

4.4 LEGO EV3

The LEGO EV3 robot was loaded with a Linux installation called ev3dev (Hempel and Lechner n.d.) and a minimal ROS installation due to limited space. The EV3 was outfitted with a wireless adapter and connected to the Testbed Network. Baseline network connectivity tests resulted in an average latency of 1.73ms (s = 2.21ms), 4% higher than the AR.Drone2.0's baseline latency. The baseline latency results are shown in Figure 4.43.

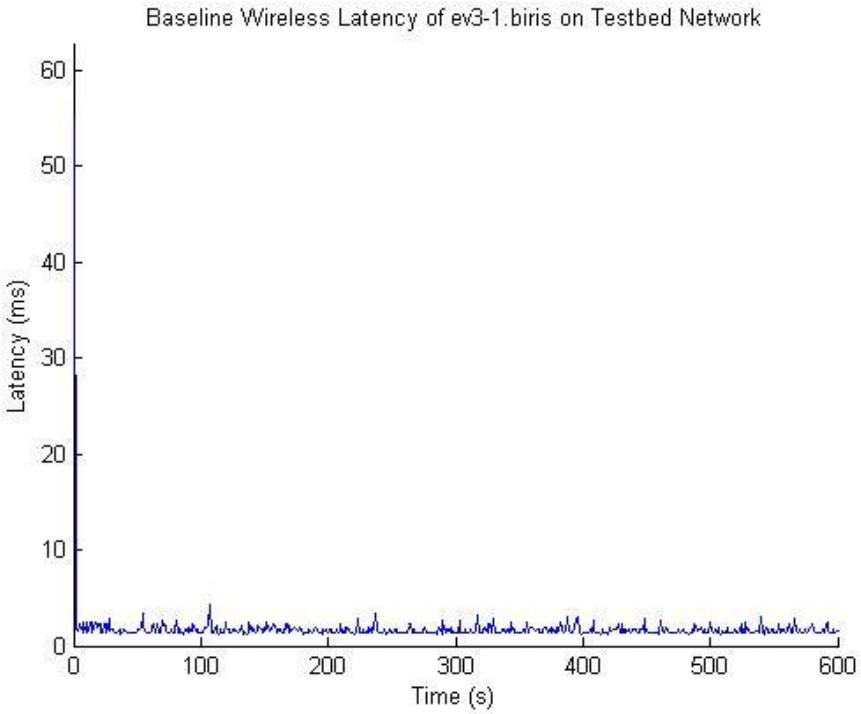

Figure 4.43 Baseline EV3-to-Cloud Latency

4.4.1 EV3 Low-Bandwidth Data Acquisition and Control

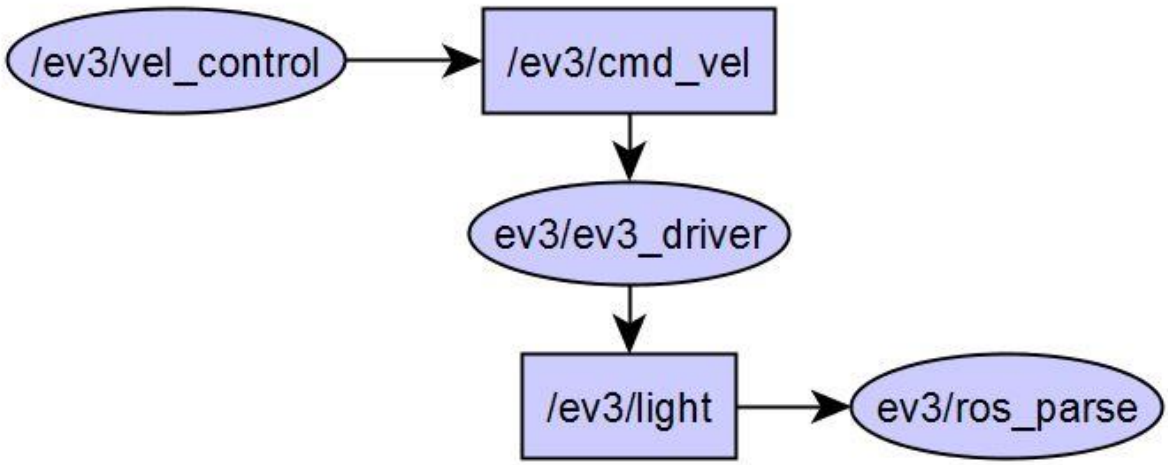

Figure 4.44 ROS Computational Graph of EV3 Low-Bandwidth Test

The ROS computational graph in Figure 4.44 indicates the data connections between the velocity controller, the EV3 ROS driver, and the sensor data parser. Latency data for the low-bandwidth test was collected and graphed in Figure 4.45 below.

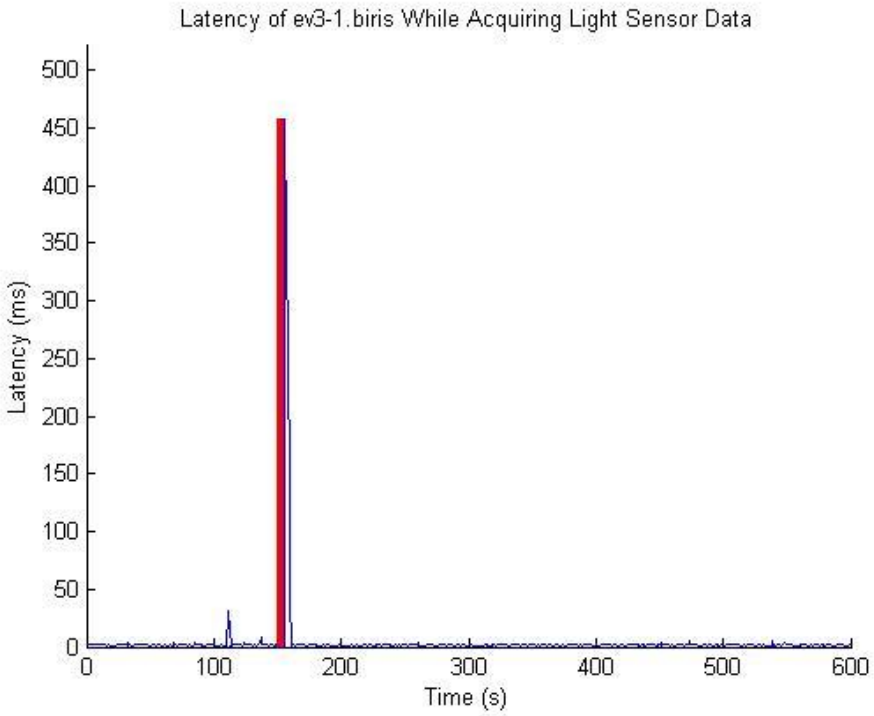

Figure 4.45. Latency of ev3-1.biris while Acquiring Light Sensor Data

While very little bandwidth was used in this test, the latency data indicates that there were six lost packets during the experiment, along with a maximum spike up to 457ms despite an average latency of 3.86ms (s = 26.4). The results of this experiment indicate that even under ideal conditions (i.e. low bandwidth and a single wireless device), it cannot be assumed wireless connectivity will be consistent or without loss. It is important that any system using the cloud, especially through a wireless connection, be capable of falling back to a fail-safe mode in the event communication with the cloud becomes disrupted.

4.4.2 EV3 Cloud-in-the-Loop Control

A control system utilizing the cloud in the loop was attempted on the EV3. Armed with a light sensor, the EV3 would attempt to follow a black line 0.75" thick on the white floor. The EV3 brick would read the reflected light value using the EV3 color sensor, and transmit that value to the cloud using ROS. A cloud-implemented controller would read the light value and determine if the EV3 needed to turn right or left to continue following the line. The cloud controller would then transmit the necessary motor values back to the EV3 brick. The block diagram for the control system is shown in Figure 4.46.

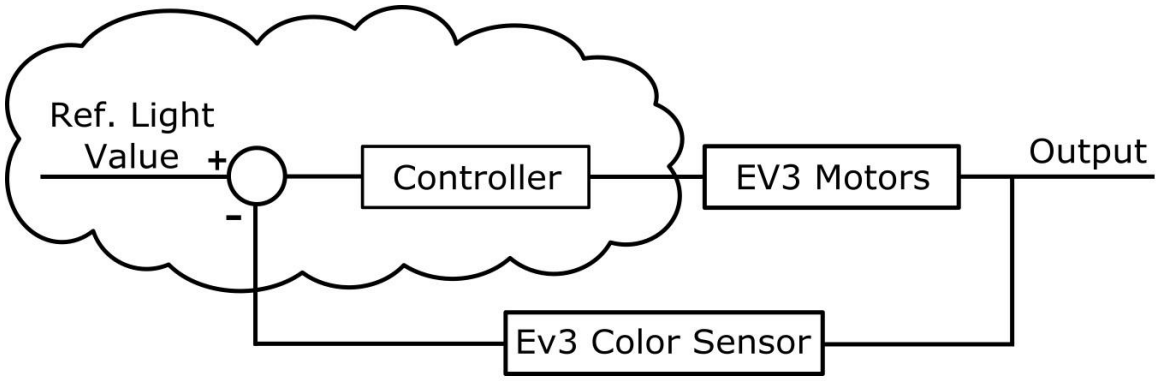

Figure 4.46 EV3 Cloud-in-the-Loop Control

While the average latency between the EV3 and the cloud did not increase while running this experiment, the standard deviation increased from the baseline value of 2.21ms to 4.18ms. The latency graph is provided in Figure 4.47. The controller was unable to properly account for the erratic transmission and reception periods of the data and thus the EV3 was unable to maintain its course on the line. Wireless communication must be much more consistent than that achieved by the EV3 in these experiments to properly utilize the cloud in latency-sensitive control systems.

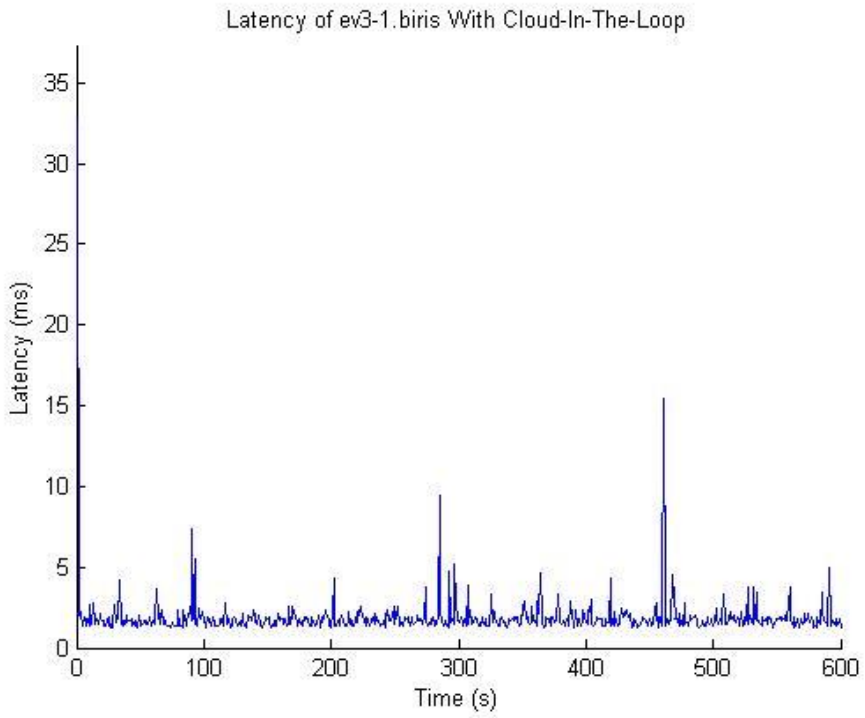

Figure 4.47 Latency of ev3-1.biris Running the "Follow the Line" Cloud-in-the-Loop Control System

4.5 Cloud-Processed Tilt Control

A less latency-sensitive experiment was designed to test the responsiveness of a robot to the manipulation of another. For this experiment, an AR.Drone2.0 was connected to the network and an *ardrone_driver* node initialized for it. Likewise, a Turtlebot was also connected to the Testbed Network and its own driver nodes started. A ROS node was designed to read the angular pose of the AR.Drone2.0 from the *navata* message sent over the */ardrone01/ardrone/navdata* topic. The node processed the rotational data and converted the degrees of rotation to a forward velocity percentage. That percentage was then transmitted to the Turtlebot over the */turtlebot01/mobile_base/commands/velocity* topic to instruct the robot to move forward at that power level. The ROS computational graph for the process is shown in Figure 4.48 below.

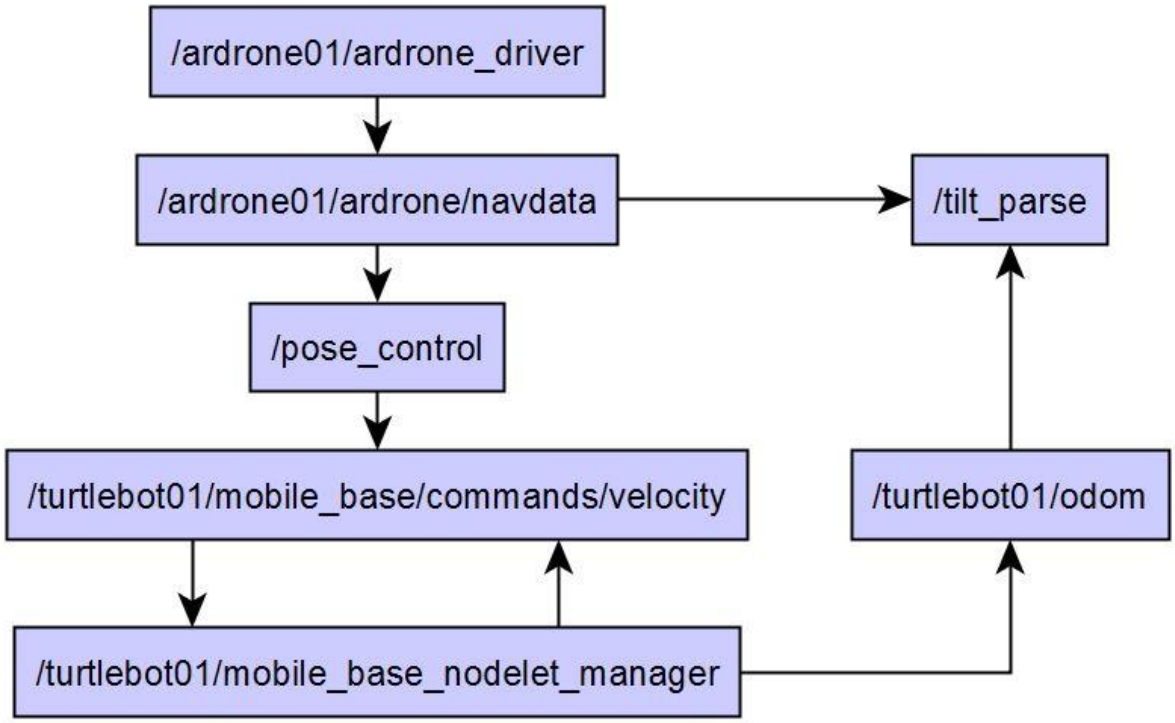

Figure 4.48 ROS Computational Graph for Cloud-Processed Tilt Control

The conversion done by the node *pose_control* is shown in Equations 1 and 2, and feature a 'dead zone', a tolerance of ±5° around 0° that would result in a zero forward velocity being transmitted to the Turtlebot. This was done to ensure the Turtlebot would not be commanded to move while the AR.Drone2.0 was sitting on a flat surface.

$$\dot{x}_T = \begin{cases} 0 & if\ |\theta_{AR}| < 5° \\ \dfrac{\theta_{AR}}{45°} & if\ |\theta_{AR}| \geq 5° \end{cases} \tag{1}$$

$$\dot{\varphi}_T = \begin{cases} 0 & if\ |\emptyset_{AR}| < 5° \\ \dfrac{\emptyset_{AR}}{45°} & if\ |\emptyset_{AR}| \geq 5° \end{cases} \tag{2}$$

A parsing script was set up to simultaneously acquire data from both the */ardrone01/ardrone/navdata* topic and the */turtlebot01/odom* topic and print it to a file for processing. The data was then processed in MATLAB to determine the time shift of the data, indicating the overall latency between the robots, perceived by humans as a sluggish response.

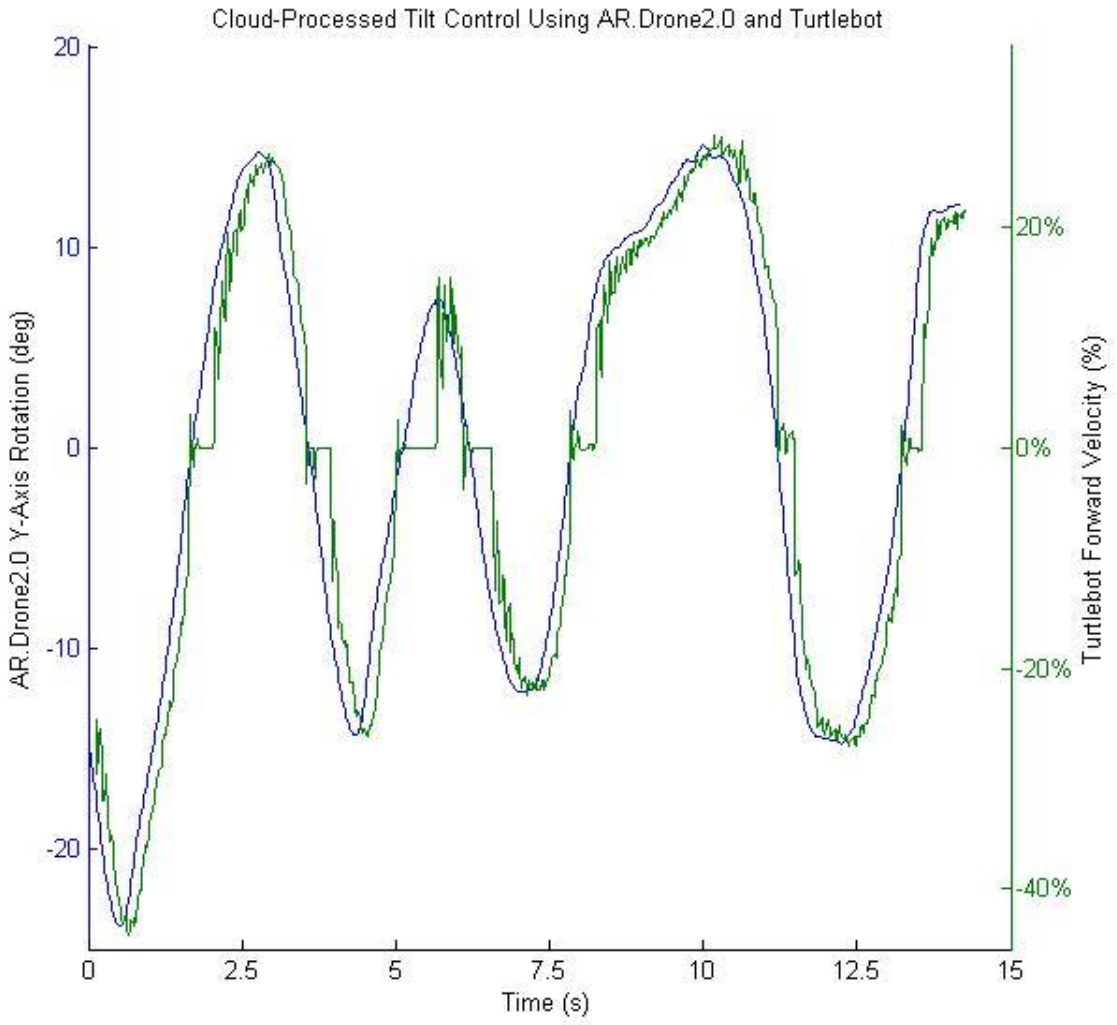

Figure 4.49 Overall Latency of Cloud-Processed Tilt Control

In Figure 4.49, the graph of AR.Drone2.0 rotation and Turtlebot forward velocity is plotted with respect to time. The shapes of the two plots are very similar with the exception of the 'dead zone' near 0° implemented by the *pose_control* node, as expected. There is an evident

time shift of the plot caused by the latency of the network connections between the AR.Drone2.0 and the Turtlebot. This time shift calculates to be approximately 80ms for the duration of the experiment.

CHAPTER 5: CONCLUSIONS AND RECOMMENDATIONS

5.1 SUMMARY OF THE PRESENT WORK

This work investigated the integrity and performance of a network as multiple heterogeneous systems, including robots and control software, were added to it. To perform this investigation, a virtual datacenter was built to support the creation and operation of software systems on virtual machines. The datacenter provided the computational resources necessary for the operation of any cloud-based software systems utilized by the robots. Furthermore, the datacenter provided the ability to build virtual machines as necessary to be the foundation of the software systems that were to be run.

The Robot Operating System was used as the backbone of the network communication that occurred from robot-to-cloud and inside the cloud. Through ROS, data streams were easily initialized, transmitted, and received throughout the network, passing along information to those agents that requested it. ROS nodes implemented on the virtual machines within the cloud provided cloud services to the robots, such as control systems to follow a black line or utilize the pose of one robot to control the velocity of another.

Varying amounts of data were passed from the robot to cloud and back, creating the load that the network performance and integrity was measured against. For each type of robot, baseline performance data was gathered, and then performance data for an increasing number of robots and an increasing amount of bandwidth usage. This data was processed to determine the latency of robot-to-cloud communications, as well as dropped data packets.

A cloud-in-the-loop control system was developed to determine the viability of implementing the controller of a time-sensitive system within the cloud. Though the functionality of the control system was preserved, the latency between the controller and the robot made the system unstable and unviable. Finally, a tilt control system was implemented to read from the pose of one robot

and interpret the data as velocity commands for another. The state of each robot was observed to determine the latency between maneuvering the control robot and receiving the velocity commands on the other.

The results confirm that it is possible to build a network that provides cloud services to heterogeneous client devices while maintaining integrity under some amount of load, though the integrity diminishes significantly as the wireless bandwidth capacity is saturated. The network supported many robots under low bandwidth applications, but few robots under high bandwidth applications.

5.2 Future Scope of Work

Minimizing latency due to interference for research purposes would be useful for conducting very controlled experiments on robot performance in an isolated setting. To this end, two improvements could be made to the study: first, the use of a 5GHz wireless access point would result in much less interference, as the campus access points causing much of the interference are on the 2.4GHz band, as was the wireless access point used for this study. Second, the construction of a Faraday cage would provide the means to examine the performance of these devices in an environment completely isolated from outside interferences.

Additional networked systems could be integrated into the environment to provide supplementary sensor feedback, such as mounted 2D and 3D cameras. These cameras would provide additional network load, though the load could be through wired connections instead of wireless, increasing the capacity of the connection.

With IPv6 fast approaching implementation out of necessity due to a growing number of networked devices, the existing network could be switched to the IPv6 protocol to examine the challenges the protocol present and determine solutions to meet those challenges.

The performance of networked devices should be evaluated based on the ratio of local processing to remote processing, and the advantages and disadvantages of such a design evaluated. For those devices unable to be programmed, an intermediary device, such as a Field Programmable Gate Array (FPGA) could be added to handle some local processing to increase stability of the system.

REFERENCES

An, Baik Song, Manhee Lee, Ki Hwan Yum, and Eun Jung Kim. "Efficient data packet compression for cache coherent multiprocessor systems." *Data Compression Conference, 2012.* Snowbird, UT: IEEE, 2012. 129-138.

Araos, Diego. "ardrone-wpa2." *GitHub.* March 29, 2013.

Armbrust, M, et al. "A view of cloud computing." *Communications of the ACM* 53, no. 4 (2010): 50-58.

Cao, Y., A. Fukunaga, and A Kahng. "Cooperative mobile robotics: antecedents and directions." *Auton. Robots*, 1997: 1-23.

Deka, Ganesh Chandra. "Cost-benefit analysis of datacenter consolidation using virtualization." *IT Professional*, 2014.

Doriya, Rajesh, Pavan Chakraborty, and G. C. Nandi. "'Robot-cloud': a framework to assist heterogeneous low cost robots ." *International Conference on Communication, Information, and Computing Technologies.* 2012. 1-5.

"Enabling a new future for cloud computing." *National Science Foundation.* August 20, 2014. http://www.nsf.gov/news/news_summ.jsp?cntn_id=132377 (accessed 10 31, 2015).

Gungor, V.C., and G.P. Hancke. "Industrial wireless sensor networks: challenges, design principles, and technical approaches." *Industrial Electronics, IEEE Transactions on* 56, no. 10 (2009): 4258-4265.

Hempel, Ralph, and David Lechner. *ev3dev.* n.d. http://www.ev3dev.org/ (accessed October 2, 2015).

Kehoe, Ben, Sachin Patil, Pieter Abbeel, and Ken Goldberg. "A survey of research on cloud robotics and automation." *IEEE Transactions on Automation Science and Engineering*, April 2015: 398-409.

Kelmendi, Rilind. "Economics of cloud computing latency." *International Journal of Academic Research* 5, no. 2 (2013): 99-101.

LEGO Education EV3. 2015. https://education.lego.com/en-us/lesi/middle-school/mindstorms-education-ev3/all-about-ev3 (accessed 10 18, 2015).

Monajjemi, Mani. "ardrone_autonomy." *GitHub.* April 24, 2015.

Nguyen, Vu-Anh-Quang, and Myungsik Yoo. "Packet loss compensation for control systems over industrial wireless sensor networks." *International Journal of Distributed Sensor Networks*, 2015: 1-9.

Parker, Lynne E., and Fang Tang. "Building multirobot coalitions through automated task solution synthesis." *Proceedings of the IEEE.* 2006. 1289-1305.

Parrot AR.Drone2.0. 2015. ardrone2.parrot.com (accessed 10 18, 2015).

Pye, Andy. "THE INTERNET OF THINGS connecting the unconnected." *Engineering & Technology (17509637)*, December 2014: 64-70.

Quidley, Morgan, et al. "ROS: an open-source Robot Operating System." *Willow Garage.* 2009. https://www.willowgarage.com/sites/default/files/icraoss09-ROS.pdf (accessed 10 25, 2015).

ROS/Concepts. Jun 21, 2014. http://wiki.ros.org/ROS/Concepts (accessed 10 18, 2015).

Sharp, Ron. "Latency in cloud-based interactive streaming content." *Bell Labs Technical Journal (John Wiley & Sons, Inc.)* 17, no. 2 (2012): 67-80.

"The NIST definition of cloud computing." *NIST.* September 2011. http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf (accessed November 18, 2015).

Turtlebot 2. n.d. http://www.turtlebot.com (accessed 10 18, 2015).

Wang, Lujia, Ming Liu, and Max Meng. "Real-time multisensor data retrieval for cloud robotic systems." *IEEE Transactions on Automation Science and Engineering* 12, no. 2 (2015): 507-518.

Weir, Bill. "Resolving signal integrity issues in cloud computing platforms." *ECN: Electronic Component News*, May 15, 2013: 16-17.

Whitson, George M., III. "Computer Networks." *Salem Press Encyclopedia of ScienceResearch Starters.* 2015.

Willcocks, Leslie P., Will Venters, and Edgar A. Whitley. *Moving to the cloud corporation: how to face the challenges and harness the potential of cloud computing.* Palgrave Macmillan, 2013.

Appendix A

LIST OF PUBLICATIONS

C. Reid and B. Samanta, "Gesture Recognition for Control in Human-Robot Interactions," in *ASME 2014 International Mechanical Engineering Congress & Exposition*, Montreal, 2014.

C. Reid, B. Samanta, and C. Kadlec, "Development of a Network Infrastructure for Heterogeneous Robot and Control Systems Interactions," in *ASME 2014 International Mechanical Engineering Congress & Exposition*, Houston, 2015.

C. Reid, B. Samanta, and C. Kadlec, "Heterogeneous Networked Systems in a ROS-Enabled Cloud Environment," in *2016 ASNE Symposium Proceedings,* Arlington, 2016.