A Distributed Software Framework for the Virginia Tech Ground Station

Paul U. David

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Electrical Engineering

Robert W. McGwier, Chair

T. Charles Clancy

R. Michael Buehrer

William C. Headley

October 5, 2015

Blacksburg, Virginia

Keywords: Satellite, Ground Station, Actor Model, Distributed Systems, Peer-to-peer

A Distributed Software Framework for the Virginia Tech Ground Station

Paul U. David

(ABSTRACT)

The key goal in this work is to enable a flexible ground station that is not constrained to a particular mission or set of hardware. In addition, with the concepts and software produced in this thesis, it will play a significant role in educating engineers and students by providing critical infrastructure and a sandbox for ground station operations. Key pieces of software were developed in this work to create a flexible and robust software-defined ground station. Several digital transmission modes were developed in order to allow communication between the ground station and common amateur radio CubeSats and SmallSats. In order to handle distributed tasks and process at a ground station with multiple servers and controllers, a specialized actor framework was written in *Python* for ease of use. Actors have the ability to send messages to one another over a network, and they maintain their own memory in order to avoid synchronization problems that come with sharing memory. In addition to the software developed in this work, a novel Peer-to-Peer (P2P) protocol for a network of ground stations is proposed in order to increase coverage and access to spacecraft without requiring centralized server infrastructure. This protocol provides the method to scale the developed software architecture beyond a single ground station.

Since the Virginia Tech Ground Station (VTGS) will have many concurrent processes running across multiple servers, it was necessary to apply the actor model in order to simplify the design of the system. The purpose of this thesis is to describe the developed software for the VTGS as well as the P2P protocol for a larger network of ground stations. There are three primary repositories: *planck-dsp*, *gr-vtgs*, and *pystation*. The *planck-dsp* library and *gr-vtgs* Out-of-tree (OOT) make up the primary digital signal processing and communications toolboxes, where GNU Radio serves as the scheduler for signal processing blocks used in flow graphs. The *pystation* module is the extensible software actor framework that connects various systems both locally and remotely. It is also responsible for scheduling and handling ground station requests. While the software was primarily created for the VTGS, it is general enough to apply to other ground station implementations.

Dedication

I dedicate this thesis to my wonderful parents for their love and patience.

I also dedicate this to my amazing family and friends

for their overwhelming support in my times of need.

Acknowledgments

I would like to thank my advisor, Dr. McGwier, for his support and help in my research. I would also like to thank everyone at the Hume Center and Virginia Tech for their help and mentorship. I would like to acknowledge Seth Hitefield who contributed significantly to the control software and framework as a whole. And finally, I want to thank Zach Leffke for transferring to me his vision of a satellite ground station at Virginia Tech. He is the true brain behind the ground station and has continued to push for its construction during his career at Virginia Tech. Where this thesis is concerned, Dr. William Headley was of significant help. Without him, I'm not sure I would have finished with as much quality in this thesis; his advice and comments were invaluable.

Contents

List of Figures

List of Tables

Acronyms

AFSK Audio Frequency-shift Keying

AFSK1200 Audio Frequency-shift Keying 1200 Baud

API Application Programming Interface

APRS Automatic Packet Reporting Service

ASOF Autonomous Satellite Operations Framework

BPSK Binary Phase-shift Keying

CBR Carson Bandwidth Rule

CCITT Consultative Committee in International Telegraph and Telephone

CLI Command-line Interface

COTS Commercial Off-the-shelf

CPFSK Continuous Phase Frequency-shift Keying

CPM Continuous Phase Modulation

DDoS Distributed Denial-of-service

DHT Distributed Hash Table

DLSAP Data-Link Service Access Point

DSP Digital Signal Processing

ESA European Space Agency

FCS Frame Check Sequence

FM Frequency Modulation

FSK Frequency-shift Keying

FSK9600 Frequency-shift Keying 9600 Baud

GENIE Generic Inferential Executor

GENSO Global Educational Network for Satellite Operations

GIL Global Interpreter Lock

GMSK Gaussian Minimum-shift Keying

GMSK9600 Gaussian Minimum-shift Keying 9600 Baud

GSML Ground Station Markup Language

GUI Graphical User Interface

HDLC High-Level Data Link Control

IPC Interprocess Communication

ISO International Organization for Standardization

ISS International Space Station

ITU International Telecommunications Union

JSON JavaScript Object Notation

JVM Java Virtual Machine

LAN Local Area Network

LEO Low Earth Orbit

LFSR Linear Feedback Shift Register

MTU Maximum Transmission Unit

NBFM Narrowband Frequency Modulation

NOAA National Oceanic and Atmospheric Administration

NRZI Non-return to Zero Inverted

OOT Out-of-tree

OSI Open Systems Interconnection

P2P Peer-to-Peer

PLL Phase-locked Loop

PSK Phase-shift Keying

REST Representational State Transfer

RPC Remote Procedural Call

SAMPEX Solar Anomalous and Magnetospheric Particle Explorer

SatNOGS Satellite Networked Open Ground Station

SDR Software-Defined Radio

SNR Signal-to-noise Ratio

SWIG Simplified Wrapper and Interface Generator

TLE Two-Line Element set

TNC Terminal Node Controller

TT&C Telemetry, Tracking, and Command

TTL Time to Live

UDP User Datagram Protocol

USRP Universal Software Radio Peripheral

USRP N210 Universal Software Radio Peripheral N210

VPN Virtual Private Network

VTGS Virginia Tech Ground Station

WSGI Web Server Gateway Interface

XML Extensible Markup Language

Chapter 1

Introduction

1.1 Overview of the VTGS

The primary motivation for creating a ground station at Virginia Tech is to support CubeSats and other communication opportunities with spacecraft in Low Earth Orbit (LEO) at altitudes between 160 km and 2000 km. Most CubeSat projects are concerned with space-based research and providing educational activities for university students. However, there are other uses for a ground station such as radio astronomy and communication with amateur radio satellites. Examples include the reception of Ham TV from the International Space Station or the detection and measurement of galactic hydrogen. Other uses include Telemetry, Tracking, and Command (TT&C) of spacecraft to support future contracts at Virginia Tech. It is important that Virginia Tech be able to handle its own CubeSat mis-

sions and allow faculty and students to monitor and communicate with their experiments in orbit, conduct experiments in a course lab, or learn about communications from the vantage point of ground station operations. While there are many ways to construct software for a ground station, certain design choices were made to enable signal processing in software and distribute tasks among dedicated servers. It was also important that the ground station software be extendable to other ground station constructions that might utilize different hardware. An extensible signal processing library and module were developed in this work to enable communications using Software-Defined Radios (SDRs). In addition, an actor model framework was created to organize these tasks and provide a message passing library for communication across the Local Area Network (LAN) while maintaining state isolation among processes to avoid system-wide lockups.

The Virginia Tech Ground Station (VTGS) supports many systems: amateur VHF/UHF, amateur L/S-band (1-4 GHz), X-Band (8-12 GHz), and amateur radio astronomy. It is physically located near the Virginia Tech campus on Prices Fork Road in an area away from dense wireless interference. The operations room is located at Space@VT facilities, and networked VPN connections exist between the operations centers and the tracking station. Figure 1.1 and Figure 1.2 describe the general layout of the tracking station, and highlight the different systems the ground station supports. A pair of doubly stacked yagi antennas are used for amateur VHF/UHF systems in order to conduct communications with amateur radio satellites as well as CubeSat spacecraft. The VTGS will also support third party spacecraft such as QB50 and future commercial contracts. The automated control software

created in this work will be able to establish links with active amateur satellites to enable wider communications and amateur radio activities.

In addition to command and control opportunities, the VTGS has the capability to monitor telemetry on a myriad of active CubeSats in orbit, such as CUTE-1, Funcube-1, and many others.

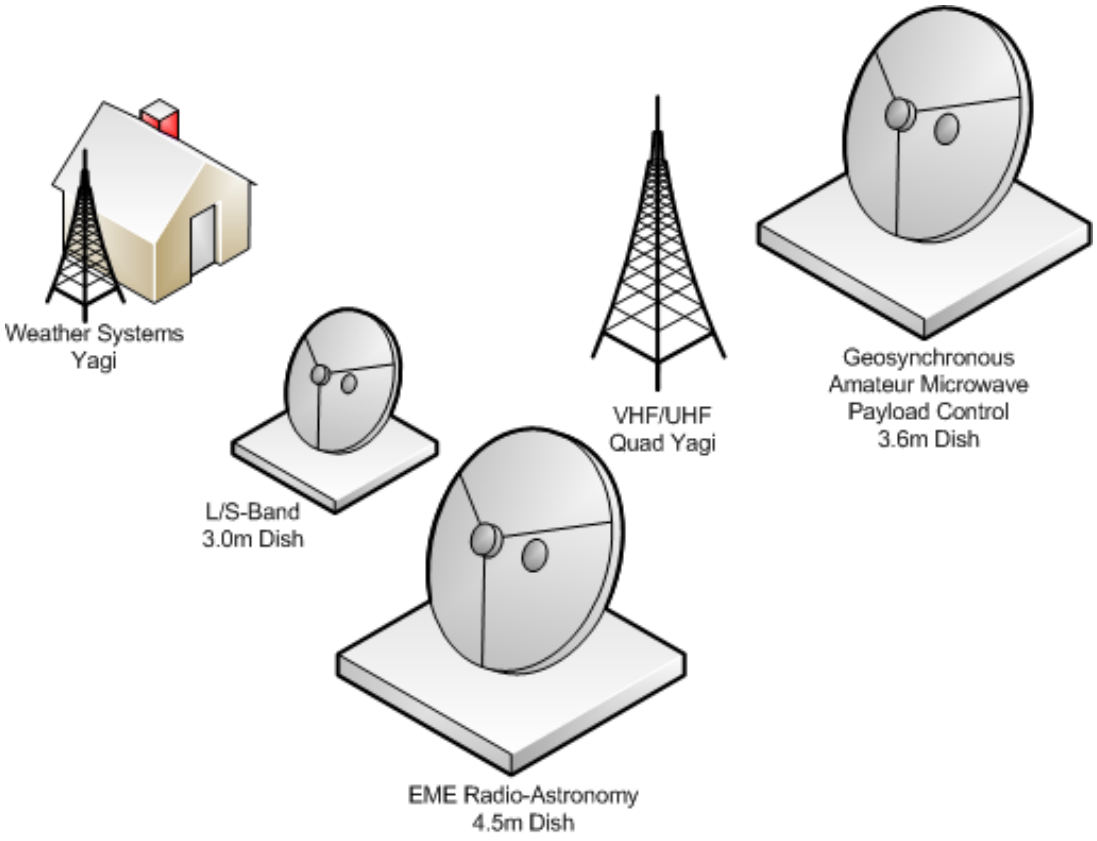

Figure 1.1: High-level overview of the Virginia Tech Ground Station antennas and their purposes.

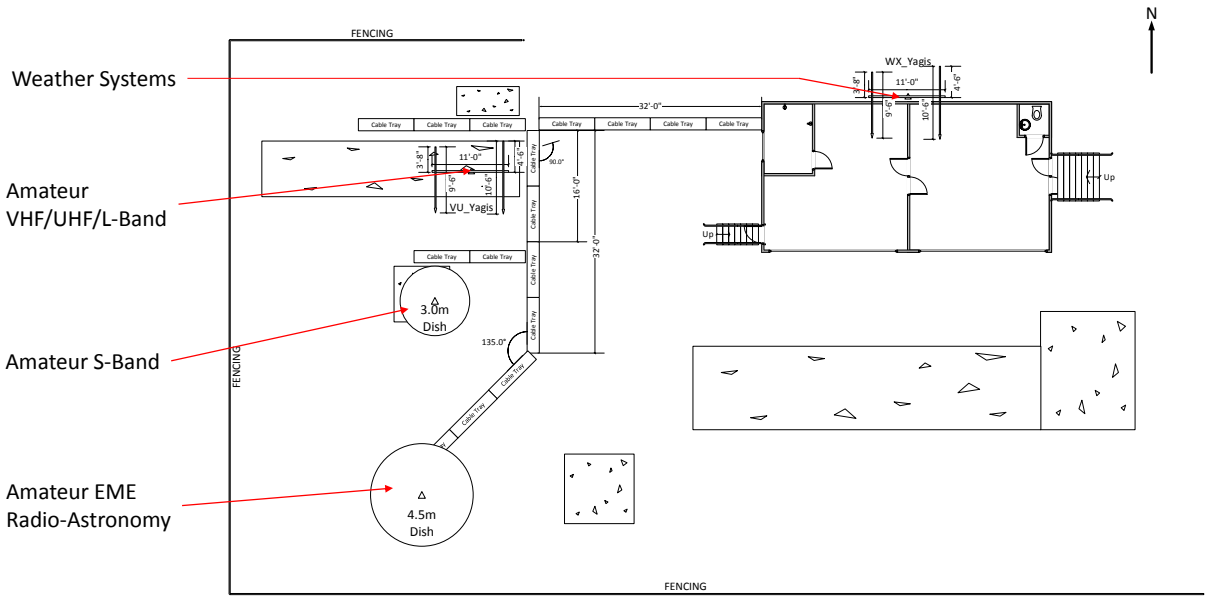

Figure 1.2: Physical layout and ground plan of the Virginia Tech Ground Station.

The S-Band dishes will handle ISS Ham TV as well as high baud rate telemetry downlinks. An L-Band dish will serve as the uplink for the amateur satellite service or as another S-band downlink depending on the station activity and schedule. Weather systems provide access to satellite imaging like National Oceanic and Atmospheric Administration (NOAA) satellites in LEO and geosynchronous orbit. Commercial systems and more advanced satellites are supported by the S-Band and X-Band dishes on the uplink and downlink, respectively, allowing for future expansions of the VTGS capabilities.

A major goal in the design of the infrastructure for VTGS was to move as much of the functionality of the ground station to software as possible. The software developed in this work attempts to do just that by providing a software framework with several components

that tackle tasks like scheduling, controls, and signal processing. All of the signal processing should be done in software with baseband samples arriving from the USRP N210 as the last transition in the transmit and receive chain in order to increase the flexibility of this system. This design goal implies the introduction of network infrastructure, which is shown in Figure 1.3.

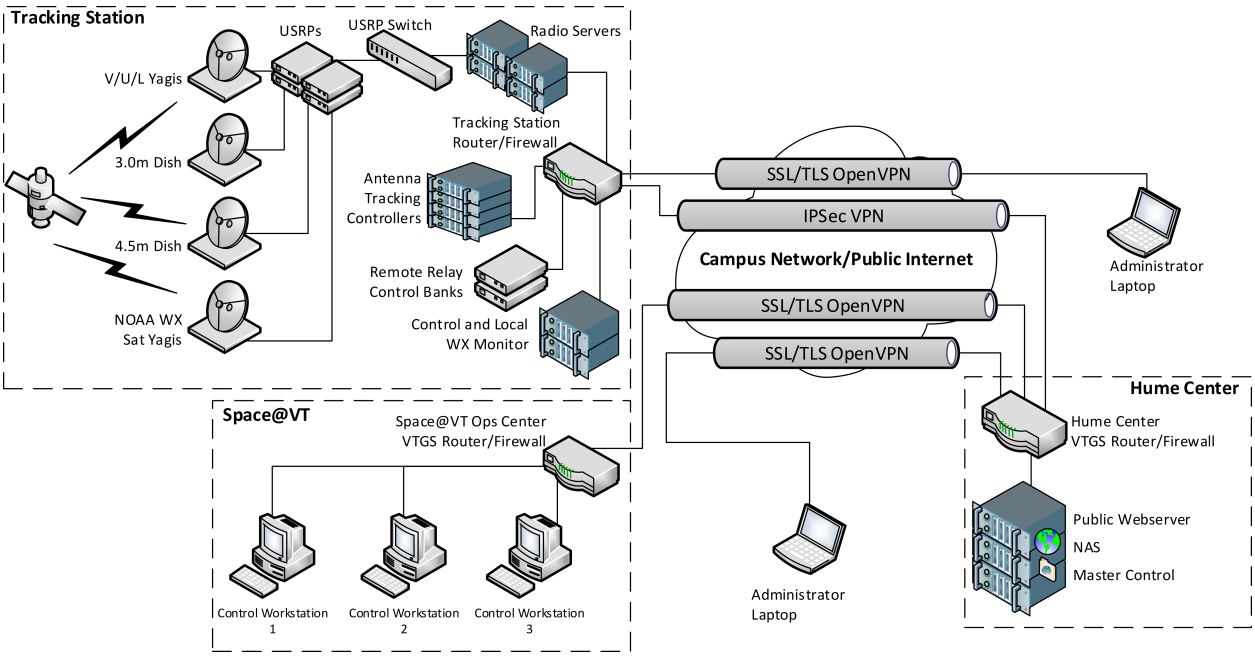

Figure 1.3: Network configuration of the Virginia Tech Ground Station.

The VTGS will be an initial peer in a network of ground stations, warranting the need to develop a protocol to facilitate this expansion. Such a network of ground stations is desirable to increase coverage and access to missions. As part of this work, a Peer-to-Peer (P2P) protocol is proposed that allows for a network of ground stations to coordinate without the need for centralized server infrastructure. This means that peers in the network can participate without a dependence on a central web site or server for controlling ground

station operations. If a ground station peer goes down, the network will remain intact and coordination can proceed with any remaining peers.

The following section provides an outline of the contributions detailed in the following chapters. Each chapter builds on the previous one, beginning with the signal processing software and ending with the proposed P2P protocol.

1.2 Contributions

The aim of this thesis is to describe the software architecture involved in controlling the VTGS and handling its signal processing routines. This software was fundamental to making the automated operation of the VTGS possible while ensuring future developers can easily add to its functionality. The two main contributions of this work consist of the application of an actor framework to the ground station internals and the proposal of a novel P2P protocol for a network of ground stations.

The thesis is organized as follows:

- Chapter 2: Review of related projects and prior works.
- Chapter 3: Waveform flow graphs and signal processing libraries, such as *gr-vtgs* and *planck-dsp*.
- Chapter 4: Actor model overview, history, and *pystation* framework.
- Chapter 5: A novel P2P protocol is proposed to handle a network of ground stations.

- Chapter 6: Concluding remarks and future work.

Chapter 2 discusses several other ground station projects focused on educational and scientific missions. These ground station projects serve as a basis for motivating some of the design choices made as part of this work.

Chapter 3 will describe the communication systems that a large proportion of CubeSats and SmallSats utilize, as well as the reasoning behind their choices. This is important because most of the descriptions concerning these methods are discussed with hardware implementations as the focal point, and there are many subtleties not often exposed in other literature or online resources. This chapter serves two purposes: to elaborate on the software developed for the VTGS and provide an accurate reference for several amateur radio transmission modes.

Chapter 4 describes the actor model and its history, as well as the *pystation* framework. The primary contribution of this chapter is the *pystation* framework, necessary for control and controlling the signal processing at the VTGS. Within the larger set of software, *pystation* is a single module with multiple dependencies, as shown in Figure 1.4.

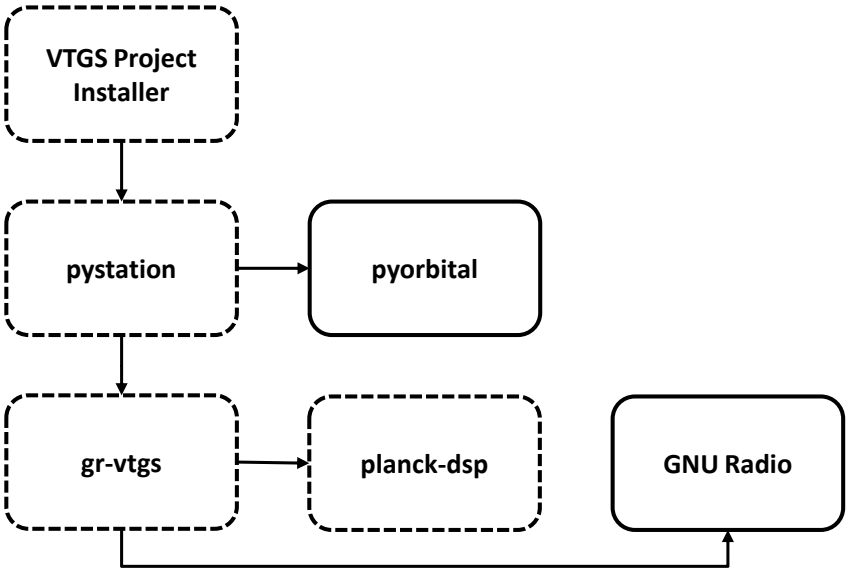

Figure 1.4: Software dependency graph of the Virginia Tech Ground Station, with *pystation* as the central module described in Chapter 4. The dotted outline indicate that the software was developed as part of this thesis, where the solid outline indicate a large dependency.

Chapter 5 describes a novel P2P protocol, where a network of ground stations do not rely on a centralized server, and utilize a Distributed Hash Table (DHT) of geohashes in order to activate a ground station peer closest to the spacecraft. The main concept behind the protocol is described as well as some of the technical details involving future integration into *pystation*.

The software developed as part of this thesis project shown in Figure 1.4 are listed as follows:

- *pystation* is the actor framework with the purpose of control messaging and management of ground station operations.
- *planck-dsp* is a small C library meant to isolate the Digital Signal Processing (DSP)

routines and unit tests from GNU Radio.

- *gr-vtgs* is an Out-of-tree (OOT) module that adds additional signal processing blocks to GNU Radio. This module wraps much of the code in *planck-dsp* and provides the following blocks:
  - Audio Frequency-shift Keying 1200 Baud (AFSK1200) modem: A full implementation featuring transmit and receive.
  - Frequency-shift Keying 9600 Baud (FSK9600) modem: Includes transmit and receive as well as the associated scrambling and descrambling routines.
  - AO-40 style encoder and decoder: Wrapped decoders based on Phil Karn's AO-40 spacecraft proposal [4].
  - An AX.25 framer and deframer: An implementation of both framers and deframers of the amateur radio protocol AX.25 not found within GNU Radio.

Chapter 2

Relevant Ground Station Projects

The choice of tools, programming languages, and hardware technologies uniquely determine the capabilities of a ground station. In order to motivate many of the design decisions made throughout this work, this section will briefly examine other relevant ground station projects and identify their primary features, similarities, and differences to the VTGS.

2.1 Generic Inferential Executor

There have been several frameworks to tackle the problem of automating satellite ground station activities. An early attempt was the Generic Inferential Executor (GENIE), created by NASA in 1996 to assist in managing spacecraft due to budget constraints and the possibility of reduced numbers of human operators. The spacecraft that this software was built to support, known as the Solar Anomalous and Magnetospheric Particle Explorer (SAMPEX),

ceased operation on June 30, 2004, after 12 years of its science mission, and re-entered the Earth's atmosphere on November 13, 2012 [6]. GENIE consisted of three components: graphical pass scripts, pass scripts, and the run-time script executor (or inference engine) as shown in 2.1. The graphical pass scripts allowed operators to define the steps necessary to mimic a ground operations team within a graphical interface. The purpose of the run-time script executor was to utilize a rule-based system to react to certain mission parameters. For instance, if a spacecraft fell below a certain measured altitude, such a rule-based system might send an uplink command to physically adjust the spacecraft, assuming the human operator agreed with that response. It served as a proof-of-concept of an expert system applied to ground station operations in order to reduce the number of tasks to be performed manually.

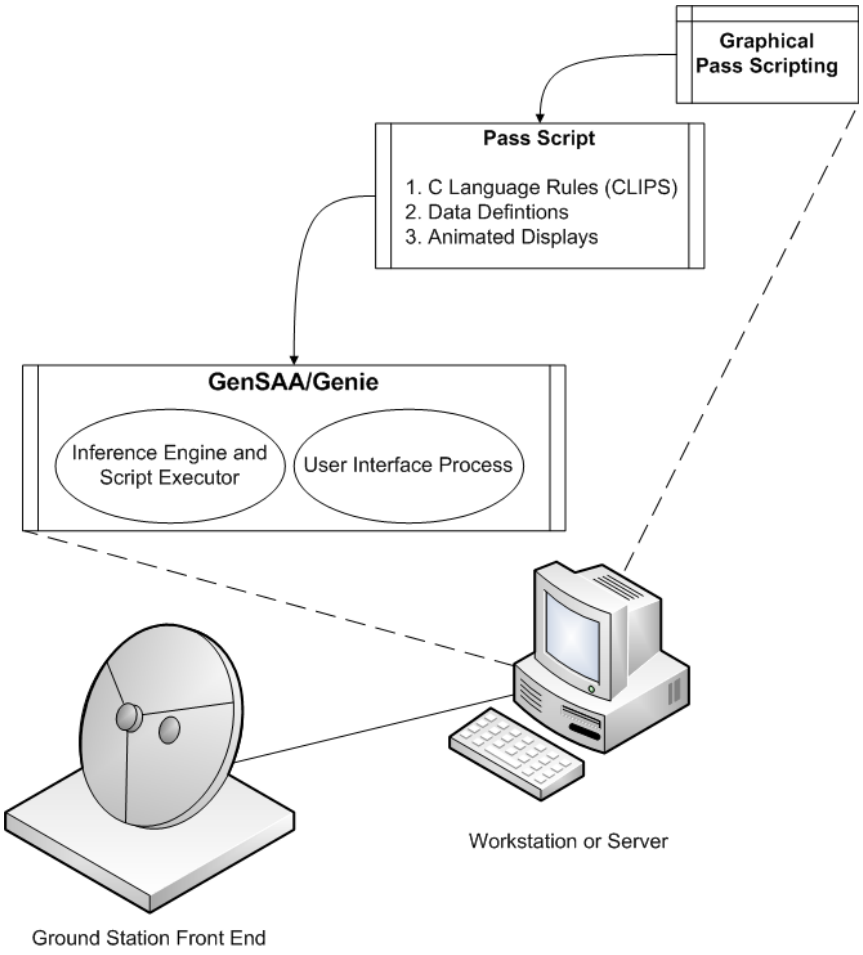

Figure 2.1: GENIE system overview as described by NASA. This figure was originally created in [1] J. Hartley, E. Luczak, and D. Stump, Spacecraft control center automation using the Generic Inferential Executor (GENIE), European Space Agency, (Special Publication) ESA SP, no. 394 PART 2, pp. 10071014, 1996. Used under fair use, 2015.

Expert systems technology found an early use in spacecraft control [1, 2] with GENIE, utilizing a rule-based inference engine based on graphical pass scripts, which allow users to easily define complex logic for spacecraft operation. This system allowed an operator to

quickly interact with the interface without needing programming experience. The graphical interface that it provided allowed an operator to build a model of the spacecraft and its various subsystems as well as define complicated rules for how to react to data from the spacecraft. GENIE appears to be one of the earliest attempts at satellite ground station automation. Part of the reasoning behind the developed software framework, as discussed in Chapter 4 for the VTGS, was to enable automation of scheduling and TT&C of spacecraft in such a way to reduce the number of performed manual tasks. If an operator was necessary for every interaction with the software developed as part of this work, it would ultimately reduce its utility for a student lab setting or any other scenarios involving shared use.

2.2 Autonomous Satellite Operations Framework

The Autonomous Satellite Operations Framework (ASOF) uses three major components in its implementation of an automation framework: a central agent, knowledge base, line of sight executive process, task file, and the Terminal Node Controller (TNC) as described in [2]. An overview is shown by Figure 2.2.

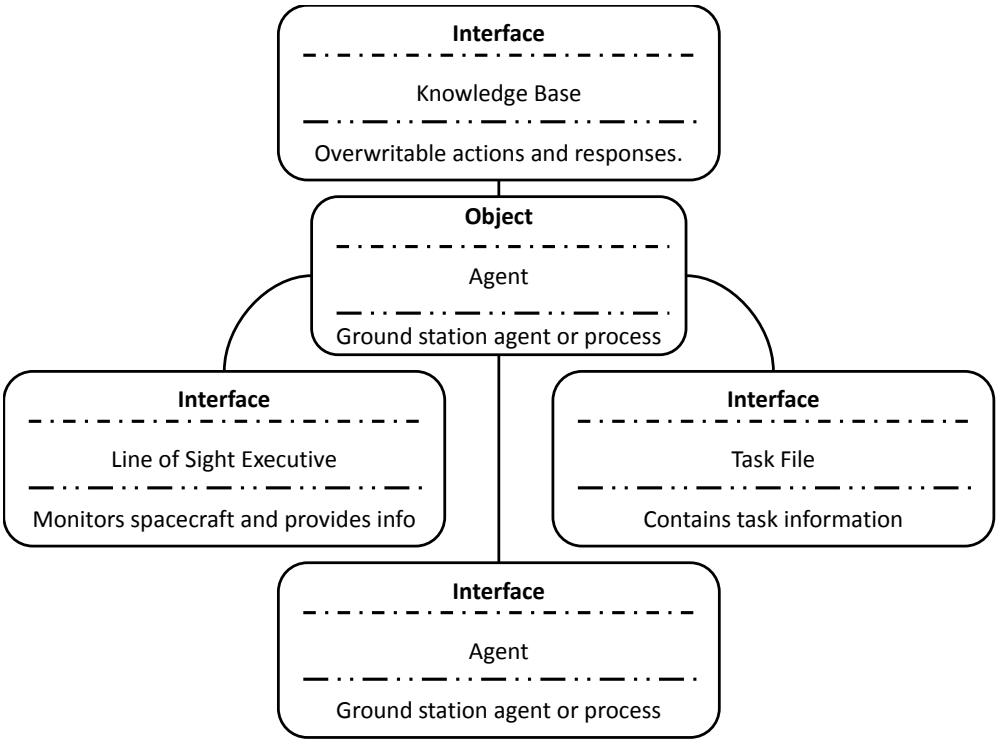

Figure 2.2: Autonomous Satellite Operations Framework overview which describes the software's overall structure, originally created in [2] J. Anderson, Autonomous Satellite Operations For CubeSat Satellites, Masters thesis, California Polytechnic State University, 2010. Used under fair use, 2015. Note that the terms *object* and *interface* apply to *Java* programming.

Automation in this system is typically carried out by its knowledge base and task files, which is loaded by a central agent. Each knowledge base implementation is uniquely defined for every supported satellite. In turn, there are uniquely defined tasks for every possible mission and some general tasks are available to all knowledge base implementations. In contrast, the VTGS utilizes an actor framework developed as part of this work for ground station operations for task distribution. In addition, a knowledge-base is maintained as flow graphs

that contain profile information for each waveform that would correspond with a particular spacecraft. These flow graph objects are accessed by the developed actor framework during operation. The VTGS also performs all of its signal processing in software using modems developed in this work, and utilizes SDRs. Instead of using a TNC, all of the framing is parsed in software, allowing for new or altered framing structures to be added quickly and without the need for new hardware. Utilizing SDRs ensures that the VTGS is not locked down to any particular protocol or transmission mode, and can be extended to support future framing structures or modems. The developed VTGS software framework in Chapter 4 also provides the infrastructure for adding more actors to perform tasks that might be needed for a future project or experiment, making it more flexible in comparison to the ASOF.

2.3 TU Wien Satellite Ground Station

A multi-mission satellite ground station was developed at the Vienna University of Technology in order to support communications and education with CubeSats. As discussed in [7], the concept of the ground station centered on separate control interfaces through a browser and *MATLAB* [8]. Remote connections through a Virtual Private Network (VPN) are also possible. In contrast with the VTGS, the TU Wien ground station relies on hardware modems for specific satellites. For instance, TNCs are used as the primary machines for decoding AX.25 packets and transmitting them to the host machine via the KISS protocol over a serial connection. However, this ground station is inflexible due to the use of

hardware modems; it is not possible to add additional signal processing routines or support for other satellites without altering the hardware layout of the ground station. Adding the use of SDRs to the VTGS RF front-end moves much of the signal processing to software, and allows for additional waveform development for future spacecraft. The software developed as part of this thesis provides the baseline signal processing capability for many of the digital transmission modes used in communication with spacecraft using amateur radio techniques. In addition to the modems developed in this work and discussed in Chapter 3, the software framework enables expansion in the case of new spacecraft with differing waveforms or protocols.

2.4 Global Educational Network for Satellite Operations

Global Educational Network for Satellite Operations (GENSO) was an international effort to create a ground station network on a global scale by involving universities and amateur radio hobbyists [9]. By allowing the registration of trusted ground stations, both uplink and downlink sessions are possible. The entire GENSO architecture works by adding a GENSO layer between the application and network layers on the Open Systems Interconnection (OSI) model. The software contains multiple components: ground station server, mission control client, and an authentication server. An individual ground station contains the following components: rotator controller, TNC, and a radio controller. All of this software

was originally written in *Java* under the supervision of the European Space Agency (ESA). It is certainly one of the original distributed ground station networks conceived and executed upon by the international community in order to conduct educational outreach and increase ground station coverage for scientific missions. The GENSO project seems to have slowed over the years, and the level of participation is unknown. Since ground station operations relies on a centralized server, if the GENSO service were to go down, the entire project would be inactive. In order to increase coverage and provide additional access time to missions supported by the VTGS, a P2P protocol was proposed in Chapter 5 to avoid the issue of relying on a centralized server. The intention is to have the VTGS be an initial peer to encourage other ground station projects to join the network of ground stations.

2.5 Distributed Ground Station Network for CubeSats

An early implementation of low-cost satellite receiver nodes can be found in [10]. It also represents an early proposal for a global distributed ground station network of low-cost satellite ground stations, with SDRs as a principle component in the design. The author in [10] utilized an embedded micro-computer known as the BeagleBone Black for control and signal processing. GNU Radio was used as the primary means of signal processing in this project [11]. After initial quadrature demodulation, an audio signal was piped through a socket connection to decoder software known as *multimon-ng* [12].

In order to focus on the downlink, most amateur radio spacecraft use a higher rate mode, like

9600 Baud Frequency-shift Keying (FSK), due to increased utilization. For the uplink, they usually would use a lower data rate like 1200 Baud. This obviously depends on whether the spacecraft even utilizes the Amateur Satellite Service. The different physical layer modes, such as AFSK and GMSK, will be part of the discussion on the signal processing software for the VTGS. The project presented in [10] was an approach to creating a distributed network of low-cost ground station receivers. The approach presented in this thesis is far more general, supporting ground stations with both transmit and receive capabilities, as well as larger and more complex operations across a network with many concurrent tasks. In addition, a P2P protocol is proposed in Chapter 5. The project reviewed in this section does not propose a protocol for linking the ground station receivers.

2.6 Satellite Networked Open Ground Station

Satellite Networked Open Ground Station (SatNOGS) is an open source implementation of a distributed ground station network of participating nodes as well as an open hardware design of a low-cost receiver [13]. The software is largely written in *Python* with a centralized web based server, making it ideal for community efforts to track satellites due to an accessible Graphical User Interface (GUI). SatNOGS allows amateur satellite hobbyists to establish their own ground stations based on open hardware kits, all of which are connected to a global ground station network. From there, users can schedule observations and downlink recordings, and ground stations are rated by their success rates. Any type of personal

computer can be used for control, such as RaspberryPi, BeagleBone Black, or Odroid, making each ground station node small and affordable.

For an open source community restricted to downlink utilization, SatNOGS is a great application of modern development platforms, micro-computers, and web technologies. At the VTGS, the software framework has been developed in such a way to allow for integration to a web interface for a limited amount of control and operation. Such an interface would be convenient since it is hard to write desktop applications that are cross platform across all possible operating systems and devices. As mentioned in Section 2.5, the work presented in this thesis provides a way to avoid a centralized web server by proposing a P2P protocol in Chapter 5. In addition, the developed software framework is general enough to scale from a small, low-cost receiver, to a large and complex network of dedicated machines at a ground station peer.

2.7 Comparison of Features for the VTGS

This chapter analyzed the similarities and differences between various ground station projects and the work developed for the VTGS. Table 2.1 represents a capability matrix between the ground station projects discussed in this chapter and the planned capabilities for the VTGS. These capabilities were enabled by the software framework developed as part of this thesis work, such as the signal processing blocks in Chapter 3, the actor model for distributing tasks over a network of dedicated servers in Chapter 4, and the proposed P2P protocol for

extending to a network of ground stations in Chapter 5. While this list does not contain all ground stations in existence, Table 2.1 serves to highlight some of the key differences between this work and others.

Features	VTGS	GENIE	ASOF	Tu Wien	GENSO	Distributed GS	SatNOGS
Internal LAN	✓						
Network of Ground Stations	✓				✓	✓	✓
Software-Defined Radio	✓					✓	✓
Modular Software Framework	✓						✓
Transmit Capabilities	✓	✓	✓	✓	✓		

Table 2.1: Capability matrix comparing some of the features of the VTGS to other various ground station projects.

Chapter 3

Waveform Development at the VTGS

3.1 Digital Transmission Modes and Protocols

While there is no set standard among SmallSat and CubeSat spacecraft for their frequency bands or communication protocols, many of them utilize amateur radio frequencies [10]. Typically, CubeSats and SmallSats will utilize amateur radio digital transmission modes and equipment, even if they are used for experimental purposes, to comply with the FCC rules on emissions and bandwidth. In addition to complying with regulations, support of sometimes cheaper and plentiful legacy equipment such as Narrowband Frequency Modulation (NBFM) radios, is often a major point in using designs derived from amateur radio. As summarized in Section 2.3 of [10], and explicitly specified by the FCC in Title 47 Code of Federal Regulation [14], small spacecraft are typically governed under Title 47, Part 97,

which concerns the amateur radio service. Operators of small satellites with scientific purposes often file for experimental licenses which are covered under Title 47, Part 5 of the Code of Federal Regulations [14]. Scientific missions sometimes derive their technology from amateur radio techniques to comply with internationally respected regulations for related services. The typical data volume from CubeSats with sensors or telemetry data is usually small enough to fit in the amateur radio bands, as shown in Table 3.1.

Common Amateur Radio Bands for CubeSats and SmallSats [15, 16]		
Range	Band	Allocation
HF	40 meter	7.000 - 7.100 MHz
	20 meter	14.000 - 14.250 MHz
	17 meter	18.068 - 18.168 MHz
	15 meter	21.000 - 21.450 MHz
	12 meter	24.890 - 25.990 MHz
	10 meter	28.000 - 29.700 MHz
	2 meter	144.000 - 146.000 MHz
VHF	70 centimeter	435.000 - 438.000 MHz
UHF	23 centimeter	1.260 - 1.270 GHz
	13 centimeter	2.400 - 2.450 GHz

Table 3.1: Common Amateur Radio Bands for CubeSats, SmallSats, and those that fall under the general umbrella of the Amateur Satellite Service. The Amateur Satellite and Amateur Radio Service are viewed under the same laws regulating bands in coordination with the International Telecommunications Unions (ITUs) rules.

AFSK1200, GMSK9600, and FSK9600 are all variants of Continuous Phase Modulation (CPM). Sometimes Gaussian pulse shaping is used, which means that Gaussian Minimum-shift Keying 9600 Baud (GMSK9600) modems can optionally employ a matched filter on the receive side. In order to avoid complexity, many modems do not implement pulse shaping on the receive side. With a high enough carrier-to-noise ratio, implementation losses due to an unmatched filter can easily be overcome at the receiver or ground station. Some amateur spacecraft, such as Funcube-1, utilize PSK in their designs in order to take advantage of the performance gain and add some sophistication in the realm of forward error correction, which is not often present in small spacecraft. Details of the Funcube-1 and AO-40 inspired spacecraft will be discussed later in this chapter.

Examples of common SmallSats and CubeSats frequencies and digital transmission modes can be found in [17] with an up-to-date list of active, re-entered, and inactive spacecraft. Due to the software-defined nature of the VTGS, additional modems can always be added to a library of GNU Radio flow graphs, enabling the ground station to support any potential protocols or digital transmission modes. The software architecture developed in this work resulted in a extensible framework that allows for easily expanding the flow graph library and integrating new signal processing. As noted in Section 2, several ground station projects utilize hardware modems as their front-ends, usually with packet decoding performed on a TNC, which is tremendously limiting if a new spacecraft is launched. Within GNU Radio, new waveforms and framing protocols can be deployed with the only cost being time, unless of course the front-end fails to support the chosen frequency bands of interest.

The main point of CubeSats, or small spacecraft in general, is to provide a low-cost space platform for education, scientific experiments, or amateur radio communications. Typically, this means that CubeSats often make use of Commercial Off-the-shelf (COTS) components and attempt to accomplish communication tasks with relatively simple transmission modes and protocols in order to support existing equipment. The following sections will attempt to elaborate on the specific modes and protocols that were developed and integrated into GNU Radio as part of this work.

3.2 Audio Frequency-shift Keying 1200 Baud

AFSK1200 is a digital transmission mode used in amateur radio for packet radio and Automatic Packet Reporting Service (APRS). It is based on the Bell-202 modem, which is a standard developed by Bell System [18]. The *gr-vtgs* and *planck-dsp* implementations provide a full AFSK1200 modem with transmit and receive capabilities as shown in Figure 3.1.

Figure 3.1: The communication flow graph of a Bell-202 style modem. The blocks in blue indicate those developed within *planck-dsp* and contained in *gr-vtgs* for use in GNU Radio.

It employs FSK with a "mark" tone at 1200 Hz, and a "space" tone at 2200 Hz, representing

a binary one and zero, respectively. The mode is often used by APRS, a packet radio system that is responsible for making positioning reports and relaying transmissions between users. A well-written description of APRS, as well as certain ambiguities in its design are available in [19]. AFSK1200 is well within the range of most devices meant to carry audio. Thus, AFSK1200 is typically placed on an FM carrier, adding an outer modulation layer and encapsulating the FSK signal. In addition to the application of inner and outer modulation schemes, data is typically line-encoded as Non-return to Zero Inverted (NRZI) as shown in Figure 3.2. This sort of line-coding can still result in a long run of ones since the transition occurs on a zero, and requires the use of some supplemental run-length limited code. This is normally provided by AX.25, an amateur radio link-layer protocol described in Section 3.5.

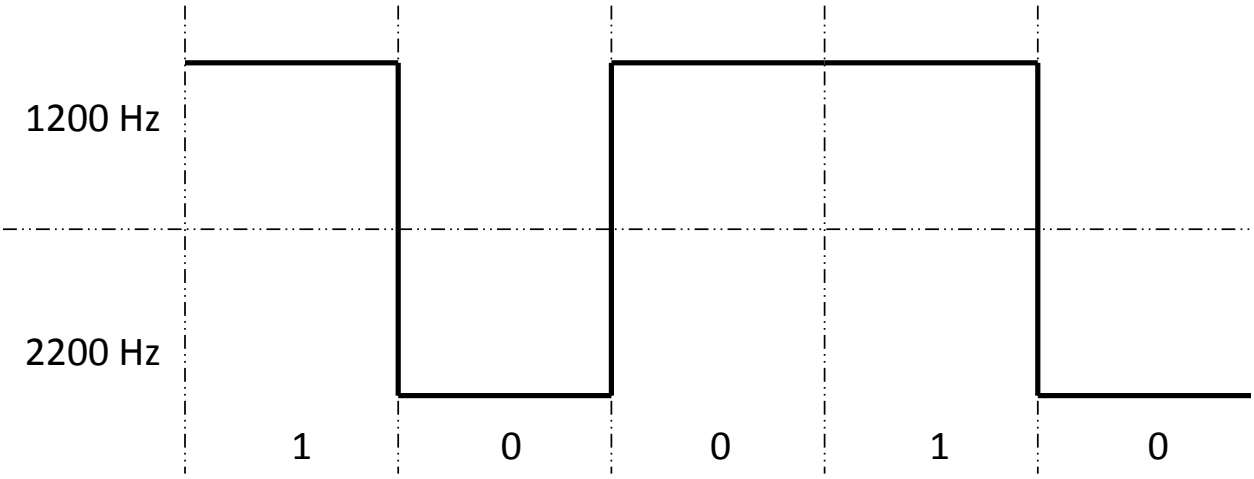

Figure 3.2: Audio Frequency-shift Keying 1200 Baud (AFSK1200) modems use Non-return to Zero Inverted (NRZI) line coding, where a zero bit indicates a change in frequency and a one bit indicates that the frequency remain the same. The phase transition is also continuous, as previously mentioned.

In order to test the AFSK modem implementation with Universal Software Radio Peripherals (USRPs), an APRS TinyTrak4 GPS encoder was used to transmit its position with a hardware radio [20]. While the conditions were not like that of the channel between a ground station and a spacecraft in LEO, these tests do provide a test that is more realistic than a closed loop simulation. Figure 3.3 shows the GNU Radio implementation as well as the terminal output using the message debug block. Note that the AX.25 framing structure discussed in Section 3.5 is used to ensure that the packets contain a passing Frame Check Sequence (FCS).

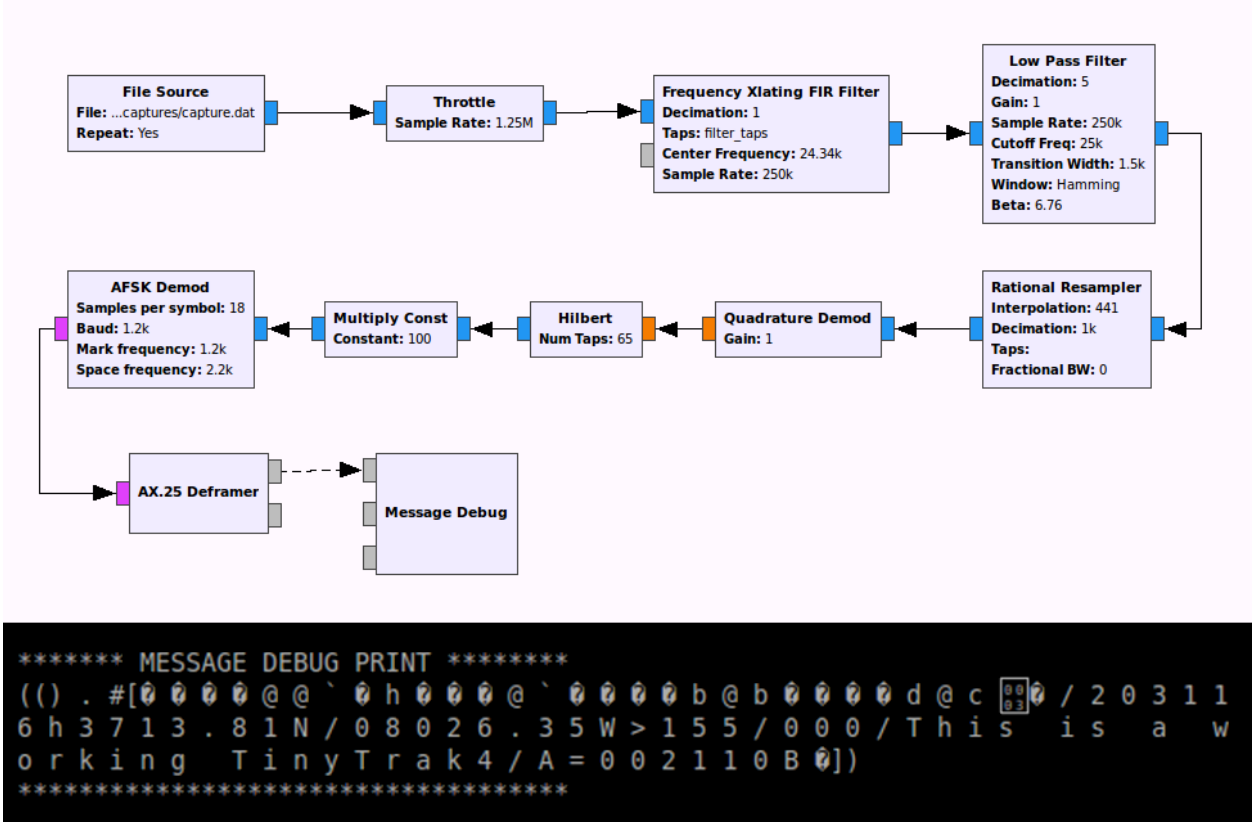

Figure 3.3: GNU Radio flow graph utilized to decode an AFSK 1200 baud APRS message with the terminal output.

3.3 Frequency-shift Keying 9600 Baud

The FSK9600 digital mode, often found in amateur radio circles for "high rate" packet radio, is a variation of Continuous Phase Frequency-shift Keying (CPFSK) with the addition of a scrambler and the optional addition of a Gaussian filter. It is well suited for voice, terrestrial and satellite communications. In particular, the KD2BD 9600 Baud modem article found in [21] describes a hardware implementation of FSK9600 but does not mention Gaussian filtering in its implementation. Figure 3.4 illustrates a particular layout of a communications chain with this amateur radio mode.

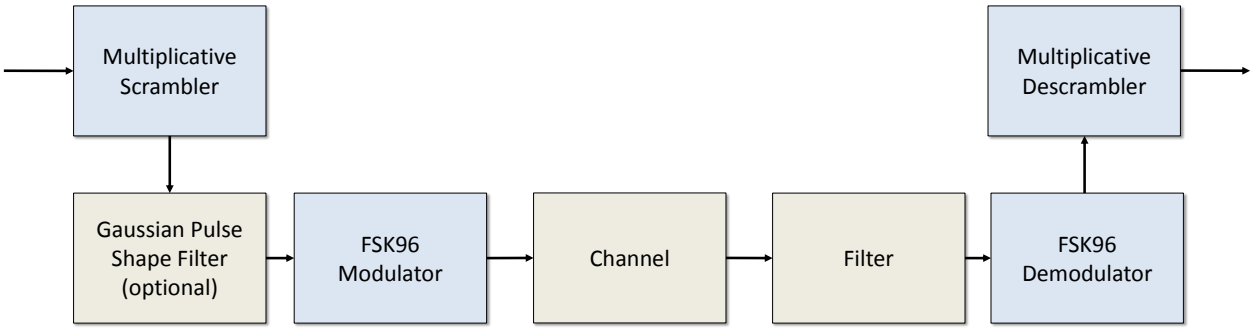

Figure 3.4: Implementation of a full FSK 9600 Baud communication chain. Note that the blocks in blue indicate those that were added to GNU Radio through the software packages *planck-dsp* and *gr-vtgs*.

Like in the previous section, an APRS TinyTrak4 GPS encoder was used to test this modem over the air [20]. The TNC, hardware radio, USRP receiver were located in the same room, yielding a high Signal-to-noise Ratio (SNR). A similar location report was decoded with a passing FCS in Figure 3.5.

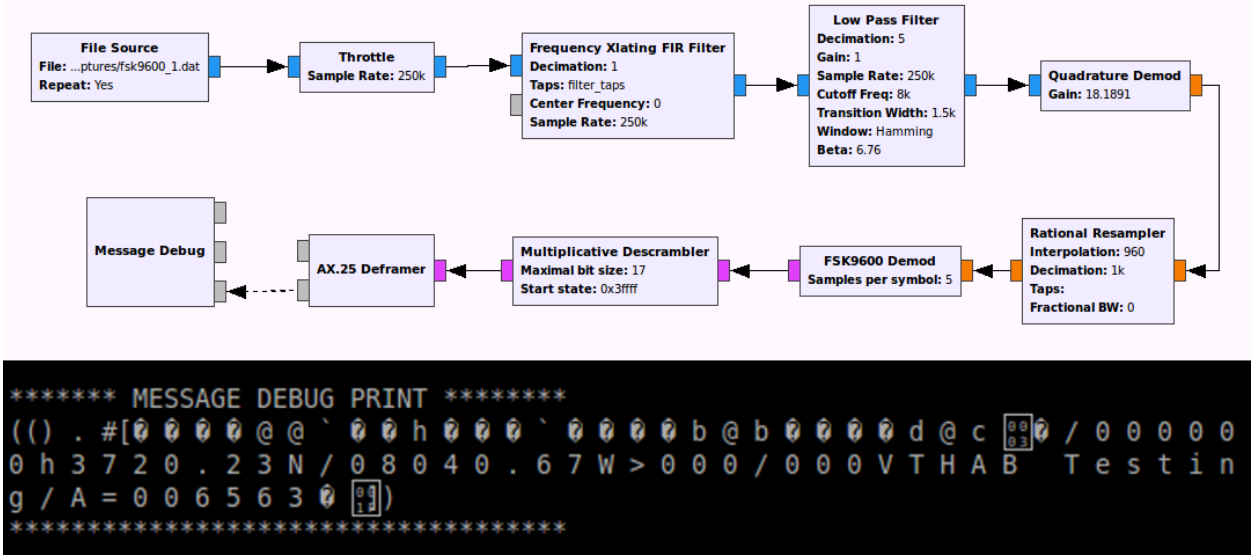

Figure 3.5: GNU Radio flow graph utilized to decode an FSK 9600 baud APRS message with the terminal output.

3.3.1 FSK9600 Scrambler

There are two main types of scramblers: additive and multiplicative. Both types of scramblers are provided in the *gr-vtgs* module to support any eventualities at the ground station. Scramblers are typically used to make the data appear more random in a statistical sense. Figure 3.6 displays the two structures of the scrambler pairs with the polynomial used in FSK9600 implementations.

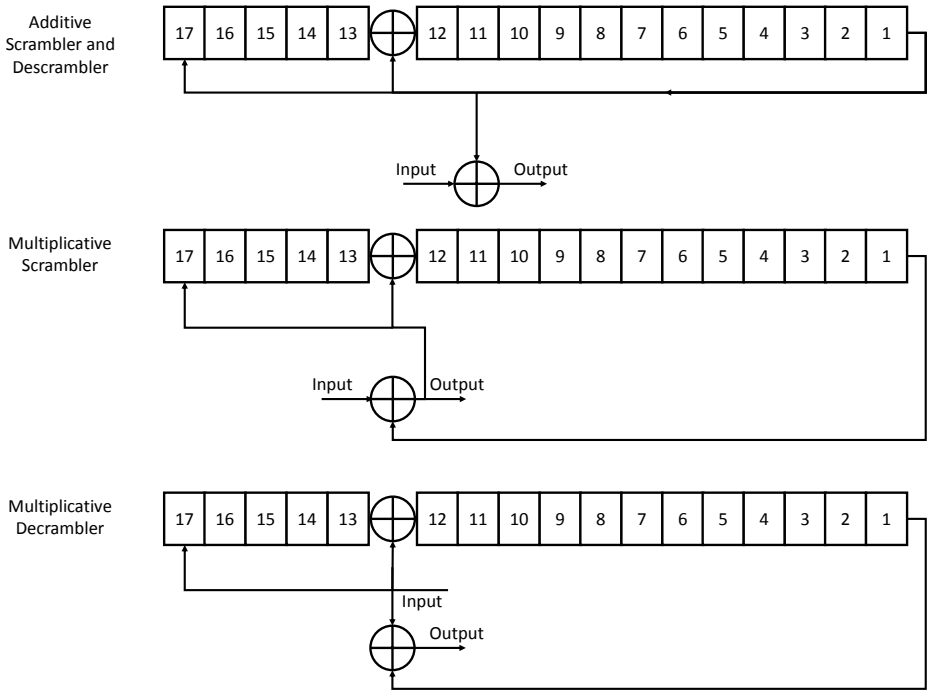

Figure 3.6: Comparison of the two different scrambler types. All of the scramblers utilize a Galois Linear Feedback Shift Register (LFSR) structure, which allows an efficient implementation with the polynomial as a bit mask [3]. The polynomial shown in these diagrams corresponds to the one utilized in the amateur mode FSK9600: $1 + x^{-12} + x^{-17}$.

As illustrated in Figure 3.6, a typical FSK9600 modem will utilize a multiplicative scrambler and descrambler for transmission and reception. This is due to the self-synchronizing property of a multiplicative descrambler. The main reason for utilizing a scrambler is to improve synchronization results by randomizing the data so that it is independent and identically distributed [22]. For packets, if some part of the data or payload contains a sequence similar to that of the frame preamble, it could affect the synchronization process or lead to false

alarms. The primary issue with multiplicative scramblers in comparison to their additive counterparts is bit error propagation. If a single bit is flipped by the channel, there is a chance that the descrambled version will contain more bit errors than it otherwise would if it had not been scrambled. A single bit error could turn into as many errors as there are registers [22].

3.4 FUNcube-1 (AO-73) Decoder

A small (10x10x10 cm) CubeSat known as FUNcube-1 (or AO-73), was launched November 21st, 2013 [23]. In order to properly test the VTGS software framework based on contributions of this thesis, it was necessary to use a CubeSat or SmallSat that broadcasts telemetry information over North America. FUNcube-1 met this criteria since one of its major goals was to support the FUNcube Dongle, an SDR receiver meant to allow users to display telemetry information with their software. FUNcube-1 is unique in the sense that it does not support the dominant modems previously discussed in this chapter. In fact, it utilizes differential Binary Phase-shift Keying (BPSK) modulation with some heavy forward error correction, which was originally proposed and written by Phil Karn in [4]. It also has its own custom framing instead of the AX.25 protocol. Figure 3.7 describes the decoder chain formed in GNU Radio.

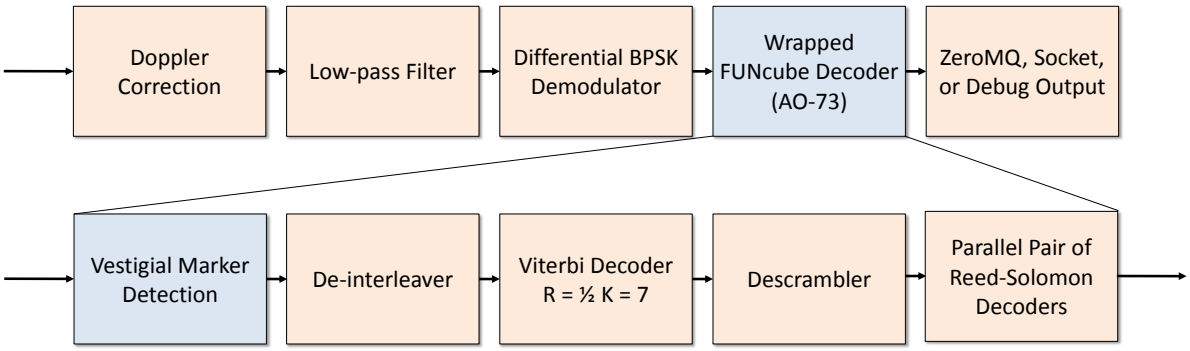

Figure 3.7: Wrapped AO-40 inspired decoder for the FUNcube-1 CubeSat (AO-73). Much of the original forward error correcting code was written by Phil Karn [4]. Note that the blocks in blue indicate functionality that was added in *planck-dsp* and *gr-vtgs*.

To ensure that the decoder works with an actual ground station receiver, some of the older ground station hardware was utilized to record the complex baseband samples of an actual pass of the FUNcube-1 CubeSat. While most of the data consists of telemetry for the FUNcube-1 dashboard project [24], periodic data about the technical team is also transmitted. This data was successfully decoded from a live capture at the current location of the VTGS, as shown in Figure 3.8, which shows the GNU Radio flow graph and terminal output.

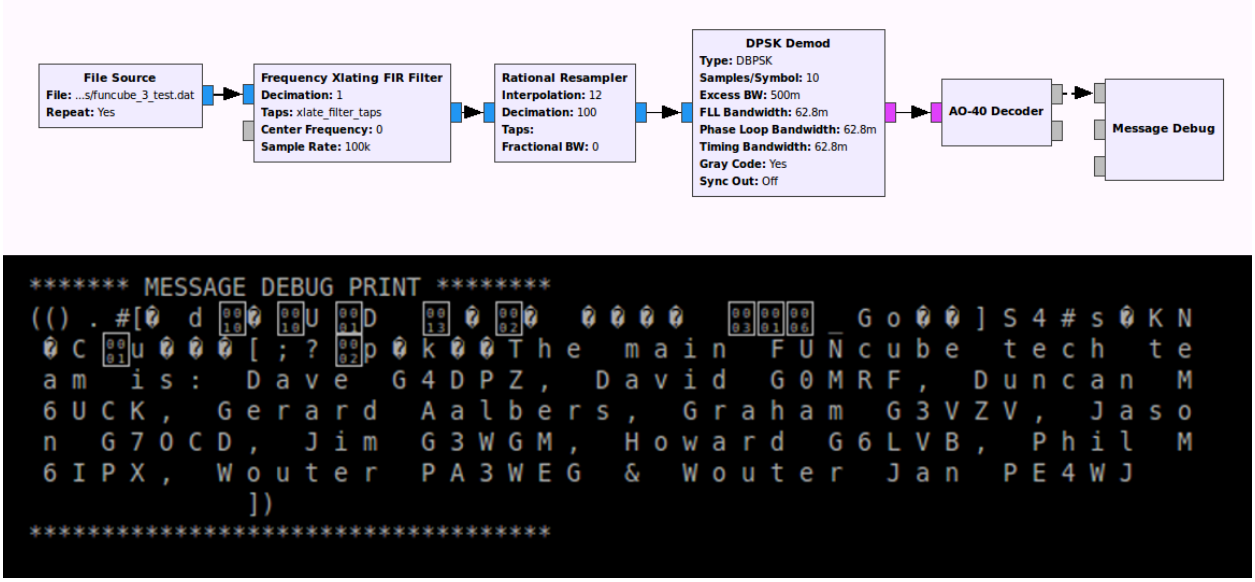

Figure 3.8: GNU Radio flow graph utilized to decode frames from the FUNcube-1 spacecraft with the terminal output crediting the FUNcube-1 team.

3.5 AX.25 Protocol

The Amateur Packet-Radio Link-Layer Protocol, known as AX.25, defines a link layer for terminals with a variety of different digital transmission modes. This chapter took a sample of some of the amateur transmission modes used on small spacecraft, which could easily serve as the physical layer underneath AX.25. AX.25 conforms to specifications set out by the International Organization for Standardization (ISO) in their High-Level Data Link Control (HDLC) protocol [25]. As with most link-layer protocols, there are more sub-layers contained within that particular layer itself.

The AX.25 protocol defines primitives which facilitate entities on the data link-layer to

communicate with each other, as shown in Figure 3.9. This is really where the logic behind the communication system resides. The segmenter handles breaking up data if it exceeds the Maximum Transmission Unit (MTU) size as defined by the AX.25 frame sizes. The link multiplexer is responsible for controlling access to the channel as the name suggests. The management data link is responsible for the parameters established for the link. And finally, the data link handles all the logic relating to making connections between two stations, like a spacecraft and ground station, and whether those frames are connection-less. Also, it is possible to have multiple links on a single piece of equipment as indicated in Figure 3.9 by the multiple Data-Link Service Access Points (DLSAPs). Figure 3.10 illustrates the three basic types of frames: unnumbered, supervisory, and information formats.

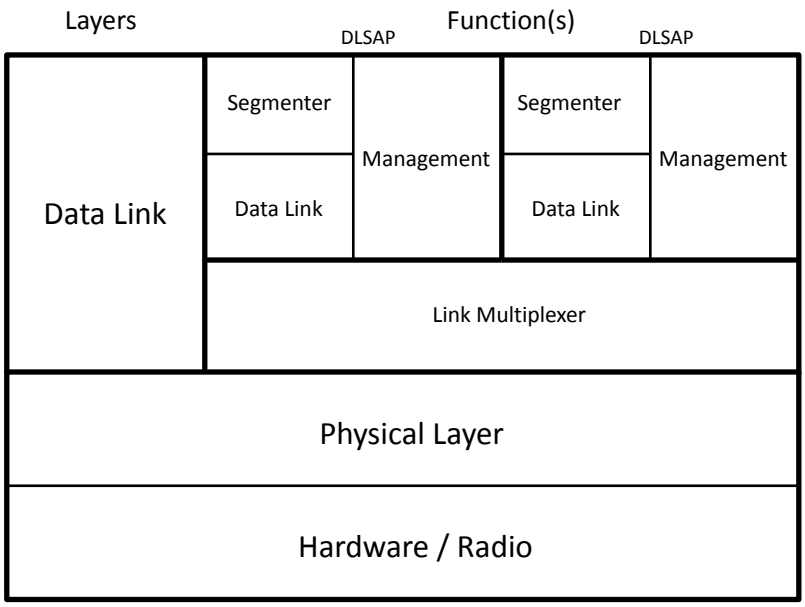

Figure 3.9: The AX.25 Protocol stack as described by the standard [5] W. A. Beech, D. E. Nielsen, and J. Taylor, AX. 25 Link Access Protocol for Amateur Packet Radio, Tucson Amateur Packet Radio Corporation, pp. 1133, 1998. Used under fair use, 2015. As shown by the multiple Data-Link Service Access Points (DLSAPs), a piece of equipment can have more than one end-point as well as multiple physical channels.

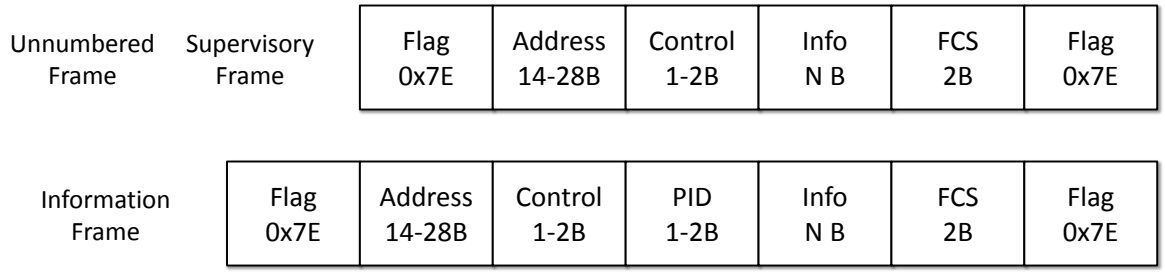

Figure 3.10: The three main AX.25 frames as described by the standard [5] W. A. Beech, D. E. Nielsen, and J. Taylor, AX. 25 Link Access Protocol for Amateur Packet Radio, Tucson Amateur Packet Radio Corporation, pp. 1133, 1998. Used under fair use, 2015. Note that the supervisory and unnumbered frames have the same size limitations in octals (bytes).

As part of the development of the basic flow graphs for the VTGS, an AX.25 framer and deframer pair of signal processing blocks were developed for GNU Radio. The framer block reads in a series of bits, packs them into bytes, and calculates the FCS using the CCITT polynomial for the checksum calculation. Prior to transmission, the framer performs an operation known as bit stuffing. In order to avoid a run of ones that would result in the AX.25 flag being misidentified, the standard [5] mandates that a zero be placed after a run of five ones. The deframer simply "unstuffs" the bits by removing a zero after receiving five ones. It is important to note that the receiver must first identify the starting AX.25 flag and wait until it identifies the terminating flag, otherwise the "unstuffing" operation might be accidentally performed on a frame terminating flag.

Chapter 4

VTGS Actor Model Framework

4.1 Background and History of the Actor Model

This section attempts to give some relevant background information on the actor model for concurrent and distributed systems as well as the details of a custom actor framework designed for the VTGS. Other actor model implementations will briefly be discussed in order to aid in reviewing some history and elaborating on differences and design decisions in the VTGS actor framework.

The actor model centers around the idea of independent primitives of computation, which communicate only via message passing and share no state. An illustration of actor communication is shown in Figure 4.1.

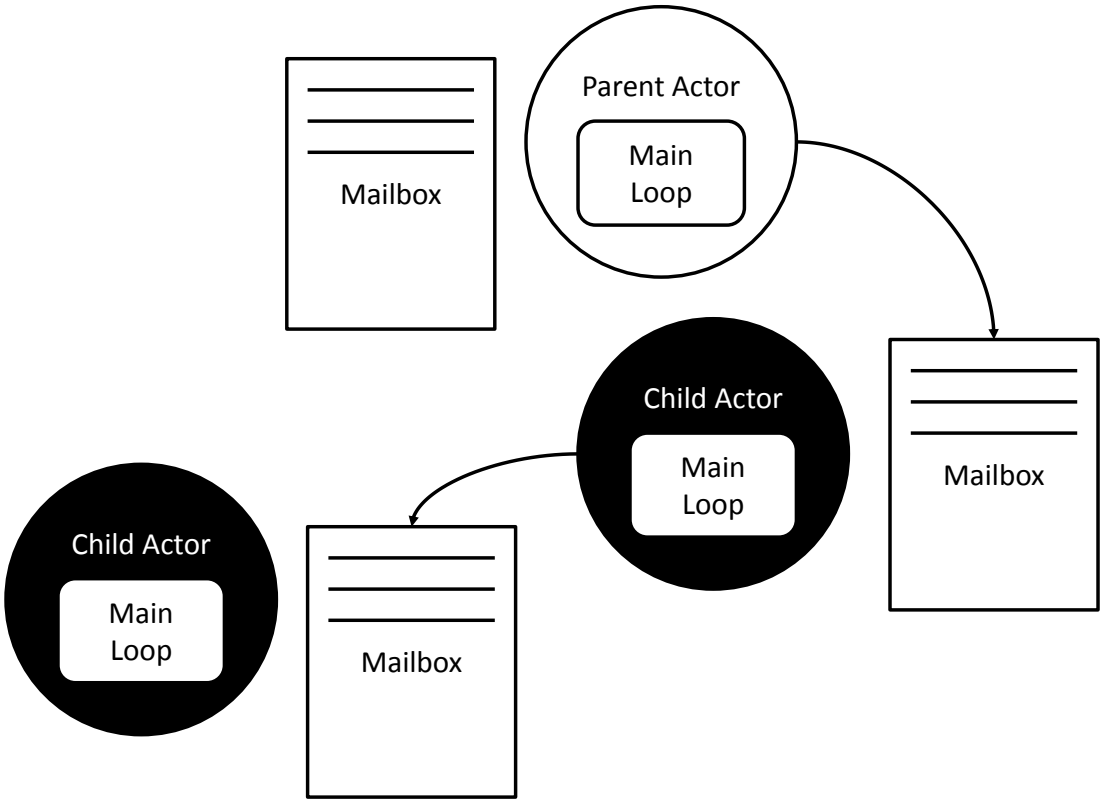

Figure 4.1: The idea of concurrent actors and isolated mailboxes. Messages are delivered to independent mailboxes without sharing memory (usually a queue of some sort).

Each actor has a mailbox from which it pulls delivered messages, where order is not guaranteed in the general model. An actor can also change its behavior in response to a message or create child actors, allowing for dependencies to be created during run time. Actors have very general properties which allow entire programming languages to be built around them [26]. Such a model circumvents the issues found in concurrent systems that make use of shared memory, where programming mistakes can lead to race conditions and deadlocks.

A simple example of this would be two threads attempting to access a memory buffer at the same time. In addition to aiding the programmer against concurrent pitfalls, actors encourage intuitive software design, allowing developers to organize a distributed system in a manner similar to creating an intuitive graph of tasks and their connections. Depending on whether an actor is a thread or process, it is conceptually independent from the underlying low level operating system mechanics. Thus, an actor is easily extendable to distributed and networked systems. The implementation details and differences between remote and local actors should be barely visible to the user of the framework as a consequence of abstractions.

The actor model has its origins in the question of formalizing concurrency and parallelism. Many authors addressed several aspects of the actor model, such as operational semantics [27], general laws [28], and other proofs related to actor language semantics [29, 30]. One of the first publications formalizing the actor model was published by Carl Hewitt [31], laying down the foundation for the model within computer science. This implementation of an actor framework, among others, follow many of the definitions developed by Gul Agha found in [32], which was first published in 1985 and reprinted in 1986. Agha defines the operational semantics of an actor programming language in his work, verifying important properties about how an actor-based program is interpreted. Karmani further expounds on the history of the actor framework in [33] and delves into more details on the advantages the actor model provides as well as important semantic properties of actor languages.

The earliest implementation of an actor language was *Act 1-3* [34, 35], followed by more popular implementations like *Erlang* and *Scala* [36, 37]. *Erlang* saw widespread usage,

beginning in the telecommunications industry, and was originally developed by Ericsson. Modern programming and scripting languages include the likes of *Akka* and *Pykka*, which are similar implementations created using the Java Virtual Machine (JVM) and *Python*, respectively [38, 39]. There have been other actor frameworks in modern *C++* and *C++11* such as Theron and the C++ Actor Framework [40, 41]. Theron even provides benchmarks in [40], proving that it has excellent performance, supporting a peak throughput of 10 million messages per second. However, the primary purpose of the actor framework developed in this work is to provide ease of use to those with limited programming experience while being capable of handling the required message passing latency for the ground station as a whole. The following section discusses this actor framework and discusses its motivation as well as software itself as applied to ground station operations.

4.2 *Pystation* Actor Framework

In order to organize all of the different services across multiple nodes, *pystation* was created as an actor framework to allow for actor-style programming and to hide many of the issues encountered when designing concurrent and distributed systems across a network. This particular actor framework is based on a small actor implementation by Donovan Preston [42], which motivated the initial use of Web Server Gateway Interface (WSGI) for node-to-node communication. His simple implementation of an actor framework in *Python* served as an initial example with networking. The incentive for using *Python* to create the ground

station software was due to the dependency on GNU Radio and its ease of use for beginners and new developers. GNU Radio uses Simplified Wrapper and Interface Generator (SWIG) to expose *C/C++* signal processing with *Python*, effectively rendering the scripting language as a glue between blocks. This makes *Python* a perfect choice for this framework since the intent is to use GNU Radio flow graphs as the physical layer of the VTGS.

There are several differences between *pystation* and other actor frameworks, which make it suitable for control messaging and organizing tasks, but inefficient for lower level tasks. It has the purpose of making local and remote procedure calls via actors possible, and exists as a bridge between scheduling services and ground station subsystems. Most actor frameworks are capable of handling all low level tasks while providing concurrency like accessing files or spawning any necessary workers. However, *pystation* is really a hybridized framework that uses actors to address problems that require maintaining state, but also includes several features that are outside the actor model. The *pystation* framework concentrates on the VTGS scheduling and controls.

4.2.1 Actor Framework Motivation

There were several design and engineering choices behind the decision to create a custom actor framework in *Python*. As mentioned in the previous section, several actor languages and frameworks exist that would have saved time in the short term, but introduced complications in the long term. For instance, *Pykka* is a *Python* actor framework, but does not include

support for multiple processes or communication over a network. This is non-ideal and would have required heavy modifications in order to adapt for the ground station. Another issue heavily affecting the choice of language was accessibility of new developers to the VTGS project as a whole. It is important that new communications engineers or students, some of which are not necessarily experienced with many programming languages or programming in general, be able to jump into the project with a minimal learning curve. Other issues would have occurred with actor-based languages such as *Erlang*. For example, exposing GNU Radio to *Erlang* would be very difficult since the language does not follow a traditional programming model. Utilizing *Python* to create an actor framework means that accessing GNU Radio simply requires an import statement and nothing more.

In short, the primary reason for building a custom framework was to enhance control and utilize *Python* as a syntactically approachable programming language.

From a security standpoint, none of the other actor model solutions offer any real advantages. For instance, the only security feature of *Erlang* is magic cookies in plain-text. If a node were compromised and the shared cookie was known, all of the other nodes would be compromised. In addition, *Erlang* does not ensure cryptographically secure communications over a distributed network [36, 43]. Hence, it is advised to conduct communications over a secured network of some kind, and not expose individual nodes with any of the aforementioned actor model frameworks or languages capable of networking. Future work concerning adding a network security layer to the *pystation* actor framework is discussed in Chapter 6.

4.3 Asynchronous Message Passing

Message passing is a fundamental part of the developed *pystation* actor model. In order to avoid sharing state among actors, information is transmitted from one actor to another via delivery to a mailbox. Message passing between remote nodes is also possible by treating remote messaging as Remote Procedural Calls (RPCs). Transparency is built into the actor framework, allowing the programmer to think up a network of actors and distribute them across multiple nodes (machines). Many messaging patterns are possible between actors, such as one-shot, request-reply, and divide-and-conquer [33]. We have already seen an example of request-reply in the ping-pong scenario from the previous section. The bit-wise left shift operator has been overridden for the purpose of actor messaging to add convenience. In the case of an address object, << represents dispatching a message to that address within the context of **destination << message object**, which is illustrated in the previous Coding Segments 4.6.5 and 4.6.6. Other methods exist built into the actor, such as dispatching messages to an internal outbox, which is accessible to external threads via the **ask** method.

Actors can be created as threads or processes in the *pystation* actor framework. Pragmatically, the primary difference between threads and processes is the presence of a memory boundary in the latter. In the case of message passing, this introduces issues with the way delivery to mailboxes are handled. With remote message handling, messages are serialized and delivered by a node's controller, which is responsible for placing the data within the queue of an actor thread or process.

The primary medium for Interprocess Communication (IPC) for *pystation* is derived from the **multiprocessing** libraries built into *Python* 2.7, which uses socket communication and its own serialization behind the scenes. The advantage of processes is the potential for true parallelism, which is otherwise not possible with *Python* threads, thereby circumventing the Global Interpreter Lock (GIL) in *Python*. This places more expense on transferring messages from or to process actors due to the induced overhead with the internal IPC mechanisms. In addition, certain features are not available in node-to-node communication when actors derive from their parent process class, such as reporting status or availability. This is due to the inherent memory boundary between the main controller process and actor process. For most actors that function within the framework, they should inherit from the parent thread actor class so that messages can more easily be transferred without the induced overhead, and memory can be shared with the main controller process. Figure 4.2 provides a visual description of how communication takes place between nodes in *pystation* and on a network, as well as the interaction between interfaces, daemons, and actors. Interfaces and daemons will be discussed in the following sections.

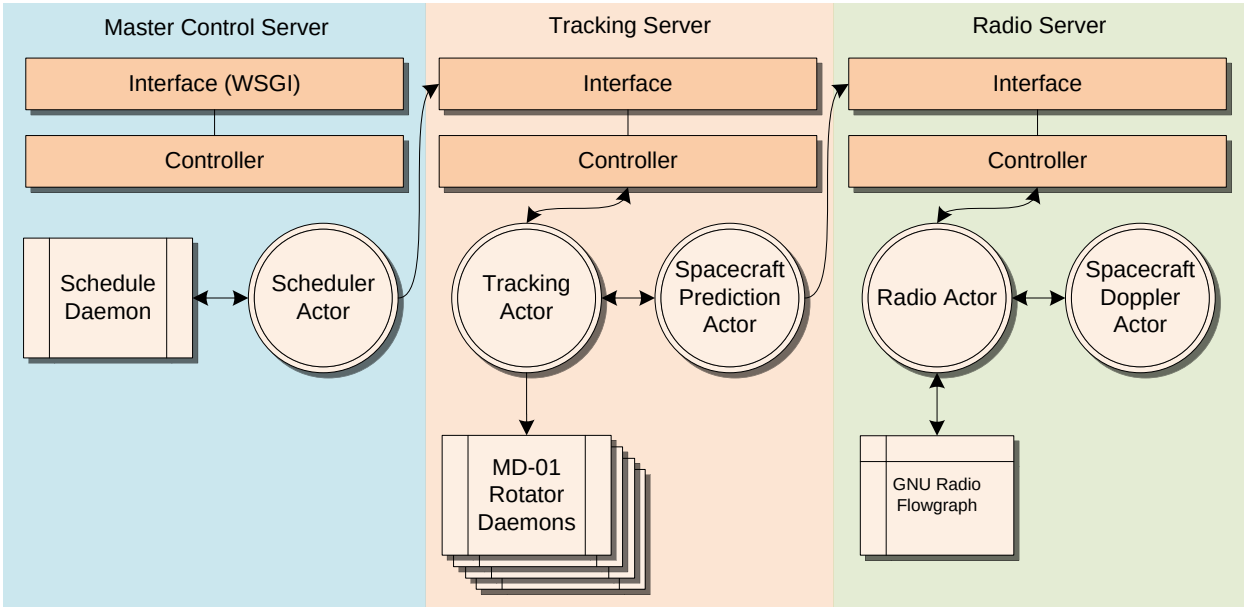

Figure 4.2: Illustration describing the interaction between interfaces, daemons, and node controllers on each server machine.

4.3.1 Interfaces

Interfaces are plugins for communication between nodes and run as threads under the controller process on each machine. This is an important feature for extending *pystation* and allow different forms of communication between nodes on different server machines. Without this feature, it would not be possible to add additional features to the controller process. In Figure 4.2, servers or nodes interact through RESTful principles, utilizing the same architecture as the web.

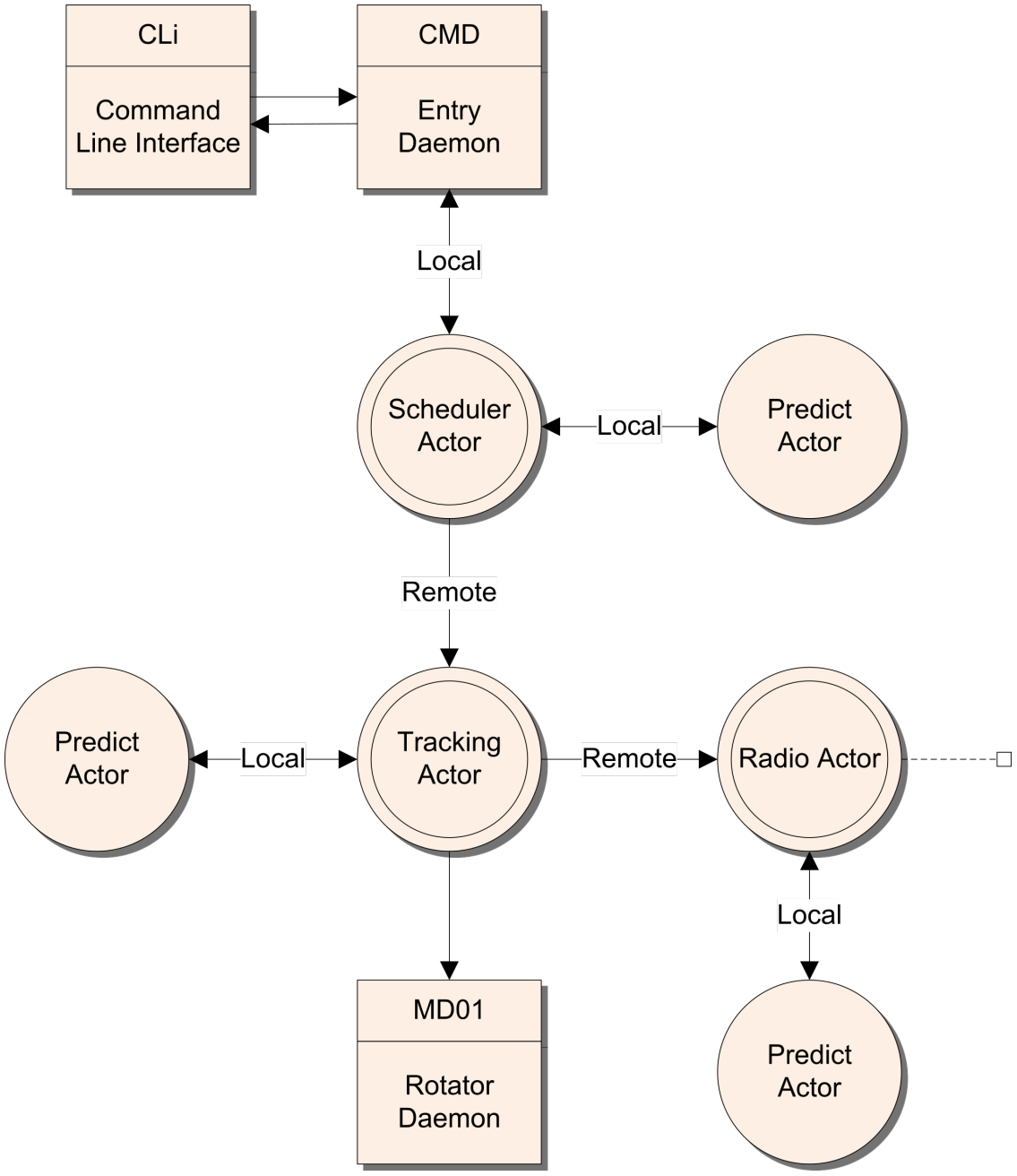

Figure 4.3: A conceptual example of how a command-line interface connects to actors during ground station operation.

This architecture encourages scalability, separation of concerns, and portability among dif-

HTTP Command	GET	POST	PUT	DELETE
Resulting Action	Get information from the node in question, such as a list of actors, interfaces, or daemons.	Post a message to a specified actor address.	Create an actor at the specified node.	Request an actor be shut down at the specified node.

Table 4.1: RESTful commands and their corresponding action at each *pystation* node's WSGI interface. Any process on a node can send these commands to any other node as long as they are on the local network.

ferent servers on the ground station's internal network. Thus, *pystation* aims to create a Representational State Transfer (REST) based controller in order to allow actors on different nodes to easily access different resources from other nodes, such as actors or information concerning the status of other services. The HTTP protocol is commonly used by software falling under this architecture, and so RESTful applications often involve resource identifiers similar to that seen when surfing the internet, like **/command/read/entry_id**. Table 4.1 gives an overview on how these commands currently correspond to the actor framework's controller process. These are just some of the available HTTP commands used to control actors and query information at a controller on a per machine basis.

The servers exist on a private network where the WSGI interface ports are not exposed. Also, all of these commands would constitute soft targets, because only code that exists in the framework can be executed. Furthermore, adding an authentication layer to this actor framework is outside the scope of this work.

There are more interfaces for facilitating node beacons and discovery, which enable machines on a LAN to discover each other on the local network. This sort of network discovery

eliminates the need to hard code or map out the network a priori. Instead, each node can send a User Datagram Protocol (UDP) beacon out and receive them in order to maintain an active list of available services and systems. Any one of these interfaces can be disabled, including the primary services for communication and remote actor creation. For instance, there might be a need to run *pystation* locally without the need for WSGI, beacons, or network discovery, in which case both could be disabled via the main configuration file.

4.3.2 Daemon Processes

Daemons aid in maintaining a clear distinction between pieces of the framework that fit within the actor model and those that do not. Specialized daemon processes handle the interaction between the framework and ground station hardware components that must remain constantly active. Anything that fails to fit within the actor model appropriately might also fall under the category of a daemon. As shown in Figure 4.3, an entry point daemon accepts messages from a Command-line Interface (CLI) and manages the primary scheduler actor. Figure 4.3 also shows the interaction with actors during operation of the ground station. The entry point daemon continues to accept messages until the main *pystation* process is terminated. If actors were used exclusively, there might be issues interacting with external processes that utilize different communication protocols. Daemons aid in maintaining a clear distinction between pieces of the framework that fit within the actor model and those that do not.

4.4 GNU Radio as the Physical Layer

Remote actors are created at server nodes connected to the USRP switch, as shown previously on the network overview Figure 1.3. Due to the network topology, not every server is capable of running GNU Radio since not all nodes have network connections to the USRPs. Thus, the node running the entry point daemon is responsible for loading a network configuration file, which specifies the capabilities of each node. With network beacons and discovery interfaces enabled on each node, the configuration file notes the host names rather than network addresses, allowing the primary node to spin up the associated chain of actors for a singular pass or multiple radios.

Radio actors are responsible for creating the GNU Radio flow graph objects which are accessible to *Python* via SWIG. After standing up the necessary blocks within a flow graph, the parent class allows the actor to call its member methods **start** and **stop**. Predefined radio actors for each spacecraft handle different communication tasks, such as recording telemetry data during a pass or an uplink and downlink transaction. Figure 4.4 presents a simplified view of the interaction between actors and GNU Radio flow graphs.

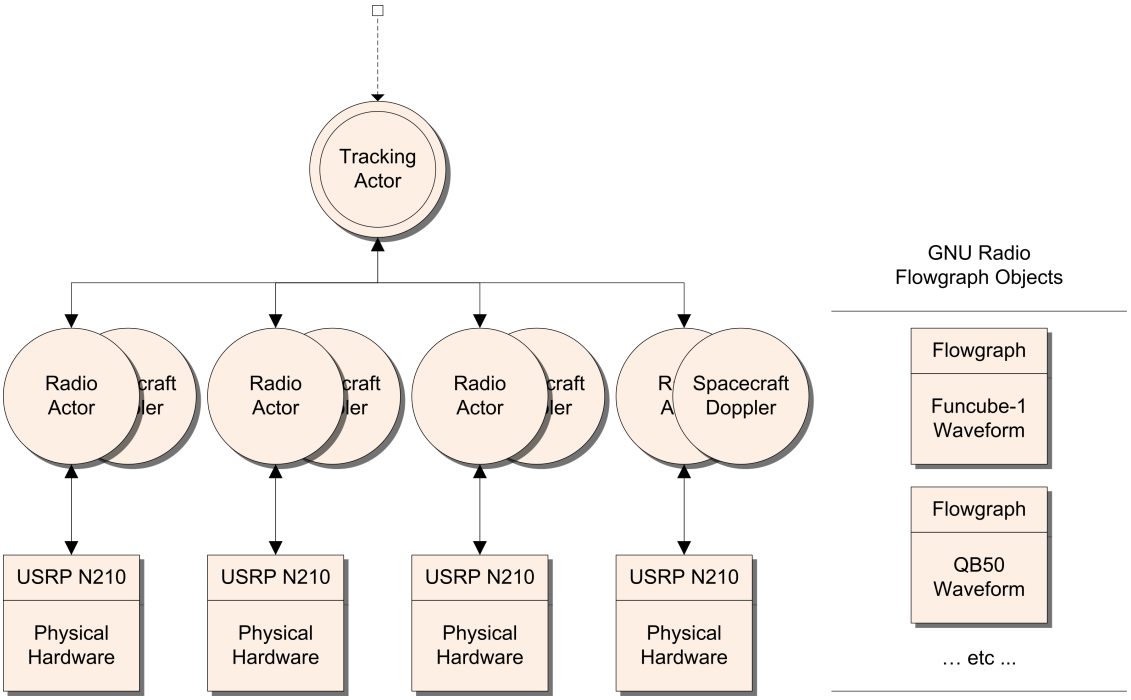

Figure 4.4: Overview of the interaction between *pystation* and GNU Radio. Specialized actors exist that inherit objects from a library of flow graphs.

4.5 Benchmarks for *pystation*

Several tests were performed locally to verify the message passing capabilities of threaded and process actors within the *pystation* framework. Table 4.2 outlines the machine used to run these benchmarks and the results can be found in Figure 4.5.

Benchmark Machine Specs	
Operating System	Ubuntu 14.04
CPU	Intel™2 Duo E8600
CPU CLOCK	3.33 GHz
RAM	8 GB

Table 4.2: Machine specifications for *pystation* benchmarks. These benchmarks were mostly used to validate the number of messages that the *pystation* process could handle internally.

The flood test simulates 50,000 messages being sent from a request actor to a reply actor all at once. The latency is measured as the time it takes for reply actor to respond to all the messages. The count test measures the time it takes for 1,000 actors to initialize, increment a counter, and deconstruct so that it equals the number of actors created. One thing to note about this performance test is the large difference between the thread and process latency. This is due to the initialization of expensive multiprocessing queues, which are standing up servers to allow for IPC. Threads utilize normal queues and share memory, so initialization time is not nearly as expensive. The ping-pong test is a simple request-reply pattern, where a reply is expected for every message sent. In this test, the marginal performance difference is due to a process actor's ability to take advantage of multiple cores.

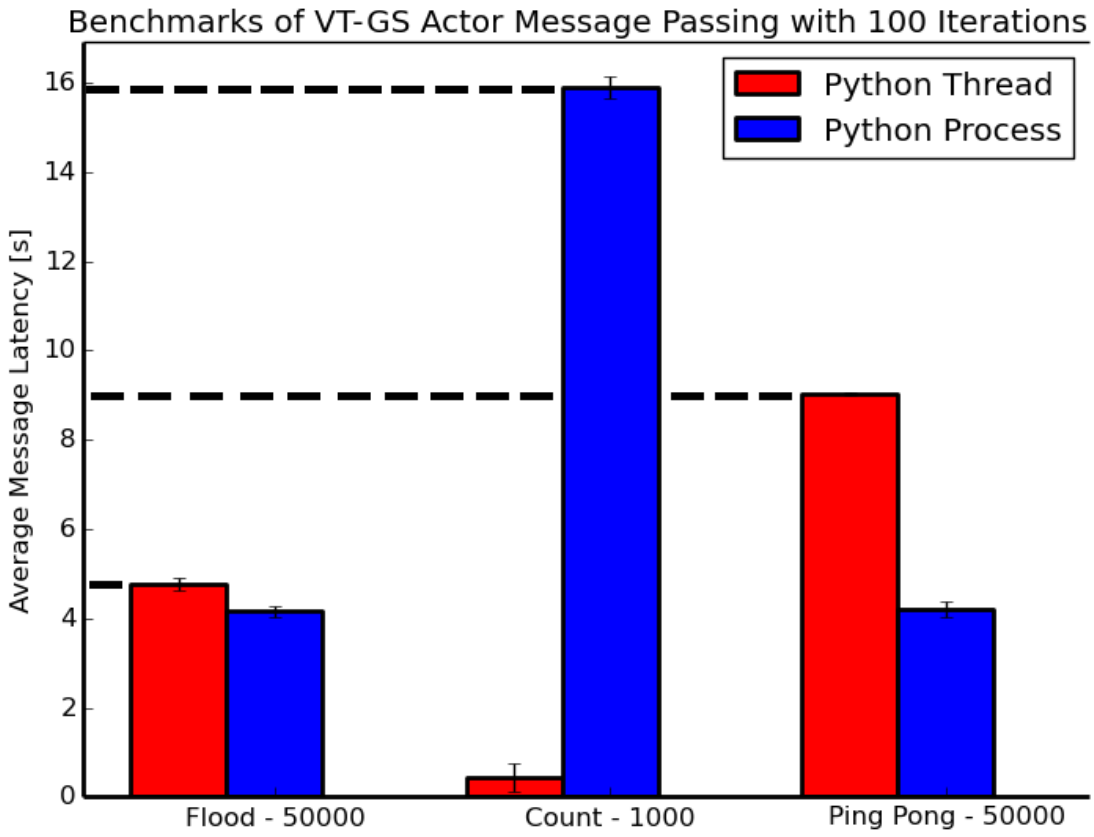

Figure 4.5: Results from averaged latency tests within *pystation*. The flood test measures the latency in a request-reply pattern when all the messages are sent at once. The count test measures the time it takes to initialize actors en masse. The ping-pong test measures basic latency between two actors in a request-reply pattern.

In order to properly measure how well the performance of message passing scales with the number of messages, multiple trials were performed over a flood test to ensure that latency grows linearly with the number of messages in processing as shown in Figure 4.6. Since a process has the potential for parallelism and multiple cores, the average latency trend has a smaller slope. In Figure 4.6, the latency to respond to a flood of messages is also

measured ensure that the framework satisfies the average maximum allowable latency. The allowable latency is based on the assumption that the bottleneck for message passing will be between the actor handling Two-Line Element set (TLE) updates and the tracking actor responsible for forwarding those messages to the MD-01 controllers in the case of the VTGS. This is because the antenna needs to be properly pointed at the spacecraft in order for it to communicate, making this the most time critical piece of a ground station. The MD-01 controllers are responsible for commanding the rotators and pointing the antenna. They are queried four times a second, yielding the requirement that message transmissions can take no longer than four messages per second between actors on the corresponding dedicated machine if the MD-01 controller is to adjust the pointing angles of the attached antenna.

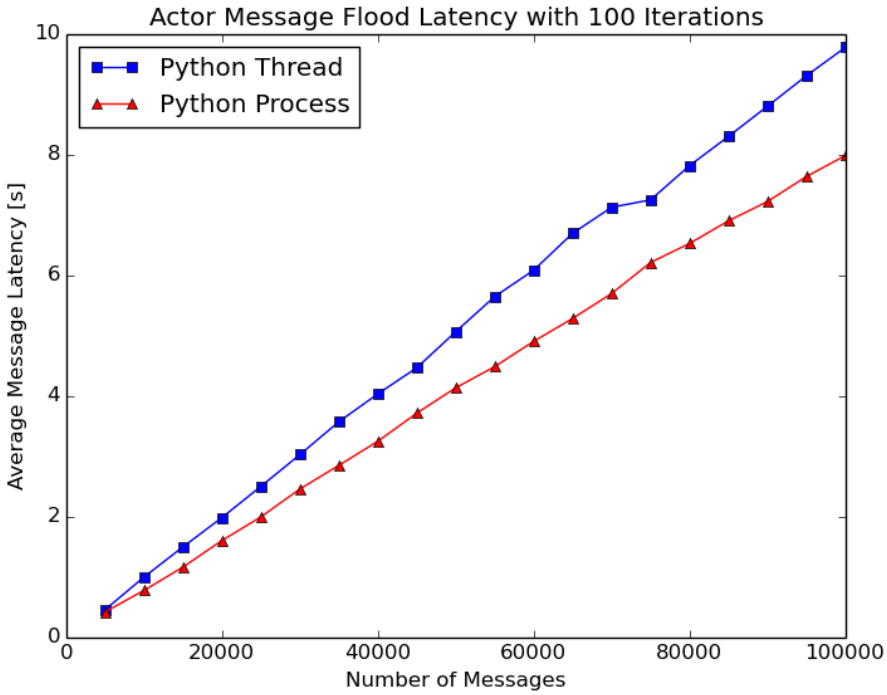

Figure 4.6: Additional results from the flood test show a linear trend in latency and ensure the maximum allowable latency is not exceeded.

4.6 Illustrative Examples of *pystation*

The following sections display several examples of the *pystation* actor framework. The first set of examples deals with applying the actor framework to a ground station through two segments of code. These examples include a scheduling actor that maintains an active plan for tracking spacecraft and a radio actor that controls the GNU Radio flow graph. The last few examples show the actors used in the ping-pong tests as well as an actor that illustrates how to grab data within the parent thread or process.

4.6.1 Ground Station Actors in *pystation*

In order to demonstrate how the actor framework developed in this work is used to track a spacecraft, two actors will be described for an example scenario where data is provided to a radio actor to transmit during a pass. Figure 4.3 provides a depiction of the full actor network during a pass at a ground station running *pystation*. These example code segments are meant to illustrate how the *pystation* framework might be utilized at a ground station. A scheduling actor is shown in Coding Segment 4.6.2. This actor creates an object that maintains state information for the spacecraft tracking schedule. The spacecraft scheduling Application Programming Interface (API) is based on *PyOrbital*, which is an Open Source project modified as part of this thesis work to provide Doppler offset correction information based on the TLE sets. It also provides the necessary functionality for retrieving and calculating orbital parameters based on the TLE sets. The scheduler actor also handles the logic

for creating the tracking actor for talking to the MD-01 controllers as well as creating the radio actors, which is not shown in this snippet for the sake of brevity.

4.6.2 Scheduling Actor Example

```
class ScheduleActor(ThreadActor):
    def __init__(self):
        ThreadActor.__init__(self)
        # Provide the planner
        # object a tuple of our
        # geographic location
        # (longitude, latitude, altitude)
        self.planner = Plan(config.LOCATION)

    def on_start(self):
        self.create_actors()

    def action(self):
        while self.is_alive:
            req = Match(self.recv(0.1))
            if req.type is not None:
                body = req.contents
                args = [body["NORAD_ID"], body["PASSES"]]
                # Map the strings within the message to integer type
                norad_id, passes = map(int, args)
                # Add the spacecraft prediction to the schedule
                self.planner.add_passes(norad_id, passes)

            self.handle_behavior()
```

In this case, the scheduling actor would create a child radio actor to handle the flow graph, as shown in Coding Segment 4.6.3, which would queue the data through a message source

with an accessor function. This data might originate from some remote actor on the LAN that has this actor's address, which might be receiving the data from another application. It is also possible to utilize sockets for entry and exit points in a GNU Radio flow graph.

4.6.3 GNU Radio Actor for an AFSK Transmitter

```
class RadioActor(ThreadActor):
    def __init__(self):
        ThreadActor.__init__(self)
        # Create a GNU Radio top block
        self.tb = gr.afsk_tx()

        # Create connections (excluded details for brevity)
        self.setup_connections()
    def action(self):
        while self.is_alive:
            # Start the flow graph (non-blocking)
            self.tb.start()
            req = Match(self.recv(0.1))
            if req.type == "frame":
                # Extract the binary data from the message
                frame_data = req.contents["AX25_PAYLOAD"]

                # Queue the frame data in the flow graph
                # source block for transmit
                self.tb.message_source.queue_data(frame_data)
        # Before exiting cleanly, close the flow graph
        self.tb.stop()
```

4.6.4 General Examples of *pystation* Actors

The ping and pong actor setup shown in Code Segments 4.6.5 and 4.6.6 illustrate one of the test cases used for the benchmarks in Section 4.5. The ping actor simply floods the pong actor with 1000 messages and counts each reply before closing the pong actor and returning from its action method.

4.6.5 Ping Actor Example

```
class Ping(ThreadActor):
    def action(self):
        # Create the child actor with the parent address.
        rep = RepActor.create_other(self.address)

        # Create 100 requests, including the return address.
        count, number = 0, 100
        msg = Request([self.unique_id, "hai"])

        # Dispatch the messages using the '<<' operator.
        for i in range(number):
            rep << msg

        # Wait for the same number of responses.
        while self.is_alive:
            self.recv()
            count = count + 1
            if count >= number:
                # Stop ourselves once we reach the number.
                rep.stop()
                break
```

4.6.6 Pong Actor Example

```
class Pong(ThreadActor):
    def action(self):
        # While we're alive...
        while self.is_alive:
            # Match the incoming messages to a specific type
            req = Match(self.recv())
            msg = req.message

            # If the message type is a request
            if req.type == "request":

                # Extract the address
                return_addr = self.get_addr(msg)
                # Respond with a message
                return_addr << Message("pong")
```

The pong actor simply loops until its stop method is called and replies to any messages sent to it. Since these actors inherit from the **ProcessActor** class, which make them processes and not threads, additional resources are used to stand up process-safe queues as the internal mailboxes of each actor. Hence, any data sent between these actors must be "picklable" so that *Python* can serialize the data to be transferred. If these actors instead inherited from **ThreadActor**, normal thread-safe queues would allow multiple types to be transferred between actors. In general, all data is serialized through standard JavaScript Object Notation (JSON) to ensure there is no confusion on what is allowed to be transported from actor to actor.

If data is needed outside a network of actors, it is possible to request data using the **ask** method. The Coding Segment 4.6.7 provides a simple example of extracting data externally from an actor. As shown, the main purpose behind this is to provide a mechanism for getting data from an actor within the body of another process, like the **main** of a *Python* script.

4.6.7 Asking an Actor for Data

```
class Ask(ProcessActor):
    def action(self):
        # While we're looping ...
        while self.is_alive:
            # Match the message.
            req = Match(self.recv())

            # If it's a question, respond with an answer
            if req.type == "question":
                self < Message("Answer")

if __name__ == "__main__":
    question = "Question"
    print "Asking a question... ", question

    # Create the actor
    actor = AskActor.create()

    # Call the ask method and print the response
    print "Reply:", actor.ask(question)

    # Stop the actor
    actor.stop()
```

Chapter 5

A Protocol for a P2P Network of Distributed Ground Stations

In this chapter, a novel P2P protocol is proposed to create a network of ground stations resilient to failure, and capable of communicating with global coverage and scalability based on the BitTorrent DHT protocol [44, 45, 46]. Ultimately, the VTGS is meant to be the start of a larger ground station network. As seen in Chapter 2, networks of ground stations utilize a centralized server architecture. Although simple to implement and manageable with a small number of ground stations and a reliable centralized host, such a system does not scale well, and would be prone to global failures if the central server were to go down. The longevity of a service is sometimes dependent on the maintainers. Thus, with a network of ground stations A central server would also be a prime target for a concentrated Distributed Denial-of-service (DDoS) attack.

The following sections will describe the P2P protocol in detail, followed by the necessary commands that a *pystation* node needs to participate. The chapter will conclude with a description of the protocol.

5.1 Geohashes and the XOR Distance Metric

Some preliminary concepts must be explained before the protocol can be described. In this section, the concept of hashing geographic latitude and longitude coordinates will be introduced, followed by a XOR distance metric for use in that hash space. The hash value or geohash is created from the latitude and longitude of a ground station peer using a geocode system invented by Gustavo Niemeyer [47]. The algorithm for producing a geohash proceeds as follows: Take the latitude value and find whether it falls between -90 and 0 (1) or 0 and +90 (0). For the longitude value, split between the intervals -180 and 0 (1) or 0 and 180 (0). Continue splitting the latitude and longitude coordinate system into smaller and smaller sub-quadrants until the desired accuracy is achieved, adding 2 bits for every set of sub-quadrants. The bits of the latitude and longitude hash values are then interleaved, with latitude as the odd bits and longitude as the even bits.

Each ground station peer has a geohash corresponding to its latitude and longitude coordinates, which are static during the duration of its existence. The geohash of the spacecraft is not static, and changes as its latitude and longitude coordinates change. It is assumed that every ground station has the ability to fetch updated TLEs at any time instant from

CelesTrak or another source [48] so that each ground station peer is aware of the geohash of a spacecraft. The spacecraft may be rapidly moving in its latitude and longitude, also known as the sub-satellite point, which would induce a small change in its geohash. There are a few special cases, such as moving across the equator, prime meridian, and the poles, which would cause a drastic change in the geohash prefix.

The XOR operation provides a cheap distance metric, as it is easy to compute the XOR value between two binary numbers. It also satisfies the properties of a distance metric in order have some notion of distance, such as coincidence, symmetry, non-negativity, and the triangle inequality:

$$\begin{aligned}
\mathbf{XOR}\left(\mathbf{x}_1, \mathbf{x}_1\right) &= 0 \\
\mathbf{XOR}\left(\mathbf{x}_1, \mathbf{x}_2\right) &= \mathbf{XOR}\left(\mathbf{x}_2, \mathbf{x}_1\right) \\
\mathbf{XOR}\left(\mathbf{x}_1, \mathbf{x}_2\right) > 0 &\iff \mathbf{x}_1 \neq \mathbf{x}_2 \\
\mathbf{XOR}\left(\mathbf{x}_1, \mathbf{x}_2\right) + \mathbf{XOR}\left(\mathbf{x}_2, \mathbf{x}_3\right) &\geq \mathbf{XOR}\left(\mathbf{x}_1, \mathbf{x}_3\right)
\end{aligned}$$

It is important to note that this distance does not correspond to geospatial distance, as the XOR metric is non-Euclidean. However, since two physically close objects tend to have geohashes in the same grid space, their geohash prefixes will be close unless they are edge cases.

5.2 Protocol Description

Inspired by advances in P2P technology, such as BitTorrent and its DHT protocol [44, 45, 46], the distributed ground stations protocol is based on a DHT where peer identification depends on a geohash value, which is generated based on the latitude and longitude of the ground station. This distributed ground station network protocol is an application of the Kademlia protocol with the main difference being that geohashes are utilized as opposed to randomly generated hashes for peer IDs [46]. A peer in terms of this protocol corresponds to an entire ground station participating in this network.

Spacecraft also have a latitude and longitude with respect to their TLEs, and so a geohash value can be computed for every instant in time. Using the XOR operation, a distance metric can be computed between geohashes of the spacecraft and ground stations in the network. Control of a spacecraft in the network is determined through a token passing scheme, in which a token representing the spacecraft is passed across the network until it reaches a ground station whose geohash value is "closest" to that of the spacecraft. The spacecraft token corresponds to a value that encodes important information about the spacecraft, like its NORAD ID, as well as information about the token itself like the time it was created. This token represents uplink command over the spacecraft to ensure that only one ground station has permission to command the spacecraft at any given time. Thus, only a ground station peer holding this token has the ability to communicate with the spacecraft and send uplink commands. However, this does not preclude multiple ground stations from receiving

signals at the same time. This process continues until the token arrives at a ground station in view of the spacecraft. Once a ground station loses contact of a spacecraft, the process continues as described. If the token is lost while in the ownership of a ground station that fails, and the spacecraft arrives in range of another ground station, a new token will be created given enough time has passed and the Time to Live (TTL) of the token has expired. This idea of a XOR-metric was first utilized in a P2P protocol named Kademlia, and has become the basis for the tracker-less version of BitTorrent [46]. The protocol developed here also benefits from the XOR-metric topology, but combines it with the idea of a token that represents control of the spacecraft. Figure 5.1 illustrates the idea of passing a spacecraft token around in a network until it reaches a ground station with a geohash "closest" to that of the spacecraft.

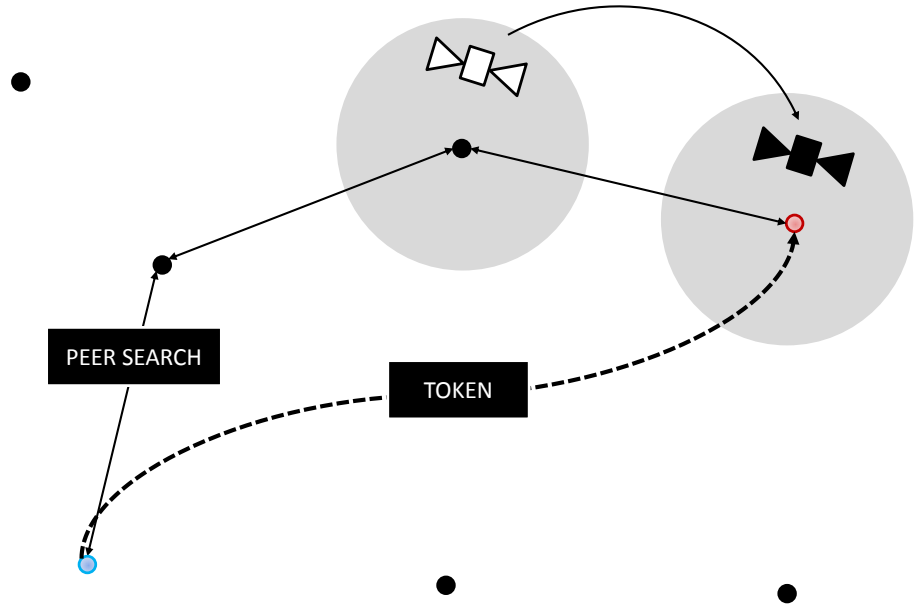

Figure 5.1: Illustration of network discovery and token transfer. The dots represent ground stations while the gray circle represents the range of the ground station. The blue ground station is transferring the spacecraft token to the red ground station after the peer discovery process.

The concept of *k-buckets* were introduced in the Kademlia protocol [46]. A *k-bucket* is merely a *list* which contains geohashes at some distance from the ground station's geohash. Thus, each *k-bucket* can be visualized as a leaf on a binary tree. For every bit in the geohash of the ground station, as in a 64-bit geohash, a peer would contain 64 *lists* or *k-buckets*. It is interesting to note that the number of entries in nearby *k-buckets* are fewer than those further away. Peers have a better awareness of neighboring peers, but maintain contact with those that are far away. If a spacecraft geohash changes drastically between edge case

quadrants, those far away contacts aid in locating distant (but spatially close) peers. Most of the time, however, the prefix of a geohash does not change drastically, and so the XOR metric somewhat corresponds to geospatial proximity.

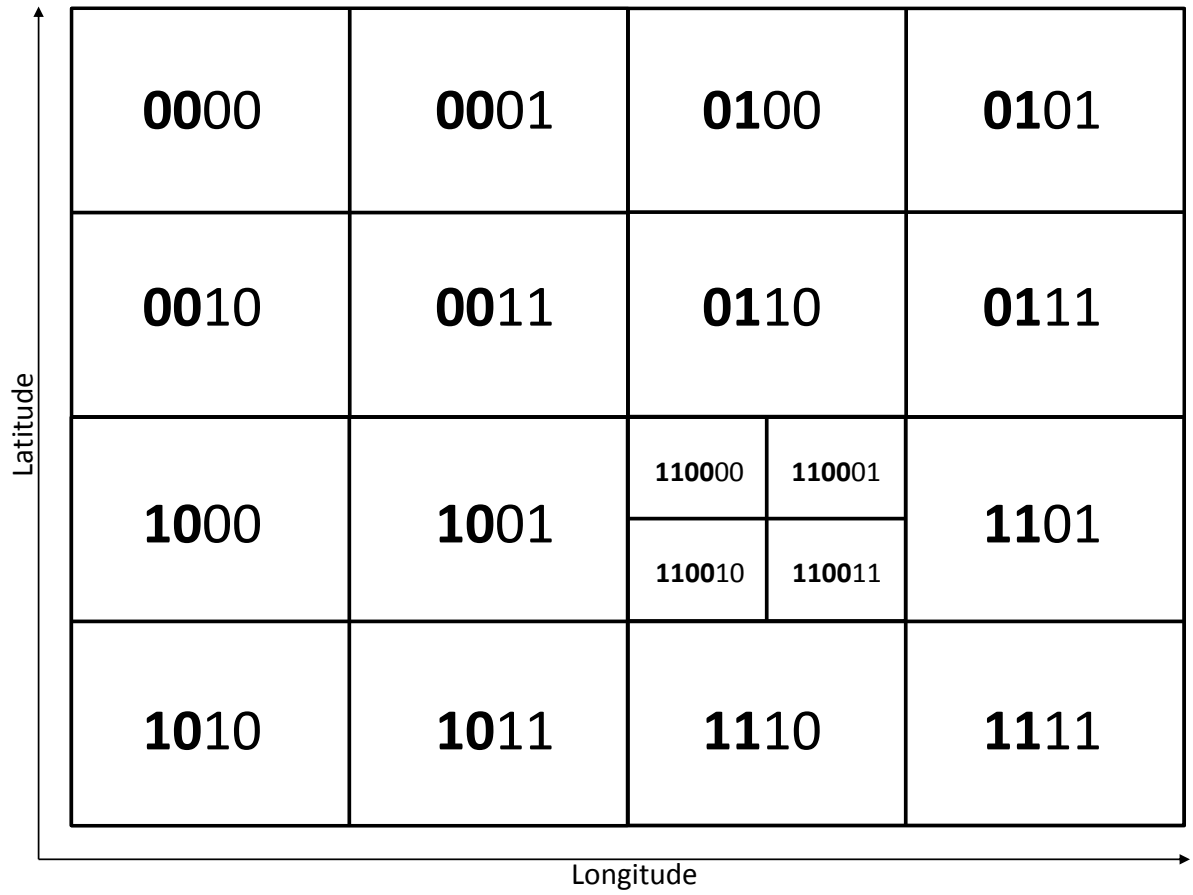

Figure 5.2: Geohashes are created by adding bits and splitting a grid into quadrants. There are edge cases located on the central horizontal line and corners where hash values differ drastically and the bits are flipped, even though the locations are spatially close.

5.3 Peer Commands

In this section, protocol commands are described in the context of the *pystation* framework. The VTGS is capable of receiving commands remotely through a scheduling daemon that is always running on the master controller. The scheduler daemon only runs on the master controller at a ground station, and creates an actor to maintain an active schedule or plan for tracking spacecraft via TLE information from CelesTrak [48]. The scheduler daemon is responsible for external communication in a larger network of ground stations as well as responding to queries and requests for information. The scheduler daemon in *pystation* would need extra commands included in Table 5.1 that it would accept through UDP.

Commands	Arguments	Returns
PING	-	ACK/NACK
UPDATE	NORAD_ID, PASSES	ACK/NACK
TRANSFER	TOKEN VALUE	ACK/NACK
FIND	GEOHASH	GEOHASH, IP, UDP PORT
STORE	KEY, VALUE	ACK/NACK
LIST	-	SCHEDULE
QUERY	NORAD_ID	ACK/NACK
CLEAR	-	ACK/NACK

Table 5.1: Commands accepted by the scheduler daemon. Only one instance of the scheduler daemon is present at each ground station. Commands can accept multiple arguments and return multiple values (tuples).

In this protocol, the PING command is used to check if a peer is alive. The TRANSFER command is used to transmit a token to another peer. The FIND command is used to locate peers with a geohash close to the one being searched. A spacecraft has two hash values

associated with it: its geohash based on its latitude and longitude, which is always changing, and a hash value of equal length generated from its NORAD ID. In order to build an overlay network for locating archived data, such as captures and data from a spacecraft, a STORE command is necessary in order for ground station peers to store key-value pairs. This allows curious peers to locate resources cataloged from the satellite of interest by asking ground station peers with a geohash "closest" to that of a unique spacecraft identifier. The QUERY command is used to ask a peer if they actually have contact with a spacecraft based on its NORAD ID.

5.4 Protocol Mechanics

This section describes different aspects of the protocol, such as the steps taken to join the network, transfer a spacecraft token, and recover scattered data. The main assumption for this protocol is that the ground stations have knowledge of the sub-satellite point of the spacecraft at all times through the use of TLE sets for calculating orbital parameters. Gaps in coverage should not be an issue, as the closest ground station will be chosen regardless of whether it is in range of the spacecraft, ensuring that the spacecraft will eventually be in view of the ground station if is placed correctly for the spacecraft orbit.

5.4.1 Joining the Network

A ground station peer joins the network in the same way a node joins in the Kademlia protocol [46]. The main difference is that a ground station peer generates its ID based on its geohash instead of random assignment. In order for a ground station peer to join the network, it must contact another participating peer. It then performs a search for its own geohash, populating other peer's routing tables as well as its own.

5.4.2 Peer Search

If the spacecraft token is located at a ground station peer that does not have contact with the spacecraft nor has the "closest" geohash, it will conduct a search for another peer with a closer geohash in the same way a node search is performed in the Kademlia protocol [46]. The process begins by the ground station peer asking some number of peers in its *k-buckets* to FIND a peer with the "closer" geohash. If the queried peers return additional peers, the process proceeds in an iterative manner, keeping the best results. Otherwise, the queried peers return nothing, and the best results are stored by the peer initiating the search. These results are stored for the token transfer process, described in the following section.

5.4.3 Token Transfer

The best search results are stored in a table, sorted by the XOR distances in ascending order. The ground station peer wishing to transfer the spacecraft token sends a QUERY command

to the peers within that table. If multiple ground station peers return an acknowledgement, the peer at the top of the table is chosen, and the token is transmitted with the TRANSFER command. Afterwards, the token is then removed from the ground station peer that initiated the transfer in order to avoid duplication of the above process. Figure 5.3 illustrates the process of locating a peer with the minimum XOR distance to the spacecraft and transferring the spacecraft token to it.

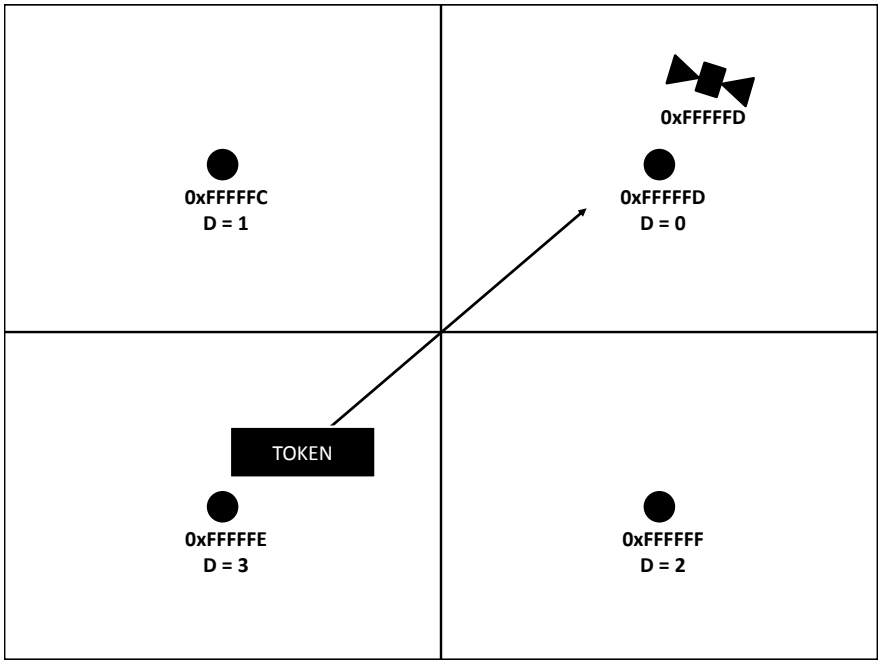

Figure 5.3: Illustration of a ground station peer measuring the XOR distance to the spacecraft from every geohash shown as hexadecimal. The D variable indicates the XOR distance from the ground station peer's geohash to that of the spacecraft.

5.4.4 Resource Location

While the peer search and token transfer handle the uplink case, there might be a need for a peer in the network to recover downlink or telemetry data scattered across multiple ground stations. This protocol handles locating resources in the same way as the Kademlia protocol [46]. However, the geohashes are not uniformly distributed, and this can cause storage hot spots depending on how spacecraft are uniquely identified. There should be a mapping between a non-changing value for the spacecraft, such as its NORAD ID, to a hash value that distributes the assigned values in a uniform way. This is required to avoid artificially centralizing the resource locations at a particular set of ground station peers. For example, consider the case where the unique identifier was simply its NORAD ID, and geohashes were defined to be 64-bit values. Since NORAD ID is a 16-bit value, a 48-bit prefix would consist only of zeros, concentrating the storage peers in the top left corner of the grid in Figure 5.2. A simple implementation would involve creating a table of key-value pairs with NORAD IDs and hash values with the same length as the geohashes. These values would be uniformly distributed over a grid space, depending on the number of entries. Each ground station peer would have to know how to construct this table in order for this portion of the protocol to work.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The work presented in this thesis provides the full software architecture for a ground station, beginning with the baseline DSP functionality in Chapter 3, building up to the control software in Chapter 4, and ending with a proposed P2P protocol to expand it to a network of ground stations in Chapter 5. The software architecture enables the creation of a software-defined ground station, which is extensible to future missions and robust to failure, and novel in comparison to the ground station projects reviewed in Chapter 2. The signal processing blocks created for GNU Radio provide a way to integrate amateur radio digital transmission modes into flow graphs controlled by *pystation* actors. The application of the actor model to concurrent operations at a *pystation* enabled ground station provided a way to organize

all of the different tasks that might be required across a ground station's internal network. Without an actor framework applied to this problem, traditional concurrent programming techniques would have increased the complexity of the underlying system.

The software architecture designed in this work will be applied to the VTGS in order to support its various systems as outlined in Chapter 1. The VTGS will also lend itself to space-based projects, such as CubeSats and SmallSats built by Virginia Tech. These projects can be controlled and maintained by a facility located on the main Virginia Tech campus in Blacksburg. Bringing ground station operations to Virginia Tech would also attract students wishing to conduct research on satellite communications to the main campus. With a ground station so close to campus, further educational activities can be created in the form of labs, where students can attempt to recreate waveforms, such as the ones found in this thesis, and test their work with a functioning ground station. With the proper scheduling and control software in place, students can schedule their particular demonstration on the system and witness their flow graphs actually decoding data from a spacecraft.

6.2 Future Work

As stressed in Chapter 1, this ground station could serve as a sandbox for future undergraduate and graduate researchers. The primary method that this author views as a path forward is utilizing the VTGS and the software developed in this work as a primitive in a larger array of experimental assets. That is, with this ground station, Virginia Tech will be

in a unique position to conduct the other side of space-based research, namely the control and operations. If a particular spacecraft is created at Virginia Tech, the VTGS can be used for operation, yielding direct access for students based in Blacksburg. The VTGS also serves as a suitable model for a larger P2P ground station network, made up of ground station peers running *pystation* and utilizing the propsed protocol found in Chapter 5.

6.2.1 Web-based Interface for *pystation*

One of the main avenues for future work is the development of a web interface in order to provide a visual way to control a *pystation* node. This would be especially important to student labs centered around the VTGS, where a scheduling interface to the main control server would be desirable as opposed to a command-line interface.

6.2.2 Security Layer and Authentication

The argument was previously made in Section 4.2.1 that the ground station internal network is assumed to be private and unexposed to the outside world. Access is provided through a secure connection on a private network. Communication between *pystation* nodes (server machines) is unencrypted and no verification is performed on a node's identity. Thus, a potential project and extension of *pystation* would be to add authentication and security between *pystation* nodes on the LAN. That way, if a malicious user were to gain access to one of the *pystation* nodes, not all of them would be compromised.

6.2.3 Exploring the Proposed P2P Protocol

Adding security features to the P2P protocol described in Chapter 5 would be an avenue for future work, as well as simulating and proving the performance. In the P2P protocol proposed in Chapter 5, adding public-key cryptographic methods to prevent duplicate tokens is an important step for securing it against malicious peers overwhelming the network. Protocols based on a DHT are often vulnerable to the Sybil attack where an attacker spoofs many nodes [49]. However, enhancing the proposed P2P protocol based on trust between ground stations provides a method to avoid that issue.

Bibliography

[1] J. Hartley, E. Luczak, and D. Stump, "Spacecraft control center automation using the Generic Inferential Executor (GENIE)," *European Space Agency, (Special Publication) ESA SP*, no. 394 PART 2, pp. 1007–1014, 1996. Used under fair use, 2015.

[2] J. Anderson, "Autonomous Satellite Operations For CubeSat Satellites," Master's thesis, California Polytechnic State University, 2010. Used under fair use, 2015.

[3] H. Beker and F. Piper, *Cipher Systems: The Protection of Communications*. A New electronics communications international book, Northwood Books, 1982.

[4] "Proposed Coded AO-40 Telemetry Format." `http://www.ka9q.net/papers/ao40tlm.html`. Version 1.2. Published 7 January 2002.

[5] W. A. Beech, D. E. Nielsen, and J. Taylor, "AX. 25 Link Access Protocol for Amateur Packet Radio," *Tucson Amateur Packet Radio Corporation*, pp. 1–133, 1998. Used under fair use, 2015.

[6] "SAMPEX Data Center." http://www.srl.caltech.edu/sampex/DataCenter/index.html. Accessed May 11th, 2015.

[7] M. Fischer, *Multi-mission Satellite Ground Station for Education and Research*. PhD thesis, Technischen Universität Wien, 2012.

[8] MATLAB, *Version R2014a*. Natick, Massachusetts: The MathWorks Inc., 2014.

[9] PA9N, Neil Melville, "Global Educational Network for Satellite Operations," in *21st Annual AIAA/USU Conference on Small Satellites*, 2008.

[10] Z. J. Leffke, "Distributed Ground Station Network for CubeSat Communications," Master's Thesis, Virginia Polytechnic Institute and State University, 2013.

[11] "GNU Radio Website." http://gnuradio.org/redmine/projects/gnuradio/wiki. Accessed May 21st, 2015.

[12] "Multimon-ng Digital Transmission Decoder." https://github.com/EliasOenal/multimon-ng. Released on 11 June 2013.

[13] "Satellite Networked Open Ground Station." https://www.satnogs.org. Accessed May 12th, 2015.

[14] "Electronic Code of Federal Regulations." https://www.fcc.gov/encyclopedia/rules-regulations-title-47. Rules and Regulations for Title 47 - current as of 13 August 2015.

[15] "FCC Online Table of Frequency Allocations." http://transition.fcc.gov/oet/spectrum/table/fcctable.pdf. Revised on 13 August 2015. Retrieved on 19 August 2015.

[16] "Online ITU Frequency Allocation Table." http://life.itu.int/radioclub/rr/rindex.htm. Last updated 6 February 2013. Retrieved 19 August 2015.

[17] "List of Active SmallSats and CubeSats." http://www.ne.jp/asahi/hamradio/je9pel/satslist.htm. Actively updated by a Japanese amateur radio operator JE9PEL.

[18] "Bell 202 Interface Specification." http://www.softelectro.ru/bell202_en.html. Published in 2010. Retrieved on 11 October, 2015.

[19] K. W. Finnegan, "Examining Ambiguities in the Automatic Packet Reporting System," master's thesis, California Polytechnic State University, 2014.

[20] "Byonics TinyTrak4 APRS Encoder." http://www.byonics.com/tinytrak4. Accessed 21 September, 2015.

[21] J. A. Magliacane, "The KD2BD 9600 Baud Modem." http://www.amsat.org/amsat/articles/kd2bd/9k6modem/9k6modem.html, 1998.

[22] R. D. Gitlin, J. Hayes, and S. B. Weinstein, *Data Communications Principles*. Springer Science & Business Media, 2012.

[23] "AMSAT Press Release and CubeSat information on FUNcube-1 (AO-73)." http://www.amsat.org/?page_id=2039. Published on 21 November, 2013.

[24] "FUNcube-1 Telemetry Dashboard." http://funcube.org.uk/working-documents/funcube-telemetry-dashboard. Version 848 and Documentation Release 1.6.

[25] "ISO 4335 HDLC Specifications." http://www.acacia-net.com/wwwcla/protocol/iso_4335.htm. Last revised 27 February, 2004.

[26] P. Haller and M. Odersky, "Actors that Unify Threads and Events," in *Coordination Models and Languages*, pp. 171–190, Springer, 2007.

[27] I. Grief, *Semantics of Communicating Parallel Processes*. PhD thesis, Massachusetts Institute of Technology, 1975.

[28] H. Baker and C. Hewitt, "Laws for Communicating Parallel Processes," *MIT Artificial Intelligence Laboratory*, 1977.

[29] R. R. Atkinson, *Automatic Verification of Serializers*. PhD thesis, MIT, 1980.

[30] C. E. Hewitt and R. R. Atkinson, "Specification and Proof Techniques for Serializers," *Software Engineering, IEEE Transactions on*, no. 1, pp. 10–23, 1979.

[31] C. Hewitt, P. Bishop, and R. Steiger, "A Universal Modular ACTOR Formalism for Artificial Intelligence," in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, (San Francisco, CA, USA), pp. 235–245, Morgan Kaufmann Publishers Inc., 1973.

[32] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press series in artificial intelligence, London, 1986.

[33] R. K. Karmani and G. Agha, "Actors," in *Encyclopedia of Parallel Computing*, pp. 1–11, Springer, 2011.

[34] H. Lieberman, *Thinking About Lots of Things At Once Without Getting Confused: Parallelism in Act-I MIT Al Memo 626.* MIT Artificial Intelligence Laboratory, 1981.

[35] H. Lieberman, *A Preview of Act 1.* MIT Artificial Intelligence Laboratory, 1981.

[36] "Distributed Erlang Manual." `http://www.erlang.org/doc/reference_manual/distributed.html`. Revision 7.0. Revised 12 May 2015.

[37] "Scala Manual and Documentation." `http://www.scala-lang.org/documentation/`. Release 2.11.7 of the Scala Language.

[38] "Pykka Actor Framework Documentation." `https://www.pykka.org/en/latest/api`. Version 1.2.1 released 20 July 2015.

[39] "Akka Concurrency Toolkit Documentation." `http://akka.io/docs`. Release 2.3.12 for Scala 2.10/2.11.

[40] "Theron Actor Framework Performance Benchmarks." `http://www.theron-library.com/index.php?t=page&p=performance`. Performance benchmarks published 3 July 2010.

[41] D. Charousset, R. Hiesgen, and T. C. Schmidt, "CAF - The C++ Actor Framework for Scalable and Resource-efficient Applications," in *Proc. of the 5th ACM SIGPLAN Conf. on Systems, Programming, and Applications (SPLASH '14), Workshop AGERE!*, (New York, NY, USA), ACM, Oct. 2014.

[42] "Python-Actors Repository." `https://bitbucket.org/fzzzy/python-actors`. Accessed March 1st, 2015.

[43] "Getting Started with Distributed Erlang Tutorial." `http://www.erlang.org/documentation/doc-7.0-rc2/doc/getting_started/users_guide.html`. Revision 7.0. Revised 12 May 2015.

[44] "BitTorrent Protocol Specification." `http://www.bittorrent.org/beps/bep_0003.html`. Last revised 11 October, 2013.

[45] "BitTorrent DHT Protocol Specification." `http://bittorrent.org/beps/bep_0005.html`. Last revised 2 April, 2013.

[46] P. Maymounkov and D. Mazieres, "Kademlia: A Peer-to-peer Information System based on the XOR Metric," in *Peer-to-Peer Systems*, pp. 53–65, Springer, 2002.

[47] "GeoHash Web Service." `http://geohash.org`. Initially Published 14 January, 2003.

[48] "CelesTrak Website for Keplerian Elements of Satellites." `https://www.celestrak.com`. Accessed July 18th, 2015.

[49] J. R. Douceur, "The Sybil Attack," in *Peer-to-peer Systems*, pp. 251–260, Springer, 2002.