2014-12

Federated ground station network model and interface specification

Felt, Aaron J.

Monterey, California: Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

FEDERATED GROUND STATION NETWORK MODEL AND INTERFACE SPECIFICATION

by

Aaron J. Felt

December 2014

Thesis Advisor: James H. Newman
Co-Advisor: Mathias N. Kölsch

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704–0188

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** December 2014	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE
FEDERATED GROUND STATION NETWORK MODEL AND INTERFACE SPECIFICATION

5. FUNDING NUMBERS

6. AUTHOR(S) Aaron J. Felt

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES)
N/A

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number ____N/A____.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE
A

13. ABSTRACT (maximum 200 words)

This thesis solves the problem of a lack of a complete, simple ground station network interface standard. A federated satellite ground station network (FGN) model and computer interface are developed that extend the use of ground stations to external users across the Internet. This should allow for reuse of existing ground stations, reducing costs and complexity of space missions. An improved model describing FGNs is proposed that defines a hierarchy of the components of the network, allowing for scalability and unified interfaces, and simplifying the process of using FGN resources. This model, which we call the Improved FGN model, is used to develop security schemes that are simple but effective. Simple but effective security schemes are then developed for this Improved FGN model, along with a standardized software interface. This interface connects external users to the network in order to extend ground station hardware to remote users as well as to simplify scheduling for the resource owners in a network. Different middleware frameworks are compared, and Apache Thrift is selected as the best fit for an FGN. This interface is then described and demonstrated with a reference implementation in Python. Recommendations for future improvements of this interface standard are discussed.

14. SUBJECT TERMS ground station, federated ground station, ground station network, earth station, earth station network, interface, web service, service-oriented architecture, CubeSat, picosatellite, M-PIPE, MC3, TT&C	**15. NUMBER OF PAGES** 183
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UU

NSN 7540–01-280-5500

Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

FEDERATED GROUND STATION NETWORK MODEL AND INTERFACE SPECIFICATION

Aaron J. Felt
Civilian, Department of the Navy
B.S., California State University, Monterey Bay, 2012
B.S., University of California, Santa Cruz, 2008

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN SPACE SYSTEMS ENGINEERING
AND
MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 2014

Author: Aaron J. Felt

Approved by: James H. Newman, Thesis Advisor
Mathias N. Kölsch, Co-Advisor

Rudolf Panholzer
Chair, Space Systems Academic Group

Peter J. Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This thesis solves the problem of a lack of a complete, simple ground station network interface standard. A federated satellite ground station network (FGN) model and computer interface are developed that extend the use of ground stations to external users across the Internet. This should allow for reuse of existing ground stations, reducing costs and complexity of space missions. An improved model describing FGNs is proposed that defines a hierarchy of the components of the network, allowing for scalability and unified interfaces, and simplifying the process of using FGN resources. This model, which we call the Improved FGN model, is used to develop security schemes that are simple but effective. Simple but effective security schemes are then developed for this Improved FGN model, along with a standardized software interface. This interface connects external users to the network in order to extend ground station hardware to remote users as well as to simplify scheduling for the resource owners in a network. Different middleware frameworks are compared, and Apache Thrift is selected as the best fit for an FGN. This interface is then described and demonstrated with a reference implementation in Python. Recommendations for future improvements of this interface standard are discussed.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF CODE LISTINGS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

ACL	access control list
AFSK	audio frequency shift keying
API	application programming interface
ASN.1	Abstract Syntax Notation 1
BPSK	binary phase-shift keying
CA	central authority
CCSDS	Consultative Committee for Space Data Systems
CGA	Neptune Common Ground Architecture®
CIA	confidentiality, integrity, availability
CONOPS	concept of operations
CSR	certificate signing request
DMZ	de-militarized zone
ESA	European Space Agency
FGN	federated ground station network
GENSO	Global Education Network for Satellite Operations
GMSK	Gaussian minimum-shift keying
GSML	ground station markup language
GSN	Ground Station Network (Japan's university network)
HTTP	Hypertext Transfer Protocol
IDL	interface definition language
JSON	JavaScript Object Notation
LEO	low Earth orbit
LNA	low-noise amplifier
MC3	Mobile CubeSat Command and Control
MGSN	Mercury Ground Station Network
NASA	National Aeronautics and Space Administration
NEN	Near Earth Network
NPS	Naval Postgraduate School
NRL	Naval Research Laboratory
OQPSK	offset quadrature phase-shift keying

P2P	peer-to-peer
PKI	public key infrastructure
REST	Representational State Transfer
RF	Radio Frequency
RPC	remote procedure call
RX	receive
SA	sub-authority
SDR	software-defined radio
SLE	Space Link Extension
SOA	service-oriented architecture
SOAP	Simple Object Access Protocol
SSL	Secure Sockets Layer
TNC	terminal node controller
TX	transmit
W3C	World Wide Web Consortium
WSDL	Web Service Definition Language
XML	Extensible Markup Language

ACKNOWLEDGMENTS

I would like to thank Dr. Jim Newman and Dr. Mathias Kölsch for your guidance and experienced advice on this effort. You provided me an opportunity-rich environment to find a project that enhanced our work being performed on the Mobile CubeSat Command and Control (MC3) ground station network. You also provided crucial guidance from years of experience that kept this project moving forward. Thank you, Dr. Kölsch, for providing crucial computer science experience and for your reviews of my writing which were instrumental in making this interface not only complete but powerful.

I want to thank Jim Horning for your initial idea to create an interface for extending the resources of our ground station network to external users, and for crucial advice in designing the model so that it was scalable and manageable, and shaping the engine and methods of the interface. Your mentorship through my entire time at the Naval Postgraduate School has been greatly appreciated through multiple projects and has been one of the most enjoyable parts of working in our lab.

Thank you to Jamie Cutler for providing the initial idea to further develop a model for our ground station networks, leading to the Improved FGN model presented in this paper. This work builds heavily upon your own master's thesis and papers. I would also like to thank Giovanni Minelli for helping to shape the requirements of this work. Thank you also to David Rigmaiden for your mentorship with radio technology and the "black magic" of radios. Thank you to John Gibson, Geoffrey Xie, and Gurminder Singh for your guidance in the world of web services and SOA.

Finally, thank you to my wonderful wife and best friend, Rebecca, for all of your support and understanding. Without you this work would not have been completed. I love you and look forward to the adventure ahead for us upon completion of this chapter of our lives.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. PURPOSE

Ground station equipment often spends a larger portion of time unused than actively tracking and communicating with satellites. A simple yet powerful interface standard is needed that can extend the resources a ground station network can provide to users across the Internet. The purpose of this research was to develop a model as well as an interface standard that allows remote users to utilize ground station network resources. Before this interface standard could be developed, a clear network model had to be described that could clearly distinguish the roles of *players* among ground station networks. This model also needed to include a clear model for what the resources are that a ground station network can provide to a user.

This research produced a new model that we call the Improved Federated Ground Station Network (FGN) model. This model was built based on our experience with the Naval Postgraduate School (NPS) and Naval Research Laboratory's (NRL) Mobile CubeSat Command and Control (MC3) ground station network. Along with the model, this research produced and interface standard that we have called the MC3 Picosatellite Interface Pipe Extension (M-PIPE). This standard is based on a single interface framework that provides all levels of development from full remote procedure call functionality to the transfer of bits across the Internet. The result is an easily implementable standard that can be integrated into new and existing software products with minimal effort. Naval Postgraduate School is beginning to use the model and interface to extend the MC3 ground station network across the Internet with a standard interface, hence the inclusion of MC3 and picosatellite into the standard's name.

This thesis first presents a background of ground station networks in Section I, then proposes the Improved FGN model in Section II. Next, Section III explores schemes for securing FGN networks using the Improved FGN model

selects the best for use in an interface. Section IV analyzes existing interfaces and demonstrates a need for a new standard. This section also analyzes different interface technologies selects one to provide FGN interface service. Section V presents the interface standard called M-PIPE that was developed in response to the need for a standard. Section VI concludes this thesis and presents areas for future work to be performed.

B. BACKGROUND

1. Ground Stations

Even before the first satellite, Sputnik, was launched into orbit, both the USSR and the USA had invested considerable resources into researching how to communicate from Earth with an object in space. Radar and radio communication stations, as well as radio and optical observatories, were commonplace by the 1950s. Early satellite ground stations spawned from the methods discovered during the development of these key technologies (Corliss, 1967). A satellite ground station at a minimum is composed of an antenna, a pre-amplifier, also known as low-noise amplifier (LNA), and a radio with a built-in amplifier, as shown in Figure 1. A common amateur-class satellite ground radio is shown Figure 2. An example pre-amplifier is shown in Figure 3. The famous 150-foot satellite dish at SRI International known as “The Dish” is shown in Figure 4 though many antennas are much smaller.

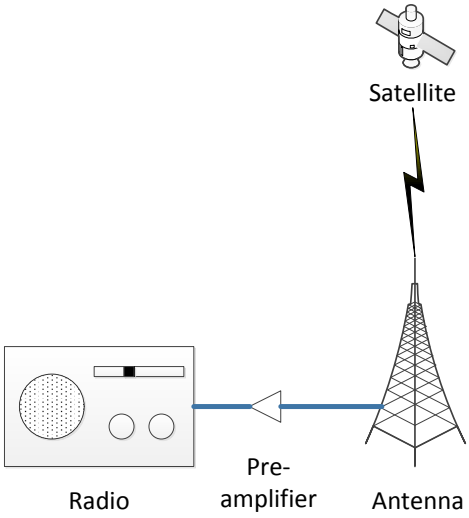


Figure 1. Simplified Satellite Ground Station Block Diagram

Figure 2. Common Amateur-Class Satellite Ground Station Radio. ICOM Ltd. (from http://www.icomuk.co.uk/IC-910H/ Amateur_Radio_Ham_Base_Stations, November 3, 2014)

Figure 3. Pre-amplifier or LNA. Mini-Circuits (from http://www.minicircuits.com/pdfs/ZQL-900MLNW+.pdf, November 3, 2014)

Figure 4. A 150-foot satellite dish antenna in Stanford, California. SRI International (from http://www.sri.com/research-development/specialized-facilities/dish-radio-antenna-facility, November 3, 2014)

Satellites would not be useful if it were not for their ability to communicate from space back to Earth. Basic satellite communications come in the form of *telecommand,* which is the remote control of a satellite, and *telemetry*, which are the data products produced by a satellite. Ground stations send telecommands, and receive telemetry.

2. Ground Station Networks

Before Sputnik was launched, the first satellite ground stations built by the USSR were geographically distributed and grouped into a network called the Command Measurement Complex, abbreviated KIK in Russian (Darrin & O'Leary, 2009). During this same time period, the United States NRL built its own satellite ground station network called Minitrack (Corliss, 1974). The American ground stations were each managed from a central location called the Vanguard Control Center at the NRL in Washington, DC. The Vanguard Control Center had communication links to each ground station with which it coordinated synchronization of clocks as well as communicated satellite tracking data (Corliss, The Evolution of the Satellite Tracking and Data Acquisition Network (STADAN), 1967). The form of the Minitrack network was similar to the network shown in Figure 5.

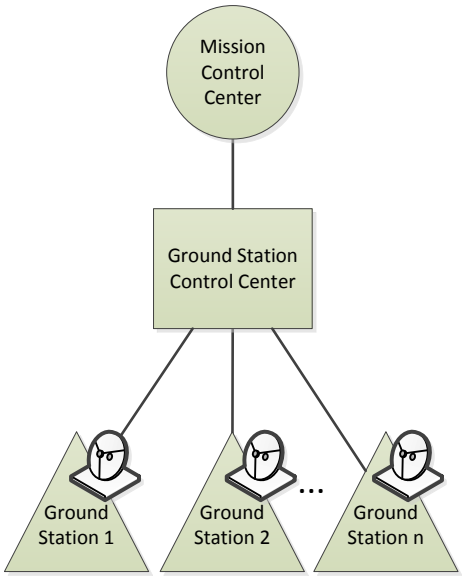


Figure 5. Basic Ground Station Network

The benefits of a satellite ground station network, as compared to a single ground station, are primarily in expanded coverage. The majority of satellites are in low Earth orbit (LEO) (Union of Concerned Scientists, 2014). A LEO satellite is only within line-of-sight of a small portion of the ground at any moment due to the geometry of a satellite in LEO above a planet of Earth's size. The portion of the Earth visible to a satellite in LEO is actually quite small as demonstrated in Figure 6. At this elevation, the satellite will only be in view of a spot on Earth for around 10 minutes per orbit.

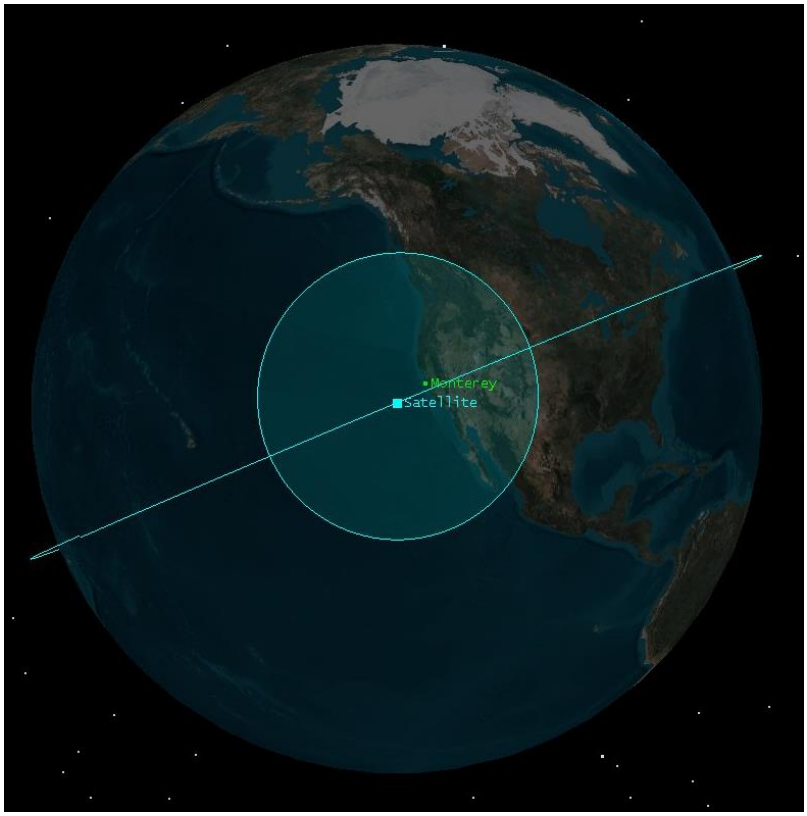


Figure 6. Coverage Area of a 500 km Circular Orbit

A LEO satellite completes its orbit in about 90 minutes (Gordon & Morgan, 1993). Ignoring the consequences of a rotating, non-spherical, inconsistently dense Earth, this means that a satellite passing above a point on the ground will return to the same point above Earth in that time period. Factoring in the reality of a rotating Earth, means that a satellite's orbit will not bring it over the same point on Earth every revolution for most orbits. For LEO satellites, it will instead slowly move westward each orbit until it circles back around after a number of orbits as seen in Figure 7.

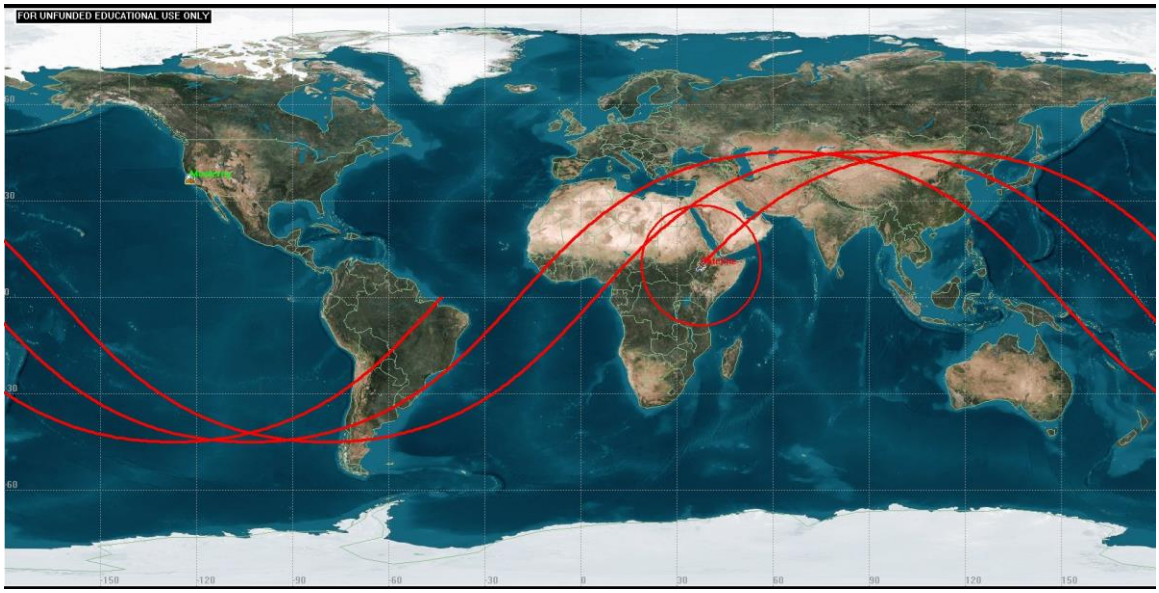


Figure 7. Satellite Ground Tracks Moving Westward

The inclination of an orbit also has an effect on how much coverage a ground station can provide. A low inclination equatorial LEO satellite will never be in sight of a polar ground station. Conversely, a polar satellite will not be able to make as much use of an equatorial ground station as from a polar ground station. These two orbits are compared with two ground stations in Figure 8.

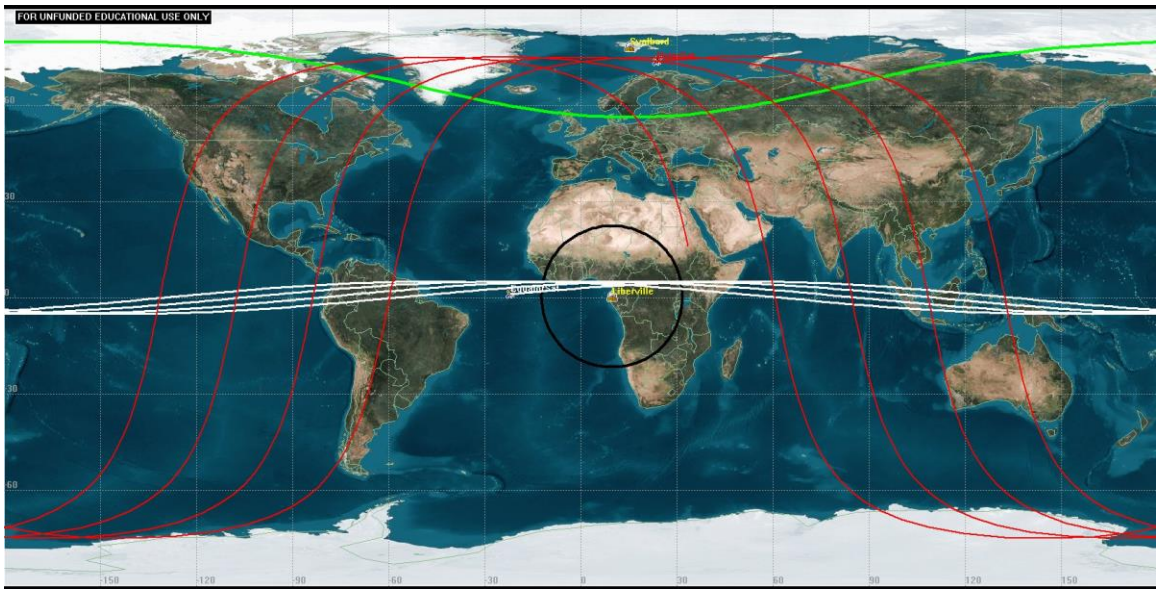


Figure 8. Polar and Equatorial Orbits Versus Ground Station Locations

These factors combined mean that a single ground station will not be able to contact a LEO satellite every time it comes around the Earth for most orbits. In fact, after a satellite's orbit moves west of a ground station, it will often not pass within range again until the far side of its orbit rotates past the ground station, up to a half day later. This illustrates the difficulty of LEO satellite communications and illustrates the benefit of duplicating ground stations around the world, increasing coverage, and therefore, contact time with satellites, as well as the types of orbits that can be reached. Due to this need for ground station networks, two categories have been developed: federated ground station networks and peer-to-peer networks.

3. Federated Ground Station Networks

The first satellite ground stations in both the United States and the USSR were also the first examples of a type of ground station network called a federated satellite ground station network. An FGN is a conglomeration of ground stations, often referred to among the satellite community as ground station *nodes*, in which a central coordinator is designated as shown in Figure 9. Cutler, Linder, and Fox (2002) present the idea of an FGN in their paper on the subject:

> This network infrastructure is a loose federation of ground stations that provides global, cross-mission support. This federated ground station network (FGN) will harness the strengths and diversity in global ground stations that are under different administrative domains to increase network connectivity to satellites and to enhance basic ground station capabilities. (p. 1)

The word federation implies a central authority that unites multiple smaller units. In the case of Minitrack, this central authority was the NRL in Washington, DC, and the smaller units were the ground stations distributed across continents. This central authority has gone by many names, but we can refer to it as the CA. The responsibilities that the CA bears vary from network to network, but a CA always serves to manage the network resources and connect a user of the network to the ground stations they wish to utilize. Some CAs serve additional duties such as coordinating time and providing a central data repository (Darrin &

O'Leary, 2009). Yet, in all instances of an FGN, a CA provides some level of coordination between multiple ground stations and their users.

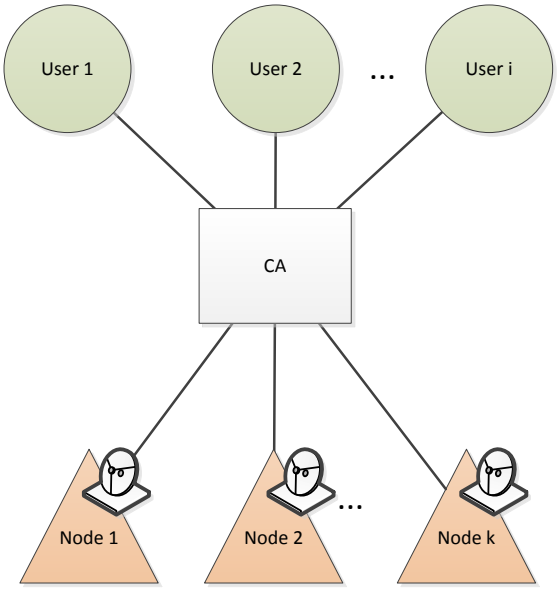


Figure 9. A Simple FGN

Each node may remain semi-autonomous in that it may be able to communicate directly with a user without coordination through a CA. A single owner may own all of the ground station nodes, as is the case with most government satellite ground station networks, or each may have different owners as is sometimes the case amongst academic ground station networks such as the Japanese Ground Station Network (Sakamoto, 2009). Current examples of FGNs are numerous. The Near Earth Network (NEN) run by the National Aeronautics and Space Administration (NASA) (NASA, 2010) is an FGN. Among small satellite communications, the MC3 network run by NPS is intended explicitly for the purpose of CubeSat communications (Minelli, et al., 2012). The Mercury Ground Station Network (MGSN) was set up to handle university satellite missions and supports CubeSat operations (Cutler, 2004).

4. Peer-to-Peer Ground Station Networks

Another type of ground station network is the peer-to-peer (P2P) network as shown in Figure 10. The P2P network mimics networks from Internet file sharing where users distribute files among themselves. P2P as it applies to ground stations means that the ground stations are connected in an ad-hoc, transient fashion from one to another without a hierarchy. The owner of a ground station donates time where other network members may use their resources similar to how a user in a file sharing community uploads pieces of a file to another user upon request. In recent years, the Global Educational Network for Satellite Operations (GENSO) attempted to form a peer-to-peer network with a CA used for authentication (Leveque, Puig-Suari, & Turner, 2007; Shirville & Klofas, 2007). In Japan, the Ground Station Network (GSN) operates almost entirely peer-to-peer, though it also utilizes a central server for server registration (Miyashita, Nakaya, Ui, & Matunaga, 2003). Currently, another P2P network is being developed called the SATNet project. SATNet is being developed at California Polytechnic State University in San Luis Obispo (Tubio, Vazquez, Puig, Kurahara, & Bellardo, 2014) and plans are to set it up as a P2P model.

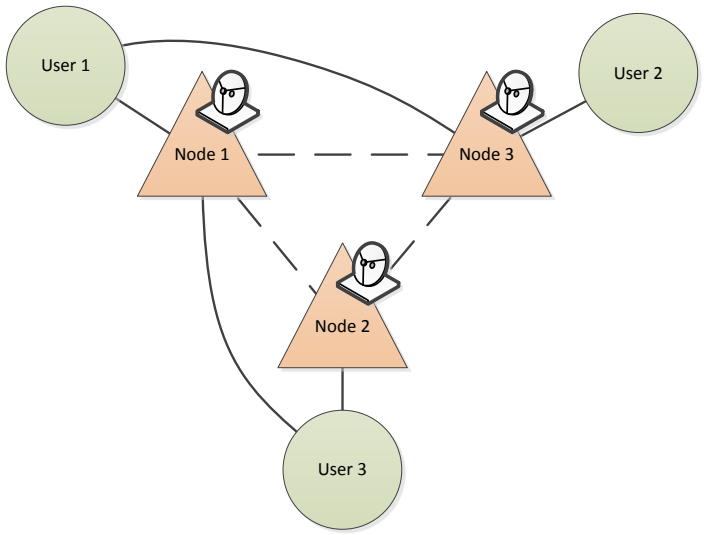


Figure 10. A Simple P2P Network

5. Ground Station Network Models Compared

Both P2P networks and FGNs have their pros and cons. The benefits of a P2P model over an FGN are that the reliance on a central server is removed. Reliance on a single point of failure may be reduced by choosing the P2P model, though in most of the P2P networks described there is still a heavy reliance on a central server. In a basic FGN, if the CA were to be disabled, the network would lose some level of functionality dependent on what responsibilities the CA carries. This drawback to FGNs may be reduced by duplication of CAs as will be discussed later. A P2P network may also be attractive for amateur ground station owners considering offering the use of their ground stations to others as power over how the network is used may be more equally shared.

For many ground station owners, the idea of a CA has its advantages, though. A trusted CA can handle the following duties:

- Trust establishment
- Coordination of resource usage
- Optimizing usage of resources
- Storing and archiving data
- Relaying communications between users and ground stations

Trust establishment can be handled through a central location that all members of the network can rely upon to securely vet new users and provide methods of identification to which all members of the network can verify the validity. A CA can also coordinate the scheduling of ground station usage with users and simplify the process of finding a satisfactory resource for a user. If the schedule is conglomerated from all ground stations at one central site, a user can simply communicate with this single source to discover availability at all member ground stations. During this process, a CA also locally has all of the information necessary to optimize the usage of these resources based upon the needs of the network users and the availability of the ground stations. Optimization cannot happen without coordination and collection of information, as optimal usage

requires a view of the data on which optimization is made. While a scheme could be imagined for peers to optimize themselves, the optimization complexity would likely need to be much greater than for an FGN with a central optimizer.

A single repository can be created at the CA, storing and archiving all data received on the network. The centralization of data archiving allows for easier navigation and analysis of data compared to navigating separate storage points. Lastly, a CA provides a single source to which users can communicate and be redirected to member ground stations. Rather than needing to reach out to each ground station to discover what resources are available, a user can contact the central authority and discover the same information from a single source. Also, a ground station can stay hidden from the user behind a layer of obscurity, as will be shown, which may be a benefit for secure, limited access ground stations.

II. IMPROVED FGN MODEL

To the author's surprise no model could be found in the literature that depicts generalized ground station layout and roles and responsibilities of all of the players within the model. In an effort to capture the complexity of the interactions within an FGN, to provide abstraction that encapsulates separate roles amongst an FGN, and to provide the scalability desirable for future FGNs, an Improved FGN model is described here. This new FGN model allows for increased growth, collaboration, and ease of scalability of the FGN model.

We have improved upon the model defining FGNs in a hierarchical model. This hierarchy allows for separate networks to be combined. Two or more of the simple networks shown in Figure 9 can be combined by adding another player to the model: a new top-level CA. This top-level CA lies one level above the previous CAs, grouping multiple subnetworks into a larger FGN as shown in Figure 11. The previous CAs can now be referred to as sub-authorities (SA) in the model. They serve as a contact point between this new top-level CA and the ground station nodes of their subnetwork. The final role described in this model is the user. Thus, the list of players in the Improved FGN model is composed of the CAs, SAs, nodes, and users.

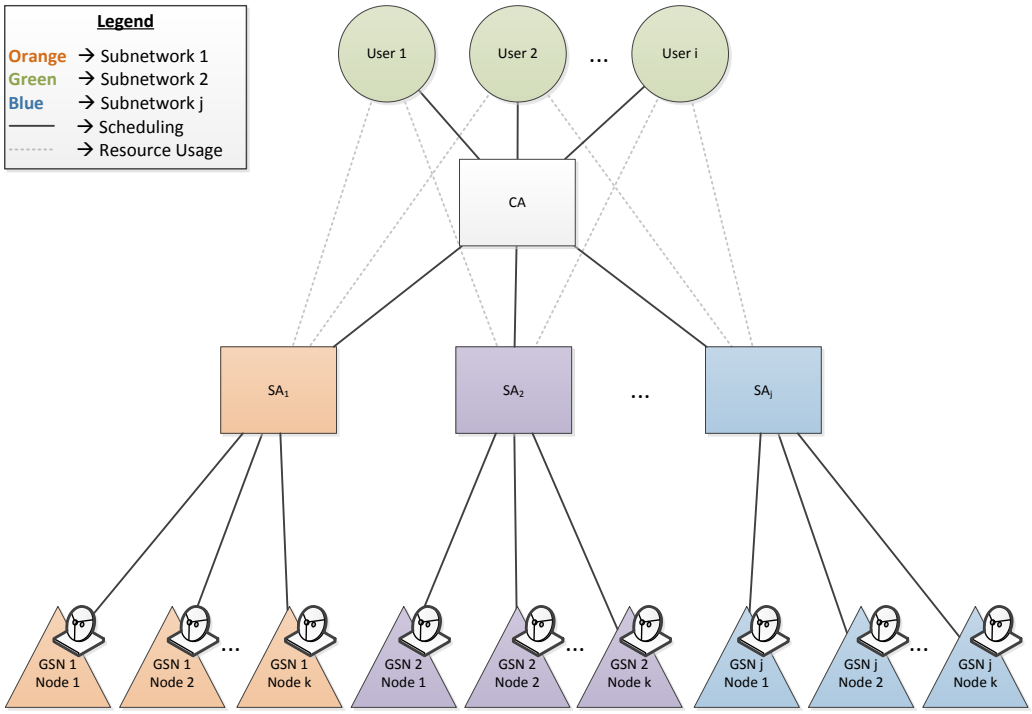


Figure 11. Multiple Subnetwork FGN

A. ROLES OF CAS AND SAS

The role of a CA is distinguished from the other players in the network by its responsibility of providing a schedule interface to users that provides information about the configuration and availability of FGN resources. SAs may also continue to serve as duplicate interfaces to their subnetworks, if they so choose, and will thus still be referred to as a hybrid CA/SA as shown in Figure 12. The role of an SA is distinguished by possessing the responsibility of communicating between its subnetwork and the CA or SA above it, or of having ground station nodes directly under it. It serves to route the communication between the layer above and below itself. At the lowest level of this hierarchy of SAs are the node-level SAs which are responsible for offering up the resources of their ground station nodes. These SAs are directly connected to their ground station nodes as are each of the SAs in this example.

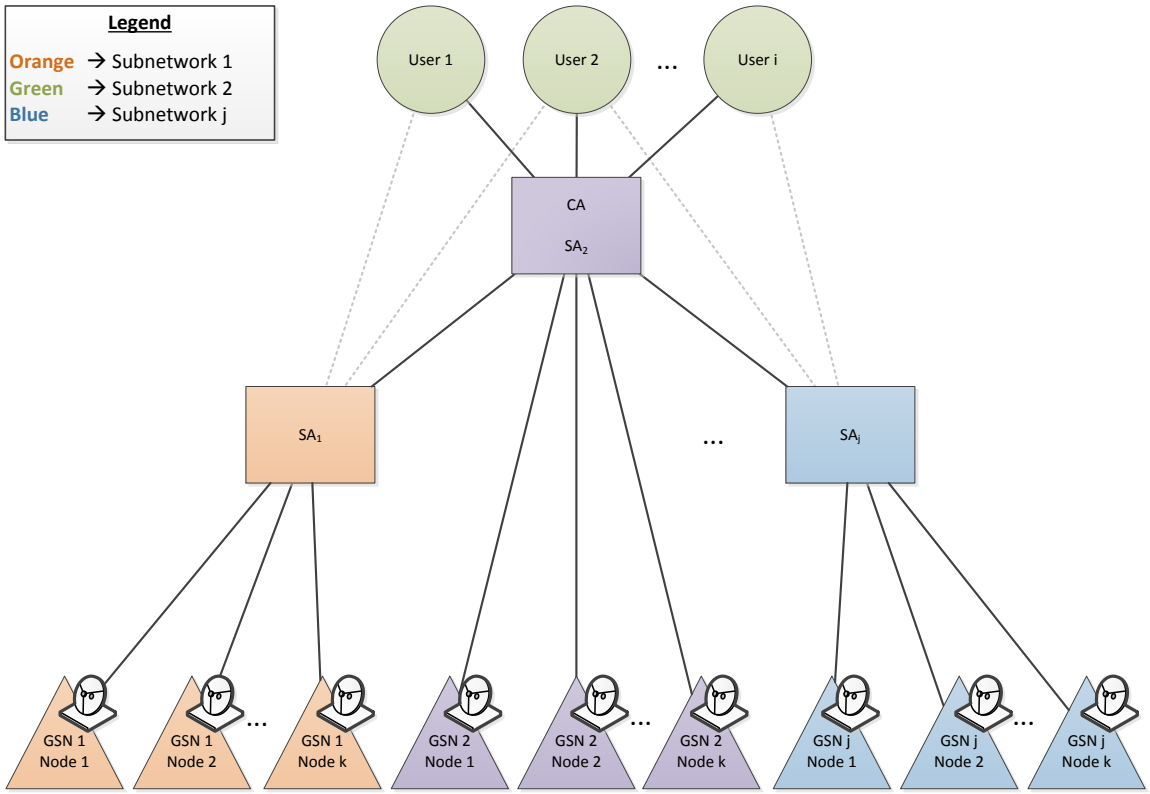


Figure 12. Hybrid CA/SA

B. HIERARCHICAL DEFINITION

A subnetwork can be defined as being composed of an SA, CA, or hybrid CA/SA, as well as any SAs residing at a lower level, and all ground station nodes below these. Subnetworks must include the ground station nodes, and therefore can be grouped from the bottom level up to a variable tree height. An FGN is composed of a federation of these subnetworks. An example grouping of subnetworks into an FGN is shown in Figure 13. This definition is hierarchical and can be repeated for many layers, where, as networks are conglomerated, each previous network can be referred to as a subnetwork in the new, larger federation. The usefulness of such a definition comes from the ability to maintain subnetworks semi-autonomous from the larger federation while becoming part of a larger network. This serves to ease the concerns of combining existing ground station networks into a unified entity.

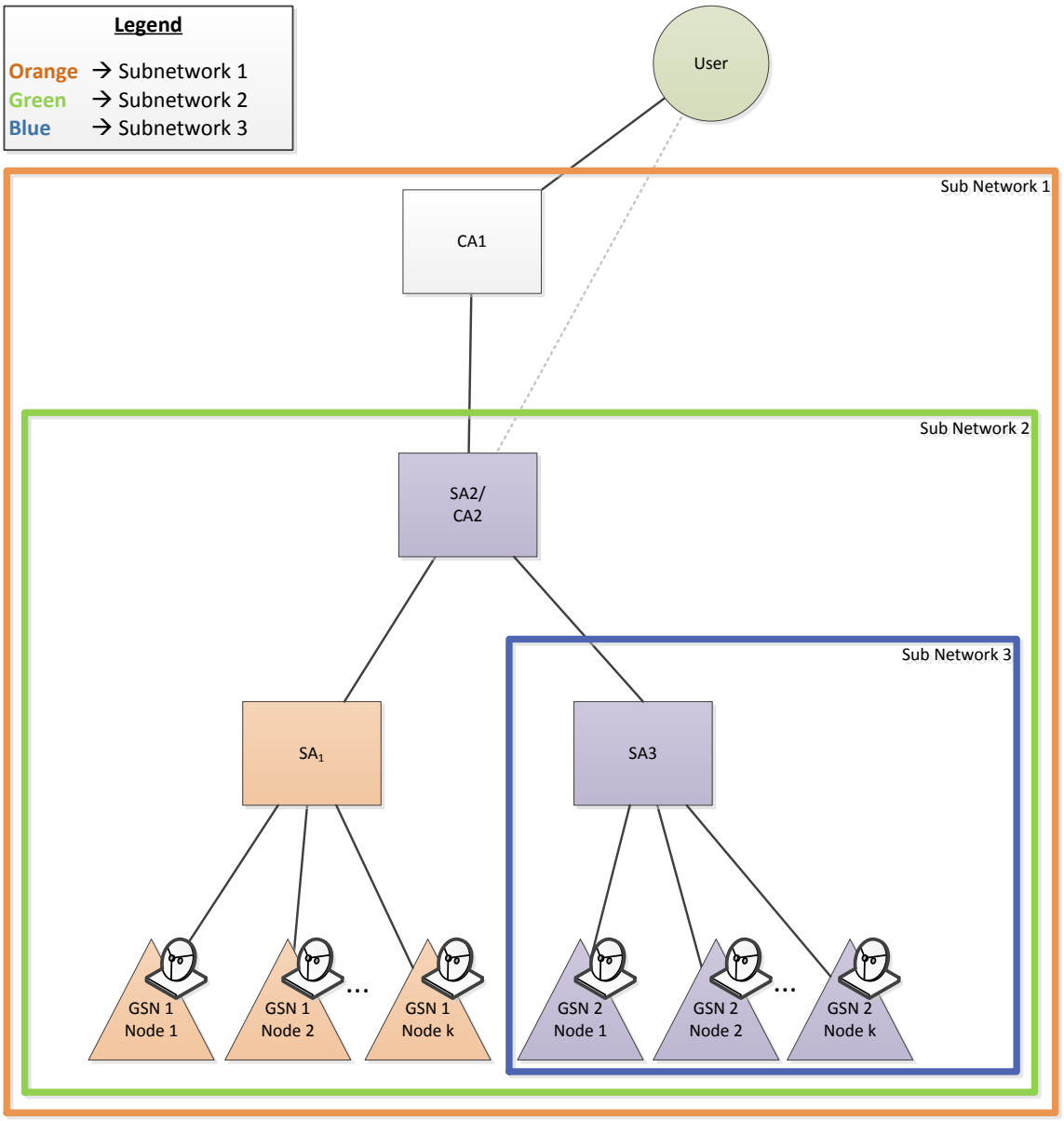


Figure 13. Multiple Levels of Subnetworks

As an example, in Figure 13, one can imagine that Subnetwork 3 started out as a lone network in which there was a single CA and a number of ground stations. Then Subnetwork 3 joined with SA1 and its ground stations, forming Subnetwork 2. To join, they selected a new player to serve as their CA, namely CA2. Once again, one can imagine this federation of two subnetworks joining together with similar networks (not shown) into an even larger federation,

Subnetwork 1. Maintaining CA2 as a CA, as well as serving its SA duties, means that Subnetwork 2 can continue to serve its existing users with the same interface it had used previously. This autonomy eases the process of conglomerating networks as the previous interface remains viable to a user and has no apparent effect. It also allows for a user to be registered only with certain subnetworks if, for example, the larger network cannot authorize the user for policy reasons.

C. GROUND STATION NETWORK RESOURCES

1. Abstraction into Pipelines

Just as the roles among an FGN can benefit from abstraction, so can the resources that an FGN provides.

a. A Traditional Packet Radio Ground Station

The traditional idea behind a ground station network is to grant control of a ground station for a certain time period, or time slot. According to Cutler (2004), due to the increasing utilization of software-defined radios, and reduction of reliance on traditional RF equipment, a simpler way to specify ground station capabilities is to define a *pipeline*. We would restrict this pipeline to be solely composed of the resources necessary for either a receive (RX) or transmit (TX) capability, but not both, as they are separate processes and may be used separately by different users. For example, in a traditional CubeSat packet radio ground station with a single terminal node controller (TNC) plugged into a typical hardware radio with a single tuner, the pipeline is defined by the entire TNC, and the entire radio, both in the uplink and downlink directions. Thus, there are two pipelines in this system: one for uplink and one for downlink. Yet both use the same exact equipment. Because no other communications can occur simultaneously on this hardware, multiplexing of this system is not possible beyond a single transmit and receive capability. A typical simplified block diagram of such a system is shown in Figure 14.

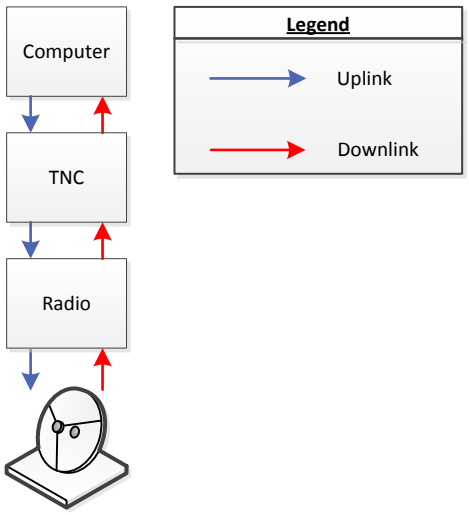


Figure 14. Typical Single Pipeline Ground Station Block Diagram

b. A Modern Software-Defined Radio Ground Station

A newer type of ground station using software-defined radios (SDR) can receive or transmit on multiple frequencies and antennas using one or more RF chains as shown in Figure 15 and thus can be multiplexed both on receive and transmit capabilities. This means that a given hardware chain can handle multiple pipelines simultaneously, and as long as multiple satellites are in view of a single antenna, these pipelines can serve various customers simultaneously. The scenario of having a number of satellites in view of an antenna is common with CubeSat missions, as well as with satellite constellation formations. The number of pipelines and specifics of the flexibility of an SDR are equipment- or implementation- dependent. This multiplexing capability creates the need to abstract the idea of a ground station resource to the provision of a pipeline rather than the traditional view consisting of a single RF chain (Cutler, Ground Station Markup Language, 2004).

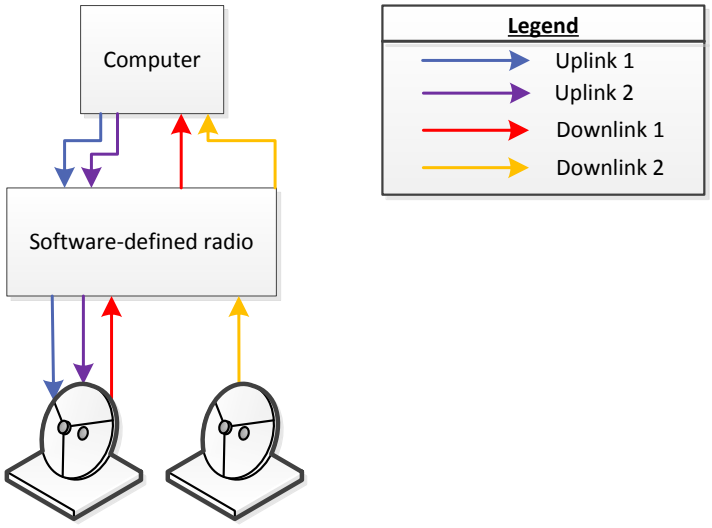


Figure 15. Simplified Multi Pipeline Ground Station Block Diagram

2. Capabilities of a Pipeline-Oriented Ground Station

With the inclusion of this multiplexing capability, what a modern FGN can provide is a pipeline for a certain period of time. Thus, the description of a pipeline is an enumeration of the possible configurations that a ground station can provide. Examples of configuration options that may describe a pipeline are the possible frequencies, modulation and demodulation formats, antenna slew rates, as well as permitted operations such as manual antenna control. In order to fully benefit from the SDRs hardware multiplexing capabilities, the separation of pipelines into TX and RX as described earlier provides the maximum amount of flexibility. Example uses of the model, or *usage scenarios*, enabled by this separation are listed below:

- Allows ground station operators to enable receive but restrict transmit capability
- Allows one user to utilize an RX pipeline while another user operates a TX pipeline
- Allows multiple RX pipelines to operate off a single hardware chain while one or more TX pipelines operate on this same hardware

3. Permissions

While security policies can vary from organization to organization, every ground station network imaginable has limited or protected resources and will want to control who is allowed to use their resources. Authorization, meaning which users are allowed to use which resources, is undoubtedly a requirement for most ground station networks. Which users are authorized will also vary from one subnetwork, or even ground station, to another depending on the parties involved and their needs. For example, “Subnetwork 1” may be willing to grant permission to a user to utilize all of its resources. “Subnetwork 2” may decide that this same user is not allowed to utilize any of their resources or may only utilize certain resources at certain times.

Due to this need for authorization, the final component that must be added to each resource is an access control list (ACL). There may be an ACL for each time slot on each pipeline. This ACL would be composed of a list of users and their permissions. Permissions may be binary, meaning the user is allowed to use the resource or not. Permissions may also be more granular to the level of certain functions of a resource which are allowed and which are barred, such as the ability to have automatic antenna steering, but not to manually command an antenna. As authorization and usage for a given channel will vary over time, a separate ACL and availability must exist for each pipeline and time slot, as shown in Table 1. An example resource definition matrix with populated fields is shown in Table 2.

Table 1. Generalized Resource Definition Matrix

		Time		
		1	2	3
Pipeline	1	• Availability • Reserved by • ACL	…	…
	2	…	…	…
	3	…	…	…

Table 2. Concrete Resource Definition Matrix Example

		Time		
		1	2	3

Pipeline	1	• Available
		• Not reserved
		• ACL
		  ○ User 1
		  ○ User 3
			• Not available
			• User 1
			• ACL
			  ○ User 1
			  ○ User 3
				• Not available
				• User 2
				• ACL
				  ○ User 2
				  ○ User 3

	2	• Not available
		• User 3
		• ACL
		  ○ User 1
		  ○ User 3
			• Not available
			• User 1
			• ACL
			  ○ User 1
			  ○ User 2
				• Not available
				• User 1
				• ACL
				  ○ User 3

	3	…	…	…

4. Communicating Pipeline Configurations

The configuration possibilities that describe a pipeline can be communicated in a number of ways. Using the improved FGN model described in the previous chapter, this information can be included in the interface provided by the CA along with the schedule information. Alternately, this data can be provided through any desired means directly from the network, or a subnetwork, such as through RPC during the beginning of a resource usage.

Sample configurations available on three pipelines can be described, as shown in Table 3. Here, GMSK stands for Gaussian minimum-shift keying, BPSK stands for binary phase shift-keying, AFSK stands for audio frequency-shift keying, and OQPSK stands for offset quadrature phase-shift keying. Note that in a realistic example, there would need to be many more columns to capture each

field that describes the capabilities of a ground station, but which have been left out for simplicity. These tables must be distributed for informational purposes, describing to a user the potential resources available.

Table 3. Example Simple Pipeline Configuration

		TX/RX	Frequency (MHz)	Modulation
Pipeline	1	TX	435-440	GMSK, AFSK
	2	RX	902-928	BPSK, GMSK
	3	RX	2200-2290	OQPSK

D. SCHEDULING

On their own, pipeline configurations do not describe the resources that an FGN provide. A resource consists of the pipeline, and its corresponding configuration options, as well as the time span for which the resource is available. Thus, what is needed for an FGN schedule is a combination of Table 1 and Table 3. This creates rows made up of individual resources, with columns detailing fields that combine to create the resource. These fields range from start and stop times, to radio options such as TX or RX, frequency ranges, modulations, encodings, as well as antenna descriptions such as slew rates and polarizations, finally followed by ground station location information. A simplified sample schedule is shown in Table 4.

Table 4. Sample Schedule

	Pipeline	Start	Stop	TX RX	Frequency (MHz)	Modu-lation	Lat	Lon
Resource	1	10/20/2014 0900	10/20/2014 0920	TX	435-440	GMSK, AFSK	37.3	-76.2
	2	10/20/2014 0900	10/20/2014 0920	RX	902-928	BPSK, GMSK	59.2	-153.3
	3	10/20/2014 0900	10/20/2014 0920	RX	902-928	BPSK, GMSK	38.0	13.3
	1	10/20/2014 0920	10/20/2014 0940	TX	435-440	GMSK, AFSK	37.3	-76.2
	3	10/20/2014 0920	10/20/2014 0940	RX	902-928	BPSK, GMSK	38.0	13.3

The abstraction provided by this resource definition enables filtering of the resources to find which can provide assistance to a user. For example, a user needs a 915 MHz GMSK receive capability for their satellite. They can provide the ephemeris and the aforementioned configured requirements to the CA. The CA can then filter resources to ones that provide the compatible configuration. In this subset of resources, it can then calculate which have line-of-sight to the ephemeris within the bounds of the start and stop time associated with each resource.

E. SCHEDULING SYNCHRONIZATION

The possibility of multiple CAs among a network presents an issue. The duplication of responsibility of scheduling requires synchronization. Without synchronization, a user could request a ground station resource and reserve it with a hybrid CA/SA at a mid-level, which would leave the resource appearing to be available at the top-level CA, when in fact it has been reserved. Synchronization solutions for this problem must be solved in any implementation of the Improved FGN model.

A solution would be to require the hybrid CA/SA at a mid-level to propagate resource request acceptance messages from the node-level SA up to the top level CA. As a CA is only aware of the schedule of the ground station nodes below it, but not in separate branches, there is no need for the message to

propagate through each branch of the tree. The message only needs to travel straight upwards to the top level CA.

A second solution could be to require that when a reservation is made, the bottom level SA, which is the owner of the resource, must request up the chain that the resource is marked reserved or deleted from the schedule. Thus, when a bottom level SA receives a request to reserve a resource, it must take action to notify the CAs above it of the change. Whichever solution is selected, the problem of synchronization must be solved in implementation with multiple CAs.

III. FGN INFORMATION SECURITY SCHEMES

The security of satellite communications has often been of extremely high priority. Even in amateur systems where some message structures are fully public and decodable by the public, some commands and messages are likely to be kept private. Thus, in a ground station network, particularly one using the Internet as a communication pathway, the following are of high importance:

- Confidentiality—know that no party but the intended destination can understand a message
- Integrity—be assured the data is not modified en route to its destination
- Availability—provide reliable access to ground station
- Authenticity—know whom they are sharing a session with
- Authorization—limit who is allowed to use their resources

A. INFORMATION SECURITY OVERVIEW

The first three concepts form the Confidentiality, Integrity, Availability (CIA) triad of information security (Stallings & Brown, 2008). Authenticity and authorization are often considered to be equally important in a secure system. These are the five concepts that must be provided to have confidence in the security of an FGN system.

Availability is usually inherent when using the Internet but methods of improving availability will be discussed in sub-section E of this section. Authorization has been discussed in Section II.C.3. This leaves three concepts to resolve with a security scheme: confidentiality, integrity, and authenticity. These can be provided in a number of ways using modern computer networks.

A widely used network authentication scheme is password authentication. This scheme in its basic form provides authentication, but does not provide either integrity or confidentiality and is thus insufficient on its own for the purposes of most satellite ground station networks. More complicated methods exist that use

passwords to achieve the tenets of information security, but generally they rely on a level of trust that has already been established using another method (RSA Laboratories, 2000).

Public key infrastructure (PKI), using matching public and private keys, solves the shortcomings of password authentication and is a good choice for providing information security in ground station networks across the Internet (Stallings & Brown, 2008). PKI solves the issues of confidentiality, integrity, and authenticity. While, PKI is not the only solution that meets security needs, it is one of the most commonly used. PKI provides a means for the CA, SA, and users all to maintain trust that their data is protected in all directions and only being understood by the intended recipient.

As far as the users of an FGN are concerned, there is only a resource they would like to use and a system in between that enables access to the resource. Figure 16 shows this abstracted view; notice that none of the details inside of the FGN concern the user, and that the FGN simply serves as a relay for communications with the resource. Security must be ensured for each party as the system is composed of multiple entities, and communication is often in an insecure medium, namely the Internet. This necessitates methods of authentication which implementations must follow. Two sample authentication schemes follow.

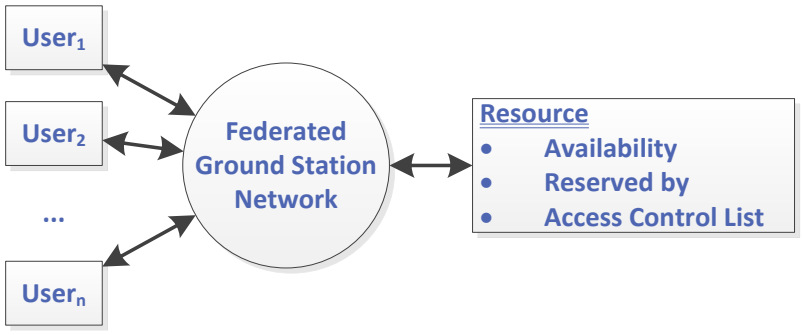


Figure 16. User's View of FGN Resources

B. SHORT INTRODUCTION TO PKI

Certificates and private keys are the crux of PKI security. The process of certificate requests and creation is standard, though as applied to this network model, the role of creating certificates can be played by varying players. The process begins with the following: a new user who desires to use the network will create a certificate request and a matching private key. The private key must be shared with no one and the user must maintain sole knowledge of its contents. The user will then send this certificate request to a trusted authority of the network. This authority must then verify the identity of this new user through out-of-channel means. This could take place through registered mail, in person, or through some secure, verifiable means. Once identity has been verified, the truth authority creates a certificate and sends it in the clear to the user. This certificate is public knowledge and not secret.

Within this certificate is what is known as the *public key*. This public key is cryptographically linked to the private key. The authority never needed to see the private key itself but was able to secure its use for the user. With users Alice and Bob, these two keys allow Bob to encrypt a message to Alice using her public key, contained in her certificate. This encrypted message can only be decrypted by Alice using her private key. This process is shown in Figure 17. Conversely, the private key can be used to *sign* a message that can be checked for validity with the user's public key. For more in-depth information on information security and public-key infrastructure refer to (Stallings & Brown, 2008).

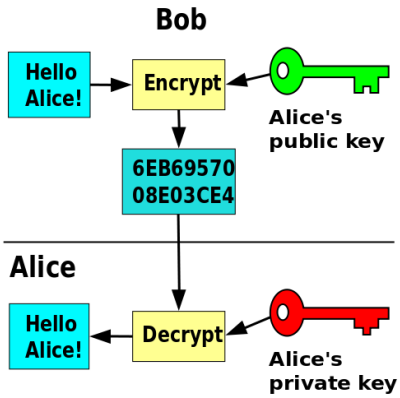


Figure 17. Public Key Encryption (from http://en.wikipedia.org/wiki/Public-key_cryptography, November 3, 2014)

C. CANDIDATE AUTHENTICATION SCHEMES FOR IMPROVED FGN MODEL

This section presents candidate authentication schemes utilizing PKI in federated ground station networks for the purposes of accomplishing a reasonable level of security, while balancing usability with ground station subnetwork autonomy. Following are two primary categories of strategies for FGN security schemes.

- Category 1: the top-level CA manages authorization of users and subnetworks by generating certificates for all members of the FGN, acting like a traditional PKI certificate authority
- Category 2: the top-level CA only provides a means of sharing resource availability and of scheduling.

The primary distinction for the second category is that the CA does not hold the responsibility to provide certification for users to be recognized by subnetworks. The SA must have signed a special subnetwork-specific certificate to identify each user it recognizes, and in this way acts as a certificate authority. A need to maintain tighter control over a subnetwork could drive a desire to use this second category of schemes. If the SAs are the generators of certificates, they hold the keys to their kingdom. Conversely, if the top-level CA is the sole generator of certificates, if any subnetwork recognizes a user, all subnetworks

must also recognize the user. This is because they implicitly trust the certificate as valid and representing a registered user. This does not imply that they must authorize the user to use the subnetwork's resources, but they must recognize the user as a valid registrant. While this provides the user no particular capabilities, security, administrative, or political concerns may dictate a simple top-level CA.

An important concept to note when considering the security of this system is that, while the user controls the node during scheduled resource usage, the user never communicates directly with the node. The user does not need to be made aware of the node's contact information, namely its Internet Protocol address. The SA handles routing between the user and node such that the node is safely hidden behind a layer of obscurity, while allowing the SA to be aware of any communications destined for or coming from a node, thereby enabling traceability. This may help determine which scheme is most applicable.

1. Category 1 Scheme—Network-wide Central Certificate Authority

With a network-wide CA, the CA, or CAs, provides a certificate to every user and SA, and, depending on implementation, possibly to each ground station node. The SA trusts that the CA verified the identity of a user in creating its certificate and thus trusts the binding between the certificate and the assumed user. If all communications between users or the CA and the ground station nodes of a subnetwork are routed through the SA as shown below in Figure 18, then each sub-authority only needs one certificate, and ground station nodes need none. Security of the communications between the SA and the ground station node is independent of the larger security model, and could be done using PKI, symmetric key cryptography, or any secure method and could be different for each subnetwork. This allows for existing subnetworks to rely on their current models and software when joining a larger FGN.

Scheduling and resource publishing is provided through an interface of the CA for the user. The SA informs the CA of all available resources at each node in its subnetwork. The CA, in turn, publishes this information for users. Resources are requested by users from the CA, which routes these requests to the appropriate SA. The SA can accept or deny these requests and the response is returned to the user by the CA. Whether the SA accepts or not depends on a check of its Access Control List (ACL) for the resource. This ACL is not part of this standard but will instead be implementation-dependent as the levels of restriction and determination for authorization can be simple or complex. The MC3 implementation will determine this based upon mappings of users to ground station nodes that they are authorized to utilize. Once a request is accepted, this enables communication between the user and SA at the scheduled time. During this resource usage, communication is directly between the user and the SA, which routes these commands to the appropriate node. The SA in turn routes all data flow, including any received downlink data, from ground station nodes to the user. Figure 18 shows a sample concept of operations (CONOPS) diagram, and Table 5 shows the meaning of each of the operations numbered in the figure.

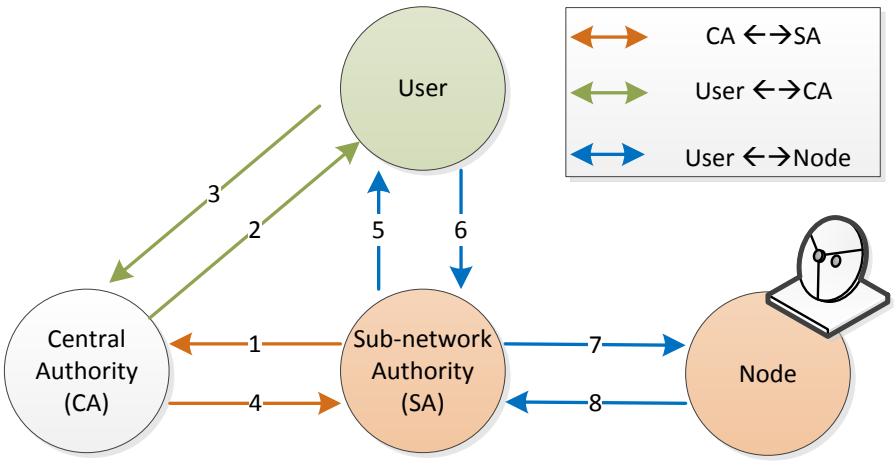


Figure 18. FGN Usage Concept of Operations

Table 5. FGN Usage Concept of Operations

Step	Operation
1	SA publishes available time slots to CA
2	CA hosts available time slots for users
3	User sends a request for a time slot to CA
4	CA pushes, or makes available, time slots requests to SA until these requests reach the lowest level SA
5	At appropriate time, SA initiates connection with user and begins forwarding all data from node
6	User sends configuration and transmit requests to SA
7	SA commands node based on user's requests

2. Category 2 Scheme—Certificate Authority Solely as a Scheduling Interface

In this scheme, high-level operations appear very similar to those in Category 1 except that for communications with a user, a certificate is created for each user by each SA and transferred in out-of-band secure communications. This certificate and its matching key are used for all communications between the SA and user. Each CA generates a certificate for each SA. This secures the CA/SA bidirectional communications. To secure user/CA bidirectional communications, the SA distributes a user's certificate to the CA. When the user needs to request a resource, this certificate is used to secure communications with the CA. This same certificate is utilized by the user to communicate with the SA during scheduled time slots. This allows for each subnetwork to strictly control initial authorization of each user and to have been the authority that generated the identification mechanism.

D. ANALYSIS OF AUTHENTICATION SCHEMES

Category 2 comes with the added complexity of managing many more certificates, with multiple certificates for a single party, but enables each subnetwork maximum confidence in the identity of each user to whom it grants access. This grants each subnetwork absolute control over who has been granted a certificate in order to establish communications. This may serve to comply with security requirements and alleviate concerns that may arise for a

subnetwork considering joining an FGN that may lack trust in the CA's ability to properly authenticate new users.

Conversely, Category 1 relies on a complete trust relationship between the SA and CA, which in turn allows the SA to trust users, while still maintaining the ability to deny authorization, either carte blanche, or specific to a given user at a given time slot. One risk is in being able to recognize an unauthorized user's certificate, but does not pose any risk of misuse of a resource as authorization can be denied just as readily as in Category 2. Another risk is if a CA cannot be trusted to properly authenticate users in a secure manner, which would mean no CA-signed certificate can be trusted.

A single certificate authority provides for a reduction in the number of certificates. This also moves the responsibility of authenticating new users to the CA, freeing the SA of this duty. Loading this duty onto a single source, may hamper scalability if certificate signing becomes a high demand task. Category 1 seems to be a winner between these two schemes as long as the CA can be trusted to securely vet new users.

E. INCREASING AVAILABILITY

Availability is the last piece of the five security principles to ensure. In either category, having multiple CAs helps to increase reliability. In a single-CA system, even if the CA has a 99.9% availability time, this means that 0.1% of the time, the entire network is unable to add new users or schedule new resources. With multiple CAs, redundancy exists that may increase reliability, which is an important consideration for many satellite owners. Conversely, as resource scheduling can be done far in advance, a user should not need to reserve a resource shortly before the resource start time. The standard procedure for using a resource on a network is to reserve it as far in advance as possible. Thus, as long as downtime is short, even if it is frequent, the network should still remain fully viable. Reliability in critical times such as during or just before a resource start time is dependent on the reliability of the SAs and nodes themselves.

IV. STANDARDIZATION OF AN INTERFACE

A. INTERFACE BACKGROUND

The abstraction that the improved FGN model provides suggests the possibility of a standard communication interface that would extend the resources of an FGN to a user. The remote use of a resource by a user across the Internet is often referred to as a service, and as being part of a Service Oriented Architecture (SOA) (Erl, 2005). The World Wide Web Consortium defines a service interface as "the abstract boundary that a service exposes. It defines the types of messages and the message exchange patterns that are involved in interacting with the service, together with any conditions implied by those messages" (W3C Working Group, 2004). Four components comprise an interface:

- Data structure formats – a syntactic specification of what data types group together to form data objects and to be used in requests and responses
- Serialization - a means of translating data to be transferred on the wire between heterogeneous computer systems
- Remote procedure call (RPC) library - the signatures of functions that can be requested of the service provider
- Interface specification – semantic documentation describing what the interface provides and the correct usage and expected outcomes and effects of usage (Bachmann, et al., 2002)

B. BENEFITS OF A STANDARD

By now, CubeSats have become widespread in government, industry, and academic use. Most missions develop their own ground station hardware and software solution but reuse of existing infrastructure would allow efforts to be focused on the satellite and researching new technologies. Often, effort is spent in CubeSat communications in setting up new networks of one-time-use ground stations. Leffke (2013) proposed a method of building simple receive-only ground stations that could be networked and provide receive service. Many other ground

stations with similar capabilities already have been built and have unused resources.

There is not always a need to build new ground stations for a mission, as existing stations are hardware compatible. Incompatibility with existing ground station systems often occurs due to software differences or proprietary software being used for control. With a standard interface, the hardware can extend common functions to any software that implements an FGN's interface. As long as the software has an integrated interface connecting the different components of an FGN in a standard way, what particular software is running at any one computer is not relevant to the computer on the other end of the interface.

As an example, this interface could be integrated into the software currently being utilized by the MC3 network, namely the Naval Research Laboratory's Neptune™ ground station software. The MC3 ground station nodes could continue using Neptune™ software to control their hardware, but would be capable of providing FGN service to users who could be using any mix of software they would like, as long as it implements the same interface. The Neptune™ software would be unaware of what type of software is running on the other end.

Multiple efforts are also under way to create ground station networks. Each will likely have a different custom interface, and thus far, CubeSat ground station networks have shown this to be true (Tubio et al., 2014; Leveque et al., 2007). New and existing networks could implement a common interface and architecture, which would allow for conglomeration. This would expand the options available to users of ground station networks. A standard interface would also simplify a user's development effort when attempting to achieve compatibility with an interface, as developing a single interface would allow for compatibility with multiple networks. If multiple networks, even separate from each other, use the same interface to network with users, the user can develop a solution once and utilize multiple FGNs.

As not all ground station networks utilize the same radio frequencies, and neither do satellites, a unification of the interface also prevents evolution among satellites from affecting the method of communication between users and ground station networks. For example, many CubeSat developers begin with Ultra-High Frequency (UHF) radios onboard their satellites and migrate to higher data rate S-Band radios. If they began using a UHF-centric network with a standard interface, they could slowly migrate to S-Band centric networks and not create the burden of redesigning their ground station network communications.

Finally, a standard allows for differing ground station networks to unify and integrate with each other. Existing ground station networks could conglomerate more simply if they all used the same interface to communicate between each other. Integration between systems using the same standard would involve less change as the architectural framework would be very close.

C. EXISTING FGN INTERFACE STANDARDS

Multiple implementations exist that extend ground station network services, some of which fully or partially describe an interface standard.

1. GSML

The Ground Station Markup Language (GSML) (Cutler, 2004) describes the data structures that are used in the interface, as well as describing a means of serialization. The Hardware Virtualization Layer of GSML is shown in Table 6. GSML does not contain an RPC library as part of the published interface standard, and instead the RPC libraries are implementation-specific. The Mercury Ground Station Network (MGSN) implements GSML, and provides a custom set of RPCs to achieve the functionality an FGN provides its users. As GSML does not explicitly document what actions are to be taken upon receiving a GSML (Extensible Markup Language, XML) document, one could argue it does not compose a full interface (Bachmann, et al., 2002). The MGSN describes its services and implements RPC using XML-RPC. Use of XML-RPC implies use of the Hypertext Transport Protocol (HTTP) as the transport protocol. A standard

defined using these tools is not available at the time of this writing at the home page of the GSML description.

Table 6. GSML Virtual Hardware Level Object Descriptions (from Cutler, 2004)

Object	State
Antenna	Azimuth Angle Elevation Angle Brake Status
Preamp (LNA)	Enabled Status Gain Level
Radio	Transmit Frequency Transmit Mode Transmit Output Power Receive Frequency Receive Mode Receive Attenuation Receive Squelch Level Receive Signal Strength
Output Amplifier	Enabled Status Output Level
Digital IO	Channel State Number of Channels
Analog IO	Channel State Number of Channels
Power Controller	Channel Value Channel Name Number of Channels
CPU	Network Bytes In Network Bytes Out CPU Usage Times Disk Usage (Free/Total) Load Averages Processes Total Processes Running Memory Usage (Free/Total) Status Uptime Name IP Address
Virtual Machine	(Includes CPU state from above.) Type of VM

Object	State
Ground Station	Name Location Pipelines Status
Environment Parameters (common to all hardware)	Temperatures Voltages

2. CCSDS SLE

The Space Link Extension (SLE), created by the international Consultative Committee for Space Data Systems (CCSDS) provides a more comprehensive standard than GSML does with extensive documentation and architectural descriptions (CCSDS, 2005). SLE explicitly details the RPC library, in addition to suggesting a method of data structure formats and serialization, which is largely Abstract Syntax Notation 1 (ASN.1).

While the SLE standard is extremely comprehensive and well-defined, it is intended to be used with a specific set of communication protocols spanning from the lowest physical level to the highest application level: the Space Link protocols (CCSDS, 2001). The CCSDS protocols at each layer were engineered to support CCSDS protocols at other layers, but were not engineered with generality in mind. The CCSDS protocols must be bent and the majority of their capabilities ignored to easily encapsulate the typical capabilities needed for CubeSat users who typically do not use CCSDS protocols at other levels of communications. For example, AX.25, a very common CubeSat protocol, is not by default compatible with CCSDS. Considerable expertise is needed to encapsulate simpler protocols inside of the CCSDS protocol layers.

The SLE definition is spread across a number of documents totaling thousands of pages of documentation putting its complexity well beyond the usefulness for the average small satellite team and showing its intended target audience, which was originally NASA and the European Space Agency (ESA). While this complexity might be useful with the level of complexity present in large satellites flown by the likes of NASA and ESA, it is largely unnecessary for the

simple type of control and ground station network usage a small satellite team would desire or want to invest valuable man hours into implementing.

While the protocol sets define most of what is needed to provide FGN service to a user, a complete implementation would be prohibitive for the typical small CubeSat team that is not interested in implementing the additional layers of CCSDS. In addition, the specificity of CCSDS protocols requires modification to support the array of protocols being utilized by CubeSat missions. As will be shown, the capabilities needed can be fairly simple to achieve use of an FGN.

D. MOTIVATION FOR A NEW STANDARD

The primary objective of any further FGN interface development should be a standard capable of meeting the goals of today's and tomorrow's FGNs. These goals consist of enabling sharing of ground station resources with granularity of control. Each piece of hardware that has configuration possibilities should be considered in the interface. An FGN user may want full control of a ground station's hardware, with the granularity of control to scan frequencies attempting to acquire a signal, or to dither an antenna back in a pattern to attempt to acquire their satellite or to test a satellite acquisition optimization algorithm. Alternatively, an FGN user may want nothing but to transmit and receive packets, which is often called *bent pipe* operation. To use a bent pipe, the SA must control the radio frequency to account for Doppler, predict the satellite position to point the antennas, and prepare any other hardware such as amplifiers and pre-amps. Figure 19 shows these two usage scenarios. Due to the granularity a user may require in controlling hardware, it is highly desirable to have abstraction of each hardware resource available on an interface such that maximum granularity and compatibility with varying models of hardware can be communicated in a single interface standard.

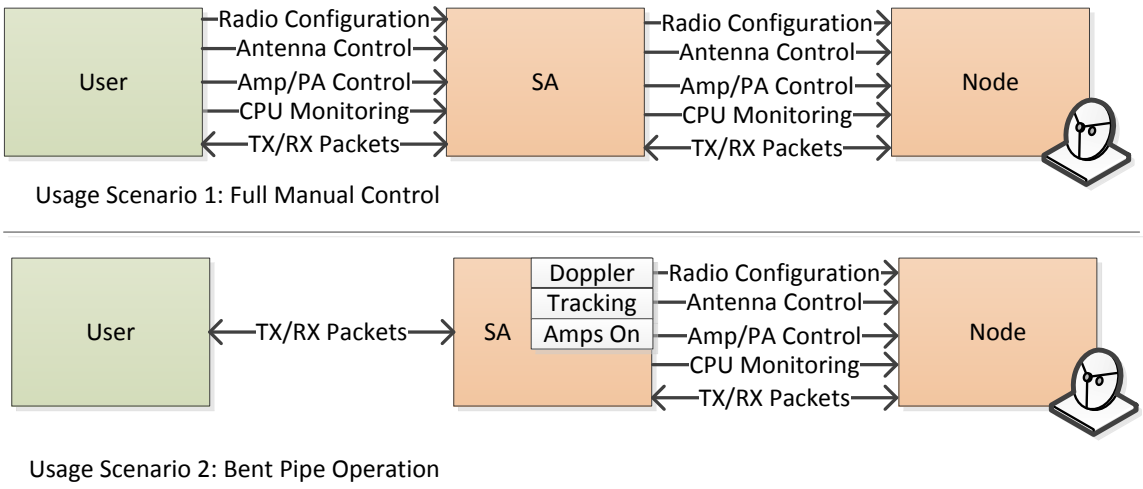


Figure 19. Two Possible Usage Scenarios of FGN Interface

Cutler's work designing the GSML Virtual Hardware Layer, Session Layer, and Network Layer as well as in defining resources as a pipeline paves the way for these objectives to be accomplished (Cutler, 2004). With minor additions and deletions from these abstraction layers, a new interface was developed to fully abstract and encompass the large enumeration of configuration options available among hardware and software-defined radio ground stations. But in addition to the abstraction of the different hardware components and their possible configuration options, a standardized set of functions for altering and checking these configurations and transmitting data was needed. These functions that software can call upon for remote control of a ground station are referred to as remote procedure calls (RPC) and when compiled along with necessary descriptions of the effects of these calls and their expected input and output data it is referred to as an application programming interface (API). Finally, the semantic behavior of the interface must be defined along with expected usage and high-level behavior and interaction descriptions. These components combined make up the full definition of an interface standard and were the products of this research (Bachmann, et al., 2002).

When designing an interface, it is necessary to define specifics such that a standard is implemented correctly between different interpreters of the specification. This level of specifics is missing from SLE as it is written only for CCSDS protocols and encompassing other standard protocols must be implemented as a side branch of the standard. Also, specific implementation recommendations are avoided in the SLE standard leaving the door open to incompatible implementations. The idea seems to have been to avoid cornering any possible implementer of the standard into a design decision they might not desire. While this has its merits, it also leaves the possibility of incompatible implementations of the standard. Defining further layers of software requirements of the interface can serve to further standardize it and maintain compatibility between different implementations extending the utility of a standard.

Standardization, in its current form of GSML and SLE, could be improved in order to encourage users to develop solutions for their mission capable of utilizing existing FGNs. We have developed a standard interface implementing a Category 1-modeled interface (see Section III.C.1). It not only defines the data structures used for input and output, by building upon the foundation of GSML, but also defines the RPC library available for a remote user, as well as the semantics of their use. Using this new interface and model, we should be able to extend the infrastructure of the MC3 FGN to other United States Government and U.S. Government-sponsored contractors and researchers.

E. INTERFACE DEFINITION TOOLS

A new interface standard should be simple enough to be easily comprehended and implemented while still providing sufficient abstraction to enable functionality among a diverse number of ground stations and users. This necessitates enough documentation and specification without ruling out unforeseen uses and demands. The documentation should also thoroughly explain intended usage and effects of RPC calls.

This new interface standard should not limit which languages it may be implemented with, ruling out some obvious options such as using JavaScript with the Asynchronous JavaScript + XML (Ajax). The user and ground station owner should be able to use their existing ground station software to implement the interface creating a need for compatibility with as many languages as possible, or at least the most common languages such as C languages, Java, PHP, JavaScript, Python, and Perl (TIOBE Software, 2014).

The interface tool chosen should also handle schema evolution well. Schema evolution is the term for when an interface changes over time or is versioned. If a data structure to describe the status of an antenna is built, it may well need to change over time as different types of antennas with more functionality are added necessitating new variables. Also, it may be the case that a data type chosen for a field is found to be insufficient or non-optimal. A well-designed interface tool supports the ability to have a client running one version of an interface while the server runs another with minimal effect. The following sections evaluate these options.

1. CORBA

While GSML uses XML to format the structure of its data, and its implementation under the MGSN uses XML-RPC to provide RPC, many different options exist for accomplishing the goal of specifying a standardized interface. The Common Object Request Broker Architecture (CORBA) is a framework for performing RPC in a language-agnostic way. CORBA is one of the oldest interface definition methods, dating back to 1991. It has since been considered by most to be outdated and difficult to implement and understand due to its design by committee. CORBA has largely been replaced by other tools such as RESTful API’s and SOAP (Henning, 2006).

2. SOAP

The Simple Object Access Protocol (SOAP) has been a common choice for web services for the better part of the last decade and remains common

today. SOAP often uses HTTP to transfer XML documents. SOAP is very similar to XML-RPC but with additional features and data types. SOAP is slower than some competitors such as CORBA because data is transferred using XML instead of a binary format. XML is a verbose, textual representation of data. SOAP is highly portable and does not rely on a particular transport protocol. This means that the large numbers of languages with XML support are already close to compatible with SOAP, if a library does not exist already. Also, SOAP can automatically generate code from descriptions of the service, allowing for interfaces to be implemented faster. A drawback of SOAP is that it can be quite verbose when using Web Service Definition Language (WSDL) documents, which it uses to define the RPC. This can be a significant drawback if there are competing standards defining a service, as implementers may be deterred by the perceived complexity of an interface defined using SOAP.

3. REST

RESTful (Representational state transfer) API's are another alternative. In approximately 2008 REST overthrew SOAP to become the most utilized interface definition tool as shown in Figure 20 (Royal Pingdom, 2010). REST attempts to set architectural constraints in the design of the interface including that the server must be stateless. This results in successive calls to a server producing the same response, allowing for caching, and for different client connections to the server to all view the same state. While this may work for many web services, this is hard to imagine applying to a ground station where state not only is different for every client that connects, but also which is constantly changing as real-world events such as the movement of the satellite are taking place. REST can be made to be stateful but this is not in line with the idea of it. One benefit of REST is the ability to discover the resources of a server dynamically. This allows for changes to the interface while it is live. Many RPC implementations of interfaces will break when the interface is updated on one side or the other. Another benefit of REST is that it has implementations in many languages.

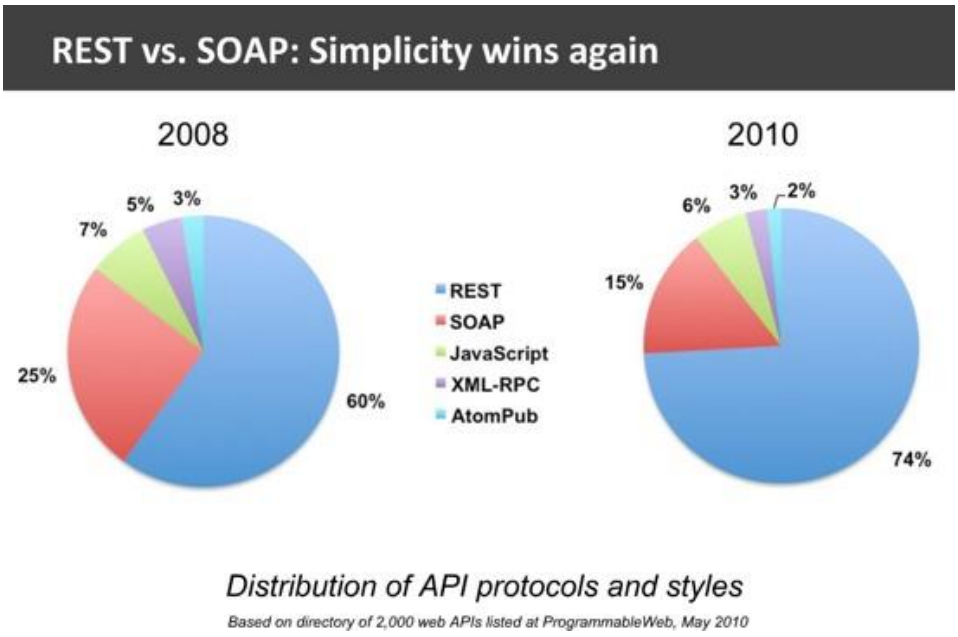


Figure 20. Rest Versus Soap (from DuVander, 2010)

4. ICE

The Internet Communication Engine (ICE) is a proprietary interface framework created by ZeroC, Inc. It has a licensing model that includes a free (GNU General Public License) version for the open source community but also has a proprietary licensing version for commercial customers. ICE is designed to be very fast, with its website advertising that ICE is typically hundreds of times faster than SOAP. ICE also claims to be easier and less verbose than XML (ZeroC Inc.). While it may be faster and simpler than SOAP, the simple addition of a license makes this framework a negative aspect of any standard that would rely on it. In order to achieve the highest levels of adoption, excess costs associated with low-level pieces of the software are to be avoided. Existing commercial ground station software that does not currently hold an ICE license would need to purchase one to integrate with the system. Therefore, regardless of the speed increases ICE may bring, its cost is a high deterrent.

5. Abstract Syntax Notation 1

Abstract Syntax Notation 1 (ASN.1) is an older standard dating back to the 1980s that only defines part of what is necessary to build an interface. ASN.1 covers serialization, which is the process of encoding and decoding data structures across a boundary such as the Internet. Thus, ASN.1 provides a method of encoding data structures, but lacks the RPC framework that is available with other options, which makes ASN.1 an incomplete solution. SLE made use of ASN.1 in its standard for defining data structures. ASN.1 contains a number of standard encodings including textual, binary, and more heavily encoded formats that perform faster than the basic textual ones, and would outperform XML encodings in terms of speed. Downsides of ASN.1 are the broad number of data types available for defining a field. While this may seem empowering, it can often lead to inconsistent implementations where certain developers use a certain data type to represent a field, and other developers use a different type for a similar field.

6. Protocol Buffers

Originally developed by Google and now an open source project, Protocol Buffers, or ProtoBuf as it is often called, also only provides serialization. ProtoBuf is used for Google's RPC communications, but only by building a custom RPC framework on top of ProtoBuf, which Google has kept proprietary. ProtoBuf is fast in terms of serializing data across a boundary and was designed to be faster than XML. The fact that ProtoBuf is open source is often considered a positive as it has withstood the rigors of public review and is constantly being updated. A benefit of ProtoBuf is its ability to handle schema evolution. Yet, ProtoBuf is still missing an RPC framework, which is a significant drawback.

7. Apache Thrift

Apache Thrift is both a serialization and RPC framework similar to CORBA, SOAP, RESTful APIs and ICE, including the RPC that is missing from ASN.1 and ProtoBuf. Thrift was originally developed by Facebook and it is still

used internally for their serialization and RPC needs. Facebook has handed over Thrift development to the open source community at the Apache Software Foundation. Being an Apache project means that Thrift is an open source project and as such all source code is available for public review and modification if needed. Thrift, at the time of this writing, is still relatively new and has yet to reach version 1.0, but is expected shortly. It has been in existence for six years, though, which is a moderate amount of time for software. Literature is scarce but a great introduction and reference guide is available from the Manning Early Access Program as “The Programmer’s Guide to Apache Thrift” by Randy Abernethy. This textbook was used to successfully create a Thrift interface that is described in Section V.

With Thrift, data structures are defined in a language-generic method using the Thrift Interface Definition Language (IDL) that is compiled into many common programming languages. It is currently compatible with 17 languages including C, C++, Objective-C, C#, Java, PHP, and Python. The full list of languages is listed in Table 7.

Table 7. Languages Supported by Apache Thrift (from Abernethy, 2014)

C	C++	C#	D
Delphi	Erlang	Go	Haskell
Java	JavaScript	Objective-C	OCaml
Perl	PHP	Python	Ruby
Smalltalk			

Serialization is provided as a plug-in and different serialization protocols can be chosen from or created to encode data structures across the wire. Data structures can be passed from server to client or vice versa across the Internet, through a data file locally, or through local memory. Regardless of the transport

type, the data structure will be maintained across the boundary. RPC is provided through means of service definitions. Services are defined with full signatures including function names, input parameters, and return data types using Thrift IDL.

A service provider can define their service using Apache Thrift IDL and implement their server-side code in Python. The provider can then share this IDL file with a user, who can compile the file into C++ and add their client code to the auto-generated stub. The user can then interact with the Python server from a C++ program, making RPC calls and passing parameters defined in C++ structures that are translated into Python structures and that trigger Python functions at the server. This process is demonstrated in Figure 21.

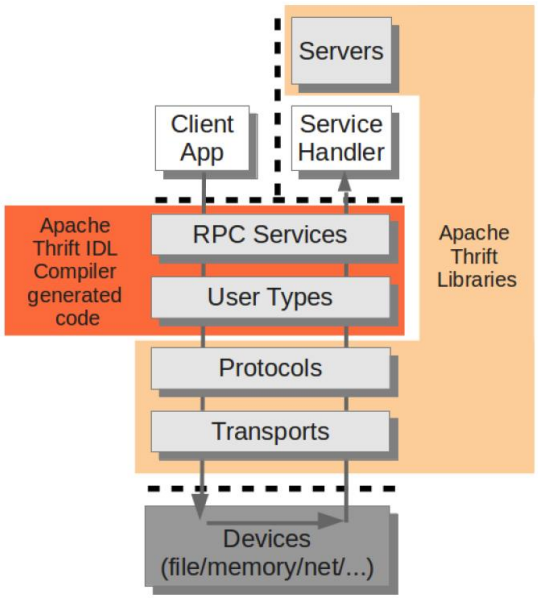


Figure 21. Apache Thrift Framework Diagram (from Abernethy, 2014)

8. Design Decisions

Apache Thrift has been determined to be the most fitting choice for a standard at this time. The primary reason for this is Thrift's completeness. It provides serialization, as each of its competitors do, but Thrift also provides RPC, which ProtoBuf and ASN.1 do not include. While ProtoBuf and ASN.1 can be

connected to RPC frameworks, this must be done externally, complicating the process of defining a standard based upon multiple separate technologies. This lack of completeness in ProtoBuf and ASN.1 leaves CORBA, SOAP, REST, ICE, and Thrift as possible contenders.

Among these tools, CORBA has been considered to be outdated for quite some time and due to the problems described in Section IV.E.1, has largely been abandoned by the programming community. SOAP and REST remain viable options. SOAP typically uses XML documents (W3C Woking Group, 2007) and as such is often slower than necessary when compared with binary protocols. While REST is a capable choice, it is a style more than a framework. Each language has its own frameworks that implement the REST style. Choosing REST would leave much work up to the designer in setting up a framework. The designer may find they are the only implementer of the standard using the framework they choose, possibly presenting unique incompatibilities. The ideal framework will require less work on the programmer's part to encourage adoption of the standard. This leaves two complete interface frameworks with ICE and Thrift.

Both ICE and Thrift perform well, are highly configurable, support a number of languages, and have support for schema evolution but the scales tip towards Thrift in a couple key areas. Thrift provides support for a larger set of languages than ICE does, covering a key design requirement mentioned at the beginning of this section. It also is completely free, whereas ICE requires a licensed product to be used in a commercial setting. Thus, Apache Thrift was chosen to define and implement an interface.

THIS PAGE INTENTIONALLY LEFT BLANK

V. M-PIPE INTERFACE

This section presents an interface that was designed to extend the resources that an FGN can provide across the Internet. The interface is called the MC3 Picosatellite Interface Protocol Extension (M-PIPE). Thought its name includes MC3 and Picosatellite, it is specific to neither and is a broad ground station network standard. It provides support for two primary categories of functionality: remote operation of ground station hardware, and scheduling of resources. An introduction to the middleware framework chosen to build our interface, Apache Thrift, follows.

A. INTRODUCTION TO APACHE THRIFT

As was explained in Section IV, Thrift uses a language called IDL to encode the data structures, services, and exceptions it can describe. Data structures come in different forms such as a struct, union, enumeration, and other common data structures from other programming languages. Code Listing 1 is an IDL code snippet from the actual M-PIPE interface that provides manual control of an antenna's steering. It defines a data type and a data structure for detailing the position of an antenna.

Code Listing 1. Type Definitions and Data Structures

```
1     typedef double Degrees
2
3     /** Defaults to a position of 0 azimuth, 90 elevation */
4     struct AntDirection {
5           /** Azimuth in degrees*/
6           1: Degrees azimuth = 0.0
7           /** Elevation in degrees */
8           2: Degrees elevation = 90.0
9     }
```

The code starts off by naming a new data type of *Degrees*, which is equivalent to a double, a common programming data type representing a floating point number with high precision. This is merely for convenience and clarity of

the meaning of data fields that utilize it. The code continues by making a data structure named *AntDirection* that is composed of an azimuth and elevation direction. These are set to a default value of 0 and 90 degrees respectively. The Thrift compiler reads from detailed C-style comments throughout the code to generate Application Programming Interface web pages that help programmers utilize the interface.

The true power of Thrift or any other interface framework lies in defining services. Services are groupings of functions that each have return types and parameters. They are semantically equivalent to defining an interface in object-oriented programming (Apache Software Foundation, 2014). The following example is further sample code from the M-PIPE interface that controls the antenna's pointing direction:

Code Listing 2. Service Definition Example

```
1     /**
2      * AntControl allows for manual control of the antenna
3      */
4     service AntControl {
5           /**
6            * AntPoint points an antenna in a given direction. Returns the
7            * input antDirection repeated indicating success, otherwise
8            * throws a PointingException.
9            */
10          AntDirection AntPoint(
11                /**
12                 * an AntDirection struct describing Az and El to point
13                 * towards
14                 */
15                1: AntDirection antDirection
16          )
17          /**
18           * exception indicating the requested antenna direction was
19           * Invalid
20           */
21          throws (1: PointingException pointingException)
22    }
```

This code begins by defining a service named *AntControl*. It has a single function available called *AntPoint* that takes one parameter that is the

AntDirection struct defined in the previous section. The function can throw an exception, defined in code available in Appendix A.E. This exception informs the caller of a problem such as an invalid direction that the antenna cannot satisfy such as a negative elevation that would exceed the physical capability of the antenna controller.

1. Server-Side Stub Code

As mentioned in the previous section, the Thrift compiler takes IDL code and converts it into stub code in the language of choice. This stub code contains both client and server implementations. The programmer then needs to fill in the details of the implementation of a service call. For example, the server-side stub code for the Antenna interface shown above would generate an empty function called *AntPoint*. This code would then be extended by the programmer to drive their ground station's antenna similar to the example in Code Listing 3.

Code Listing 3. Server-side Stub Code Example

```
1     import sys
2     sys.path.append('gen-py')
3     from thrift.transport import TSocket
4     from thrift.server import TServer
5     from mpipe_antenna import AntControl
6
7     class AntHandler(AntControl.Iface):
8           def __init__(self):
9                 self.status = True # Set initial status to good
10                self.currDirection = AntControl.AntDirection(azimuth=0.0,
11                                                elevation=90.0)
12          def AntPoint(self, antDirection):
13                if antDirection.azimuth < 0.0 \
14                      or antDirection.azimuth >= 360.0 \
15                      or antDirection.elevation < 0.0 \
16                      or antDirection.elevation >= 180.0:
17                      print("[Server] Bad antenna point parameters.
18                            Raising PointingException")
19                      raise AntControl.PointingException(antDirection)
20                print("Pointing antenna to %s" % (antDirection))
21                # Insert real driver control here
22                self.currDirection = antDirection
23                return antDirection
```

The AntControl.Iface abstract class is part of the stub code created by the Thrift compiler. The AntHandler class builds upon this abstract class to give meaning to the function call AntPoint. It begins by setting up an __init__ function, that is Python's way of constructing a new instance of an object. This function will be called whenever a new AntHandler is created. It begins by setting up status and current direction variables and setting them to known states. Next, the *AntPoint* function is implemented. First, some checks on the input parameters from the client are performed, and, if the data fails to pass checks, an exception is raised, which Thrift automatically handles by returning the exception to the client. The client would receive this exception in their implementation language's exception class types when they attempt to make a call with bad parameters. If the parameters pass, the code continues with a comment indicating where actual driver-level code would be inserted (see the comment on line 21 of Code Listing 3) For some antenna controllers such as the Yaesu antenna controller commonly used on CubeSat ground stations, this would be as simple as opening a serial

port to the antenna controller, sending the text “W2 000 090” to set the azimuth to zero and the elevation to 90 degrees, and closing the port again.

All of the complexity of setting up a socket server, serializing and deserializing data, and designing the RPC language that tells the server which function the client is calling are all handled by the Thrift stub code plus a handful of configuration lines, shown in Python below for the antenna example.

Code Listing 4. Server-side Python Configuration Example

```
1      svr_trans = TSocket.TServerSocket(port=12345)
2      processor = AntControl.Processor(AntHandler())
3      server = TServer.TSimpleServer(processor, svr_trans)
4      server.serve()
```

The first line opens a server-side socket listening for client connections on port 12345. In this and all other examples shown, communications are on an insecure socket connection. The actual M-PIPE implementation will need to use Secure Socket Layer (SSL) sockets to securely communicate. The second line hooks in the *AntHandler* class defined in the previous code snippet to a *processor*, which is the level of Thrift that determines if a particular series of bits coming from a client are a request for *AntPoint* or some other function and finds the input parameters and connects them to their named variables. The amount of work involved in creating the level of code of the processor and serializer from nothing is large in comparison to the example. Thrift provides a benefit in being able to start building from this framework and begin at the level of actual function call implementations. The third line sets up a full server linking the listening socket and the processor. The final line simply starts the listening process and does not return until the server is shut down. More complex types of servers such as threaded or multiprocessing servers are also available in Thrift for Python and many other languages. For details on this, refer to Abernethy (2014). As one can see, the server-side implementation is quite simple using Apache Thrift.

2. Client-side Stub Code

The Antenna interface client side Python implementation is simpler than the server-side code, which the author believes was simple to begin with.

Code Listing 5. Client-side Example Implementation in Python

```
1     import sys
2     sys.path.append(“gen-py”)
3     from thrift import Thrift
4     from thrift.transport import TSocket, TTransport
5     from thrift.protocol import TBinaryProtocol
6     from mpipe_antenna import AntControl
7
8     socket = TSocket.TSocket(“localhost,” 12345)
9     socket = TTransport.TBufferedTransport(socket)
10    socket.open()
11    protocol = TBinaryProtocol.TBinaryProtocol(socket)
12    client = AntControl.Client(protocol)
13    try:
14          az = 41.3
15          el = 11.2
16          pointDirection = AntControl.AntDirection(azimuth=az,
17                                                   elevation=el)
18          msg = client.AntPoint(pointDirection)
19    except AntControl.PointingException as pe:
20          print(“[Client] Server rejected our antenna pointing request of
21                az: %f, el: %f” % (az, el)
```

A large amount of the code shown above is merely the imports. Following these, a socket is connected to the server on port 12345. Then a buffered transport layer is wrapped around the socket to optimize the number of packets sent to complete communications. Next, the socket is opened. Following this, the Thrift Binary protocol is chosen as the serialization method. The server in the previous section chose Thrift Binary by default. A different serialization method could be used, including JavaScript Object Notation (JSON) or more compact serializers such as the Thrift Compact method by providing parameters to the server code (Abernethy, 2014). Next, a client is created from the *AntControl* client-side stub code. An antenna direction structure is populated with the desired pointing directions. Next, a call to *AntPoint* is made. This single call handles the passing of the RPC data, including the function name and parameters, through

the serializer, to the server, and then awaits a response from which it saves to msg. This single line handles a large amount of underlying code that the developer need not implement. Finally, as we are within a try block, if this call receives an exception from the server for an invalid pointing direction, it will be caught and reported here and could be handled as desired. Clearly, the hardware control Thrift interface is quite powerful and simple to implement.

3. Thrift Schema Evolution

Thrift provides powerful tools for handling schema evolution. Schema evolution is the process of an interface and its definition changing over time. In the definition of the *AntDirection* struct, it may have been noted that the fields were numbered. Thrift needs these numbers to identify the meaning of a field as it comes across the wire and to correctly map a value to a variable when deserializing. These numbers are not required and, by default, every time Thrift compiles and IDL file it will assign numbers to the fields. Thus, if they are not manually numbered as shown above, they will arbitrarily be assigned numbers. Thus, if a new field is added to the server's interface, but not to the client's, their numbering will differ. Numbering fields removes this uncertainty.

One of the most important ways Thrift can support schema evolution is by making use of *requiredness*. Requiredness is an attribute of fields that describes whether or not it must be written or read when serializing and deserializing the struct it belongs to. This allows for some fields to be marked required such that Thrift will throw an error if it receives a struct not containing the field, or to be marked optional such that it does not need to be in the struct to be passed through Thrift. Finally, the default requiredness option in Thrift necessitates that the field be written by the struct creator, but does not need it to be present for the struct reader (Abernethy, 2014).

These requiredness options provide tools that the interface designer can use to maximize the ease of schema evolution. Most fields in M-PIPE are marked optional. This allows for these fields to be removed if needed. This may be

necessitated by a newer definition of a field with a different type. This also allows for a single struct describing both hardware and software radios as well as the many different types of equipment that may be used, each having their own fields. It is intended that minor differences in radios should not necessitate changes in the names or types of the fields in the M-PIPE interfaces. Making fields optional also allows for the ground station implementer to decide which fields they want to share on a field-by-field basis. An example of one such structure making use of requiredness is shown in the optional fields of a computer status that the ground station may want to avoid sharing.

Code Listing 6. Requiredness Example

```
1     /** CPUStatus describes the current status of the server’s computer */
2     struct CPUStatus {
3           /**
4            * CPU health status. True means healthy, False indicates
5            * degraded
6            */
7           1: bool status
8           /** Time in seconds since computer booted */
9           2: optional i64 uptime
10          /**Number of processes total */
11          3: optional i64 numProcessesTotal
12          /** Number of processes running */
13          4: optional i64 processesRunning
14          /** Average CPU load as a percent */
15          5: optional mpipe_types.Percent loadAve
16          /** Disk usage as a percent */
17          6: optional mpipe_types.Percent diskUsage
18          /** Disk free in MB */
19          7: optional i64 diskFree
20          /** RAM usage as a percent */
21          8: optional mpipe_types.Percent ramUsage
22          /** RAM free in MB */
23          9: optional i64 ramFree
24    }
```

B. INTENDED USAGE OF M-PIPE INTERFACE

As was shown in the previous sub-section, both the server- and client-side implementations of the M-PIPE interface are simple to implement and will be easy to integrate with existing ground station software.

1. Hardware Control

Hardware control was broken down into the component pieces of a radio frequency (RF) ground station chain as described by Cutler et al. for GSML in the Hardware Abstraction Layer. This includes the antenna controller as well as radio hardware. Whereas a ground station transmit RF chain often has an amplifier, a ground station receive RF chain often contains a preamp. In many cases, the receive and transmit chain are both using the same hardware, and thus both may be present in an RF chain. As the combination of this hardware varies as described, the amplifier and preamp were kept separate in the interface design. Table 8 lists all of the abstracted hardware interfaces included with M-PIPE in its first release.

Table 8. Available M-PIPE Hardware Interfaces

Hardware Interfaces	Description
Antenna	Controls antenna pointing and polarization
Preamp	Controls power and gain
Amplifier	Controls power and gain
Radio	Controls frequency and configuration
CPU	Checks status and health
Packet	Forward data for transmit or from receiver

Each interface is separate from any other interface and any combination of these can be hosted by a ground station. If a ground station prefers not to provide control of the amplifier or does not want to share statistics regarding the CPU, they simply do not need to implement the interface. If they would prefer to hide the antenna control and instead only provide automatic antenna tracking, they can do so.

As well as specific hardware control, there is a Session Level interface similar to the Session Level described by Cutler et al. in GSML. This session level control provides for two primary functions: automatic tracking of a satellite with the antenna, and automatic Doppler compensation for radios. Both require a two-line element (TLE), which is a description of the orbital parameters of a satellite and that provides the information needed to calculate the position and speed of a satellite relative to the ground station's position.

There are two large categories of satellite ground software. The first is oriented more towards the satellite itself, and can be referred to as mission control software. The second category is oriented towards the ground station hardware, and is often referred to as the ground station software. These may be integrated, or may be separate pieces of software communicating with each other. M-PIPE is designed to connect a ground station that contains ground station software with a user who is either using ground station software with integrated mission control software, or who is solely using mission control software.

a. Usage Scenario 1—Integrated Ground Station Software

A user with integrated ground station software can integrate the hardware control functionality of M-PIPE with their existing software such that when the existing software calculates the satellite prediction to necessitate pointing at a certain azimuth and elevation, its existing calculations are connected to calls to point the antenna using the client call shown in the previous sub-section, *AntPoint*. The user can choose to map all hardware control from their existing software to calls in M-PIPE for each piece of hardware included in the hardware control interfaces.

b. Usage Scenario 2—Mission Control Software

If instead a user is solely using mission control software, the responsibility to control hardware shifts largely towards the ground station itself. A ground station utilizing M-PIPE can choose to provide automatic Doppler correction and

antenna pointing when provided with the satellite's orbital elements, or TLEs. The ground station then only needs to know how to configure the radios, pre-amp, and amplifier and will continue passing any uplink or downlink data through the Packet interface. The radio configuration will be different for each satellite, and thus the mission control software the user has must be extended to be capable of manipulating the Radio interface to the point of setting modulations, modes, center frequencies, and other RF settings. If the ground station can choose default values satisfactory for its amplifier and pre-amp hardware, these interfaces can be ignored by the mission control software as well. CPU monitoring is not a required task. Therefore, beyond radio configuration, the only task the mission control software must accomplish in terms of M-PIPE is formatting its uplink packets into the Packet interface for transmission, and in decoding the downlink packets on another Packet interface for downlink.

c. Usage Scenarios Summary

Thus, two usage scenarios can be supported by a flexible ground station implementation of M-PIPE. No requirement to provide the entire functionality set exists in order to ease the difficulty of creating an M-PIPE-capable ground station. Figure 22 shows the basic usage scenarios described. Usage Scenario 1 illustrates full hardware control by the user's ground station software. Usage Scenario 2 illustrates the extension of mission control software to integrate radio configuration control as well as the Packet interface. There are various other enumerations that could be created including implementing parts of the hardware interfaces that mission control software would not typically contain without needing to implement all of them.

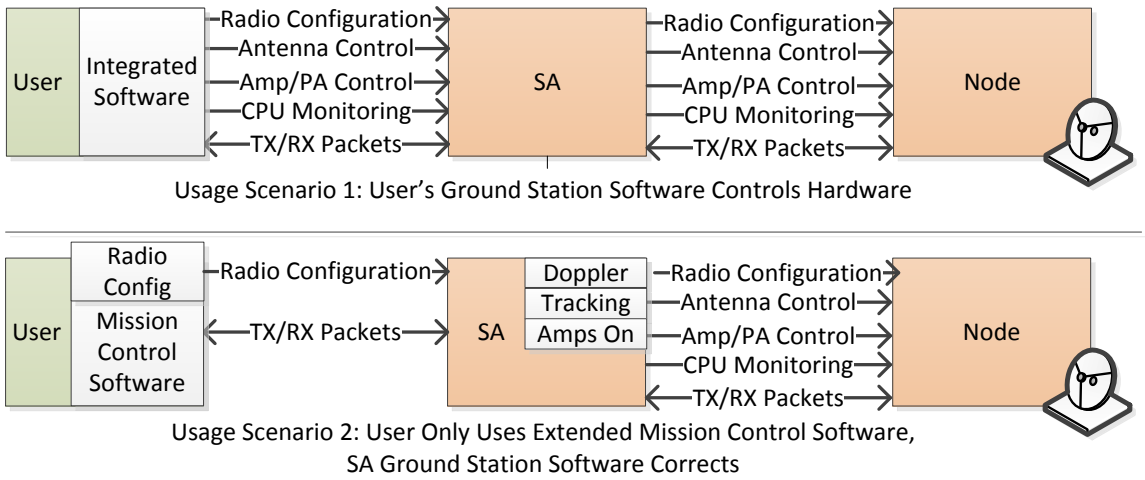


Figure 22. Two Basic M-PIPE Usage Scenarios

2. Scheduling

In addition to hardware control, methods for resource scheduling are included.

a. User Interface

An interface must be provided that supports a way for users to schedule resources they would like to utilize. It is envisioned that initially a website could be set up to provide a way to view and request resources. This website would be hosted at a CA, perhaps in a de-militarized zone (DMZ) for network security purposes. The scheduling interface could later be created in a Thrift interface as a future version of M-PIPE and this will be discussed in the section on future work.

To become a registrant of an M-PIPE FGN, a user must obtain a public and private key. The public key will be contained in a certificate along with other identifying information for the user. The common steps for creating a certificate are shown in Table 9. This process provides a signed certificate that any network member can use to secure communications with the user, identical to how secure website communications operate. As mentioned previously, SSL sockets

will be utilized for M-PIPE and certificates and private keys generated through this process will secure the connection between the user and the GSN.

Table 9. Certificate Creation Process

Step	Description
1	User generates public/private key pair using freely available software
2	User generates a certificate signing request (CSR)
3	User provides CSR to CA in secure out-of-band method such as secure email
4	CA verifies information with user
5	CA creates a certificate for user and returns to user

The user must authenticate to the CA web server. This enables the CA to provide the SA with the user's public certificate and username when resources are requested. With the user's certificate, the SA can validate whether or not the user is authorized to use the resource. Regardless of the shape of the user interface, it must be capable of linking this information to requests for resources to allow the SA to identify the requestor.

b. Intra-Network Interface

Inside the FGN's network of CAs and SAs, scheduling communication needs to occur as well. The bottom-level SA must be able to communicate with the CAs to report what resources it has available. Inversely, the CAs must be able to communicate with the SAs to request a resource on behalf of a user. Accomplishing this goal necessitates an interface hosted by the CA and another hosted by the SA.

The CA must host an interface that allows for an SA to offer a resource to the network. A report containing the configuration options of a pipeline using the

resource laid out in the Improved FGN model must be passed to the CA. A resource description in M-PIPE is composed of three parts: the top-level description, the radio options, and the radio restrictions. Table 10 shows a summary of the contents of these configuration descriptions.

A resource is composed of a top-level description, which contains a single *radio options* description, which itself contains zero or more *radio restrictions*. Radio options are possible settings of a radio for each configurable setting of the radio. Radio restrictions are incompatible sets of settings. For example, imagine a radio that can provide baud rates of 9600 and 1 megabit, but can only provide 9600 baud with GMSK modulation and 1 megabit with OQPSK modulation. In this case, radio restrictions should include a pairing of 9600 with OQPSK as well as a pairing of 1 megabit with GMSK. Thus, a resource will contain one radio options consisting of all available settings, but may contain multiple radio restrictions for each invalid set. The number of radio restrictions is expected to be small as incompatibilities are usually between categories such as baud rates and modulations. Figure 23 shows a UML object diagram indicating the relationship of M-PIPE resources. Variables which are decided by the radio are placed under *radio options* while more generic variables such as frequency and bandwidth are placed under at the top level under *resource*.

Table 10. M-PIPE Resource Description

Layer	Components
Top-level	Name of the SA
	Latitude, longitude, and elevation coordinates of the ground station
	Range of frequencies available
	Time range associated with the resource
	Models of the hardware
	Directionality (TX or RX)
	Permissions regarding hardware control
Radio options	Mode
	Modulation
	Bandwidth
	Link Layer protocol
	Encoding scheme
	Baud
	Viterbi, pseudo-random number randomization, Reed-Solomon en/decoding, Turbo coding
Radio restrictions	Incompatibilities among radio option pairings

The actual configuration description in M-PIPE is lengthy and may at first appear to be a drawback, but this resource description, once completed for each ground station, will likely be repeated until the actual hardware chain is modified. Additionally, the completeness provided by this resource description allows for a thorough check of compatibility between the satellite and the ground station. In practice, a satellite and ground station may be compatible in 99% of their configuration options, but this last 1% can cause incompatibility between the two, barring communications.

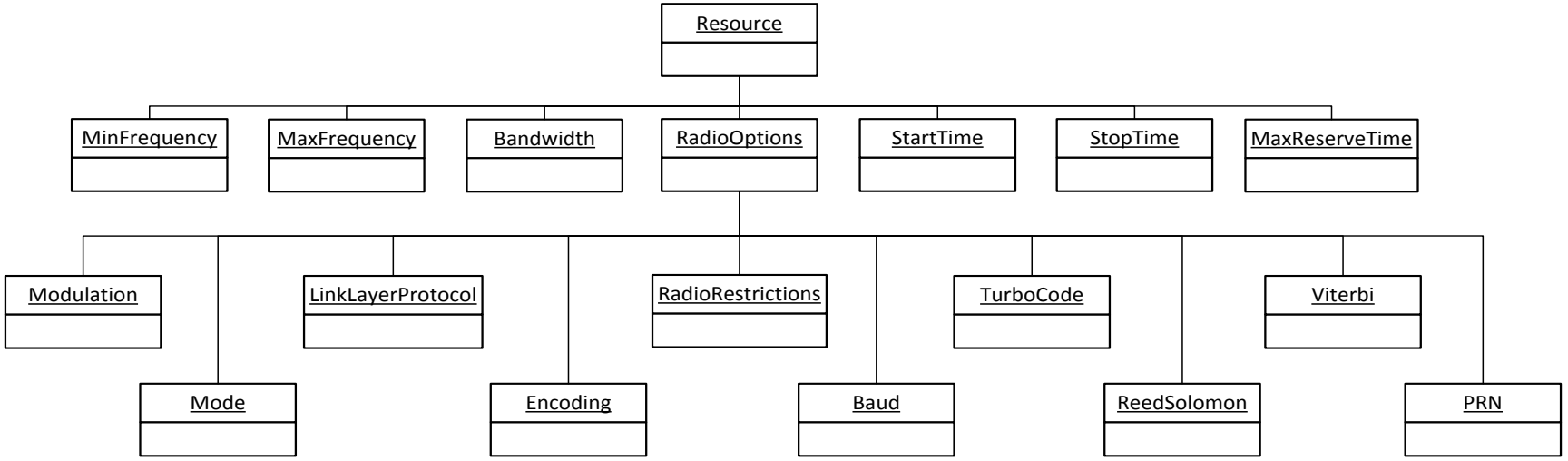


Figure 23. Resource Object Diagram

The use of the Improved FGN model as described for M-PIPE poses an issue that must be dealt with. In the Improved FGN model, SAs can also serve as CAs for their subnetwork. This creates the possibility of multiple servers offering the same resource simultaneously. Thus, synchronization between the node-level SA and all of the CAs in the tree above it arises as a requirement. An example that illustrates the possible desynchronization of states is explained in Table 11 which references Figure 13.

As this example makes evident, a method is required for posting resources up the tree beyond its neighboring parent node. Step 2 of Table 11 requires that SA3 passes the resource to CA1 even though there is no direct communication between the two. This necessitates that when a CA receives a resource offering from a player below it, whether it is a CA or an SA, the resource must then continue being offered upwards in the tree until it reaches a top-level CA. The example also shows how deletions must be passed on up the tree for the same reason.

Table 11. Example of Desynchronization of CAs

Step	Description
1	SA3 posts resource to CA2 above
2	SA3 posts resource to CA1 above
3	CA2 and CA1 both host resource on their scheduler
4	User interfaces with CA2 and requests resource
5	CA2 requests resource from SA3 on behalf of user
6	SA3 accepts
7	CA2 relays acceptance to user
8	SA3 requests CA2 delete resource from its schedule as it is reserved now
9	CA1 continues to host resource in schedule even though now reserved

The requirement for different players in the network to reference the same resource repeatedly drives a need for a unique identifier for each resource. Without a unique identifier, any reference to the resource must include the full description of it, and require a lookup on each player's database to find the resource being referenced. A unique identifier simplifies this problem greatly. Two methods were designed and analyzed that could be pursued to get a unique identifier for resources.

The first method is that each CA can create an identifier including its own name such that it always knows its resource count and can give the next number to attach to the ID. Then each SA with a resource to offer requests an ID from each CA to which it would want to post a resource. It then can talk with each CA about the resource in question using the CA's unique identifier for it.

The second method is that only the top-level CA can create ID numbers. Any request for an ID number is routed up the tree until it reaches the top-level CA who responds. This ID is unique in the entire FGN.

Comparing these two methods, there are pros and cons to each. Method one, with each CA creating a unique ID, is bulky and overly complicated to maintain. Each resource would need to have a mapping between the CA and the ID maintained at the node-level SA. Also, each request to delete must be a separate request directed to a specific CA, harming the scalability of the interface. But one important benefit of this method is that the SA can choose which CAs host their resource in their schedule. Method two is quite a bit simpler. It does have costs, though, including the fact that if a new top-level CA is chosen, all current resources must be remapped to a new identifier. Also, this method needs all CAs up the tree to show the same resource. No partitioning can be performed.

The drawbacks of method two were considered acceptable. The frequency of adding a new top-level CA is likely to be extremely infrequent with any utilization scenario of M-PIPE. Also, if the idea is to prevent unauthorized use of a resource, a request for it can simply be rejected. Thus, method two, where the top-level CA generates unique identifiers for the whole network was selected. The concept of operations (CONOPS) for scheduling shown in Figure 24 and Table 12 represents the method chosen.

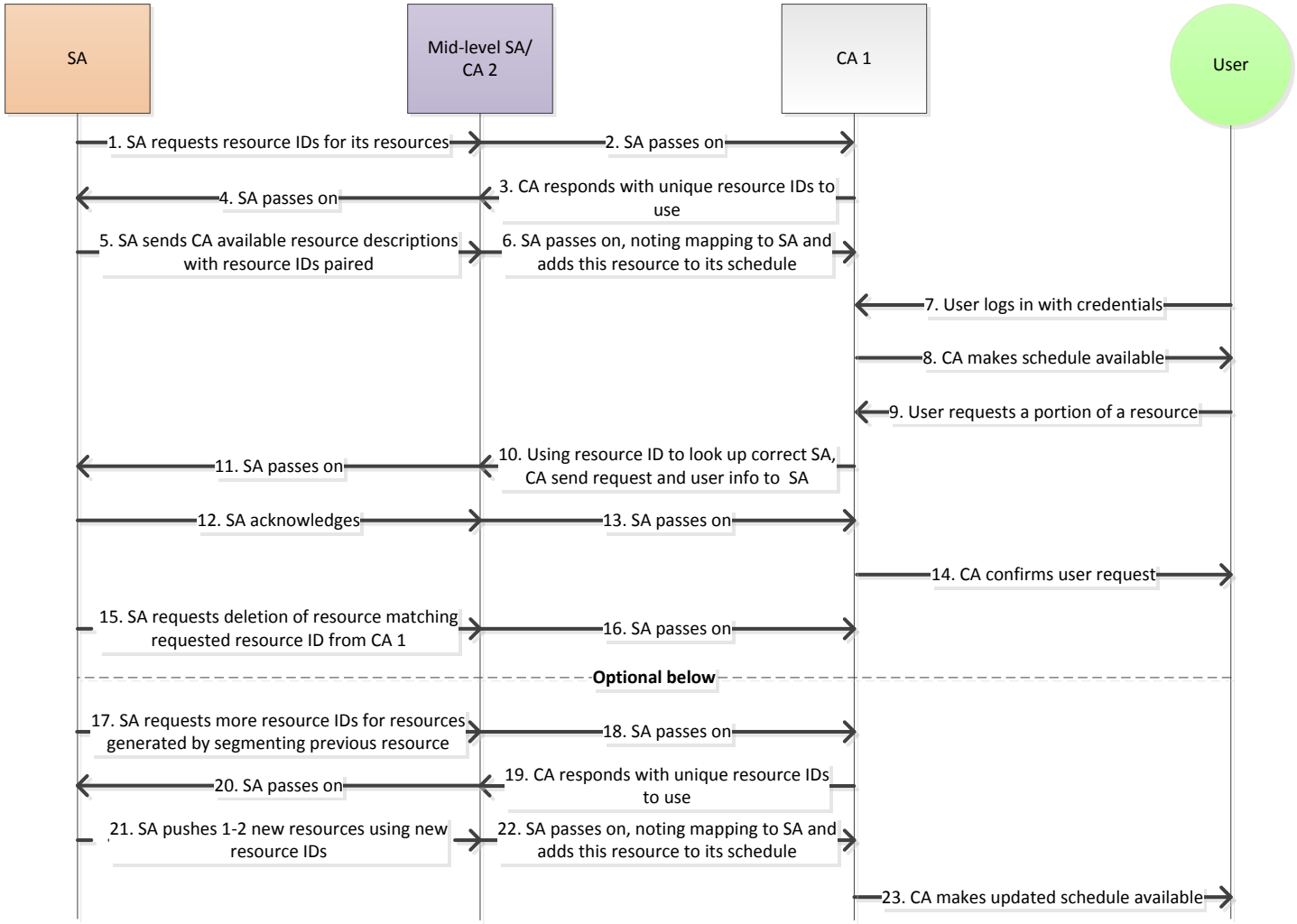


Figure 24. M-PIPE Scheduling CONOPS

Table 12. M-PIPE Scheduling Steps

Step	Description
1	SA requests resource IDs for its resources from mid-level SA
2	Mid-level SA passes on to CA
3	CA responds to mid-level SA with unique resource IDs to use
4	Mid-level SA passes on to SA
5	SA sends mid-level SA available resource descriptions with resource IDs paired
6	Mid-level SA passes on to CA, noting mapping between SA and resource and adds to its schedule
7	User logs in to CA interface with credentials
8	CA makes schedule available to user
9	User requests some portion of a resource from CA
10	Using resource ID to look up correct SA, CA send request and user info to mid-level SA
11	Mid-level SA looks up owner of resource and passes on to SA
12	SA acknowledges request to mid-level SA
13	Mid-level SA passes acknowledgement on to CA
14	CA confirms user request to user
15	SA requests deletion of resource matching requested resource ID from mid-level SA to clear from schedule
16	Mid-level SA passes on delete request to CA
17	SA requests more resource IDs from mid-level SA for resources generated by segmenting previous resource
18	Mid-level SA passes on ID request

19	CA responds with unique resource IDs to use
20	Mid-level SA passes on IDs to SA
21	SA pushes 1–2 new resources using new resource IDs
22	Mid-level SA passes on, noting mapping to SA and adds this resource to its schedule
23	CA makes updated schedule available for users

The important behavior to note is that all messages to add or delete a resource or to request a resource ID are routed all the way up the chain, as described in method two. Also, requests for a resource are routed back from the CA to the node-level SA using the mapping that was created by any SAs in between as the resource was added. This mapping is not a complete route, but, instead each mid-level SA along the way notes from which SA below it the resource was routed. Thus, in this example, the mid-level SA does not necessarily know that the SA below it is the owner of the resource. It simply needs to know that the request for the resource should be passed to it. If that SA needs to pass on the message, it does not concern the mid-level SA. This simplifies routing, as only neighbors one step away need to be routable. No direct communication needs to occur more than one level away.

There is a dotted line in Figure 24 that is labeled “Optional below.” This is because below that dotted line, the node-level SA is attempting to offer up the unused portion of the original resource. As a resource covers a certain time period, if a reservation does not request the entire time period, this resource can be split into two or three resources, with one being the reserved portion. Figure 25 demonstrates what happens when a resource is reserved for a portion of time in the middle of its coverage time. New resources can be created at the beginning and end of the resource. A minimum length of time should be considered before splitting a resource. If only seconds are left, there is no valid reason to create a new resource for this unused portion of time. If new resources

are not created, the portion of the CONOPS below the “Optional below” line does not need to occur.

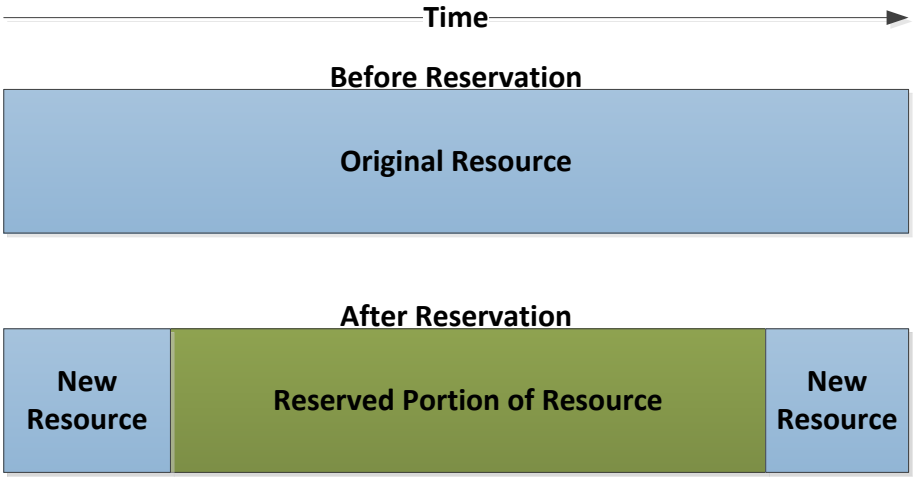


Figure 25. Resource Split by Reservation

State information must be kept by each player in the network such that neighbor mappings are the sole responsibility of the player, and should not need to be re-requested from other players. It is planned that for the first implementation of M-PIPE, state information will be kept in a file. The top-level CA also needs to keep a permanent tally of resource IDs so that it never causes a collision by re-assigning an ID. Keeping state in a file is an easy task as Thrift already provides methods of reading and writing files. Thrift allows for serialization to a file, and this way the state can be stored after each modification to the schedule. Programming languages also provide their own methods of storing data for later use, such as Python and its Pickle module that can be used to serialize data to disk and later be deserialized into memory once again (Python Software Foundation, n.d.).

The M-PIPE interface that is offered by a higher level authority to a lower level authority is named the Resource Scheduler service. The name of the interface offered by a lower level authority to a higher level authority is the Resource Offerer service. The Resource Scheduler provides a method to request

a resource ID for identifying new resources to the system, a method to offer a resource to the network, and another to delete resources. The Resource Offerer service provides a method to request a resource on behalf of a user. The names of the two services may seem backwards, but the service will be hosted by the player to whom the name applies, as shown in Figure 26. The figure shows the same CONOPS as shown in Figure 24 but with actual method calls from the M-PIPE interface. In the figure, the calls to methods arrive at the Resource Scheduler and Offerer servers. The returns calls and clients are not shown for simplicity but each player has a client to the server it is making calls to. Thus, the arrows arrive at the server but start from a client at each player.

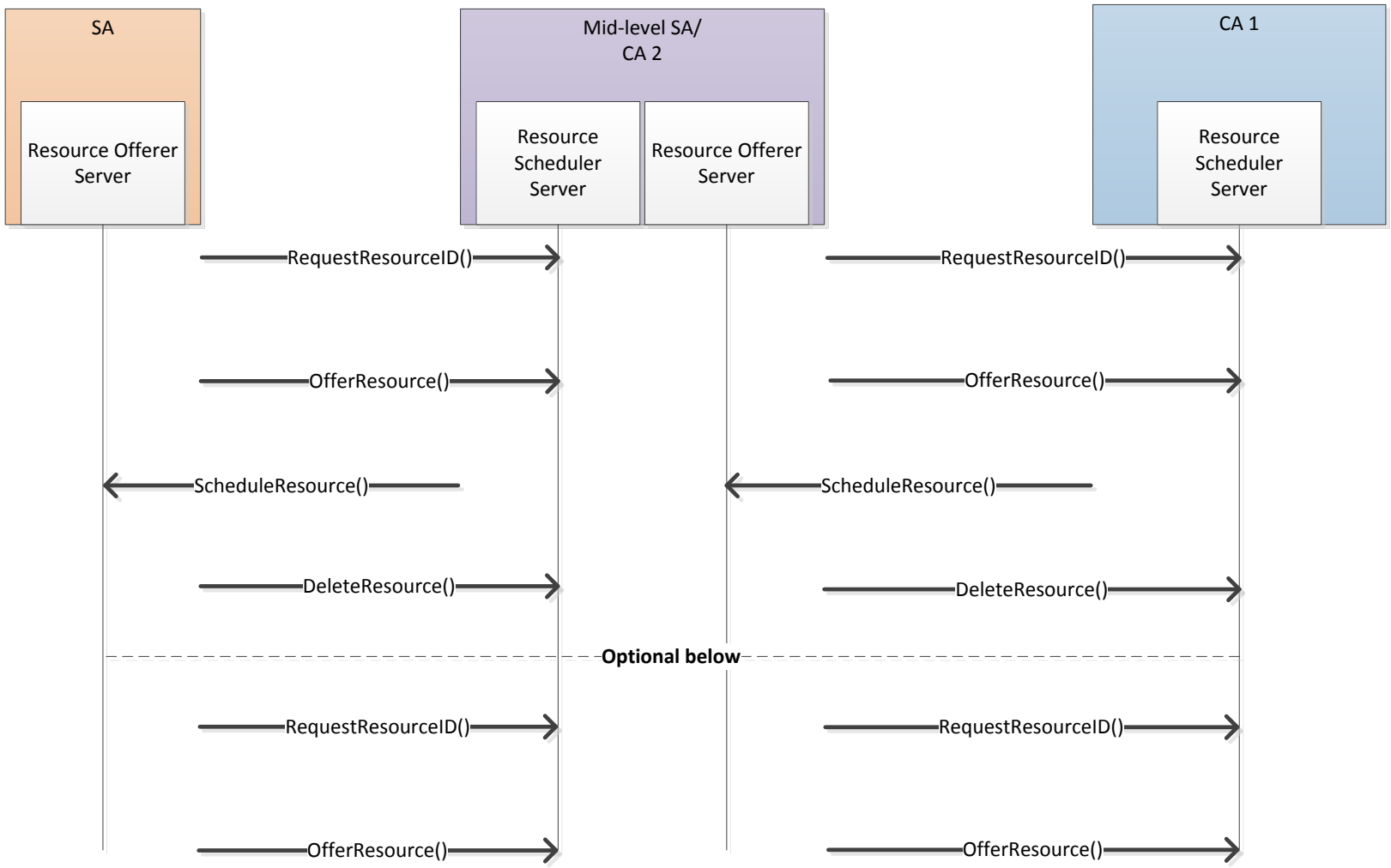


Figure 26. CONOPS with M-PIPE Service Method Calls

c. Retaining Resource Configuration Information

The resource configuration options communicated during scheduling from the CA to the user must have been retained by the user so that they know the bounds that they must stay within to control hardware. The SA will be aware of these bounds as they have created them, and are likely based on real limits of the hardware. Thus, the SA can throw an exception if a hardware control request is received with out-of-bounds parameters, returning the exception including the violated bounds. With an HTML schedule, this will require some work on the part of the implementer for the first implementation of M-PIPE. The future work section of this thesis discusses creating a Thrift interface for the scheduling between the user and the CA to simplify the storage of these resource configuration options.

3. Design Decisions

A number of areas required critical design decision making in the creation of the interface. The abstraction of resources included in the Improved FGN model requires careful description to communicate the options available. Thus, a complete and extensible resource description was created that left no option undescribed that could be initially thought of. Undoubtedly, some options will not have been captured by the initial interface, thus an interface with high capability to evolve was chosen in selecting Thrift. Also, extensive use of optional requiredness was employed in defining the fields of M-PIPE structures easing the evolution process and planning ahead for future improvements and modifications to the standard.

A decision was made to branch from the abstraction of GSML in terms of hardware control where GSML allows for analog I/O and power controllers. It was decided that power control should be part of the physical component to which power is being switched, even if the control is implemented through a separate physical power controller. This reduces the complexity and amount of knowledge an operator needs to have of the ground station setup in order to power on a

preamp for example. With a separate power controller interface, the user must know that switching power control port one, for example, may switch the radio power, while port two switches the preamp power. Thus, power control was made an option for each component to which power control might be needed. As for analog I/O, it was decided that if any future hardware that requires a separate interface in order to describe it must be integrated, a new service should be created rather than tying it into a generalized analog I/O interface. The motivation for this is to put the burden of control on the ground station rather than the operator by reducing the ground station-specific ways the interfaces are used.

A decision also had to be made in terms of communicating ground station configuration details during a resource usage period. While the user is connected to the SA, they have a direct line where the user could be able to request the configuration options from the SA. This was decided to be non-ideal as this allowed for two pathways to receive configuration details. Duplicating information pathways can be a source for error and branching of behavior where different implementers use the two pathways differently. Instead it was decided that the configuration details would instead need to be stored by the user from the time the resource is reserved. If the user requests hardware control that is out of bounds, they will instead receive an exception. This exception will contain the bounds and in this way, even if configuration is improperly stored by the user in between reservation and usage time, the user's software can still correct and update its boundaries programmatically.

C. PYTHON SAMPLE IMPLEMENTATION

A full implementation of the M-PIPE interfaces present in the current version was created for each player in the network using the Python programming language. Previous examples shown in sub-section A are simplified portions of this Python implementation. This code was written from both client and server perspectives for each interface. Both successful use of the interface, as well as exceptional circumstances such as incorrect antenna

pointing directions, were tested. The test code can be found in most of the Python appendix files. Scheduling functions were also tested by making an SA pass resources up to a CA and then requesting them with a user. Hardware drivers were not added to implementations. Drivers will be discussed in the section on future work.

The code could be incorporated into existing ground station software to provide access to the interfaces and its functions called from existing locations in software where hardware control is present. The code could also be extended to perform as standalone ground station software. From a ground station owner's point of view, this code could be used to extend access to their ground station by implementing drivers for their hardware and adding a layer that offers up resources based upon actual scheduling. From a mission operator's point of view, this code could be used to extend ground station hardware control to their implementation of their satellite's command and control interface.

D. M-PIPE API DOCUMENTATION

The M-PIPE Thrift IDL files are coded with embedded documentation known as docstrings. These lines in the code are consumed by the Thrift compiler when run with the following command:

```
thrift -r --gen html mpipe.thrift
```

This will cause the Thrift compiler to read in the docstrings and compile structured HTML pages that serve together with this thesis document as a full API. This documentation can be generated by each user of the Thrift interface or it can be hosted at a central location for each network describing the current state and version of the interface. An example of the Antenna interface's API on its enumerations and structs is shown in Figure 27 as viewed in a browser. A further example of the service portion of the Antenna interface's API on its services as viewed in a browser is shown in Figure 28.

Thrift module: antenna

Module	Services	Data types	Constants
antenna	AntControl • AntBrake • AntConfigure • AntGetStatus • AntPoint	AntConfig AntDirection AntStatus BadConfigException NoBrakeException PointingException Polarization	

Enumerations

Enumeration: Polarization

An enum describing the possible polarization configurations. These may or may not be configurable for a ground station.

`Vertical`	1	Linear vertical
`Horizontal`	2	Linear horizontal
`LHC`	3	Left-handed Circular
`RHC`	4	Right-handed Circular

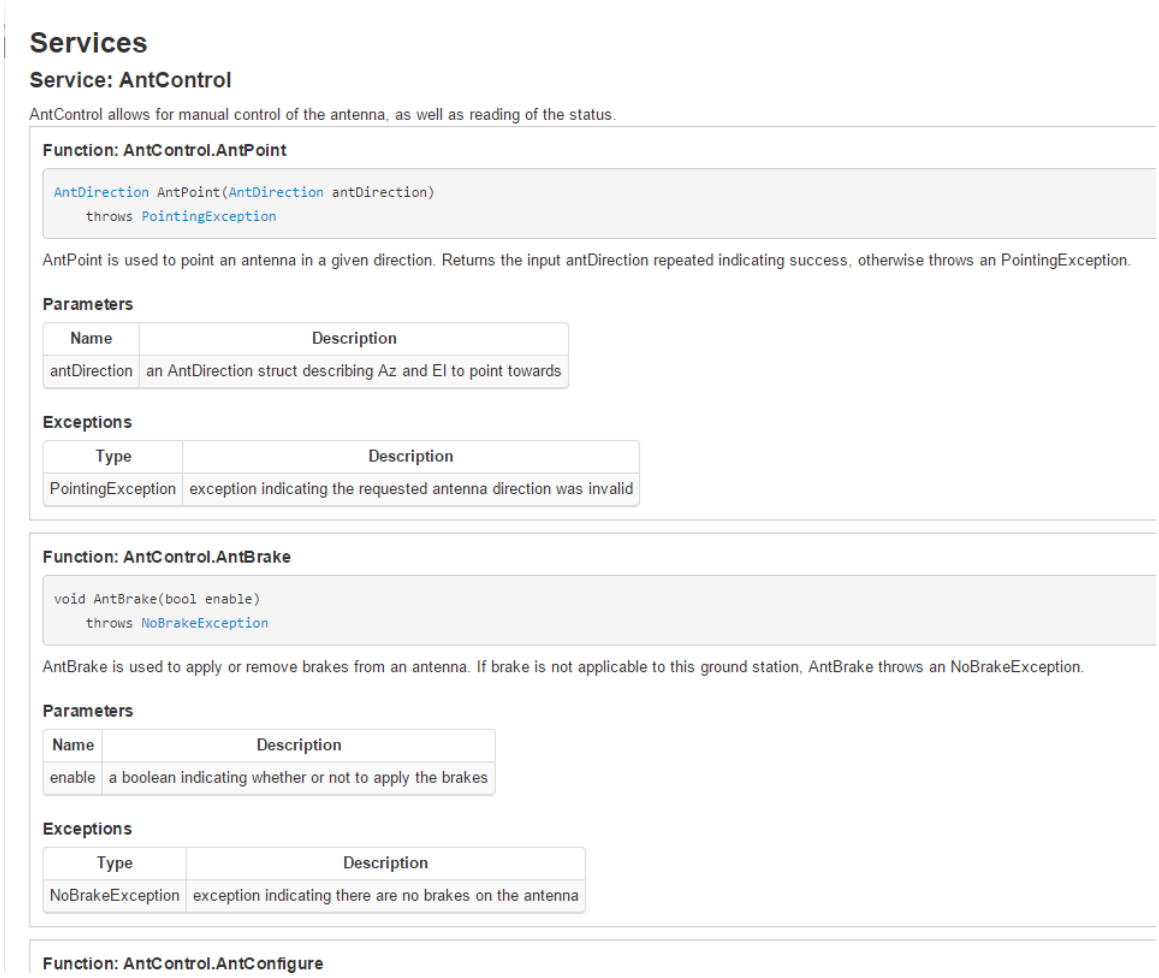
Data structures

Struct: AntDirection

Key	Field	Type	Description	Requiredness	Default value
1	azimuth	`mpipe_types.Degrees`	Azimuth in degrees	default	0
2	elevation	`mpipe_types.Degrees`	Elevation in degrees	default	90

AntDirection describes pointing coordinates for an antenna using azimuth and elevation. Defaults to "bird bath" coordinates, pointing straight up.

Figure 27. M-PIPE Antenna Enums and Structs API Example



Services

Service: AntControl

AntControl allows for manual control of the antenna, as well as reading of the status.

Function: AntControl.AntPoint

```
AntDirection AntPoint(AntDirection antDirection)
    throws PointingException
```

AntPoint is used to point an antenna in a given direction. Returns the input antDirection repeated indicating success, otherwise throws an PointingException.

Parameters

Name	Description
antDirection	an AntDirection struct describing Az and El to point towards

Exceptions

Type	Description
PointingException	exception indicating the requested antenna direction was invalid

Function: AntControl.AntBrake

```
void AntBrake(bool enable)
    throws NoBrakeException
```

AntBrake is used to apply or remove brakes from an antenna. If brake is not applicable to this ground station, AntBrake throws an NoBrakeException.

Parameters

Name	Description
enable	a boolean indicating whether or not to apply the brakes

Exceptions

Type	Description
NoBrakeException	exception indicating there are no brakes on the antenna

Function: AntControl.AntConfigure

Figure 28. M-PIPE Antenna Services API Example

VI. CONCLUSION AND FUTURE WORK

A. CONCLUSION

This thesis described a method of defining roles and hierarchy in federated ground station networks as well as a standardized interface to be created that extends the functions of satellite ground stations. By defining the roles inherent in operating an FGN and structuring these roles into layers, scalability and conglomeration of networks is simplified. Said standard will allow for increased utilization of ground station resources by satellite mission operators.

Analysis showed existing standardization to either be incomplete or overly complex for a typical CubeSat satellite team to utilize with their existing ground station software. Thus, an interface was created to tie together the following three functions: the ability for a user to interact with a ground station, for a user to schedule with a central interface, and for communication between the different ground station nodes in a network with the central interface. Apache Thrift was selected as the best technology for creating interfaces, allowing for greatly simplified implementations of hardware control interfaces into existing ground station software, increasing the appeal of the interface and decreasing the burden on ground station software implementers to include such an interface in their existing software. Lowering the burden of software development necessary to integrate with ground station networks will entice program managers to utilize existing ground station networks rather than duplicate efforts and build one-use ground stations or ground station software. Sample implementations of interfaces from M-PIPE were demonstrated in the Python programming language. These implementations tested each interface presented by the M-PIPE standard and were shown to be functional.

B. FUTURE WORK

Listed below are areas of possible improvements of the M-PIPE interface and suggestions for how to implement them.

1. SSL

The current Python reference implementation of M-PIPE does not use SSL sockets. This is in part because the Thrift Python SSL library is missing server-side certificate parameters and only allows the client to check the validity of certificates. The Thrift Python SSL library could easily be modified to allow for SSL validation of the client's certificate at the server and preliminary efforts have been made to add this functionality but are incomplete at the time of this writing.

2. Online Certificate Status Protocol

PKI certificates can be revoked or suspended by their certifying CAs. This can happen due to a compromise of the user's or CA's private key. The most common method for communicating these revocations and suspensions in the past has been with certificate revocation lists. These are slowly being replaced with the Online Certificate Status Protocol (OCSP), which allows for automated checking of the current status of a certificate. The CA could be set up to be an OCSP responder and network components could check the validity of the certificate before accepting it.

3. Scheduling CA interface

The current M-PIPE standard does not specify a scheduling interface between the user and the CA. This was a design decision, as HTML is likely the best way for a human to view this information. HTML, though, does not provide a good means of automation between the user and the CA, and thus it may be best to have two interfaces between the user and the CA: one in HTML that humans can view, and another in Thrift that software can interact with to make resource reservations and view availability in existing ground station software. Another benefit that a Thrift scheduling interface could provide is in the ability to save

ground station configuration options between when the program makes a reservation to when it operates during a resource usage. Currently, this will have to be handled with another mechanism such as manual entry into a configuration file.

There is currently no sample HTML-driven CA interface and this should be developed. This interface would display the resource information that the M-PIPE Scheduler interface communicated. This would require a web server and to integrate the sample implementation in Python to drive the backend data.

4. Driver implementations

The sample implementation currently does not exercise any hardware. Drivers will be added to control the hardware at the Naval Postgraduate School's MC3 ground station to demonstrate the functionality of the M-PIPE system. Also, the NPS expects to offer the functionality its ground stations provide to other government and government-sponsored users and will use M-PIPE to do so. To best leverage existing U.S. Government investment in ground station software, an M-PIPE interface should be integrated with the Neptune™ software on the MC3 network.

5. Turbo and LDPC Codes

Turbo and Low-Density Parity-Check (LDPC) codes are complex but efficient methods of performing forward-error correction (FEC). They currently do not have parameters to enable their configuration in the radio.thrift file. This functionality could be added by parameterizing both fields. This would enable the remote configuration of custom Turbo and LDPC codes in compatible receivers.

C. SUMMARY

This thesis solved the lack of a simple yet powerful interface which can connect remote users to ground station networks. A model was created which provides:

- Hierarchy and role definitions
- Scheduling
- Security

Utilizing this model, an interface called M-PIPE was created which provides:

- Remote hardware control
    - Automatic and manual tracking
- Scheduling
    - Resource sharing and reservations
    - Synchronization

Finally, reference implementations of the interface, which can be found in the appendix, were demonstrated. These will provide a starting point for future implementers of the M-PIPE standard. This standard might lead to increased usage of federated ground station networks and help to overcome the complexity of integrating varying software packages for ground station and satellite control.

APPENDIX. M-PIPE THRIFT IDL FILES

A. MPIPE.THRIFT

```
/**
 * M-PIPE interface: MC3 Picosatellite Interface Pipeline Extension
 * Author: Aaron Felt, Naval Postgraduate School
 */
namespace * mpipe

include “antenna.thrift”
include “preamp.thrift”
include “amp.thrift”
include “radio.thrift”
include “cpu.thrift”
include “session.thrift”
include “packet.thrift”
include “scheduler.thrift”

/** The version of the M-PIPE Interface */
const string VERSION = “0.1.0”
```

B. MPIPE_TYPES.THRIFT

```
/** Azimuth/Elevation as a float in degrees */
typedef double Degrees
/** Slew Rate as a float in degrees per second */
typedef double SlewRate
/** Decibels as a double */
typedef double Decibels
/** Frequency in MHz as a double */
typedef double Frequency
/** Percentage */
typedef double Percent
/** Time in Unix time */
typedef i64 Time
/** A unique 32-bit integer identifying a resource network-wide */
typedef i32 ResourceID
```

C. MPIPE_GLOBALS.THRIFT

```
/**
 * M-PIPE interface: MC3 Picosatellite Interface Pipeline Extension
 * Author: Aaron Felt, Naval Postgraduate School
 *
 * TODO A radio expert could go through these to see what needs to be added or
 * collapsed
 */
namespace * mpipe_globals


/** Radio Modulation Mode */
enum Mode {
       /** Frequency Modulation */
       FM     = 1
       /** Lower Side Band */
       LSB    = 2
       /** Upper Side Band */
       USB    = 3
       /** Continuous Wave */
       CW     = 4
       /** Amplitude Modulation */
       AM     = 5
       /** Phase Modulation */
       PM     = 6
       /** Quadrature Modulation */
       QM     = 7
}

/** Modulation type */
enum Modulation {
       /** Audio Frequency-Shift Keying */
       AFSK   = 1
       /** Minimum-Shift Keying */
       MSK    = 2
       /** Gaussian Minimum-Shift Keying */
       GMSK   = 3
       /** Binary Phase-Shift Keying */
       BPSK   = 4
       /** Quadrature Phase-Shift Keying */
       QPSK   = 5
       /** Offset Quadrature Phase-Shift Keying */
       OQPSK  = 6
}

/** Physical level bit encoding */
enum Encoding {
       /** Non-return-to-zero, level */
       NRZL   = 1
       /** Non-return-to-zero, inverted */
       NRZI   = 2
       /** Non-return-to-zero, space */
```

```
        NRZS   = 3
        /** Return-to-zero */
        RZ     = 4
        /** Return-to-zero, inverted */
        RZI    = 5
        /** Manchester Encoding / Phase Encoding */
        ME     = 6
        /** Differential Manchester Encoding */
        DME    = 7
        /** Biphase Mark Coding */
        BMC    = 8
        /** Biphase Space Coding */
        BSC    = 9
        /** Biphase Level Coding */
        BLC    = 10
        /** Bipolar */
        BP     = 11
}

/** RSConfig describes parameters for a Reed-Solomon code */
struct RSConfig {
        /** Block length */
        1: i32 n
        /** Message length */
        2: i32 k
}

/** Data Link Layer protocols */
enum LinkLayerProt {
        /** High-Level Data Link Control protocol */
        HDLC          = 1
        /** CCSDS Telemetry Space Data Link protocol */
        CCSDS_TM      = 2
        /** CCSDS Telecommand Space Data Link protocol */
        CCSDS_TC      = 3
        /** CCSDS Advanced Orbiting System Data Link protocol */
        CCSDS_AOS     = 4
}
```

D. AMP.THRIFT

```
/**
 * M-PIPE interface: MC3 Picosatellite Interface Pipeline Extension
 * Author: Aaron Felt, Naval Postgraduate School
 */
namespace * mpipe_amp

include “mpipe_types.thrift”

/**
 * AmpStatus describes the current status of the amplifier, returning an error
 * code if health is degraded.
 */
struct AmpStatus {
       /** Current amp gain setting */
       1: mpipe_types.Decibels currGain
       /** Current power status of amp */
       2: bool currPowerState
       /** True for healthy, False for degraded */
       3: bool healthy
       /** A code indicating degraded status cause */
       4: optional i16 degradedCode
}

/**
 * GainException is thrown when an invalid gain is requested.
 */
exception GainException {
       /** the invalid requested gain setting*/
       1: mpipe_types.Decibels dB
       /** an optional field indicating the minimum gain the amp supports */
       2: optional mpipe_types.Decibels minGain
       /** an optional field indicating the maximum gain the amp supports */
       3: optional mpipe_types.Decibels maxGain
}

/** AmpControl allows for manual control of amplifier status. */
service AmpControl {
       /**
        * AmpEnable is used to power-on/enable or power-off/disable the amp.
        * Returns the new power status after making change to power.
        */
       bool AmpEnable(
              /** True will power-on, False will power-off */
              1: bool isPowered
       )

       /**
        * AmpSetGain is used to set the amp gain. If a valid gain is provided,
        * the value will be set and returned. If an invalid gain is provided, a
        * GainException will be thrown.
        */
```

```
        mpipe_types.Decibels AmpSetGain(
               /**
                * Specifies setting the gain to the given value
                */
               1: mpipe_types.Decibels dB
        )
        /** exception indicating the requested gain was invalid */
        throws (1: GainException gainException)

        /**
         * AmpGetStatus is used to request the health of the amp that is
         * returned as a AmpStatus struct.
         */
        AmpStatus AmpGetStatus()
}
```

E. ANTENNA.THRIFT

```
/**
 * M-PIPE interface: MC3 Picosatellite Interface Pipeline Extension
 * Author: Aaron Felt, Naval Postgraduate School
 */
namespace * mpipe_antenna

include “mpipe_types.thrift”

/**
 * An enum describing the possible polarization configurations. These may or
 * may not be configurable for a ground station.
 */
enum Polarization {
       /** Linear vertical */
       Vertical     = 1
       /** Linear horizontal */
       Horizontal   = 2
       /** Left-handed Circular */
       LHC          = 3
       /** Right-handed Circular */
       RHC          = 4
}

/**
 * AntDirection describes pointing coordinates for an antenna using azimuth
 * and elevation. Defaults to “bird bath” coordinates, pointing straight up.
 */
struct AntDirection {
       /** Azimuth in degrees */
       1: mpipe_types.Degrees azimuth = 0.0
       /** Elevation in degrees */
       2: mpipe_types.Degrees elevation = 90.0,
}

/** AntConfig describes configuration parameters for an antenna controller */
struct AntConfig {
       /** Azimuth slew rate in degrees per second */
       1: mpipe_types.SlewRate azSlewRate
       /** Elevation slew rate in degrees per second */
       2: mpipe_types.SlewRate elSlewRate
       /** Polarization type */
       3: Polarization polarization
}

/**
 * AntStatus describes the current status of the antenna hardware, returning
 * an error code if health is degraded.
 */
struct AntStatus {
       /** Current antenna direction (Az, El) */
       1: AntDirection currDirection
```

```
	/** True for healthy, False for degraded */
	2: bool healthy
	/** A code indicating degraded status cause */
	3: optional i16 degradedCode
}

/**
 * PointingException is thrown when an invalid antenna direction is requested.
 */
exception PointingException {
	/** an AntDirection struct which was rejected as invalid */
	1: AntDirection badDirection
	/**
	 * an optional field indicating the minimum azimuth the antenna supports
	 */
	2: optional mpipe_types.Degrees minAz
	/**
	 * an optional field indicating the maximum azimuth the antenna supports
	 */
	3: optional mpipe_types.Degrees maxAz
	/**
	 * an optional field indicating the minimum elevation the antenna
	 * supports
	 */
	4: optional mpipe_types.Degrees minEl
	/**
	 * an optional field indicating the maximum elevation the antenna
	 * supports
	 */
	5: optional mpipe_types.Degrees maxEl
}

/**
 * NoBrakeException is thrown when there are no brakes on an antenna but a
 * client uses the AntBrake() function.
 */
exception NoBrakeException {}

/**
 * BadConfigException is thrown when an invalid configuration is requested.
 */
exception BadConfigException {
	/** an AntConfig struct that caused the exception*/
	1: AntConfig antConfig
	/** an optional field indicating the requested azSlewRate was too low */
	2: optional mpipe_types.SlewRate minAzSlewRate
	/**
	 * an optional field indicating the requested azSlewRate was too high
	 */
	3: optional mpipe_types.SlewRate maxAzSlewRate
	/** an optional field indicating the requested elSlewRate was too low */
	4: optional mpipe_types.SlewRate minElSlewRate
	/**
	 * an optional field indicating the requested elSlewRate was too high
```

```
        */
       5: optional mpipe_types.SlewRate maxElSlewRate
       /** an optional field indicating the requested polarization isn't
        * supported
        */
       6: optional Polarization polarization
}

/**
 * AntControl allows for manual control of the antenna, as well as reading of
 * the status.
 */
service AntControl {
       /**
        * AntPoint is used to point an antenna in a given direction. Returns
        * the input antDirection repeated indicating success, otherwise throws
        * a PointingException.
        */
       AntDirection AntPoint(
              /**
               * an AntDirection struct describing Az and El to point towards
               */
              1: AntDirection antDirection
       )
       /**
        * exception indicating the requested antenna direction was invalid
        */
       throws (1: PointingException pointingException)

       /**
        * AntBrake is used to apply or remove brakes from an antenna. If brake
        * is not applicable to this ground station, AntBrake throws a
        * NoBrakeException.
        */
       void AntBrake(
              /** a boolean indicating whether or not to apply the brakes */
              1: bool enable
       )
       /** exception indicating there are no brakes on the antenna */
       throws (1: NoBrakeException noBrakeException)

       /**
        * AntConfigure is used to configure parameters of the antenna. Returns
        * new configuration of the antenna. If requested configuration is not
        * applicable to this ground station, AntConfigure throws a
        * BadConfigException and does not make any updates to the
        * configuration.
        */
       AntConfig AntConfigure(
              /**
               * an AntConfig struct describing the parameters to set the
               * antenna to.
               */
              1: AntConfig antConfig
```

```
        )
        /** exception indicating the requested configuration is invalid */
        throws (1: BadConfigException badConfigException)

        /**
         * AntGetStatus is used to get the current status of the antenna.
         * Returns an AntStatus struct describing the health of the antenna, as
         * well as current pointing info.
         */
        AntStatus AntGetStatus()
}
```

F. CPU.THRIFT

```
/**
 *     M-PIPE interface: MC3 Picosatellite Interface Pipeline Extension
 *     Author: Aaron Felt, Naval Postgraduate School
 */
namespace * mpipe_cpu

include “mpipe_types.thrift”

/** CPUStatus describes the current status of the server’s computer */
struct CPUStatus {
       /** CPU health status. True means healthy, False indicates degraded */
       1: bool status
       /** Time in seconds since computer booted */
       2: optional i64 uptime
       /**Number of processes total */
       3: optional i64 numProcessesTotal
       /** Number of processes running */
       4: optional i64 processesRunning
       /** Average CPU load as a percent */
       5: optional mpipe_types.Percent loadAve
       /** Disk usage as a percent */
       6: optional mpipe_types.Percent diskUsage
       /** Disk free in MB */
       7: optional i64 diskFree
       /** RAM usage as a percent */
       8: optional mpipe_types.Percent ramUsage
       /** RAM free in MB */
       9: optional i64 ramFree
}

/** NetStatus describes the current status of the network */
struct NetStatus {
       /** Bytes received on the network socket */
       1: optional i64 bytesIn
       /** Bytes transmitted on the network socket */
       2: optional i64 bytesOut
}

/** CPUControl allows the client to check on the server’s computer status. */
service CPUControl {
       /**
        * GetCPUStatus is used to check on the operating system-level server
        * status.
        */
       CPUStatus GetCPUStatus()

       /**
        * GetNetStatus is used to check on the network-level server status.
        */
       NetStatus GetNetStatus()
```

```
}
```

G. PACKET.THRIFT

```
/**
 * M-PIPE interface: MC3 Picosatellite Interface Pipeline Extension
 * Author: Aaron Felt, Naval Postgraduate School
 */
namespace * mpipe_packet

include “mpipe_types.thrift”

/**
 * Packet is a container for an uplink or downlink packet containing the data
 * and an integer identifier
 */
struct Packet {
	/**
	 * A packet counter to uniquely identify and order packets. Receiving a
	 * packet with identical packetIDs has undefined meaning but should
	 * not be created on purpose by a server.
	 */
	1: i32 packetID
	/** Packet of data */
	2: string data
}

/**
 * PacketSizeException is thrown when a packet too large for the system is
 * received to be uplinked
 */
exception PacketSizeException {
	/** The packet that was too large to be uplinked */
	1: Packet badPacket
	/** The maximum acceptable packet size */
	2: i32 maxPacketSize
}

/** CPUControl allows the client to check on the server’s computer status. */
service PacketService {
	/**
	 * SendPacket is used to send packets between players. It can be used
	 * for uplink or downlink packets.
	 */
	void SendPacket(
		/** the packet to be sent */
		1: Packet packet
	)
	/**
	 * Indicates a packet too large for the system has been requested to be
	 * uplinked.
	 * NOTE: This is meaningless with downlink packets.
	 */
	throws (1: PacketSizeException packetSizeException)
}
```

H. PREAMP.THRIFT

```
/**
 * M-PIPE interface: MC3 Picosatellite Interface Pipeline Extension
 * Author: Aaron Felt, Naval Postgraduate School
 */
namespace * mpipe_preamp

include “mpipe_types.thrift”

/**
 * PAStatus describes the current status of the preamp hardware, returning an
error
 * code if health is degraded.
 */
struct PAStatus {
	/** Current preamp gain setting */
	1: mpipe_types.Decibels currGain
	/** Current power status of preamp */
	2: bool currPowerState
	/** True for healthy, False for degraded */
	3: bool healthy
	/** A code indicating degraded status cause */
	4: optional i16 degradedCode
}

/**
 * GainException is thrown when an invalid gain is requested.
 */
exception GainException {
	/** the invalid requested gain setting */
	1: mpipe_types.Decibels dB
	/** an optional field indicating the minimum gain the preamp supports */
	2: optional mpipe_types.Decibels minGain
	/** an optional field indicating the maximum gain the preamp supports */
	3: optional mpipe_types.Decibels maxGain
}

/** PAControl allows for manual control of preamp status. */
service PAControl {
	/**
	 * PAEnable is used to power-on/enable or power-off/disable the preamp.
 * Returns the new power status after making change to power.
	 */
	bool PAEnable(
		/** True will power-on, False will power-off */
		1: bool isPowered
	)

	/**
	 * PASetGain is used to set the preamp gain. If a valid gain is
	 * provided, the value will be set and returned. If an invalid gain is
	 * provided, a GainException will be thrown.
```

```
        */
       mpipe_types.Decibels PASetGain(
              /**
               * Specifies setting the gain to the given value
               */
              1: mpipe_types.Decibels dB
       )
       /** exception indicating the requested gain was invalid */
       throws (1: GainException gainException)

/**
 * PAGetStatus is used to request the health of the preamp that is returned
 * as a PAStatus struct.
        */
       PAStatus PAGetStatus()
}
```

I. RADIO.THRIFT

```
/**
 * M-PIPE interface: MC3 Picosatellite Interface Pipeline Extension
 * Author: Aaron Felt, Naval Postgraduate School
 *
 * Description: Radio here encompasses both transmit and receive radios. As
 * explained in the supporting documentation, a radio, though it may
 * physically be capable of both transmit and receive, is  treated in this
 * architecture as two different radios following the Improved FGN model.
 */
namespace * mpipe_radio

include “mpipe_types.thrift”
include “mpipe_globals.thrift”

/** A list of error codes and their values */
enum ConfigError {
       /**
        * Indicates that the ground station does not support the requested
        * value
        */
       Unsupported         = 1
       /**
        * Indicates a conflict between two requested field-value pairs that
        * cannot co-exist
        */
       IncompatibleChoices = 2
       /** Value out of supported range for a given field */
       OutOfSupportedRange = 3
}

/** RConfig describes settings for the radio parameters */
struct RConfig {
       /** Frequency in MHz */
       1: optional mpipe_types.Frequency freq
       /** Radio mode from the Mode enum */
       2: optional mpipe_globals.Mode mode
       /** Percent of power level for transmit (0%-100%) */
       3: optional mpipe_types.Percent powerLevel
       /** Percent of attenuation level for receive (0%-100%) */
       4: optional mpipe_types.Percent attenuation
       /** Percent of squelch level for receive (0%-100%) */
       5: optional mpipe_types.Percent squelch
       /** Loop bandwidth for Phase Locked Loop (PLL) carrier synchronizer */
       6: optional double freqLBW
       /** Loop bandwidth for Phase Locked Loop (PLL) bit synchronizer */
       7: optional double modLBW
       /** Percent of Automatic Gain Control level (0%-100%) */
       8: optional mpipe_types.Percent agcLevel
       /** Encoding style */
       9: optional mpipe_globals.Encoding encoding
       /** Pseudorandom bit randomization */
```

```
        10: optional i64 prnSequence
        /** Differential encoding enable/disable */
        11: optional bool useDiffEncode
        /** Viterbi encoding enable */
        12: optional bool useViterbi
        /**
         * Viterbi rate specifier. Meaningless if useViterbi is not True.
         * Rate should be specified as a ratio such as “1/2” or “3/4”
         */
        13: optional string viterbiRate
        /** Reed-Solomon coding enable */
        14: optional bool useReedSolomon
        /** Reed-Solomon parameters */
        15: optional mpipe_globals.RSConfig rsParams
        /** Turbo Code enable */
        16: optional bool useTurboCode
        // TODO add a struct describing Turbo codes as a parameter
        // TODO add LDPC code
        /** Data Link Layer protocol */
        17: optional mpipe_globals.LinkLayerProt llProt
        /** Modulation type */
        18: optional mpipe_globals.Modulation modulation
        /** Baud rate (symbols per second) */
        19: optional i32 baud
        /** Filter bandwidth in MHz */
        20: optional mpipe_types.Frequency filterBW
        /** IF (Intermediate Frequency) Bandwidth */
        21: optional mpipe_types.Frequency ifBW
        /**
         * In systems with scanning PLLs for acquisition of carrier, this
         * variable defines the range in MHz to box around center frequency. If
         * center frequency is 915MHz and acqRange is 1MHz, this will scan from
         * 914.5 to 915.5 MHz for the carrier.
         */
        22: optional mpipe_types.Frequency acqRange
}

/** RReceiveInfo contains RSSI and SNR from a receive radio */
struct RReceiveInfo {
        /**
         * Received Signal Strength Indicator (RSSI) in dB if provided by radio
         */
        1: optional mpipe_types.Decibels rssi
        /** Signal-to-Noise Ratio (SNR) as a float if provided by radio */
        2: optional double snr
}

/**
 * ConfigException is thrown when an invalid config is requested.
 */
exception ConfigException {
        /** The configuration request being rejected */
        1: RConfig badConfig
        /** A map linking the field(s) to the error related to the field(s) */
```

```
	2: map<i16, ConfigError> fieldErrors
}

/** ProgramException is thrown when an SDR program cannot be started. */
exception ProgramException {
	/** the name of the program attempting to be started */
	1: string name
	/** the error message from attempting to start the SDR program */
	2: string errorMessage
}

/**
 * ParamException is thrown if errors are detected with parameters sent from
 * the client. Errors may be due to an unacceptable parameter string due to
 * use of restricted symbols or keywords, or may be due to parameters
 * recognized as missing by the server handler.
 */
exception ParamException {
	/** the client's parameters input to be started with the SDR program */
	1: string params
	/**
	 * the error message the server handler would like to return to the
	 * client indicating failure
	 */
	2: string errorMessage
}

/**
 * RControl allows for manual control of the radio, as well as reading of its
 * status.
 */
service RControl {

	/**
	 * RadioConfigure is used to configure a radio. Returns the input
	 * RConfig repeated indicating success, otherwise throws a
	 * ConfigException.
	 */
	RConfig RadioConfigure(
		/** an RConfig struct describing settings for the radio */
		1: RConfig config
	)
	/** exception indicating the requested antenna direction was invalid */
	throws (1: ConfigException configException)

	/**
	 * RSDRSelect is used to start and configure an SDR program, which must
	 * already exist at the ground station system. The transfer of the SDR
	 * program is beyond the scope of this interface. Returns True on
	 * success, otherwise returns an exception.
	 */
	void RSDRSelect(
		/** the filename of an SDR program */
		1: string name
```

```
            /**
             * parameters to be passed when calling the SDR program
             *
             * NOTE: Implementers of this parameter are advised to thoroughly
             * inspect parameters for malicious behavior.
             */
            2: string params
        )
        /** exception indicating the requested antenna direction was invalid */
        throws (
            /**
             * Thrown when an error occurs starting a program such as
             * inability to find the program, or inability to start program
             * due to bad parameters. Detection of bad parameters is an
             * optional feature and is not required, but may be provided for
             * a user.
             */
            1: ProgramException progException
            2: ParamException paramException
        )

        /**
         * Returns the information about received signal strength if available.
         * If a value is not accessible to the ground station, it will not be
         * included. This function only makes sense to call on a receiver radio.
         * Values are meaningless on a transmitter and the service handler may
         * return an empty RReceiveInfo.
         */
        RReceiveInfo RGetReceiveInfo()
}
```

J. SESSION.THRIFT

```
/**
 * M-PIPE interface: MC3 Picosatellite Interface Pipeline Extension
 * Author: Aaron Felt, Naval Postgraduate School
 */
namespace * mpipe_session

include “mpipe_types.thrift”

/** A string representing the Two-Line Element ephemeris for an object */
typedef string TLE

/**
 * SessionInfo describes information regarding the current session such as
 * satellite being tracked, as well as ground station location
 */
struct SessionInfo {
       /**
        * The Satellite Catalog Number for the current session, for example
        * 39400 (1- 5 digits typically)
        */
       1: optional i32 CatalogNumber
       /** Ground station latitude */
       2: optional double latitude
       /** Ground station longitude */
       3: optional double longitude
       /** Ground station altitude using WGS84 in meters */
       4: optional double altitude
}

/**
 * SessionInfo describes information regarding the current session such as
 * satellite being tracked, as well as ground station location
 */
struct SessionConfig {
       /** The TLE for the currently tracked satellite */
       1: optional TLE currTLE
       /** Configured for automatic antenna track this session */
       2: bool isAutoAntTrack
       /** Configured for automatic Doppler shifting of radio this session */
       3: bool isAutoDopplerTrack
}

/**
 * ConfigError is an error code indicating what reason the configuration was
 * rejected
 */
enum ConfigError {
       /** Indicates automatic antenna tracking is not available */
       noAutoAntTrack                  = 1
       /** Indicates automatic doppler shfting is not available */
       noAutoDopplerTrack  = 2
```

```
	/** Indicates no TLE was available to make predictions from */
	noTLEForTrack                    = 3
}
/**
 * ConfigException is used to indicate that a configuration was rejected for
 * ConfigError reason
 */
exception ConfigException {
	/** The SessionConfig that triggered the exception */
	1: SessionConfig badConfig
	/** A value from ConfigError indicating the cause of the exception */
	2: optional ConfigError error
}

/** SessionControl allows the client to configure automatic tracking. */
service SessionControl {
	/**
	 * GetSessionInfo is used to check on the operating system-level server
	 * status.
	 */
	SessionInfo GetSessionInfo()

	/**
	 * SetSessionConfig is used to set the session configuration, for
	 * example updating the TLE, or switching to manual control of the
	 * antenna.
	 */
	SessionConfig SetSessionConfig (
		/** The SessionConfig to set the session to */
		1: SessionConfig config
	)
/**
 * Thrown when a configuration is rejected because some form of automatic
 * tracking requested is not allowed.
 */
throws (1: ConfigException configException)
}
```

K. SCHEDULER.THRIFT

```
/**
 * M-PIPE interface: MC3 Picosatellite Interface Pipeline Extension
 * Author: Aaron Felt, Naval Postgraduate School
 */
namespace * mpipe_scheduler

include “mpipe_types.thrift”
include “mpipe_globals.thrift”
include “antenna.thrift”

/** Indicates the directionality of a resource, either transmit or receive */
enum Direction {
	/** Transmit */
	TX = 1
	/** Receive */
	RX = 2
}

/**
 * A structure for describing a single incompatibility of radio options for
 * this resource. Every field is optional so that only fields that are
 * incompatible with each other are included. For example, if BPSK and a baud
 * rate of 1200 are incompatible on this resource, this struct will only be
 * populated with BPSK and 1200. If BPSK and LSB are also incompatible, there
 * will need to be a second RadioRestrictions with this incompatibility. If
 * this struct had BPSK, 1200 baud, and LSB in it, it would imply that the
 * three of those in combination are not compatible but doesn’t restrict BPSK,
 * LSB, 2400 baud.
 */
struct RadioRestrictions {
	/** Mode of operation such as FM, LSB, etc. */
	1: optional mpipe_globals.Mode mode
	/** Radio modulation such as GMSK, BPSK, etc. */
	2: optional mpipe_globals.Modulation modulation
	/**
	 * The bandwidth desired on an SDR. Only this field or
	 * hardwareFrequencies should be populated but not both. This field
	 * shall be used when the resource contains an SDR
	 */
	3: optional mpipe_types.Frequency sdrBandwidth
	/**
	 * The specific hardware bandwidth desired. Only this field or
	 * sdrBandwidthMax should be populated but not both. This field shall be
	 * used when the resource contains a traditional radio as opposed to an
	 * SDR
	 */
	4: optional mpipe_types.Frequency hardwareBandwidth
	/** Desired link layer protocol such as HDLC, etc. */
	5: optional mpipe_globals.LinkLayerProt linkLayerProt
	/** Desired encoding such as NRZ-I, etc. */
	6: optional mpipe_globals.Encoding encoding
```

```
        /** Desired baud rate */
        7: optional i32 baud
        /** Describes whether or not viterbi is desired */
        8: optional bool useViterbi
        /**
         * Describes whether or not PRN randomization/derandomization is desired
         */
        9: optional bool usePRN
        /** Describes whether or not Reed-Solomon en/decoding is desired */
        10: optional bool useRS
        /** Describes whether or not Turbo Coding is desired */
        11: optional bool useTurboCode
}

/** A structure defining the options available on the radio of a resource */
struct RadioOptions {
        /** List of radio modes of operation such as FM, LSB, etc. */
        1: list<mpipe_globals.Mode> mode
        /** List of radio modulations such as GMSK, BPSK, etc. */
        2: list<mpipe_globals.Modulation> modulation
        /**
         * The maximum bandwidth the SDR can support. Only this field or
         * hardwareFrequencies should be populated but not both. This field
         * shall be used when the resource contains an SDR
         */
        3: optional mpipe_types.Frequency sdrBandwidthMax
        /**
         * A list of bandwidths the hardware radio can support. Only this field
         * or sdrBandwidthMax should be populated but not both. This field shall
         * be used when the resource contains a traditional radio as opposed to
         * an SDR
         */
        4: optional list<mpipe_types.Frequency> hardwareBandwidths
        /** List of link layer protocols such as HDLC, etc. */
        5: list<mpipe_globals.LinkLayerProt> linkLayerProt
        /** List of encodings such as NRZ-I, etc. */
        6: list<mpipe_globals.Encoding> encoding
        /** List of baud rates */
        7: list<i32> baud
        /** Describes whether or not viterbi is a supported option */
        8: bool viterbiSupported
        /**
         * Describes whether or not PRN randomization/derandomization is a
         * supported option
         */
        9: bool prnSupported
        /**
         * Describes whether or not Reed-Solomon en/decoding is a supported
         * option
         */
        10: bool rsSupported
        /** Describes whether or not Turbo Coding is supported */
        11: bool turboCodeSupported
}
```

```
/**
 * A structure for defining the choices selected from a RadioOptions struct */
struct RadioSelections {
       /**
        * Center frequency desired. This is mostly important for auto Doppler-
        * shifted resources
        */
       1: mpipe_types.Frequency centerFrequency
       /** Mode of operation such as FM, LSB, etc. */
       2: mpipe_globals.Mode mode
       /** Radio modulation such as GMSK, BPSK, etc. */
       3: mpipe_globals.Modulation modulation
       /**
        * The bandwidth desired on an SDR. Only this field or
        * hardwareFrequencies should be populated but not both. This field
        * shall be used when the resource contains an SDR
        */
       4: optional mpipe_types.Frequency sdrBandwidth
       /**
        * The specific hardware bandwidth desired. Only this field or
        * sdrBandwidthMax should be populated but not both. This field shall be
        * used when the resource contains a traditional radio as opposed to an
        * SDR
        */
       5: optional mpipe_types.Frequency hardwareBandwidth
       /** Desired link layer protocol such as HDLC, etc. */
       6: mpipe_globals.LinkLayerProt linkLayerProt
       /** Desired encoding such as NRZ-I, etc. */
       7: mpipe_globals.Encoding encoding
       /** Desired baud rate */
       8: i32 baud
       /** Describes whether or not viterbi is desired */
       9: bool useViterbi
       /**
        * Describes whether or not PRN randomization/derandomization is desired
        */
       10: bool usePRN
       /** Describes whether or not Reed-Solomon en/decoding is desired */
       11: bool useRS
       /** Describes whether or not Turbo Coding is desired */
       12: bool useTurboCode
}

/**
 * ResourceOptions describes a resource being offered with all radio options
 * included
 */
struct ResourceOptions {
       /**
        * An ID sourced from the top-level CA, originated from a
        * RequestResourceID call uniquely identifying the resource
        */
       1: mpipe_types.ResourceID id
```

```
/** The node-level SA who this resource belongs to */
2: string saName
/** The Unix time that this resource begins being available */
3: mpipe_types.Time start
/** The Unix time that this resource stops being available */
4: mpipe_types.Time stop
/** The maximum number of seconds this resource can be reserved */
5: i32 maxSecReserve
/** Either transmit (TX) or receive (RX) */
6: Direction direction
/** The lowest frequency this resource can provide */
7: mpipe_types.Frequency lowFreq
/** The highest frequency this resource can provide */
8: mpipe_types.Frequency highFreq
/** The manufacturer of the radio*/
9: string radioManufacturer
/** The model of the radio */
10: string radioModel
/** The manufacturer of the antenna control unit */
11: string acuManufacturer
/** The model of the antenna control unit */
12: string acuModel
/** The manufacturer of the pre-amp */
13: optional string paManufacturer
/** The model of the pre-amp */
14: optional string paModel
/** The manufacturer of the amplifier */
15: optional string ampManufacturer
/** The model of the amplifier */
16: optional string ampModel
/** The minimum azimuth slew rate of the antenna */
17: mpipe_types.SlewRate azSlewRateMin
/** The maximum azimuth slew rate of the antenna */
18: mpipe_types.SlewRate azSlewRateMax
/** The minimum elevation slew rate of the antenna */
19: mpipe_types.SlewRate elSlewRateMin
/** The maximum elevation slew rate of the antenna */
20: mpipe_types.SlewRate elSlewRateMax
/** The polarization options available on the antenna  */
21: list<antenna.Polarization> polarizations
/** The gain of the antenna */
22: mpipe_types.Decibels gain
/**
 * True if the antenna can be manually controlled; False if auto-
 * tracking must be used
 */
23: bool antControlEnabled
/** The minimum azimuth the antenna can be directed to */
24: mpipe_types.Degrees minAz
/** The maximum azimuth the antenna can be directed to */
25: mpipe_types.Degrees maxAz
/** The minimum elevation the antenna can be directed to */
26: mpipe_types.Degrees minEl
/** The maximum elevation the antenna can be directed to */
```

```
	27: mpipe_types.Degrees maxEl
	/**
	 * The minimum transmit elevation allowed. This only needs to be
	 * populated for a TX config but should be populated even if auto-
	 * tracking is enabled to inform the user the pass length
	 */
	28: optional mpipe_types.Degrees minElTx
	/**
	 * True if the pre-amp can be manually controlled; False if it will be
	 * controlled
	 */
	29: bool paControlEnabled
	/** The minimum gain in dB the pre-amp can support */
	30: mpipe_types.Decibels paMinGain
	/** The maximum gain in dB the pre-amp can support */
	31: mpipe_types.Decibels paMaxGain
	/**
	 * True if the radio frequency can be manually controlled; False if
	 * automatic  doppler-shifting must be used.
	 */
	32: bool freqControlEnabled
	/** The latitude of the ground station */
	33: optional double latitude
	/** The longitude of the ground station */
	34: optional double longitude
	/** Ground station altitude WGS72 in meters */
	35: optional double altitude
}

/**
 * ResourceConfig describes a resource being requested with all options having
 * been selected
 */
struct ResourceConfig {
	/**
	 * An ID sourced from the top-level CA, originated from a
	 * RequestResourceID call uniquely identifying the resource
	 */
	1: mpipe_types.ResourceID id
	/** The node-level SA who this resource belongs to */
	2: string saName
	/** The Unix time that this resource begins being available */
	3: mpipe_types.Time start
	/** The Unix time that this resource stops being available */
	4: mpipe_types.Time stop
	/** Either transmit (TX) or receive (RX) */
	5: Direction direction
}

/**
 * A struct for communicating a username and certificate in a resource request
 */
struct User {
	/** The username associated with the requesting user */
```

```
        1: string userName
        /**
         * The public certificate associated with the user in X.509 PEM format
         */
        2: string publicCertificate
}

/** Thrown to indicate the resource ID was not recognized */
exception DeleteException {
        1: mpipe_types.ResourceID id
}

/** A list of error codes and their values */
enum ConfigError {
        /**
         * Indicates that the ground station does not support the requested
         * value
         */
        Unsupported         = 1
        /**
         * Indicates a conflict between two requested field-value pairs that
         * cannot co-exist
         */
        IncompatibleChoices = 2
        /** Value out of supported range for a given field */
        OutOfSupportedRange = 3
        /**
         * Special error code for ResourceIDs that are not recognized by the
         * node-level SA
         */
        IDNotRecognized     = 4
}

/** ConfigException is thrown when an invalid resource is passed. */
exception ConfigException {
        /** The resource being rejected */
        1: ResourceConfig badConfig
        /** A map linking the field(s) to the error related to the field(s) */
        2: map<i16, ConfigError> fieldErrors
}

/** A list of error codes and their values */
enum AuthError {
        /** Indicates the user is not authorized to use the resource */
        NotAuthorized             = 1
        /**
         * Indicates the certificate did not match the public certificate the SA
         * expected for the user
         */
        UnrecognizedCertificate   = 2
}

/**
 * AuthException is thrown when user requests a resource but is not authorized
```

```
 * for use of the resource.
 */
exception AuthException {
	/** The authentication error behind the exception */
	1: AuthError error
}

/**
 * ResourceOwnerException is thrown when an SA attempts to add a resource but
 * the resourceID is already registered in the schedule with a  different SA
 */
exception ResourceOwnerException {}

/**
 * BandwidthOptionException is thrown when a struct mentions both an SDR and
 * hardware bandwidth, which is ambiguous as both cannot exist
 */
exception BandwidthOptionException {}

/**
 * ResourceScheduler is served by a CA and used by an SA to offer a resource
 * to a CA as available for scheduling.
 */
service ResourceScheduler {
	/**
	 * RequestResourceID is offered by a CA to allow an SA to request a
	 * unique ID for keying a resource in a network-wide manner. This
	 * message shall be propagated through each mid-level SA until it
	 * reaches the top-level CA, all the way back to the originator which
	 * should be a node-level SA.
	 */
	mpipe_types.ResourceID RequestResourceID()
	/**
	 * OfferResource is used by an SA to offer a resource to a CA as
	 * available for scheduling. If the schedule already has this ResourceID
	 * in use, and the saName that is part of the resource matches the
	 * saName in its current schedule under that key, it will update its
	 * schedule with this new resource. If the saNames do not match, this
	 * throws a ResourceIDInUseException.
	 */
	void OfferResource(
		/** A description of the resource being offered */
		1: ResourceOptions resource
		/** The radio options available in this resource */
		2: RadioOptions radioOptions
		/** The restricted options that subtract from radioOptions */
		3: list<RadioRestrictions> radioRestrictions
	)
	/**
	 * Thrown when an SA attempts to add a resource but is not shown as the
	 * owner of the resource in the schedule
	 */
	throws (1: ResourceOwnerException resourceOwnerException)
	/**
```

```
         * DeleteResource is used by an SA to request a CA delete a previously
         * offered resource. If the resource ID is not recognized an exception
         * will be thrown.
         */
        void DeleteResource(
               /** The ResourceID matching the resource to delete */
               1: mpipe_types.ResourceID id
        )
        throws (
               /**
                * Thrown when the requested resource ID has not been assigned
                * yet
                */
               1: DeleteException deleteException
               /**
                * Thrown when the requested resource ID does not belong to the
                * SA requesting the deletion
                */
               2: ResourceOwnerException resourceOwnerException
        )
}

/**
 * ResourceOfferer is served by an SA and used by a CA to request a resource
 * reservation for a user.
 */
service ResourceOfferer {
        /**
         * ScheduleResource is used by a CA to relay that a user has requested a
         * resource. The ResourceConfig shall be populated with the specific
         * selections of the user, no longer containing multiple options as may
         * have been the case when the resource was offered.
         */
        void ScheduleResource(
               /**
                * Description of the selections made in requesting a resource
                * such as start time, and more.
                */
               1: ResourceConfig config
               /**
                * Desription of the selections made from the options the radio
                * provides this resource
                */
               2: RadioSelections radioSelections
               /** The user requesting the resource */
               3: User user
        )
        throws (
               /** Exception indicating the resource contained an error */
               1: ConfigException configException
               /** Exception indicating an authorization error */
               2: AuthException authException
        )
}
```

L. EVENTTSERVERSOCKET.PY

```
from thrift.transport import TSocket
import socket
from thrift.transport.TTransport import *

# Adds an event field to the TServerSocket
# Event is set when a connection is formed. Can be used to detect incoming
# connections in threading

class EventTServerSocket(TSocket.TServerSocket):
	def __init__(self, event, host=None, port=9090, unix_socket=None):
		self.host = host
		self.port = port
		self._unix_socket = unix_socket
		self.handle = None
		self.event = event

	def accept(self):
		result = TSocket.TServerSocket.accept(self)
		self.event.set()
		return result
```

M. AMP_CLIENT.PY

```
'''
	M-PIPE interface: MC3 Picosatellite Internet Protocol Extension
	Author:
		Aaron Felt, Naval Postgraduate School
	Description:
		This script is meant to exercise, test, and demonstrate the amp
		Thrift Interface as a client and is intended to be used with
		amp_server.py. As it doesn't specify a protocol, it defaults to
		TBinaryProtocol but does use sockets as expected.
'''
import sys
sys.path.append("gen-py")

from thrift import Thrift
from thrift.transport import TSocket, TTransport
from thrift.protocol import TBinaryProtocol
from mpipe_amp import AmpControl
from mpipe import constants
from socket import error as sockError

try:
	socket = TSocket.TSocket("localhost," 8585)
	socket = TTransport.TBufferedTransport(socket)
	socket.open()
	protocol = TBinaryProtocol.TBinaryProtocol(socket)

	client = AmpControl.Client(protocol)

	# Test AmpEnable works
	msg = client.AmpEnable(True)
	print("[Client] Attempted to power on amp. Set to: %r" % msg)
	msg = client.AmpEnable(False)
	print("[Client] Attempted to power off amp. Set to: %r" % msg)
	msg = client.AmpEnable(True)
	print("[Client] Attempted to power on amp. Set to: %r" % msg)

	# Test AmpSetGain works the first time and throws an exception the
	# second and third time
	firstGain = 5.0
	msg = client.AmpSetGain(firstGain)
	print("[Client] Attempted to set gain to %f. Set to: %f"\
		% (firstGain,msg))
	try:
		badGain1 = -1.0
		msg = client.AmpSetGain(badGain1)
	except AmpControl.GainException as ge:
		print("[Client] Attempted to set a bad gain of %f and" + \
			" correctly  received an exception." % badGain1)
		print("[Client] %s" % ge)
	try:
		badGain2 = 10.1
```

```
            msg = client.AmpSetGain(badGain2)
        except AmpControl.GainException as ge:
            print("[Client] Attempted to set a bad gain of %f and" +  \
                  " correctly received an exception." % badGain2)
            print("[Client] %s" % ge)

        # Test AmpGetStatus works the first time and returns the values set
        # above (power-on, gain 5.0)
        msg = client.AmpGetStatus()
        print("[Client] %s" % msg)

except KeyboardInterrupt:
        print("[Shutdown] Close command received. Goodbye!")
except sockError as se:
        print("[Error] Socket was aborted. socket.error %s" % ( se))
except TTransport.TTransportException as tte:
        print("[Error] Server response timed out or data was lost." + \
            " TTransportException(%d): %s" % (tte.type, tte))
except Thrift.TApplicationException as ta:
        print("[Error] There appears to be an interface mismatch. Expected" + \
            " version of interface is %s: %s" % (constants.VERSION, ta))
except Exception as e:
        print("[Error] Exception: %s %s" % (type(e), e))
except:
        print("[Error] BaseException: %s" % sys.exc_info()[0])
```

N. AMP_SERVER.PY

```
'''
	M-PIPE interface: MC3 Picosatellite Internet Protocol Extension
	Author:
		Aaron Felt, Naval Postgraduate School
	Description:
		This script is meant to exercise, test, and demonstrate the amp
		Thrift Interface as a server and is intended to be used with
		amp_client.py. As it doesn't specify a protocol, it defaults to
		TBinaryProtocol but does use sockets as expected.
'''
import sys
sys.path.append('gen-py')
from thrift.transport import TSocket
from thrift.server import TServer
from mpipe_amp import AmpControl

class AmpHandler(AmpControl.Iface):

	def __init__(self, minGain, maxGain, isPowered, gain):
		# Sample class variables for a real amp
		self.minGain = minGain
		self.maxGain = maxGain
		self.isPowered = isPowered
		self.gain = gain

	def AmpEnable(self, isPowered):
		print("[Server] Amp power set to %r" % isPowered)
		self.isPowered = isPowered
		return isPowered

	def AmpSetGain(self, dB):
		if dB >= self.minGain and dB <= self.maxGain:
			self.gain = dB
			print "[Server] Amp gain set to %f" % dB
			return dB
		elif dB < self.minGain:
			print("[Server] Amp gain of %f below min gain of %f." + \
				" Raising GainException" % (dB, self.minGain))
			raise AmpControl.GainException(dB=dB,
							minGain=self.minGain)
		else:
			print("[Server] Amp gain of %f above max gain of %f." + \
				" Raising GainException" % (dB, self.maxGain))
			raise AmpControl.GainException(dB=dB,
							maxGain=self.maxGain)

	def AmpGetStatus(self):
		ampStatus = AmpControl.AmpStatus(currGain=self.gain,
			currPowerState=self.isPowered,
			healthy=True)
		print("[Server] Amp status is %s" % ampStatus)
```

```
            return ampStatus

if __name__ == “__main__”:
      minGain = 0.0
      maxGain = 10.0
      isPowered = False
      gain = 3.0
      svr_trans = TSocket.TServerSocket(port=8585)
      processor = AmpControl.Processor(AmpHandler(minGain,
                                                  maxGain,
                                                  isPowered,
                                                  gain))
      server = TServer.TSimpleServer(processor, svr_trans)
      server.serve()
```

O. ANTENNA_CLIENT.PY

```
'''
	M-PIPE interface: MC3 Picosatellite Internet Protocol Extension
	Author:
		Aaron Felt, Naval Postgraduate School
	Description:
		This script is meant to exercise, test, and demonstrate the
		antenna Thrift interface as a client and is intended to be used
		with antenna_server.py. As it doesn't specify a protocol, it
		defaults to TBinaryProtocol but does use sockets as expected.
'''
import sys
sys.path.append("gen-py")

from thrift import Thrift
from thrift.transport import TSocket, TTransport
from thrift.protocol import TBinaryProtocol
from mpipe_antenna import AntControl
from mpipe_antenna.AntControl import Polarization
from mpipe import constants
from socket import error as sockError

try:
	socket = TSocket.TSocket("localhost," 8585)
	socket = TTransport.TBufferedTransport(socket)
	socket.open()
	protocol = TBinaryProtocol.TBinaryProtocol(socket)

	client = AntControl.Client(protocol)

	# Test AntPoint works the first time and throws an exception the second
	# time
	pointDirection = AntControl.AntDirection(azimuth=23.0, elevation=72.5)
	msg = client.AntPoint(pointDirection)
	print("[Client] Attempted to steer antenna. Set to: %s" % msg)
	try:
		pointDirection = AntControl.AntDirection(azimuth=-1.0,
							elevation=181.2)
		msg = client.AntPoint(pointDirection)
	except AntControl.PointingException as pe:
		print("[Client] Attempted to send a bad AntDirection and" + \
			" successfully caught error")

	# Test AntBrake works the first time and throws an exception the second
	# time
	client.AntBrake(False)
	print("[Client] Attempted to set brake False. Set to: %s" % msg)
	try:
		msg = client.AntBrake(False)
	except AntControl.NoBrakeException:
		print("[Client] Attempted to use brake and correctly" + \
			" received an exception")
```

```
        # Test AntConfigure works the first time and throws an exception the
        # second time
        config = AntControl.AntConfig(azSlewRate=2.3,
                                      elSlewRate=1.2,
                                      polarization=Polarization.LHC)
        msg = client.AntConfigure(config)
        print("[Client] Attempted to set configuration. Set to: %s" % msg)
        try:
              config = AntControl.AntConfig(azSlewRate=3.1,
                                            elSlewRate=0.1,
                                            polarization=Polarization.LHC)
              msg = client.AntConfigure(config)
        except AntControl.BadConfigException as bce:
              print("[Client] Attempted to enter a bad config and" + \
                    " correctly received an exception")
              print("[Client] %s" % bce)

        # Test AntStatus
        msg = client.AntGetStatus()
        print("[Client] Attempted to check ant status. Received: %s" % msg)

except KeyboardInterrupt:
        print("[Shutdown] Close command received. Goodbye!")
except sockError as se:
        print("[Error] Socket was aborted. socket.error %s" % ( se))
except TTransport.TTransportException as tte:
        print("[Error] Server response timed out or data was lost." + \
              " TTransportException(%d): %s" % (tte.type, tte))
except Thrift.TApplicationException as ta:
        print("[Error] There appears to be an interface mismatch. Expected" + \
              " version of interface is %s: %s" % (constants.VERSION, ta))
except Exception as e:
        print("[Error] Exception: %s %s" % (type(e), e))
except:
        print("[Error] BaseException: %s" % sys.exc_info()[0])
```

P. ANTENNA_SERVER.PY

```
'''
	M-PIPE interface: MC3 Picosatellite Internet Protocol Extension
	Author:
		Aaron Felt, Naval Postgraduate School
	Description:
		This script is meant to exercise, test, and demonstrate the
		antenna Thrift interface as a server and is intended to be used
		with antenna_client.py. As it doesn't specify a protocol, it
		defaults to TBinaryProtocol but does use sockets as expected.
'''
import sys
sys.path.append('gen-py')
from thrift.transport import TSSLSocket
from thrift.server import TServer
from mpipe_antenna import AntControl

antBrakeTestToggle = False

class AntHandler(AntControl.Iface):
	def __init__(self):
		self.status = True
		self.currDirection = AntControl.AntDirection(azimuth=1.0,
							elevation=23.9)

	def AntPoint(self, antDirection):
		if antDirection.azimuth < 0.0 \
			or antDirection.azimuth >= 360.0 \
			or antDirection.elevation < 0.0 \
			or antDirection.elevation >= 180.0:
			print("[Server] Bad antenna point parameters." + \
				" Raising PointingException")
			raise AntControl.PointingException(antDirection)
		print("Pointing antenna to %s" % (antDirection))
		self.currDirection = antDirection
		return antDirection

	def AntBrake(self, enable):
		global antBrakeTestToggle
		# Toggles every call
		antBrakeTestToggle = not antBrakeTestToggle
		if antBrakeTestToggle:
			print("[Server] Antenna brake set to %r" % enable)
			return enable
		else:
			print("[Server] Antenna has no brake." + \
				" Raising NoBrakeException")
			raise AntControl.NoBrakeException()

	def AntConfigure(self, antConfig):
		if antConfig.elSlewRate < 0.25 :
			print("[Server] Bad config received as input." + \
```

```
                        " Raising BadConfigException")
                  raise AntControl.BadConfigException(antConfig=antConfig,
                                                      minElSlewRate=0.25)
            else:
                  print("[Server] Antenna being configured to %s" \
                        % antConfig)
                  return antConfig

      def AntGetStatus(self):
            status = AntControl.AntStatus(healthy=self.status,
                                          currDirection=self.currDirection)
            print("[Server] Antenna status being returned as %s" % status)
            return status

if __name__ == "__main__":
      svr_trans = TSSLSocket.TSSLServerSocket(port=8585)
      processor = AntControl.Processor(AntHandler())
      server = TServer.TSimpleServer(processor, svr_trans)
      server.serve()
```

Q. CPU_CLIENT.PY

```
'''
	M-PIPE interface: MC3 Picosatellite Internet Protocol Extension
	Author:
		Aaron Felt, Naval Postgraduate School
	Description:
		This script is meant to exercise, test, and demonstrate the CPU
		Thrift Interface as a client and is intended to be used with
		cpu_server.py. As it doesn't specify a protocol, it defaults to
		TBinaryProtocol but does use sockets as expected.
'''
import sys
sys.path.append("gen-py")

from thrift import Thrift
from thrift.transport import TSocket, TTransport
from thrift.protocol import TBinaryProtocol
from mpipe_cpu import CPUControl
from mpipe import constants
from socket import error as sockError

try:
	socket = TSocket.TSocket("localhost," 8585)
	socket = TTransport.TBufferedTransport(socket)
	socket.open()
	protocol = TBinaryProtocol.TBinaryProtocol(socket)

	client = CPUControl.Client(protocol)

	# Test GetCPUStatus works
	msg = client.GetCPUStatus()
	print("[Client] Attempted to get CPU status. Received: %s" % msg)

	# Test GetNetStatus works
	msg = client.GetNetStatus()
	print("[Client] Attempted to get network status. Received: %s" % msg)

except KeyboardInterrupt:
	print("[Shutdown] Close command received. Goodbye!")
except sockError as se:
	print("[Error] Socket was aborted. socket.error %s" % ( se))
except TTransport.TTransportException as tte:
	print("[Error] Server response timed out or data was lost." + \
		" TTransportException(%d): %s" % (tte.type, tte))
except Thrift.TApplicationException as ta:
	print("[Error] There appears to be an interface mismatch. Expected" + \
		" version of interface is %s: %s" % (constants.VERSION, ta))
except Exception as e:
	print("[Error] Exception: %s %s" % (type(e), e))
except:
	print("[Error] BaseException: %s" % sys.exc_info()[0])
```

R. CPU_SERVER.PY

```
'''
	M-PIPE interface: MC3 Picosatellite Internet Protocol Extension
	Author:
		Aaron Felt, Naval Postgraduate School
	Description:
		This script is meant to exercise, test, and demonstrate the CPU
		Thrift Interface as a server and is intended to be used with
		cpu_client.py. As it doesn't specify a protocol, it defaults to
		TBinaryProtocol but does use sockets as expected.
'''
import sys
sys.path.append('gen-py')
from thrift.transport import TSocket
from thrift.server import TServer
from mpipe_cpu import CPUControl

# Use external Python psutil library to get CPU/network statistics
import psutil

class CPUHandler(CPUControl.Iface):

	def GetCPUStatus(self):
		diskUsage = psutil.disk_usage("C:\\")
		print diskUsage
		memUsage = psutil.virtual_memory()
		print memUsage
		cpuStat = CPUControl.CPUStatus(
			status=True,
			uptime = 3216021,
			numProcessesTotal = 321,
			processesRunning = 27,
			loadAve = psutil.cpu_percent(interval=1),
				diskUsage = diskUsage.percent,
				# Convert bytes to MB
				diskFree = diskUsage.free / (2**20),
				ramUsage = memUsage.percent,
				# Convert bytes to MB
				ramFree = memUsage.free / (2**20)
		)
		return cpuStat

	# Note that this currently just returns the bytes in and out on the
	# interface, not specific to the M-PIPE connection, but this is just
	# meant to show capability
	def GetNetStatus(self):
		netIO = psutil.net_io_counters()
		netStat = CPUControl.NetStatus(bytesIn = netIO.bytes_recv,
						bytesOut = netIO.bytes_sent)
		print("[Server] Network status requested")
		return netStat
```

```
if __name__ == “__main__”:
       svr_trans = TSocket.TServerSocket(port=8585)
       processor = CPUControl.Processor(CPUHandler())
       server = TServer.TSimpleServer(processor, svr_trans)
       server.serve()
```

S. PACKET_CLIENT.PY

```
'''
	M-PIPE interface: MC3 Picosatellite Internet Protocol Extension
	Author:
		Aaron Felt, Naval Postgraduate School
	Description:
		This script is meant to exercise, test, and demonstrate the
		Packet Thrift interface as a client and is intended to be used
		with packet_server.py. As it doesn't specify a protocol, it
		defaults to TBinaryProtocol but does use sockets as expected.
'''
import sys
sys.path.append("gen-py")

from thrift import Thrift
from thrift.transport import TSocket, TTransport
from thrift.protocol import TBinaryProtocol
from mpipe_packet import PacketService
from mpipe import constants
from socket import error as sockError

currPacketID = 0

try:
	socket = TSocket.TSocket("localhost," 8585)
	socket = TTransport.TBufferedTransport(socket)
	socket.open()
	protocol = TBinaryProtocol.TBinaryProtocol(socket)

	client = PacketService.Client(protocol)

	# Test SendPacket works
	packet = PacketService.Packet(packetID=currPacketID,
					data="You have mail!")
	msg = client.SendPacket(packet)
	print("[Client] Successfully sent a packet")

	currPacketID = currPacketID + 1

	# Test SendPacket excepts correctly
	try:
		packet = PacketService.Packet(packetID=currPacketID, data="You
have mail!You have mail!You have mail!You have mail!You have mail!You have
mail!You have mail!You have mail!You have mail!You have mail!You have mail!You
have mail!You have mail!You have mail!You have mail!You have mail!You have
mail!You have mail!")
		msg = client.SendPacket(packet)
		print("[Client] !!!!ERROR!!!!!! ---- Attempted to send" + \
			" a bad packet but exception was NOT caught")
	except PacketService.PacketSizeException as pse:
		print("[Client] Correctly received size exception for" + \
			" big packet")
```

```
            print("[Client] %s" % pse)

except KeyboardInterrupt:
      print("[Shutdown] Close command received. Goodbye!")
except sockError as se:
      print("[Error] Socket was aborted. socket.error %s" % ( se))
except TTransport.TTransportException as tte:
      print("[Error] Server response timed out or data was lost." + \
            " TTransportException(%d): %s" % (tte.type, tte))
except Thrift.TApplicationException as ta:
      print("[Error] There appears to be an interface mismatch. Expected" + \
            " version of interface is %s: %s" % (constants.VERSION, ta))
except Exception as e:
      print("[Error] Exception: %s %s" % (type(e), e))
except:
      print("[Error] BaseException: %s" % sys.exc_info()[0])
```

T. PACKET_SERVER.PY

```
'''
	M-PIPE interface: MC3 Picosatellite Internet Protocol Extension
	Author:
		Aaron Felt, Naval Postgraduate School
	Description:
		This script is meant to exercise, test, and demonstrate the
		Packet Thrift interface as a server and is intended to be used
		with packet_client.py. As it doesn't specify a protocol, it
		defaults to TBinaryProtocol but does use sockets as expected.
'''
import sys
sys.path.append('gen-py')
from thrift.transport import TSocket
from thrift.server import TServer
from mpipe_packet import PacketService

MAX_PACKET_SIZE = 223

class PacketHandler(PacketService.Iface):

	def SendPacket(self, packet):
		if len(packet.data) > MAX_PACKET_SIZE:
			print("[Server] Packet size of %d received. Beyond" + \
				" max packet size of %d. Raising exception." % \
				(len(packet.data), MAX_PACKET_SIZE))
			raise PacketService.PacketSizeException(badPacket=packet,
									maxPacketSize = MAX_PACKET_SIZE)
		print("[Server] Packet received of size %d: %s" \
			% (len(packet.data), packet))

if __name__ == "__main__":
	svr_trans = TSocket.TServerSocket(port=8585)
	processor = PacketService.Processor(PacketHandler())
	server = TServer.TSimpleServer(processor, svr_trans)
	server.serve()
```

U. PREAMP_CLIENT.PY

```
'''
	M-PIPE interface: MC3 Picosatellite Internet Protocol Extension
	Author:
		Aaron Felt, Naval Postgraduate School
	Description:
		This script is meant to exercise, test, and demonstrate the
		preamp Thrift interface as a client and is intended to be used
		with preamp_server.py. As it doesn't specify a protocol, it
		defaults to TBinaryProtocol but does use sockets as expected.
'''
import sys
sys.path.append("gen-py")

from thrift import Thrift
from thrift.transport import TSocket, TTransport
from thrift.protocol import TBinaryProtocol
from mpipe_preamp import PAControl
from mpipe import constants
from socket import error as sockError

try:
	socket = TSocket.TSocket("localhost," 8585)
	socket = TTransport.TBufferedTransport(socket)
	socket.open()
	protocol = TBinaryProtocol.TBinaryProtocol(socket)

	client = PAControl.Client(protocol)

	# Test PAEnable works
	msg = client.PAEnable(True)
	print("[Client] Attempted to power on preamp. Set to: %r" % msg)
	msg = client.PAEnable(False)
	print("[Client] Attempted to power off preamp. Set to: %r" % msg)
	msg = client.PAEnable(True)
	print("[Client] Attempted to power on preamp. Set to: %r" % msg)

	# Test PASetGain works the first time and throws an exception the second
	# and third time
	firstGain = 5.0
	msg = client.PASetGain(firstGain)
	print("[Client] Attempted to set gain to %f. Set to: %f" %
		(firstGain,msg))
	try:
		badGain1 = -1.0
		msg = client.PASetGain(badGain1)
	except PAControl.GainException as ge:
		print("[Client] Attempted to set a bad gain of %f and" + \
			" correctly received an exception." % badGain1)
		print("[Client] %s" % ge)
	try:
		badGain2 = 10.1
```

```
            msg = client.PASetGain(badGain2)
        except PAControl.GainException as ge:
            print("[Client] Attempted to set a bad gain of %f and" + \
                  " correctly received an exception." % badGain2)
            print("[Client] %s" % ge)

        # Test PAGetStatus works the first time and returns the values set above
        # (power-on, gain 5.0)
        msg = client.PAGetStatus()
        print("[Client] %s" % msg)

except KeyboardInterrupt:
        print("[Shutdown] Close command received. Goodbye!")
except sockError as se:
        print("[Error] Socket was aborted. socket.error %s" % ( se))
except TTransport.TTransportException as tte:
        print("[Error] Server response timed out or data was lost." + \
              " TTransportException(%d): %s" % (tte.type, tte))
except Thrift.TApplicationException as ta:
        print("[Error] There appears to be an interface mismatch. Expected" + \
              " version of interface is %s: %s" % (constants.VERSION, ta))
except Exception as e:
        print("[Error] Exception: %s %s" % (type(e), e))
except:
        print("[Error] BaseException: %s" % sys.exc_info()[0])
```

V. PREAMP_SERVER.PY

```
'''
	M-PIPE interface: MC3 Picosatellite Internet Protocol Extension
	Author:
		Aaron Felt, Naval Postgraduate School
	Description:
		This script is meant to exercise, test, and demonstrate the
		preamp Thrift interface as a server and is intended to be used
		with preamp_client.py. As it doesn't specify a protocol, it
		defaults to TBinaryProtocol but does use sockets as expected.
'''
import sys
sys.path.append('gen-py')
from thrift.transport import TSocket
from thrift.server import TServer
from mpipe_preamp import PAControl

class PAHandler(PAControl.Iface):

	def __init__(self, minGain, maxGain, isPowered, gain):
		# Sample class variables for a real preamp
		self.minGain = minGain
		self.maxGain = maxGain
		self.isPowered = isPowered
		self.gain = gain

	def PAEnable(self, isPowered):
		print("[Server] Preamp power set to %r" % isPowered)
		self.isPowered = isPowered
		return isPowered

	def PASetGain(self, dB):
		if dB >= self.minGain and dB <= self.maxGain:
			self.gain = dB
			print "[Server] Preamp gain set to %f" % dB
			return dB
		elif dB < self.minGain:
			print("[Server] Preamp gain of %f below min gain of" + \
				" %f. Raising GainException" % (dB, self.minGain))
			raise PAControl.GainException(dB=dB, minGain=self.minGain)
		else:
			print("[Server] Preamp gain of %f above max gain of" + \
				" %f. Raising GainException" % (dB, self.maxGain))
			raise PAControl.GainException(dB=dB, maxGain=self.maxGain)

	def PAGetStatus(self):
			paStatus = PAControl.PAStatus(currGain=self.gain,
						currPowerState=self.isPowered,
						healthy=True)
			print("[Server] Preamp status is %s" % paStatus)
			return paStatus
```

```
if __name__ == “__main__”:
       minGain = 0.0
       maxGain = 10.0
       isPowered = False
       gain = 3.0
       svr_trans = TSocket.TServerSocket(port=8585)
       processor = PAControl.Processor(PAHandler(minGain, maxGain, isPowered,
gain))
       server = TServer.TSimpleServer(processor, svr_trans)
       server.serve()
```

W. RADIO_CLIENT.PY

```
‘‘‘
	M-PIPE interface: MC3 Picosatellite Internet Protocol Extension
	Author:
		Aaron Felt, Naval Postgraduate School
	Description:
		This script is meant to exercise, test, and demonstrate the radio
		Thrift Interface as a client and is intended to be used with
		radio_server.py. As it doesn’t specify a protocol, it defaults to
		TBinaryProtocol but does use sockets as expected.
‘‘‘
import sys
sys.path.append(“gen-py”)

from thrift import Thrift
from thrift.transport import TSocket, TTransport
from thrift.protocol import TBinaryProtocol
from mpipe_radio import RControl
from mpipe import constants
from socket import error as sockError
from mpipe_globals.ttypes import Mode, Modulation, Encoding, RSConfig,
LinkLayerProt

try:
	socket = TSocket.TSocket(“localhost,” 8585)
	socket = TTransport.TBufferedTransport(socket)
	socket.open()
	protocol = TBinaryProtocol.TBinaryProtocol(socket)

	client = RControl.Client(protocol)

	# Test RadioConfigure works the first time and fails the second
	config = RControl.RConfig(
			freq = 2415.2,
			mode = Mode.PM,
			powerLevel = 100.0,
			encoding = Encoding.NRZI,
			useDiffEncode = True,
			useReedSolomon = True,
			rsParams = RSConfig(
					n = 223,
					k = 255),
			llProt = LinkLayerProt.HDLC,
			modulation = Modulation.BPSK,
			baud = 57600)
	msg = client.RadioConfigure(config)
	print(“[Client] Attempted to configure for test transmit. Set to:” + \
		“ %s” % msg)
	try:
		config = RControl.RConfig(
				freq = 2415.2,
				mode = Mode.PM,
```

```
                        powerLevel = 100.0,
                        encoding = Encoding.NRZI,
                        useDiffEncode = True,
                        useReedSolomon = True,
                        # Purposely not including rsParams to trigger
                        llProt = LinkLayerProt.HDLC,
                        modulation = Modulation.BPSK,
                        baud = 57600)
                msg = client.RadioConfigure(config)
                print("[Client] !!!!ERROR!!!!!! ---- Attempted to send a bad" + \
                      " config but exception was NOT caught")
        except RControl.ConfigException as ce:
                print("[Client] Attempted to send a bad config but exception" + \
                      " was correctly caught")
                print("[Client] %s" % ce)

        # Test RSDRSelect works the first time and throws exceptions on 2nd and
        # 3rd
        msg = client.RSDRSelect(name="test.sdr," params="--param1 goodValue")
        print("[Client] Started an SDR.")
        try:
                msg = client.RSDRSelect(name="badName.sdr,"
                                        params="--param1 goodValue")
                print("[Client] !!!!ERROR!!!!!! ---- Attempted to send a bad" + \
                      " SDR name but exception was NOT caught")
        except RControl.ProgramException as pe:
                print("[Client] Correctly got exception for bad SDR name")
                print("[Client] %s" % pe)
        try:
                msg = client.RSDRSelect(name="test.sdr," params="badParams")
                print("[Client] !!!!ERROR!!!!!! ---- Attempted to send a bad" + \
                      " SDR param but exception was NOT caught")
        except RControl.ParamException as pe:
                print("[Client] Correctly got exception for bad params")
                print("[Client] %s" % pe)

        # Test RGetReceiveInfo works
        msg = client.RGetReceiveInfo()
        print("[Client] Attempted to read receive strength: %s" % (msg))

except KeyboardInterrupt:
        print("[Shutdown] Close command received. Goodbye!")
except sockError as se:
        print("[Error] Socket was aborted. socket.error %s" % ( se))
except TTransport.TTransportException as tte:
        print("[Error] Server response timed out or data was lost." + \
              " TTransportException(%d): %s" % (tte.type, tte))
except Thrift.TApplicationException as ta:
        print("[Error] There appears to be an interface mismatch. Expected" + \
              " version of interface is %s: %s" % (constants.VERSION, ta))
except Exception as e:
        print("[Error] Exception: %s %s" % (type(e), e))
except:
        print("[Error] BaseException: %s" % sys.exc_info()[0])
```

X. RADIO_SERVER.PY

```
‘‘‘
	M-PIPE interface: MC3 Picosatellite Internet Protocol Extension
	Author:
		Aaron Felt, Naval Postgraduate School
	Description:
		This script is meant to exercise, test, and demonstrate the radio
		Thrift Interface as a server and is intended to be used with
		radio_client.py. As it doesn’t specify a protocol, it defaults to
		TBinaryProtocol but does use sockets as expected.
‘‘‘
import sys
sys.path.append(‘gen-py’)
from thrift.transport import TSocket
from thrift.server import TServer
from mpipe_radio import RControl

class RHandler(RControl.Iface):

	def RadioConfigure(self, config):
		if config.useReedSolomon and config.rsParams == None:
			errorMap = {
				14: RControl.ConfigError.IncompatibleChoices,
				15: RControl.ConfigError.IncompatibleChoices
			}
			print(“[Server] Config with conflicting parameters” + \
				“ requested. Rejecting.”)
			print(“[Server] Bad config: %s” % config)
			raise RControl.ConfigException(badConfig=config,
							fieldErrors=errorMap)
		print(“[Server] Radio config set to %s” % config)
		return config

	def RSDRSelect(self, name, params):
		if name == “badName.sdr”:
			print(“[Server] Bad SDR named %s being rejected” % (name))
			raise RControl.ProgramException(name, “Program %s not” + \
				“ found” % name)
		if params == “badParams”:
			print(“[Server] Bad params being rejected”)
			raise RControl.ParamException(name, “Parameters are” + \
				“ invalid”)
		print(“[Server] Starting %s with params %s” % (name, params))
		print(“[Server] Command would look like: %s %s” % (name, params))

	def RGetReceiveInfo(self):
		print(“[Server] Returning receive information”)
		return RControl.RReceiveInfo(rssi=-90.0, snr=15.0)

if __name__ == “__main__”:
	svr_trans = TSocket.TServerSocket(port=8585)
	processor = RControl.Processor(RHandler())
```

```
server = TServer.TSimpleServer(processor, svr_trans)
server.serve()
```

Y. SESSION_CLIENT.PY

```
'''
	M-PIPE interface: MC3 Picosatellite Internet Protocol Extension
	Author:
		Aaron Felt, Naval Postgraduate School
	Description:
		This script is meant to exercise, test, and demonstrate the
		Session Thrift interface as a client and is intended to be used
		with session_server.py. As it doesn't specify a protocol, it
		defaults to TBinaryProtocol but does use sockets as expected.
'''
import sys
sys.path.append("gen-py")

from thrift import Thrift
from thrift.transport import TSocket, TTransport
from thrift.protocol import TBinaryProtocol
from mpipe_session import SessionControl
from mpipe import constants
from socket import error as sockError

try:
	socket = TSocket.TSocket("localhost," 8585)
	socket = TTransport.TBufferedTransport(socket)
	socket.open()
	protocol = TBinaryProtocol.TBinaryProtocol(socket)

	client = SessionControl.Client(protocol)

	# Test GetSessionInfo works
	msg = client.GetSessionInfo()
	print("[Client] Attempted to get session status. Received: %s" % msg)

	# Test SetSessionConfig throws an exception the first time and works the
	# second time
	try:
		config = SessionControl.SessionConfig(
			isAutoAntTrack = True,
			isAutoDopplerTrack = True
		)
		msg = client.SetSessionConfig(config)
		print("[Client] !!!!ERROR!!!!!! ---- Attempted to send a bad" + \
			" config but exception was NOT caught")
	except SessionControl.ConfigException as ce:
		print("[Client] Correctly received config exception for" + \
			" bad config")
		print("[Client] %s" % ce)
	config = SessionControl.SessionConfig(
		currTLE = "AGOODTLE,"
		isAutoAntTrack = True,
		isAutoDopplerTrack = True
	)
```

```
        msg = client.SetSessionConfig(config)
        print("[Client] Set new config. Received: %s" % msg)

    except KeyboardInterrupt:
        print("[Shutdown] Close command received. Goodbye!")
    except sockError as se:
        print("[Error] Socket was aborted. socket.error %s" % ( se))
    except TTransport.TTransportException as tte:
        print("[Error] Server response timed out or data was lost." + \
              " TTransportException(%d): %s" % (tte.type, tte))
    except Thrift.TApplicationException as ta:
        print("[Error] There appears to be an interface mismatch. Expected" + \
              " version of interface is %s: %s" % (constants.VERSION, ta))
    except Exception as e:
        print("[Error] Exception: %s %s" % (type(e), e))
    except:
        print("[Error] BaseException: %s" % sys.exc_info()[0])
```

Z. SESSION_SERVER.PY

```
'''
	M-PIPE interface: MC3 Picosatellite Internet Protocol Extension
	Author:
		Aaron Felt, Naval Postgraduate School
	Description:
		This script is meant to exercise, test, and demonstrate the
		Session Thrift interface as a server and is intended to be used
		with session_client.py. As it doesn't specify a protocol, it
		defaults to TBinaryProtocol but does use sockets as expected.
'''
import sys
sys.path.append('gen-py')
from thrift.transport import TSocket
from thrift.server import TServer
from mpipe_session import SessionControl

class SessionHandler(SessionControl.Iface):

	def GetSessionInfo(self):
		info = SessionControl.SessionInfo(
				CatalogNumber=39400,
				latitude=36.651231,
				longitude=-121.81231,
				altitude=30.12
				)
		return info

	def SetSessionConfig(self, config):
		if ((config.isAutoAntTrack or config.isAutoDopplerTrack) \
				and config.currTLE == None):
			print("[Server] Exception thrown for bad config")
			raise SessionControl.ConfigException(badConfig=config,
				error=SessionControl.ConfigError.noTLEForTrack)
		print("[Server] Config received")
		print("[Server] %s" % config)
		return config

if __name__ == "__main__":
	svr_trans = TSocket.TServerSocket(port=8585)
	processor = SessionControl.Processor(SessionHandler())
	server = TServer.TSimpleServer(processor, svr_trans)
	server.serve()
```

AA. CA_SCHEDULER.PY

```
'''
	M-PIPE interface: MC3 Picosatellite Internet Protocol Extension
	Author:
		Aaron Felt, Naval Postgraduate School
	Description:
		This script is meant to exercise, test, and demonstrate the
		Scheduler Thrift interface as a CA and is intended to be used
		with sa_scheduler.py. As it doesn't specify a protocol, it
		defaults to TBinaryProtocol but does use sockets as expected.
'''
import sys, getopt
sys.path.append("gen-py")

from thrift import Thrift
from thrift.transport import TSocket, TTransport
from thrift.protocol import TBinaryProtocol
from thrift.server import TServer
import EventTServerSocket
from socket import error as sockError
import threading
from time import sleep
from mpipe_scheduler import ResourceOfferer, ResourceScheduler
from mpipe import constants
from resource_scheduler_server import ResourceSchedulerHandler
from mpipe_globals.ttypes import Mode, Modulation, Encoding, RSConfig,
LinkLayerProt
from mpipe_scheduler.ResourceScheduler import Direction
from mpipe_antenna.ttypes import Polarization


def main(argv):
	# Default random ports
	caPort = 8585
	saPort =caPort+1
	try:
		opts, args = getopt.getopt(argv,"hs:c:","["sport=","cport="])
	except getopt.GetoptError:
		print("sa_example.py -s <SA port> -c <CA port>")
		sys.exit(2)
	for opt, arg in opts:
		if opt == '-h':
			print("sa_example.py -s <SA port> -c <CA port>")
			sys.exit()
		elif opt in ("-s," "--sport"):
			saPort = int(arg)
		elif opt in ("-c," "--cport"):
			caPort = int(arg)
	print("SA port is ," saPort)
	print("CA port is ," caPort)

	# Create an event that can signal when a connection comes in from the
```

```
# SA indicating we can move on
connEvent = threading.Event()

# Start a ResourceScheduler server
rsHandler = ResourceSchedulerHandler()
rsProcessor = ResourceScheduler.Processor(rsHandler)

# Use a custom event-handling threaded server
rsTrans = EventTServerSocket.EventTServerSocket(event=connEvent,
                                                port=caPort)
rsServer = TServer.TThreadedServer(rsProcessor, rsTrans)

# Serve in a thread as a daemon
rsServer_thread = threading.Thread(target=rsServer.serve)
# Exit the server thread when the main thread terminates
rsServer_thread.daemon = True
rsServer_thread.start()

# Wait for a connection to come in from the SA on our service
print “Waiting for connection from SA to initiate service”
connEvent.wait()

# Give servers time to start up
sleep(1)

# Start a ResourceOfferer client
roSocket = TSocket.TSocket(“localhost,” saPort)
roSocket = TTransport.TBufferedTransport(roSocket)
roSocket.open()
roProtocol = TBinaryProtocol.TBinaryProtocol(roSocket)
roClient = ResourceOfferer.Client(roProtocol)

# Make sure the resource has been added and finalized before reserving
# it
sleep(1)

config =ResourceOfferer.ResourceConfig(id = 1,
       saName = “SA1,”
       # Unix time for 3 PM UTC on Sep 18, 2014 plus 5 minutes to cause
       # need for resource to be split
       start = 5.0*60.0 + 1412349478.884,
       # Unix time for 3 PM UTC on Sep 18, 2014 plus 10 minutes to cause
       # need for resource to be split again
       stop = 10.0*60.0 + 1412349478.884,
       direction = Direction.TX
)
radioSelections = ResourceOfferer.RadioSelections(\
                          centerFrequency = 922.325,
                          mode = Mode.FM,
                          modulation = Modulation.BPSK,
                          hardwareBandwidth = 1.5,
                          linkLayerProt = LinkLayerProt.HDLC,
                          encoding = Encoding.NRZI,
                          baud = 115200,
```

```
                                    useViterbi = True,
                                    usePRN = True,
                                    useRS = False,
                                    useTurboCode = False
        )


        user = ResourceOfferer.User(userName = “Bob,”
                                    publicCertificate = “a certificate”
)
        roClient.ScheduleResource(config, radioSelections, user)
        print(“[CA] Resource scheduled for user:”)
        print(user)

if __name__ == “__main__”:
        main(sys.argv[1:])
```

BB. SA_SCHEDULER.PY

```
'''
	M-PIPE interface: MC3 Picosatellite Internet Protocol Extension
	Author:
		Aaron Felt, Naval Postgraduate School
	Description:
		This script is meant to exercise, test, and demonstrate the
		Scheduler Thrift interface as an SA and is intended to be used
		with ca_scheduler.py. As it doesn't specify a protocol, it
		defaults to TBinaryProtocol but does use sockets as expected.
'''
import sys, getopt
sys.path.append("gen-py")

from thrift import Thrift
from thrift.transport import TSocket, TTransport
from thrift.protocol import TBinaryProtocol
from thrift.server import TServer
from socket import error as sockError
from time import sleep
import threading
import Queue
from mpipe_scheduler import ResourceOfferer, ResourceScheduler
from resource_offerer_server import ResourceOffererHandler
from mpipe_globals.ttypes import Mode, Modulation, Encoding, RSConfig,
LinkLayerProt
from mpipe_scheduler.ResourceScheduler import Direction
from mpipe_antenna.ttypes import Polarization

myName = "SA1"

def main(argv):
	# Default random ports
	caPort = 8585
	saPort =caPort+1
	try:
		opts, args = getopt.getopt(argv,"hs:c:","["sport=","cport="])
	except getopt.GetoptError:
		print("sa_example.py -s <SA port> -c <CA port>")
		sys.exit(2)
	for opt, arg in opts:
		if opt == '-h':
			print("sa_example.py -s <SA port> -c <CA port>")
			sys.exit()
		elif opt in ("-s," "--sport"):
			saPort = int(arg)
		elif opt in ("-c," "--cport"):
			caPort = int(arg)
	print("SA port is ," saPort)
	print("CA port is ," caPort)

	# Connect as a client to the CA's ResourceScheduler interface
```

```
rsSocket = TSocket.TSocket(“localhost,” 8585)
rsSocket = TTransport.TBufferedTransport(rsSocket)
rsSocket.open()
rsProtocol = TBinaryProtocol.TBinaryProtocol(rsSocket)
rsClient = ResourceScheduler.Client(rsProtocol)

# Start a ResourceOfferer server interface for the CA to connect to
roProcessor = ResourceOfferer.Processor(ResourceOffererHandler())
roTrans = TSocket.TServerSocket(port=saPort)
roServer = TServer.TThreadedServer(roProcessor, roTrans)

# Serve in a thread as a daemon
roServer_thread = threading.Thread(target=roServer.serve)
# Exit the server thread when the main thread terminates
roServer_thread.daemon = True
roServer_thread.start()

# Save a queue for new/unused resource IDs
unusedResourceIDs = Queue.Queue()
# Save a map for resources
resourceMap = dict()
radioOptionsMap = dict()
restrictionsMap = dict()

#Get a resource ID
id = rsClient.RequestResourceID()
try:
       int(id)
except ValueError:
       print(“[ERROR] The scheduler returned a non-int as a” + \
              “ resource ID”)
       sys.exit(1)
# Throw the ID in the queue and pull it right back out (just for good
# practice here)
unusedResourceIDs.put(id)
resourceID = unusedResourceIDs.get()

radioOptionsMap[resourceID] = ResourceScheduler.RadioOptions(
       mode = [Mode.FM,
              Mode.LSB],
       modulation = [Modulation.AFSK,
                     Modulation.GMSK,
                     Modulation.BPSK],
       hardwareBandwidths = [0.0015,
                             0.003,
                             0.006,
                             1.5,
                             3,
                             6],
       linkLayerProt = [LinkLayerProt.HDLC],
       encoding = [Encoding.NRZI,
                   Encoding.NRZS],
       baud = [1200,
               9600,
```

```
                19200,
                57600,
                115200,
                921600],
        viterbiSupported = True,
        prnSupported = True,
        rsSupported = False,
        turboCodeSupported = False
)

restrictionsMap[resourceID] = [ResourceScheduler.RadioRestrictions(
                                    mode = Mode.LSB,
                                    modulation = Modulation.BPSK),
                               ResourceScheduler.RadioRestrictions(
                                    modulation = Modulation.BPSK,
                                    baud = 1200),
                               ResourceScheduler.RadioRestrictions(
                                    modulation = Modulation.BPSK,
                                    baud = 9600),
                               ResourceScheduler.RadioRestrictions(
                                    modulation = Modulation.BPSK,
                                    baud = 19200),
                               ResourceScheduler.RadioRestrictions(
                                    modulation = Modulation.GMSK,
                                    baud = 57600),
                               ResourceScheduler.RadioRestrictions(
                                    modulation = Modulation.GMSK,
                                    baud = 115200),
                               ResourceScheduler.RadioRestrictions(
                                    modulation = Modulation.GMSK,
                                    baud = 921600)]

resourceMap[resourceID] = ResourceScheduler.ResourceOptions(
        id = resourceID,
        saName = myName,
        start = 1412349478.884, # Unix time for 3 PM UTC on Sep 18, 2014
        stop = 1412349478.884 + 60.0*15.0, # 15 minutes out
        maxSecReserve = 60*15,
        direction = Direction.TX,
        lowFreq = 902.0,
        highFreq = 928.0,
        radioManufacturer = “Radios Inc,”
        radioModel = “100,”
        acuManufacturer = “Yaesu,”
        acuModel = “G-5500,”
        paManufacturer = “ICOM,”
        paModel = “920A,”
        ampManufacturer = “ICOM,”
        ampModel = “AS45,”
        azSlewRateMin = 1.0,
        azSlewRateMax = 5.0,
        elSlewRateMin = 1.0,
        elSlewRateMax = 5.0,
        polarizations = [Polarization.LHC,
```

```
                                    Polarization.RHC],
                gain = 21.7,
                antControlEnabled = True,
                minAz = 0.0,
                maxAz = 450.0,
                minEl = 0.0,
                maxEl = 180.0,
                minElTx = 10.0,
                paControlEnabled = True,
                paMinGain = 0.0,
                paMaxGain = 30.0,
                freqControlEnabled = True,
                latitude=36.651231,
                longitude=-121.81231,
                altitude=30.12
        )

        # Should be successful
        rsClient.OfferResource(resourceMap[resourceID],
                               radioOptionsMap[resourceID],
                               restrictionsMap[resourceID])
        print("[SA] Offered a resource")

        # Run until test ended
        while(True):
                sleep(1)

if __name__ == "__main__":
        main(sys.argv[1:])
```

CC. USER_CONTROL_EXAMPLE.PY

```
'''
	M-PIPE interface: MC3 Picosatellite Internet Protocol Extension
	Author:
		Aaron Felt, Naval Postgraduate School
	Name:
		user_control_example.py
	Description:
		This script is meant to exercise, test, and demonstrate the user
		side of a resource-control interface and is intended to be used
		with sa_control_example.py. This script demonstrates control of
		every piece of hardware as well as session control interfaces
		remotely by a user. It begins with a user hosting a received
		packet server interface, waiting for a connection from the SA.
		The SA connects and opens server interfaces to the user for
		hardware and session control, and the user connects to these
		interfaces and tests each interface.
'''
import sys, getopt
sys.path.append("gen-py")

from thrift import Thrift
from thrift.transport import TSocket, TTransport
from thrift.protocol import TBinaryProtocol
from thrift.server import TServer
from socket import error as sockError
from mpipe_packet import PacketService
from mpipe_session import SessionControl
from mpipe_antenna import AntControl
from mpipe_cpu import CPUControl
from mpipe_preamp import PAControl
from mpipe_amp import AmpControl
from mpipe_radio import RControl
from mpipe import constants
import packet_server
import threading
import EventTServerSocket
from time import sleep
from mpipe_globals.ttypes import Mode, Modulation, Encoding, RSConfig,
LinkLayerProt


MAX_PACKET_SIZE = 223

def main(argv):
	# Default random ports
	userPort = 8585
	saPort = userPort+10
	try:
		opts, args = getopt.getopt(argv,"hs:u:,"["sport=,""uport="])
	except getopt.GetoptError:
		print("user_control_example.py -s <SA port> -u <user port>")
```

```
      sys.exit(2)
for opt, arg in opts:
      if opt == '-h':
            print("user_control_example.py -s <SA port> -u" + \
                  " <user port>")
            sys.exit()
      elif opt in ("-s," "--sport"):
            saPort = int(arg)
      elif opt in ("-u," "--uport"):
            userPort = int(arg)
print("SA port is %d" % saPort)
print("User port is %d" % userPort)

# Create an event that can signal when a connection comes in from the
# SA indicating pass start
connEvent = threading.Event()

# Start the RX packet server
rxPacketHandler = packet_server.PacketHandler()
rxPacketProcessor = PacketService.Processor(rxPacketHandler)
# Use a custom event-handling threaded server
rxPacketTrans = EventTServerSocket.EventTServerSocket(event=connEvent,
                                                      port=userPort)
rxPacketServer = TServer.TThreadedServer(rxPacketProcessor,
                                         rxPacketTrans)

# Serve in a thread as a daemon
rxPacketServer_thread = threading.Thread(target=rxPacketServer.serve)
# Exit the server thread when the main thread terminates
rxPacketServer_thread.daemon = True
rxPacketServer_thread.start()

# Wait for a connection to come in from the SA on our packet service
print "Waiting for connection from SA to initiate service"
connEvent.wait()

# Give servers time to start up
sleep(1)

# Starting TX packet client
txPacketSocket = TSocket.TSocket("localhost," saPort)
txPacketSocket = TTransport.TBufferedTransport(txPacketSocket)
txPacketSocket.open()
txPacketProtocol = TBinaryProtocol.TBinaryProtocol(txPacketSocket)
txPacketClient = PacketService.Client(txPacketProtocol)

# Starting session client
sessSocket = TSocket.TSocket("localhost," saPort+1)
sessSocket = TTransport.TBufferedTransport(sessSocket)
sessSocket.open()
sessProtocol = TBinaryProtocol.TBinaryProtocol(sessSocket)
sessClient = SessionControl.Client(sessProtocol)

# Starting the antenna client
```

```
antSocket = TSocket.TSocket(“localhost,” saPort+2)
antSocket = TTransport.TBufferedTransport(antSocket)
antSocket.open()
antProtocol = TBinaryProtocol.TBinaryProtocol(antSocket)
antClient = AntControl.Client(antProtocol)

# Starting the RX radio client
rxRadSocket = TSocket.TSocket(“localhost,” saPort+3)
rxRadSocket = TTransport.TBufferedTransport(rxRadSocket)
rxRadSocket.open()
rxRadProtocol = TBinaryProtocol.TBinaryProtocol(rxRadSocket)
rxRadClient = RControl.Client(rxRadProtocol)

# Starting the TX radio client
txRadSocket = TSocket.TSocket(“localhost,” saPort+4)
txRadSocket = TTransport.TBufferedTransport(txRadSocket)
txRadSocket.open()
txRadProtocol = TBinaryProtocol.TBinaryProtocol(txRadSocket)
txRadClient = RControl.Client(txRadProtocol)

# Starting the pre-amp client
paSocket = TSocket.TSocket(“localhost,” saPort+5)
paSocket = TTransport.TBufferedTransport(paSocket)
paSocket.open()
paProtocol = TBinaryProtocol.TBinaryProtocol(paSocket)
paClient = PAControl.Client(paProtocol)

# Starting the amp client
ampSocket = TSocket.TSocket(“localhost,” saPort+6)
ampSocket = TTransport.TBufferedTransport(ampSocket)
ampSocket.open()
ampProtocol = TBinaryProtocol.TBinaryProtocol(ampSocket)
ampClient = AmpControl.Client(ampProtocol)

# Starting the CPU client
cpuSocket = TSocket.TSocket(“localhost,” saPort+7)
cpuSocket = TTransport.TBufferedTransport(cpuSocket)
cpuSocket.open()
cpuProtocol = TBinaryProtocol.TBinaryProtocol(cpuSocket)
cpuClient = CPUControl.Client(cpuProtocol)

# Ready to go! Send commands
print(“Full session with SA established”)

# Get ground station info (should know this from resource reservation
# anyways but can check here)
sessInfo = sessClient.GetSessionInfo()
print(“Satellite ID: %d” % sessInfo.CatalogNumber)
print(“Lat: %f\tLon: %f\tAlt: %f” % (sessInfo.latitude,
                                     sessInfo.longitude,
                                     sessInfo.altitude))

# Configure session
try:
```

```
	config = SessionControl.SessionConfig(
		isAutoAntTrack = False,
		isAutoDopplerTrack = False
	)
	sessClient.SetSessionConfig(config)
except SessionControl.ConfigException as ce:
	print("The SA did not accept the session" + \
		" configuration requested.")
	print(ce)

# Configure slew rate and polarization
try:
	config = AntControl.AntConfig(azSlewRate=3.1,
			elSlewRate=3.1,
			polarization=AntControl.Polarization.RHC)
	msg = antClient.AntConfigure(config)
except AntControl.BadConfigException as bce:
	print("Attempted to enter an antenna config but received" + \
		" an exception")
	print(bce)

# Get current antenna position
antStatus = antClient.AntGetStatus()
print("Current antenna pointing direction: %s" \
	% antStatus.currDirection)

# Point the antenna at the lowest elevation allowed of 10 degrees at
# 23.4 Az where the satellite will come into view pointDirection =
# AntControl.AntDirection(azimuth=23.4, elevation=10.0)
msg = antClient.AntPoint(pointDirection)
print(pointDirection)

# Turn off the brake
try:
	msg = antClient.AntBrake(False)
	print("Brake released on antenna")
except AntControl.NoBrakeException:
	print("Attempted to use brake but ground station antenna" + \
		" doesn't have a brake.")

# Test AntStatus and pointing success
antStatus = antClient.AntGetStatus()
if not (antStatus.healthy):
	print("Antenna status is degraded")
else:
	newPointDirection = antStatus.currDirection
	print("Checking that antenna has pointed to correct direction")
	if (abs(newPointDirection.azimuth - pointDirection.azimuth) \
			> 1.0 or abs(newPointDirection.elevation - \
			pointDirection.elevation) > 1.0):
		print("Greater than 1 degree off target in at least" + \
			" one axis still")
	else:
		print("Pointing within a degree in both axes")
```

```
# Configure the RX radio
rxConfig = RControl.RConfig(
              freq = 2415.2,
              mode = Mode.PM,
              attenuation = 0.0,
              encoding = Encoding.NRZI,
              useDiffEncode = True,
              useReedSolomon = True,
              rsParams = RSConfig(
                     n = 200,
                     k = 250),
              llProt = LinkLayerProt.HDLC,
              modulation = Modulation.BPSK,
              baud = 57600,
              acqRange = 0.5)
print("Setting RX radio config")
print(rxConfig)
try:
       msg = rxRadClient.RadioConfigure(rxConfig)
except RControl.ConfigException as ce:
       print("SA rejected RX radio configuration")
       print(ce)

# Configure the TX radio
txConfig = RControl.RConfig(
              freq = 2310.8,
              mode = Mode.FM,
              powerLevel = 100.0,
              encoding = Encoding.NRZI,
              useDiffEncode = True,
              llProt = LinkLayerProt.HDLC,
              modulation = Modulation.GMSK,
              baud = 57600)
print("Setting TX radio config")
print(txConfig)
try:
       msg = txRadClient.RadioConfigure(txConfig)
except RControl.ConfigException as ce:
       print("SA rejected TX radio configuration")
       print(ce)

# Test RGetReceiveInfo works
msg = rxRadClient.RGetReceiveInfo()
print("Current RX radio receive strength: %s" % (msg))

# Try to enable the pre-amp
msg = paClient.PAEnable(True)
print("Powered on pre-amp")

# Set the pre-amp gain
try:
       newPAGain = 30.0
       msg = paClient.PASetGain(newPAGain)
```

```
        print("Set pre-amp gain to 30.0dB")
except PAControl.GainException as ge:
        print("SA did not accept requested pre-amp gain")
        print("%s" % ge)

# Verify that pre-amp started
paStatus = paClient.PAGetStatus()
if (not paStatus.healthy):
        print("Pre-amp is in a degraded state with error code: %d" \
                % paStatus.degradedCode)
elif (not paStatus.currPowerState):
        print("Pre-amp didn't power on as requested")
elif (paStatus.currGain != newPAGain):
        print("Pre-amp gain does not match requested gain")
else:
        print("Pre-amp started correctly and settings are correct")

# Try to enable the amp
msg = ampClient.AmpEnable(True)
print("Powered on amp")

# Set the amp gain
try:
        newAmpGain = 100.0
        msg = ampClient.AmpSetGain(newAmpGain)
        print("Set amp gain to 100.0dB")
except AmpControl.GainException as ge:
        print("SA did not accept requested amp gain")
        print("%s" % ge)

# Verify that amp started
ampStatus = ampClient.AmpGetStatus()
if (not ampStatus.healthy):
        print("Amp is in a degraded state with error code: %d" \
                % ampStatus.degradedCode)
elif (not ampStatus.currPowerState):
        print("Amp didn't power on as requested")
elif (ampStatus.currGain != newAmpGain):
        print("Amp gain does not match requested gain")
else:
        print("Amp started correctly and settings are correct")

# Check the CPU status
cpuStatus = cpuClient.GetCPUStatus()
print("CPU status at SA: %s" % cpuStatus)

# Check the network status
netStatus = cpuClient.GetNetStatus()
print("Network status at SA: %s" % netStatus)

# Send TX data forever
currPacketID = 1
# # Send received data forever
while(True): # Normal test would be until resource allocation time is up
```

```
            sleep(1)
            packet = PacketService.Packet(packetID=currPacketID,
                        data="Current TX packet you are" + \
                        " receiving: %d" % currPacketID)

            try:
                  msg = txPacketClient.SendPacket(packet)
            except PacketService.PacketSizeException as pse:
                  print("Packet size exception received in attempt to" + \
                        " send a received packet")
                  print(pse)
            currPacketID = currPacketID + 1

if __name__ == "__main__":
      main(sys.argv[1:])
```

DD. SA_CONTROL_EXAMPLE.PY

```
'''
	M-PIPE interface: MC3 Picosatellite Internet Protocol Extension
	Author:
		Aaron Felt, Naval Postgraduate School
	Name:
		sa_control_example.py
	Description:
		This script is meant to exercise, test, and demonstrate the SA
		side of a resource-control interface and is intended to be used
		with user_control_example.py. This script demonstrates control of
		every piece of hardware as well as session control interfaces
		remotely by a user. It begins with a user hosting a received
		packet server interface, waiting for a connection from the SA.
		The SA connects and opens server interfaces to the user for
		hardware and session control, and the user connects to these
		interfaces and tests each interface.
'''
import sys, getopt
sys.path.append("gen-py")

from thrift import Thrift
from thrift.transport import TSocket, TTransport
from thrift.protocol import TBinaryProtocol
from thrift.server import TServer
from socket import error as sockError
from mpipe_packet import PacketService
from mpipe_session import SessionControl
from mpipe_antenna import AntControl
from mpipe_cpu import CPUControl
from mpipe_preamp import PAControl
from mpipe_amp import AmpControl
from mpipe_radio import RControl
from mpipe import constants
import threading
import packet_server
from session_server import SessionHandler
from cpu_server import CPUHandler
from radio_server import RHandler
from preamp_server import PAHandler
from amp_server import AmpHandler
from antenna_server import AntHandler
from time import sleep


MAX_PACKET_SIZE = 223

def main(argv):
	# Default random ports
	userPort = 8585
	saPort = userPort+10
	try:
```

```
      opts, args = getopt.getopt(argv,”hs:u:,”[“sport=,””uport=“])
except getopt.GetoptError:
      print(“sa_ control_example.py -s <SA port> -u <user port>“)
      sys.exit(2)
for opt, arg in opts:
      if opt == ‘-h’:
            print(“sa_control_example.py -s <SA port> -u <user port>“)
            sys.exit()
      elif opt in (“-s,” “--sport”):
            saPort = int(arg)
      elif opt in (“-u,” “--uport”):
            userPort = int(arg)
print(“SA port is ,” saPort)
print(“User port is ,” userPort)

# Starting RX packet client to kick off the session
rxPacketSocket = TSocket.TSocket(“localhost,” userPort)
rxPacketSocket = TTransport.TBufferedTransport(rxPacketSocket)
rxPacketSocket.open()
rxPacketProtocol = TBinaryProtocol.TBinaryProtocol(rxPacketSocket)
rxPacketClient = PacketService.Client(rxPacketProtocol)

# Starting the TX packet server
txPacketTrans = TSocket.TServerSocket(port=saPort)
txPacketProcessor = \
      PacketService.Processor(packet_server.PacketHandler())
txPacketServer = TServer.TSimpleServer(txPacketProcessor, txPacketTrans)
# Serve in a thread as a daemon
txPacketServer_thread = threading.Thread(target=txPacketServer.serve)
# # Exit the server thread when the main thread terminates
txPacketServer_thread.daemon = True
txPacketServer_thread.start()

# Starting session server
sessTrans = TSocket.TServerSocket(port=saPort+1)
sessProcessor = SessionControl.Processor(SessionHandler())
sessServer = TServer.TSimpleServer(sessProcessor, sessTrans)
# Serve in a thread as a daemon
sessServer_thread = threading.Thread(target=sessServer.serve)
# Exit the server thread when the main thread terminates
sessServer_thread.daemon = True
sessServer_thread.start()

# Starting the antenna server
antTrans = TSocket.TServerSocket(port=saPort+2)
antProcessor = AntControl.Processor(AntHandler())
antServer = TServer.TSimpleServer(antProcessor, antTrans)
# Serve in a thread as a daemon
antServer_thread = threading.Thread(target=antServer.serve)
# Exit the server thread when the main thread terminates
antServer_thread.daemon = True
antServer_thread.start()

# Starting the RX radio server
```

```
rxRadTrans = TSocket.TServerSocket(port=saPort+3)
rxRadProcessor = RControl.Processor(RHandler())
rxRadServer = TServer.TSimpleServer(rxRadProcessor, rxRadTrans)
# Serve in a thread as a daemon
rxRadServer_thread = threading.Thread(target=rxRadServer.serve)
# Exit the server thread when the main thread terminates
rxRadServer_thread.daemon = True
rxRadServer_thread.start()

# Starting the TX radio server
txRadTrans = TSocket.TServerSocket(port=saPort+4)
txRadProcessor = RControl.Processor(RHandler())
txRadServer = TServer.TSimpleServer(txRadProcessor, txRadTrans)
# Serve in a thread as a daemon
txRadServer_thread = threading.Thread(target=txRadServer.serve)
# Exit the server thread when the main thread terminates
txRadServer_thread.daemon = True
txRadServer_thread.start()

# Configuring the pre-amp for boot
paMinGain = 0.0
paMaxGain = 30.0
paIsPowered = False
paGain = 3.0

# Configuring the amp for boot
ampMinGain = 0.0
ampMaxGain = 130.0
ampIsPowered = False
ampGain = 3.0

# Starting the pre-amp server
paTrans = TSocket.TServerSocket(port=saPort+5)
paProcessor = PAControl.Processor(PAHandler(paMinGain,
                                            paMaxGain,
                                            paIsPowered,
                                            paGain))
paServer = TServer.TSimpleServer(paProcessor, paTrans)
# Serve in a thread as a daemon
paServer_thread = threading.Thread(target=paServer.serve)
# Exit the server thread when the main thread terminates
paServer_thread.daemon = True
paServer_thread.start()

# Starting the amp server
ampTrans = TSocket.TServerSocket(port=saPort+6)
ampProcessor = AmpControl.Processor(AmpHandler(ampMinGain,
                                            ampMaxGain,
                                            ampIsPowered,
                                            ampGain))
ampServer = TServer.TSimpleServer(ampProcessor, ampTrans)
# Serve in a thread as a daemon
ampServer_thread = threading.Thread(target=ampServer.serve)
# Exit the server thread when the main thread terminates
```

```
        ampServer_thread.daemon = True
        ampServer_thread.start()

        # Starting the CPU server
        cpuTrans = TSocket.TServerSocket(port=saPort+7)
        cpuProcessor = CPUControl.Processor(CPUHandler())
        cpuServer = TServer.TSimpleServer(cpuProcessor, cpuTrans)
        # Serve in a thread as a daemon
        cpuServer_thread = threading.Thread(target=cpuServer.serve)
        # Exit the server thread when the main thread terminates
        cpuServer_thread.daemon = True
        cpuServer_thread.start()

        # Ready to go, can now send RX data as received

        currPacketID = 1

        # Send received data forever
        while(True): # Normal test would be until resource allocation time is up
               sleep(1)
               packet = PacketService.Packet(packetID=currPacketID,
                                   data="Current RX packet you" + \
                                   " are receiving: %d" % currPacketID)

               try:
                      msg = rxPacketClient.SendPacket(packet)
               except PacketService.PacketSizeException as pse:
                      print("Packet size exception received in attempt to" + \
                             " send a received packet")
                      print(pse)
               currPacketID = currPacketID + 1

if __name__ == "__main__":
       main(sys.argv[1:])
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

Abernethy, R. (2014). *The programmer's guide to Apache Thrift.* Shelter Island, NY: Manning.

Apache Software Foundation. (2014). *Apache Thrift - Thrift types*. Retrieved November 07, 2014, from apache.org: https://thrift.apache.org/docs/types

Bachmann, F., Bass, L., Clements, P., Garlan, D., Ivers, J., Little, R. … Stafford, J. (2002). *Documenting software architectures: Documenting interfaces.* Pittsburgh, PA: Carnegie Mellon Software Engineering Institute.

Consultative Committee for Space Data Systems. (2001). *Overview of Space Link protocols.* Oxfordshire, UK: CCSDS Secretariat.

Consultative Committee for Space Data Systems. (2005). *Cross support reference model - Part 1: Space Link Extension services.* Washington, DC: CCSDS Secretariat.

Corliss, W. (1967). *The evolution of the Satellite Tracking and Data Acquisition Network (STADAN).* Greenbelt, Maryland: NASA, Goddard Space Flight Center.

Corliss, W. (1974). *Histories of the Space Tracking and Data Acquisition Network (STADAN), the Manned Space Flight Network (MSFN), and the NASA Communication Network (NASCOM).* NASA.

Cutler, J. (2004). Global CubeSat operations. *CubeSat Developers' Workshop.* San Luis Obispo, CA: Stanford University. Retrieved November 07, 2014, from cubesat.org: http://www.cubesat.org/images/cubesat/presentations/DevelopersWorkshop2004/1c_stanford.pdf

Cutler, J. (2004). Ground Station Markup Language. *IEEE Aerospace Conference Proceedings*, *5*, 3337–3343. doi: 10.1109/AERO.2004.1368140

Cutler, J., Linder, P., & Fox, A. (2002). A federated ground station network. *AIAA SpaceOps Conference.* doi: 10.2514/6.2002-T2-72

Darrin, A., & O'Leary, B. (2009). *Handbook of space engineering, archaelogy, and heritage.* Boca Raton, FL: CRC Press.

DuVander, A. (2010, July 09). *New job requirement: Experience building RESTful APIs*. Retrieved October 04, 2014, from ProgrammableWeb:

http://www.programmableweb.com/news/new-job-requirement-experience-building-restful-apis/2010/06/09

Elbert, B. (2001). *The satellite communication ground segment and earth station handbook.* Norwood, MA: Artech House.

Erl, T. (2005). *Service-oriented architecture: Concepts, technology, and design.* Upper Saddle River, NJ: Prentice Hall.

Gordon, G., & Morgan, W. (1993). *Principles of communications satellites.* New York, NY: John Wiley & Sons.

Henning, M. (2006). The rise and fall of CORBA. *ACMQueue*, *4*(5), 28–34. Retrieved November 17, 2014 from acm.org: http://queue.acm.org/detail.cfm?id=1142044

Leffke, Z. (2013). *Distributed ground station network for CubeSat communications.* Blacksburg, VA: Virginia Polytechnic Institute and State University.

Leveque, K., Puig-Suari, J., & Turner, C. (2007). Global Educational Network for Satellite Operations (GENSO). *AIAA/USU Conference on Small Satellites.* Retrieved November 07, 2014 from usu.edu: http://digitalcommons.usu.edu/cgi/viewcontent.cgi?article=1506&context=smallsat

Minelli, G., Ibbitson, P., Felt, A., Yzquierdo, E., Horning, J., Rigmaiden, D. … Newman, J. (2012). Mobile CubeSat Command and Control (MC3) ground stations. *2012 Annual Summer CubeSat Developers' Workshop.* Retrieved November 15, 2014 from cubesat.org: http://www.cubesat.org/images/cubesat/presentations/SummerWorkshop2012/Day_2/1145_Giovanni_Minelli.pdf

Miyashita, N., Nakaya, K., Ui, K., & Matunaga, S. (2003). Internet and XML-based extensible and low-cost ground station system. *International Astronautical Congress of the International Astronautical Federation, the International Academy of Astronautics, and the International Institute of Space Law, 54.* doi: 10.2514/6.IAC-03-U.2.a.06

NASA, G. S. (2010). *Near Earth Network (NEN) users' guide.* Greenbelt, MD: NASA.

Python Software Foundation. (n.d.). *11.1.pickle - Python object serialization - Python 2.7.8 documentation*. Retrieved November 14, 2014, from Python.org: https://docs.python.org/2/library/pickle.html

Royal Pingdom. (2010, October 15). *REST in peace, SOAP*. Retrieved November 3, 2014, from Pingdom: http://royal.pingdom.com/2010/10/15/rest-in-peace-soap/

RSA Laboratories. (2000). *PKCS #5: Password-Based Cryptography Specification version 2.0.* Bedford, MA: RSA Laboratories.

Sakamoto, Y. (2009, 01 14). *UNISEC GSN-WG*. Retrieved November 20, 2014, from Tohoku University: http://www.astro.mech.tohoku.ac.jp/~gsn/en/

Shirville, G., & Klofas, B. (2007). GENSO: A global ground station network. *AMSAT Conference Proceedings.* Retrieved October 24, 2014 from klofas.com: http://www.klofas.com/papers/AMSAT_2007.pdf

Stallings, W., & Brown, L. (2008). *Computer security: Principles and practice.* Upper Saddle River, NJ: Pearson Education.

TIOBE Software. (2014, October). *TIOBE Software: Tiobe Index*. Retrieved October 2, 2014, from TIOBE: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

Tubio, R., Vazquez, A. J., Puig, J., Kurahara, N., & Bellardo, J. (2014). The SATNet Project: Towards an open-source ground station network for CubeSats. *Annual Spring CubeSat Developers' Workshop.* Retrieved October 13, 2014 from: http://mstl.atl.calpoly.edu/~bklofas/Presentations/DevelopersWorkshop2014/Tubio_SATNet.pdf

Union of Concerned Scientists. (2014, August 01). *UCS Satellite Database*. Retrieved October 26, 2014, from Union of Concerned Scientists: http://www.ucsusa.org/nuclear_weapons_and_global_security/solutions/space-weapons/ucs-satellite-database.html

W3C Woking Group. (2007, April 27). *SOAP Version 1.2 Part1: Messaging Framework (Second Edition)*. Retrieved November 05, 2014, from W3C: http://www.w3.org/TR/2007/REC-soap12-part1-20070427/

W3C Working Group. (2004, February 11). *Web Services Architecture*. Retrieved October 27, 2014, from W3.org: http://www.w3.org/TR/ws-arch/

Wertz, J., Everett, D., & Puschell, J. (2011). *Space Mission Engineering: The New SMAD.* Hawthorne, CA: Microcosm Press.

ZeroC Inc. (n.d.). *ZeroC - Ice vs. SOAP*. Retrieved November 24, 2014, from ZeroC.com: http://www.zeroc.com/iceVsSoap.html

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California