Carleton University

COMP 4905

Network Management for Software Defined Radio Applications

Zach Renaud

supervised by

Prof. Michel Barbeau, School of Computer Science

April 13, 2016

0.1 Abstract

In this project, a GNU Radio component is extended to support remote management of its internal state, using the popular and well supported Simple Network Management Protocol. The specific component of GNU Radio in focus for this project is a Location-free Link State Routing (LLSR) extension that provides wireless routing through radio technologies.[1] For this added support, a Python based SNMP agent is used above the NET-SNMP library in conjunction with a translation and state management layer in order to handle incoming requests that retrieve or modify the state of variables used within the routing software. Using the integrated SNMP functionality, the state of the GNU Radio module can be accessed and modified through pre-packaged client side tools on popular platforms and distributions, including Windows, Mac OSX, and Linux. With the additions made for this project, GNU Radio applications can be better managed in development and remote production environments.

0.2 Acknowledgments

I would like to express my sincere appreciation for Professor Michel Barbeau's guidance and overall participation in this project as well as his contributions of knowledge, making the end result possible.
I would also like to thank Neal Charbonneau and Meir Tseitlin for publishing their work on a pySNMP agent example.

[1] Barbeau, Michel, Stephane Blouin, Gimer Cervera, Joaquin Garcia-Alfaro, and Evangelos Kranakis. "Michelbarbeau/gr-llsr: Location-free Link State Routing (LLSR) Implementation for GNU Radio." GitHub. Accessed April 3, 2016. https://github.com/michelbarbeau/gr-llsr.

Contents

List of Figures

List of Tables

Chapter 1

1.1 Introduction

The current state of our technology facilitates interconnecting so many devices and people. A consequence of such rich and expansive networking capabilities is the deployment of systems far outside of our physical grasp, where we nonetheless assume to have constant access to them. With these higher standards of service delivery comes a common need for managing networked services and systems in generic ways. A network management protocol allows a user to remotely connect to and monitor the state of a system, querying certain components of the system and modifying the running system's state. Using a management protocol, a system administrator is able to check in on a service or device and make potentially important changes as circumstances require. This can be used for tasks varying from metric gathering to maintaining the systems that people and companies depend on. The use of a good management protocol gives a lot of power to those administrating it, as well as a sense of security that no matter the physical distance between us and our software, our high demands for accessibility will continue to be met. This project will addresses the importance of network management integration into vital software services with a focus on one particular application, discussed in detail throughout this report.

1.1.1 Background

The second technology of focus for this project is GNU radio, an open source tool kit that provides an abstraction to various radio functions such as filters, channel codes, synchronization elements, equalizers, demodulators, vocoders, and decoders.[1] It also provides a framework for developing applications using radio resources. GNU Radio is implemented primarily in the Python programming language, with performance critical aspects such as the signal processing path implemented in C++.[2] This project will consist of integrating a ubiquitous network management protocol known as the the Simple Network Management Protocol (SNMP) into the GNU Radio framework, with the example application of focus being the Location-free Link State Routing (LLSR) module for GNU Radio. SNMP incorporates a set of requests that allow querying the state of the system in question, as well as updating the state of individual variables remotely. Requests can be made to poll the system, while responses can also be received asynchronously from the managed service to indicate the change in an observed variable. With SNMP, services running on managed devices are monitored by interactions with agents from other devices called managers that communicate over UDP on designated ports. The protocol communicates the state of variables, known as objects, by defining them each with a specific identifier (OID). A set of object identifiers along with their descriptions are contained in a repository known as a Management Information Base (MIB), which makes the hosts aware of the existence of the objects. A custom MIB is defined to allow SNMP clients to interface to the management services added by this project.

The GNU Radio application development framework in this case imposes a way of developing custom Out of Tree (OOT) modules in a way that fit with the larger ecosystem and can be integrated seamlessly into an existing control application.[3] The general structure of a GNU Radio application consists of a series of blocks that are connected and linked together at runtime. The way this works is by defining the created functions as sinks and sources of data, which are then relayed to the overall control system through installation of the Python files and loading of an XML file with the relevant information about the code.[4] This application design is quite modular, allowing externally developed and maintained projects to leverage the radio resources. One such

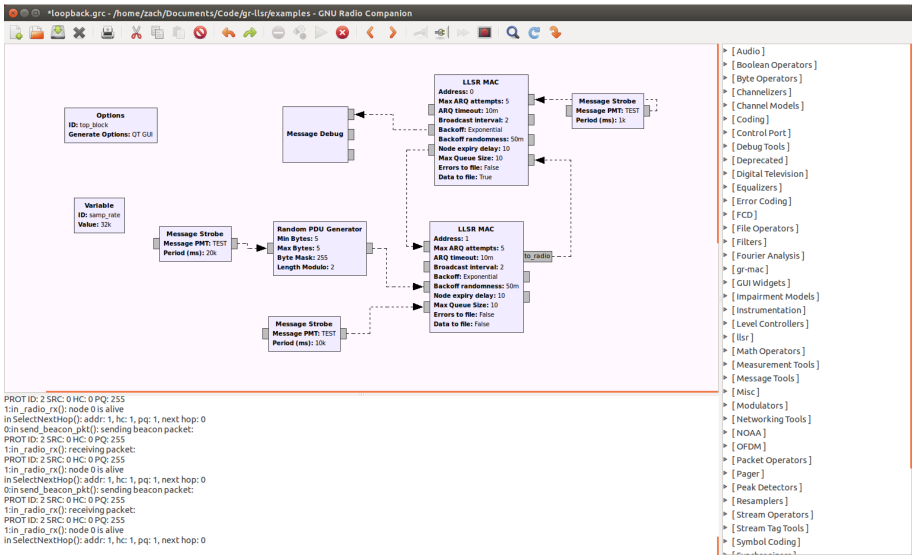
[1] Hu, Fei, and Sunil Kumar. Multimedia Over Cognitive Radio Networks: Algorithms, Protocols, and Experiments. 2015.

[2] "Overview - GNU Radio - Gnuradio.org," GNU Radio, accessed April 3, 2016, https://gnuradio.org/redmine/projects/gnuradio.

[3] "OutOfTreeModules - GNU Radio - Gnuradio.org," GNU Radio, accessed April 3, 2016, http://gnuradio.org/redmine/projects/gnuradio/wiki/OutOfTreeModules.

[4] "OutOfTreeModules - GNU Radio - Gnuradio.org," GNU Radio, accessed April 3, 2016, http://gnuradio.org/redmine/projects/gnuradio/wiki/OutOfTreeModules#Making-your-blocks-available-in-GRC.

control application is the GNU Radio Companion program available on Linux. Using this program, an extension is loaded by selecting the configuration XML file, which renders the specific module's blocks in a diagram illustrating how they are interconnected.[5] This is a great resource for testing a custom module to see the flow of data and signals through the program. An example of how this appears for the LLSR application can be seen in Fig. 1.1.



Options
ID: top_block
Generate Options: QT GUI

Variable
ID: samp_rate
Value: 32k

Message Strobe
Message PMT: TEST
Period (ms): 20k

Random PDU Generator
Min Bytes: 5
Max Bytes: 5
Byte Mask: 255
Length Modulo: 2

Message Strobe
Message PMT: TEST
Period (ms): 10k

LLSR MAC
Address: 0
Max ARQ attempts: 5
ARQ timeout: 10m
Broadcast interval: 2
Backoff: Exponential
Backoff randomness: 50m
Node expiry delay: 10
Max Queue Size: 10
Errors to file: False
Data to file: True

Message Strobe
Message PMT: TEST
Period (ms): 1k

LLSR MAC
Address: 1
Max ARQ attempts: 5
ARQ timeout: 10m
Broadcast interval: 2
Backoff: Exponential
Backoff randomness: 50m
Node expiry delay: 10
Max Queue Size: 10
Errors to file: False
Data to file: False

```
PROT ID: 2 SRC: 0 HC: 0 PQ: 255
1:in _radio_rx(): node 0 is alive
in SelectNextHop(): addr: 1, hc: 1, pq: 1, next hop: 0
0:in send_beacon_pkt(): sending beacon packet:
PROT ID: 2 SRC: 0 HC: 0 PQ: 255
1:in _radio_rx(): receiving packet:
PROT ID: 2 SRC: 0 HC: 0 PQ: 255
1:in _radio_rx(): node 0 is alive
in SelectNextHop(): addr: 1, hc: 1, pq: 1, next hop: 0
0:in send_beacon_pkt(): sending beacon packet:
PROT ID: 2 SRC: 0 HC: 0 PQ: 255
1:in _radio_rx(): receiving packet:
PROT ID: 2 SRC: 0 HC: 0 PQ: 255
1:in _radio_rx(): node 0 is alive
in SelectNextHop(): addr: 1, hc: 1, pq: 1, next hop: 0
```

Figure 1.1: GNU Radio Companion with LLSR

1.1.2 Problem

The integration of technologies and applications brings with it a set of problems. The most obvious being the method by which the systems in question are made to interact, and to what degree the existing APIs or architectures facilitate this interaction. A more subtle but equally valid problem to be addressed is the design of the greater integrated system. This problem follows logically the first in that the primary matter of importance is how to make these technologies work together, with the second focusing on achieving this in such a way that reflects sound software engineering practices and designs.

[5] "GNURadioCompanion - GNU Radio - Gnuradio.org," GNU Radio, accessed April 3, 2016, https://gnuradio.org/redmine/projects/gnuradio/wiki/GNURadioCompanion#Adding-Custom-Blocks.

1.1.3 Result

The integration of the Simple Network Management Protocol in the GNU Radio project is done using modular extensions between LLSR and a custom SNMP agent process. The added modules handle messages of two basic types: getting, and setting variable values. The SNMP Agent handles parsing of the OID tree, reducing the communication details into a simplified format which is relayed to the state manager server module extending the GNU Radio application. This allows SNMP state management with very little modification to the existing code base. The result of this project is a remotely manageable routing system and a design that easily extends to other GNU Radio applications as well as other general software systems.

1.1.4 Outline

This report discusses the research done on the various technologies incorporated in this project, as well as detailing the design considerations made throughout the research and development phases. Specifically, Section 2.1 introduces the motivation for this project, suggesting the importance of this work and the larger picture of which it is a part. Section 2.2 provides the high level design, as well as the low level implementation details for the integration of the relevant technologies and software systems. Section 2.3 elaborates on the results achieved in this project, describing use cases and highlighting important features. Section 3.1 closes by summarizing the work done throughout the project, mentioning some of the future work possible on the current implementation and design, as well as discussing where there is room for improvement. The report concludes by briefly highlighting some achievements of the project in the context of the broader goals.

Chapter 2

2.1 Motivation

The motivation for adding network management support for the link state routing application and other GNU Radio applications is to allow remotely observing and modifying the attributes and configurations of a very dynamic system in a potentially mission critical environment. Given one of the use cases for LLSR being in underwater routing, the uses may range from research to critical communication technologies. While particularly in the latter, both cases would benefit greatly from the ability to check in on current processes, control flow, and internal state, including variables such as the number of packets transmitted, the failure and re-transmission counts, bytes received, channel state, and other networking information. As seen in Fig. 1.1, the GNU Radio Companion application provides useful information about the system as it runs, making it very convenient for developing and debugging the program. However, once deployed, this information is no longer available. There is is of course the possibility that it will run successfully in the field, once tested in development, although it is not always the case that testing in a controlled environment reflects operations at scale, in a production setting. In order to perform tests remotely and check on the system, as well as to satisfy the general curiosity of how the system operates, network management will become a valuable tool.

Once deployed successfully, the routing software can communicate with the target devices in the field, while a system manager is able to remotely check into the state of the system to detect any potential failures or alarming symptoms, as well as update node addresses and other definitions as needed. The use of such integrated functionality should greatly reduce the efforts in configuration and long term operation of deployed routing systems, benefiting from an established management protocol used by mature products and tools such as those provided by the NET-SNMP Linux package, including snmpget, snmpset, and snmpwalk.[1] The benefit of using a common protocol is that there are a number of client applications to choose from that represent the data in useful ways. Furthermore, a seasoned system administrator would be able to forgo the learning curve associated with remotely managing the LLSR application with previous experience using SNMP, and the MIB to add to their client-side application. The management framework's use of a MIB also facilitates operating a potentially large collections of systems, whereby each is monitored by the same means, where one person can oversee the many systems through a single interface.

2.2 Methodology

2.2.1 Design

High Level Design

There were various design choices to be made in the process of integrating the SNMP functionality into LLSR. The main choices involved were based on how to extend the SNMP agent to understand the state of the GNU Radio software, as well as how to integrate the agent into the LLSR software. Research was done into the open source documentation for NET-SNMP, which is the standard Linux implementation of the SNMP protocol.[2] The first option considered was building a shared object binary file that would be configured into the running SNMP Agent and, with a custom MIB would allow handling all requests for objects descending from a certain node in the identifier tree. With a proof of concept implemented, the next consideration would be on how to communicate between the SNMP Agent and the LLSR module for managing the state. Before this was done, however, it became clear that the C code for this shared object is not ideal, as it is very verbose and contains much boiler plate code for the MIB interaction. The decision was made to instead seek out a Python implementation of the NET-SNMP interface. A good Python module called pySNMP was found which supports MIB parsing to generate the OID tree, as well as GET, SET,

[1]Net-SNMP, accessed April 3, 2016, http://www.net-snmp.org/.
[2]Net-SNMP, accessed April 3, 2016, http://www.net-snmp.org/.

and TRAP requests.[3] Given that the LLSR module is written in Python, this option was selected and development continued in this direction.

Using pySNMP, the custom Python SNMP agent has been written as a standalone program, rather than as part of the GNU Radio project. Whereas the alternative method would be straight to the point of this project, it would entail starting up an SNMP server process or thread from within the unrelated core functionality of the radio software. The result of which would be an efficient, yet highly coupled design, that would ignore principles of software design pertaining to single responsibilities, as extrapolated to the processes themselves.[4] The separate, custom SNMP agent therefore requires a different method of communication with the core radio in order to manage its state. Given multiple options, the chosen communication method for this task is TCP/IP, implemented in Python modules entitled grStateManager, and grStateRequester. These modules provide a monolithic TCP server, and client respectively, with a basic set of supported messages for getting and setting the value of a given variable within the managed object. This design is slightly redundant, as it essentially provides a communication protocol, albeit basic, from the agent for another communication protocol (SNMP). However, the benefit of this design is that the managed object need not be aware of any management protocol or the high level intentionality. Instead, it only has the concept of being "managed" by the lightweight grStateManager module, where it instantiates a StateManagerServer object and periodically calls the handle_request method. The server instance is created with a reference to the managed object, offloading all state management out of the core functionality. The high level block design of the interaction between these components is shown in the sequence diagram in Fig 2.1. It also illustrates the ordering of the communication between the processes and modules.

[3] "Library Reference PySNMP," SNMP Library for Python PySNMP, accessed April 3, 2016, http://pysnmp.sourceforge.net/docs/api-reference.html.

[4] "Single Responsibility Principle — Object Oriented Design," Design Patterns — Object Oriented Design, accessed April 3, 2016, http://www.oodesign.com/single-responsibility-principle.html.

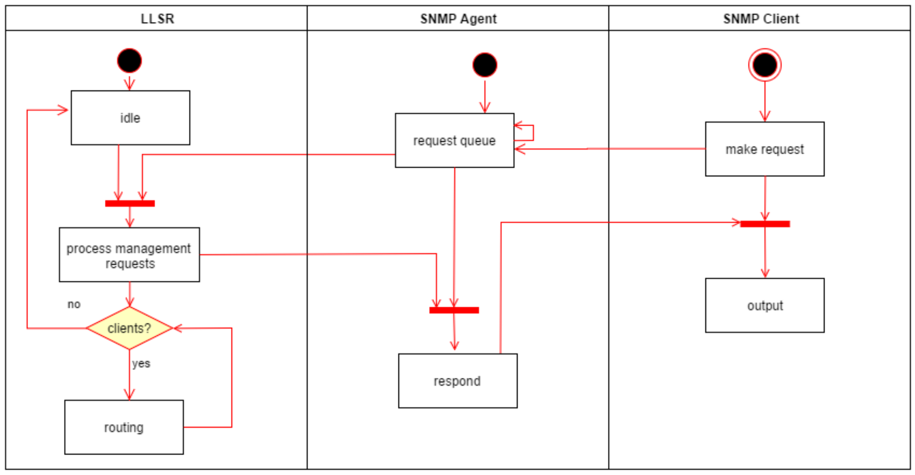


Figure 2.1: GNU Radio + SNMP Sequence Diagram

The connection of the various hosts, packages, and components in use for this integration project can be seen in the component diagram in Fig. 2.2. This shows how the LLSR module is packaged with the grStateManager, as well as the SNMP Agent and the grStateRequester. While the figure depicts the managed service and SNMP layers on the same host, the use of a TCP/IP communication steam between these packages means that they can be distributed between hosts.

The MIB

The Management Information Base is the static representation of the state that is being managed for a particular system. It contains its own syntax so that it can be parsed programatically by various SNMP libraries and referenced by client side applications as well. The MIB is composed in a tree structure, where each Object IDentifier (OID) is represented as a chain, or series of identifiers, delimited by periods. The significance of this syntactic structure is that each object provides the list of its ancestors in the OID tree, back to the root object.[5] This method of declaring a management variable is also convenient for future growth, as further subdivisions can be branched off of any given node.

[5] "SNMP Management: OIDs, MIBs, Traps, Notifications & More," Network Management Software - Reviews & Network Monitoring Tools, accessed April 3, 2016, http://www.networkmanagementsoftware.com/snmp-tutorial-part-2-rounding-out-the-basics/.

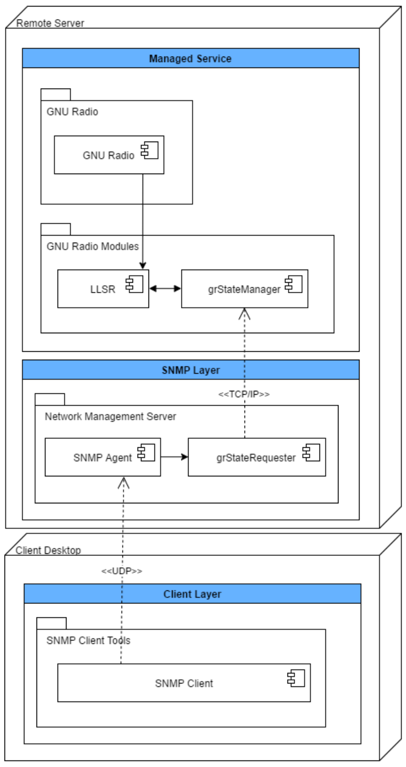


Figure 2.2: Component Diagram

Similarly to how human readable host names are used as aliases for IPv4 and IPv6 address identifiers in the Internet Protocol, a MIB offers similar labeling for the OIDs of interest. It also provides additional information such as syntax, modifier types, and a description for each of the objects. The SNMP protocol requires the presence of a MIB in order to successfully handle communication between the client and daemon processes.[6] Therefore, creating this static object database is a crucial task for adding SNMP support where none currently exists. Learning the syntax has a moderate learning curve, containing a wide range of syntactic structures.[7] However, given the ubiquity of the protocol, there exist static parsers or linters for validating a given MIB file, with the example of smilint from the libsmi package.[8] The use of such a tool is highly advisable to those unfamiliar with the details of the language, to provide instructional

[6]https://tools.ietf.org/html/rfc1157#section-3.2.6

[7]https://tools.ietf.org/html/rfc3418#page-2

[8]"Smilint," Institute of Operating Systems and Computer Networks, accessed April 3, 2016, https://www.ibr.cs.tu-bs.de/projects/libsmi/smilint.html.

feedback regarding the syntax.

The starting point for creating a new MIB is choosing the OID node to branch off of. Once branched, all future additions can descend off of a domain specific root node, defined in the local MIB. For this project, the MIB file was named GNU-RADIO, and consists of most of the state variables within the LLSR's llsr_mac class. The starting point for the module's OID tree is off of the net-SnmpPlaypen module, provided for private, proof of concept works that have not yet become mainstream or publicly available.[9] This would ordinarily be replaced by an organization name during official adoption into a package. The first definition in the file is the module identity tag, called gnuRadio. The remainder of the MIB defines 12 variables under the gnuRadio branch. The full set of identifiers defined in the GNU-RADIO MIB are described in table 2.1.

State Requester

The two light weight modules added between the SNMP Agent and the LLSR program maintain a very simple protocol. As previously mentioned, this protocol operates on top of TCP/IP, and consists of a single byte message type header preceding each request, followed by the request specific payload. There are currently only two types of requests implemented, both initiated by the client side, the SNMP agent, to the server side embedded in LLSR. The first is a GET request. This indicate that the requester object would like to reference the value of some internal state in the managed object. Following the 8-bit integer indicating the type of request, the client sends over the length of the variable name in bytes, as a 4-byte unsigned integer, followed by the UTF-8 string of the variable name being requested. The response to a GET request is the string length of the value being sent back encoded as a string, followed by the value itself. In the context of network administration, it is acceptable for all data types to be encoded as strings for the simplification of the communication process, as the clients themselves merely display the results to the user. If ever a client extension were added which required the datatype that reflects the object instance, it would have access to this information in the objects MIB definition.

The second type of request is a SET, which comes in two forms: SETINT, and SETSTR for integer and string data types respectively. Similarly, the requester

[9]Net-SNMP, accessed April 3, 2016, http://www.net-snmp.org/docs/mibs/NET-SNMP-MIB.txt.

Label	**Identifier**	**Type**	**Encoding**	**Summary**
gnuRadio	netSnmpPlaypen.1	MODULE-IDENTITY	–	The root for the MIB module
nodeAddr	gnuRadio.1	OBJECT-TYPE	STRING	The address of the node
packetCount	gnuRadio.2	OBJECT-TYPE	UINT32	The current packet count
arqCount	gnuRadio.3	OBJECT-TYPE	UINT32	The number of transmitted ARQ packets
rearqCount	gnuRadio.4	OBJECT-TYPE	UINT32	The number of re-transmitted ARQ packets
failedARQ	gnuRadio.5	OBJECT-TYPE	UINT32	The number of failed ARQ re-transmissions
maxRetry	gnuRadio.6	OBJECT-TYPE	UINT32	The maximum number of re-transmissions allowed
bytesRecv	gnuRadio.7	OBJECT-TYPE	UINT32	The number of bytes received so far
channelState	gnuRadio.8	OBJECT-TYPE	INT32	The current state identifier of the channel
ackNumber	gnuRadio.9	OBJECT-TYPE	INT32	The current expected ACK number
retransmissionTimeout	gnuRadio.10	OBJECT-TYPE	STRING	The timeout value for a re-transmission
expBackoff	gnuRadio.11	OBJECT-TYPE	STRING	True if exponential backoff is enabled
expBackoffPerc	gnuRadio.12	OBJECT-TYPE	STRING	The percentage used in exponential backoff

Table 2.1: GNU-RADIO MIB Definition

will first send the 8-bit message header indicating the type of request, followed by the length and name of the variable, and finally by the new state of the variable to be set. The value of integer type variables is sent using 4 bytes in big-endian format. The value for string variables is delivered in similar fashion to other strings in the protocol, by first sending the length as a 4 byte unsigned int, followed by the string itself.

State Manager

The StateManager class is the counterpart to the StateRequester. It is what handles the requests over TCP/IP on a designated port, 8585, under the simplistic protocol outlined in the State Requester section. This class inherits from the Python standard library's SocketServer.BaseRequestHandler, making it a specialized request handler for TCP connections. It contains a managedObject variable on which it uses Python's built-in getattr and setattr functions to get and set the state for incoming requests. Also provided in the grStateManager module is another class called StateManagerServer. This class inherits from SocketServer.TCPServer, and other than a custom constructor, it has no methods. It is simply a specialized TCP server defined to have a zero second timeout when handling requests on an empty queue, which also sets the request handler to the class mentioned above, with the managed object passed into the server's constructor. The flow of events to have a state manager for a particular object is thus to instantiate grStateManager.StateManagerServer, passing in the management object. Once created, the server will start, after which point the managed object is able to accept any management to occur with a call to the handle_request method for the instance. The interaction between GNU Radio's LLSR and the State Manager can be seen in more detail in Fig. 2.3.[10]

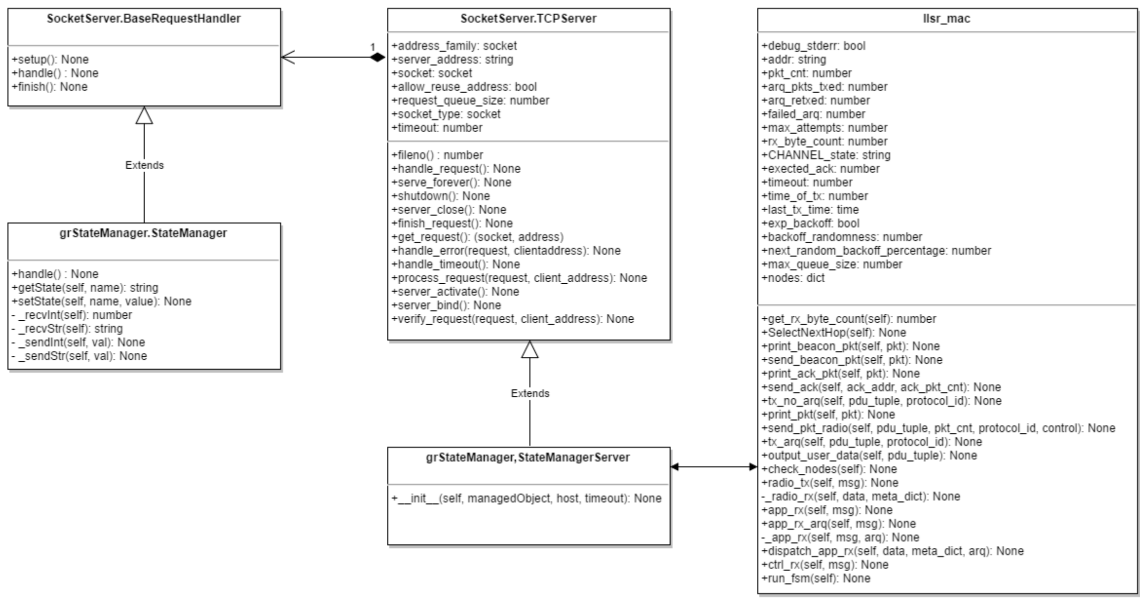

SocketServer.BaseRequestHandler
+setup(): None
+handle() : None
+finish(): None
Extends
grStateManager.StateManager
+handle() : None
+getState(self, name): string
+setState(self, name, value): None
- _recvInt(self): number
- _recvStr(self): string
- _sendInt(self, val): None
- _sendStr(self, val): None
1
SocketServer.TCPServer
+address_family: socket
+server_address: string
+socket: socket
+allow_reuse_address: bool
+request_queue_size: number
+socket_type: socket
+timeout: number
+fileno() : number
+handle_request(): None
+serve_forever(): None
+shutdown(): None
+server_close(): None
+finish_request(): None
+get_request(): (socket, address)
+handle_error(request, clientaddress): None
+handle_timeout(): None
+process_request(request, client_address): None
+server_activate(): None
+server_bind(): None
+verify_request(request, client_address): None
Extends
grStateManager,StateManagerServer
+__init__(self, managedObject, host, timeout): None
llsr_mac
+debug_stderr: bool
+addr: string
+pkt_cnt: number
+arq_pkts_txed: number
+arq_retxed: number
+failed_arq: number
+max_attempts: number
+rx_byte_count: number
+CHANNEL_state: string
+exected_ack: number
+timeout: number
+time_of_tx: number
+last_tx_time: time
+exp_backoff: bool
+backoff_randomness: number
+next_random_backoff_percentage: number
+max_queue_size: number
+nodes: dict
+get_rx_byte_count(self): number
+SelectNextHop(self): None
+print_beacon_pkt(self, pkt): None
+send_beacon_pkt(self, pkt): None
+print_ack_pkt(self, pkt): None
+send_ack(self, ack_addr, ack_pkt_cnt): None
+tx_no_arq(self, pdu_tuple, protocol_id): None
+print_pkt(self, pkt): None
+send_pkt_radio(self, pdu_tuple, pkt_cnt, protocol_id, control): None
+tx_arq(self, pdu_tuple, protocol_id): None
+output_user_data(self, pdu_tuple): None
+check_nodes(self): None
+radio_tx(self, msg): None
-_radio_rx(self, data, meta_dict): None
+app_rx(self, msg): None
+app_rx_arq(self, msg): None
-_app_rx(self, msg, arq): None
+dispatch_app_rx(self, data, meta_dict, arq): None
+ctrl_rx(self, msg): None
+run_fsm(self): None

Figure 2.3: LLSR State Management Class Diagram

[10] "20.17. SocketServer A Framework for Network Servers Python 2.7.11 Documentation," Overview Python 3.5.1 Documentation, accessed April 3, 2016, https://docs.python.org/2/library/socketserver.html.

SNMP Agent

The custom SNMP Agent, called grSnmpAgent, is built on top of the pySNMP library for NET-SNMP, which natively handles incoming SNMP requests. The base boilerplate code used in this project for the agent's foundation originates from a blog post by Neal Charbonneau on the implementation of python based SNMP agent, with extended support for the SET operation from Meir Tseitlin.[11][12] The MIB definitions are compiled into a Python module of the same name using a tool in the pySNMP package called build-pysnmp-mib. The project-specific code written for the SNMP Agent consists mostly of registering, and implementing user-defined handler functions for both GET and SET operations on each of the desired objects from the loaded MIB. The handler functions are contained within their own class, which holds onto a local grStateRequester object to which incoming SNMP requests are delegated, once parsed from the protocol by the base code. The design is such that each of the handler functions is a simple one-line function which calls one of get, setInt, or setStr methods on the requester instance for the specified object name.

One of the design goals for the grSnmpAgent is to minimize the logic performed at the SNMP Agent layer. Similarly to LLSR code, the goal is to avoid interleaving custom, domain-specific code with third party or boilerplate code. To accomplish this, all of the logic is moved out of the agent and into the grStateRequester module, leaving only hooks between them in the agent's Python file, which are themselves very small. Using this approach, the SNMP Agent is left to handle interpreting the incoming protocol from the end user client, effectively decoupling the original protocol with the way that the end attributes are referenced and modified. The result of this design choice is an independent implementation of the management concepts, allowing easily extending the support to other protocols in the future. While SNMP is the target protocol for this project, the aforementioned design would make the work of supporting another management protocol a simple plug-and-play exercise.

[11]Neal Charbonneau, www.nealc.com/blog/blog/2013/02/23/writing-an-snmp-agent-with-a-custom-mib-using-pysnmp/.

[12]Meir Tseitlin, "Developing PySNMP Based Agent with Custom MIBCloud Rocket," Cloud Rocket, accessed April 3, 2016, http://www.cloud-rocket.com/2013/08/writing-custom-mib-for-pysnmp-agent/.

Implementation Example

The following code snippets show the API of the management layer that handles retrieving and updating the state of the attributes being requested over the higher level protocol, in this case SNMP. The first block shows an example class called Test that is being managed. It has two attributes, v1 and v2, which are accessible to the StateManagerServer. This example also highlights a use case scenario not otherwise mentioned, namely operating the management server from outside of the management object. Whereas in the LLSR code, the server object is instantiated by the management class itself, if the framework permits, a controller object with access to the managed instance can handle requests for its state on its behalf.

```python
class Test():
    def __init__(self, v1, v2):
        self.v1 = v1
        self.v2 = v2

    def __str__(self):
        return ("{ %s, %d }" % (self.v1, self.v2))

def runTest():
    # Server connection details
    host, port = "0.0.0.0", grStateRequests.Protocol.PORT
    # The managed object
    test = Test("Hello", 4)
    # The state manager server instance
    server = grStateManager.StateManagerServer(test)

    while True:
        time.sleep(1)
        print("Doing work...")
        # Process state management requests on managed object
        server.handle_request()
        # Display the current state of the object
        print(test)

```

The State Requester client, in our case the SNMP Agent, will employ the following API in order to perform the management through the server code. In the case of this example, the runTest method defined above will handle one request at a time between sleeping. Therefore, its output around the time of the requests will be “{ Hello, 4 }”, “{ World! 4 }”, and finally “{ World! 5 }”.

```
# Create a requester, manage state of Test instance
requester = grStateRequester.StateRequester("localhost")
print("Getting v1: %s" % requester.get("v1")) # Prints "Hello"
print("Getting v2: %s" % requester.get("v2")) # Prints "4"
requester.setStr("v1", "World!")
requester.setInt("v2", 5)
print("Getting v1: %s" % requester.get("v1")) # Prints "World!"
print("Getting v2: %s" % requester.get("v2")) # Prints "5"

```

Flexibility

The high level design using the middle communication layer between the SNMP Agent and the managed module allows the system to be easily ported over to a different managed service. Under the current design, the incorporation of the manager requires simply instantiating the StateManager object from within the application and calling handle_request method. Any client can be made to perform the simple requests to the manager server over a network using the basic message types defined by the protocol.

Drawbacks

Admittedly, the monolithic nature of the request handler server has some undesirable consequences, such as requiring a call to the handle_requests method in a main loop. While the name of this method retains the generality that is intended with the separation of duties, it also requires the LLSR module to be more explicitly aware of the server that it is running. A potentially preferable solution to this design would be for the grStateManager module's StateManagementServer to create a new thread, in which it concurrently handles state management requests. The reason for not implementing this solution is the fact that the multi-threaded design would make locking a necessity for all state held by the managed object. This is not necessarily a bad thing, though it means introducing added complexity throughout the core module since there are many references to internal state throughout the module. Sufficient locking would entail one lock object per instance in the managed object to avoid unnecessary losses in performance. Given that the nature of the software under management is routing oriented, the speed at which it operates is an important factor that cannot be sacrificed for simplicity of design.

Another drawback that currently exists is in the form of error handling. While the components themselves aspire to be robustly implemented, the fact that the SNMP Agent and GNU Radio are communicating through a hidden intermediary protocol means that propagating error messages requires some translation between the protocols. While definitely possible, the Mix-in style framework of the pySNMP API makes it slightly difficult to do so. The API allows registering a handler for SNMP GET and SET requests. However, the handler method passed to the API must return the value that was received during a GET request, and is handled within the library itself for the SNMP GET response.[13] It would be possible to integrate an exception that can be thrown in the handler and caught by the library to indicate an error to propagate to the end user. Currently, the solution is to simply log the error and return a default, zero string.

One annoyance of the implementation of this project's goals is in the maintenance of the management objects. It is rather simple to update the MIB and re-compile the pySNMP MIB module independently of the rest of the system, however due to the requirement of camelCase variables in the MIB, the strings do not match the variable names in the managed object.[14] Therefore, there is a need to update the individual attribute name strings mapped in the grSnmpAgent at the same time.

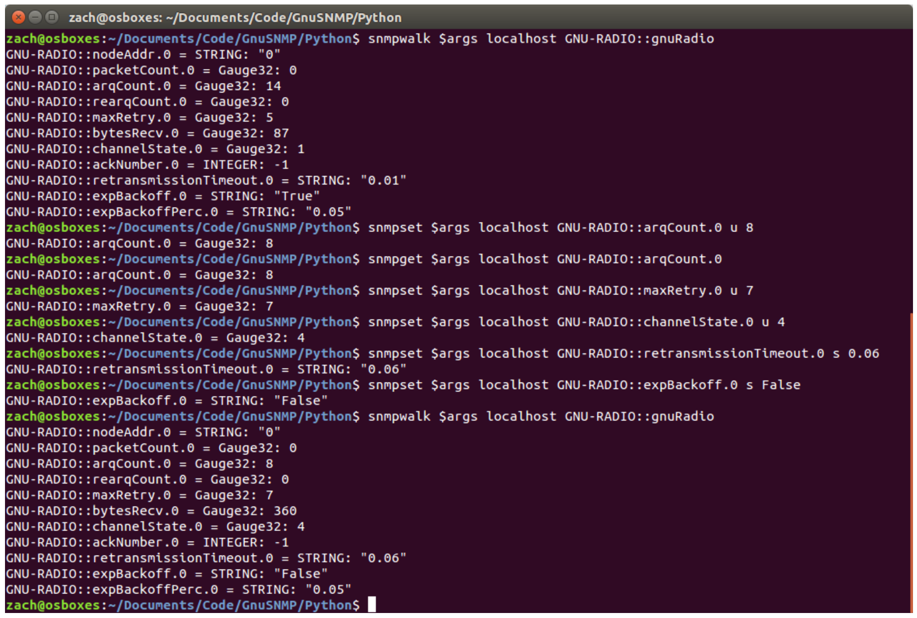
2.3 Results

The high level, end results of this project is added support for the Simple Network Management Protocol within the LLSR module of GNU Radio. This work achieved a custom Management Information Base for LLSR, with an object identifier structure that establishes GNU Radio as a branch in the chain. It provides labels, data types, and descriptions of each of the objects under

[13]Neal Charbonneau, www.nealc.com/blog/blog/2013/02/23/writing-an-snmp-agent-with-a-custom-mib-using-pysnmp/.

[14]"17 Common MIB Design Errors - MIB Smithy User's Guide — Muonics, Inc," Network Management Software, Services & Consulting — Muonics, Inc, accessed April 3, 2016, http://www.muonics.com/Docs/MIBSmithy/UserGuide/commonerrors.php#underscore.

management within the routing module. Using the pySNMP library, a custom SNMP Agent was built by compiling the MIB into Python code through a utility from the same package, adding custom handler functions which interact with the grStateRequester module. The state requester and manager modules handle communication between the SNMP Agent and the GNU Radio component under management. These modules were also written in Python and benefit from a very simple implementation and basic design. The accomplishments of combining these various components is a system that can be remotely and easily queried for the state of individual variables, bulk variables, and updated for the value of any of the managed variables. Given the SNMP protocol's popularity, administration of the LLSR software can be done with different user interfaces and well-defined scripts of requests that translate to the simple GET and SET commands issued between the agent and the radio driver. Fig. 2.4 demonstrates how the basic command line utilities in the NET-SNMP package on Linux will work to traverse the managed state as well as update individual variables with new values.

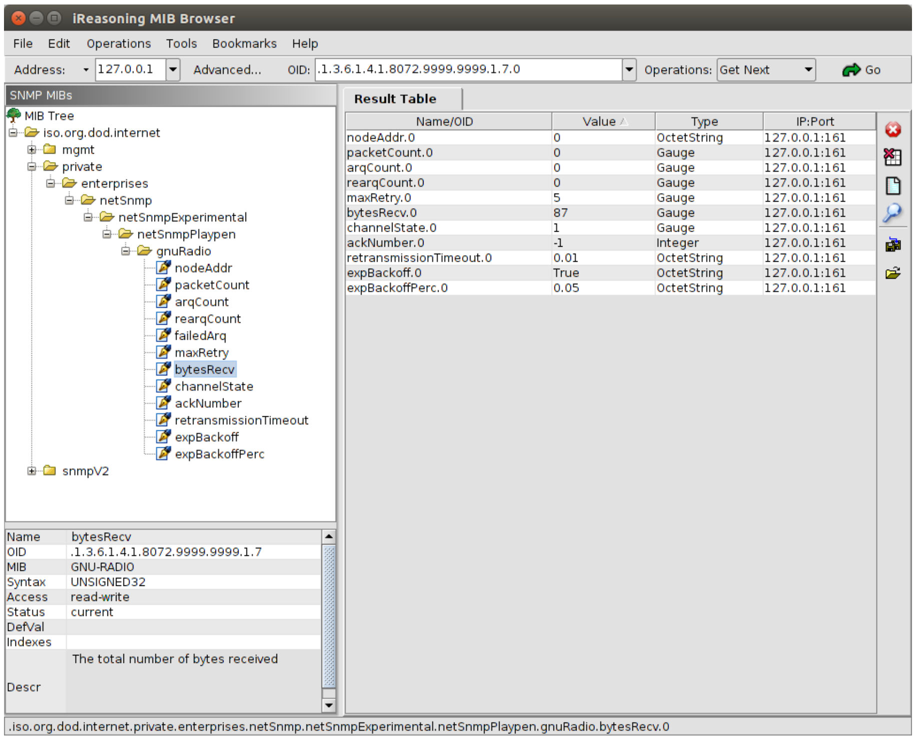


```
zach@osboxes: ~/Documents/Code/GnuSNMP/Python
zach@osboxes:~/Documents/Code/GnuSNMP/Python$ snmpwalk $args localhost GNU-RADIO::gnuRadio
GNU-RADIO::nodeAddr.0 = STRING: "0"
GNU-RADIO::packetCount.0 = Gauge32: 0
GNU-RADIO::arqCount.0 = Gauge32: 14
GNU-RADIO::rearqCount.0 = Gauge32: 0
GNU-RADIO::maxRetry.0 = Gauge32: 5
GNU-RADIO::bytesRecv.0 = Gauge32: 87
GNU-RADIO::channelState.0 = Gauge32: 1
GNU-RADIO::ackNumber.0 = INTEGER: -1
GNU-RADIO::retransmissionTimeout.0 = STRING: "0.01"
GNU-RADIO::expBackoff.0 = STRING: "True"
GNU-RADIO::expBackoffPerc.0 = STRING: "0.05"
zach@osboxes:~/Documents/Code/GnuSNMP/Python$ snmpset $args localhost GNU-RADIO::arqCount.0 u 8
GNU-RADIO::arqCount.0 = Gauge32: 8
zach@osboxes:~/Documents/Code/GnuSNMP/Python$ snmpget $args localhost GNU-RADIO::arqCount.0
GNU-RADIO::arqCount.0 = Gauge32: 8
zach@osboxes:~/Documents/Code/GnuSNMP/Python$ snmpset $args localhost GNU-RADIO::maxRetry.0 u 7
GNU-RADIO::maxRetry.0 = Gauge32: 7
zach@osboxes:~/Documents/Code/GnuSNMP/Python$ snmpset $args localhost GNU-RADIO::channelState.0 u 4
GNU-RADIO::channelState.0 = Gauge32: 4
zach@osboxes:~/Documents/Code/GnuSNMP/Python$ snmpset $args localhost GNU-RADIO::retransmissionTimeout.0 s 0.06
GNU-RADIO::retransmissionTimeout.0 = STRING: "0.06"
zach@osboxes:~/Documents/Code/GnuSNMP/Python$ snmpset $args localhost GNU-RADIO::expBackoff.0 s False
GNU-RADIO::expBackoff.0 = STRING: "False"
zach@osboxes:~/Documents/Code/GnuSNMP/Python$ snmpwalk $args localhost GNU-RADIO::gnuRadio
GNU-RADIO::nodeAddr.0 = STRING: "0"
GNU-RADIO::packetCount.0 = Gauge32: 0
GNU-RADIO::arqCount.0 = Gauge32: 8
GNU-RADIO::rearqCount.0 = Gauge32: 0
GNU-RADIO::maxRetry.0 = Gauge32: 7
GNU-RADIO::bytesRecv.0 = Gauge32: 360
GNU-RADIO::channelState.0 = Gauge32: 4
GNU-RADIO::ackNumber.0 = INTEGER: -1
GNU-RADIO::retransmissionTimeout.0 = STRING: "0.06"
GNU-RADIO::expBackoff.0 = STRING: "False"
GNU-RADIO::expBackoffPerc.0 = STRING: "0.05"
zach@osboxes:~/Documents/Code/GnuSNMP/Python$
```

Figure 2.4: SNMP CLI Client with LLSR

As seen in Fig. 2.4, the snmpwalk program will issue GET requests for each object descendant from the provided identifier in the MIB file. Given this functionality, a user is able to obtain the full managed state of the LLSR system by specifying the gnuRadio root object. Fig. 2.5 illustrates the use of a graphical client application called iReasoning MIB Browser, which helps the administrator follow the object structure of various systems, where they branch off and in some cases, where they lead.[15] The left column of interface shown in the image is a static browser which does not require any networking to display. This uses the local MIB, loaded by the user if not installed in a standard location. The OID structure is shown here near the top of the window, displaying the numerical representation for the "bytesRecv" object label. The right side of the vertical separator is retrieved by making an actual connection to the agent. The only prerequisite configuration is the address in the top left, here being the local-host loop back, and the community string defined for security reasons, not shown here. The port used by the agent is the standard port 161 for SNMP and thus requires no additional configuration on the client side. Using a graphical client tool to manage the state of GNU Radio demonstrates the purpose and success of this project as a proof of concept of a useful integration of network management with software defined radio.

[15] "Network Management / Network Monitoring / SNMP Monitoring / MIB Browser," Network Management / Network Monitoring / SNMP Monitoring / MIB Browser, accessed April 3, 2016, http://ireasoning.com/mibbrowser.shtml.



Address: 127.0.0.1 OID: .1.3.6.1.4.1.8072.9999.9999.1.7.0 Operations: Get Next

```
MIB Tree
iso.org.dod.internet
    mgmt
    private
        enterprises
            netSnmp
                netSnmpExperimental
                    netSnmpPlaypen
                        gnuRadio
                            nodeAddr
                            packetCount
                            arqCount
                            rearqCount
                            failedArq
                            maxRetry
                            bytesRecv
                            channelState
                            ackNumber
                            retransmissionTimeout
                            expBackoff
                            expBackoffPerc
    snmpV2
```

Name/OID	Value	Type	IP:Port
nodeAddr.0	0	OctetString	127.0.0.1:161
packetCount.0	0	Gauge	127.0.0.1:161
arqCount.0	0	Gauge	127.0.0.1:161
rearqCount.0	0	Gauge	127.0.0.1:161
maxRetry.0	5	Gauge	127.0.0.1:161
bytesRecv.0	87	Gauge	127.0.0.1:161
channelState.0	1	Gauge	127.0.0.1:161
ackNumber.0	-1	Integer	127.0.0.1:161
retransmissionTimeout.0	0.01	OctetString	127.0.0.1:161
expBackoff.0	True	OctetString	127.0.0.1:161
expBackoffPerc.0	0.05	OctetString	127.0.0.1:161

Name	bytesRecv
OID	.1.3.6.1.4.1.8072.9999.9999.1.7
MIB	GNU-RADIO
Syntax	UNSIGNED32
Access	read-write
Status	current
DefVal	
Indexes	
Descr	The total number of bytes received

.iso.org.dod.internet.private.enterprises.netSnmp.netSnmpExperimental.netSnmpPlaypen.gnuRadio.bytesRecv.0

Figure 2.5: iReasoning MIB Browser/SNMP Client with LLSR

Chapter 3

3.1 Conclusion

3.1.1 Summary of Work

Through various separated tasks regarding different software systems, this project has successfully implemented a MIB file defining custom object identifiers that represent the LLSR radio application, which is built using the GNU Radio toolkit and framework. By first defining the object information, it was possible to use the pySNMP library to generate a working daemon for interpreting network management requests. On top of this agent, a module for attribute access has been developed which implements a basic communication protocol for retrieving and updating the state of a given Python variable in the target system. In compliment of the requester module is a state manager module which contains two basic components. The first is a class extending the Python standard library TCP server, which is special in its acceptance of an extra constructor parameter for the instance object of the class under control. This server creates an instance of the module's TCP request handler derived class, which is given the instance object as a class member. Throughout the lifetime of the GNU Radio based application, this module will queue requests for state management originating from the requester. When the managing object invokes the handle_request method, the queue is popped, and a single interaction is made with the radio software's state before communicating back the result to the client in the case of a GET type request. The client in this scenario being the SNMP daemon, will encode the received state into an SNMP response and send it back to the original end user. The entire process benefits from a simple, though potentially redundant design in the fact that there is an additional

TCP/IP client-server model embedded between the outer SNMP requests and the managed service. However, benefits of the slightly redundant design include allowing the GNU Radio module to remain rather ignorant of the fact that a heavy management protocol is operating on its behalf. Furthermore, this additional communication layer remains lightweight and decouples the end-user's management protocol from the attribute handler, allowing for future network management protocols to be implemented without extensive changes to the current implementation.

3.1.2 Future Work

While the results of the the project are functional and implement the main desired behavior of remotely accessing and modifying the state of a running, remote radio routing system, this project may serve as a proof of concept and foundation for future development.

The current design favors a lightweight addition to the core system under management, although it currently requires the injection of management code in the main process loop in-order to pass control flow to the manager of an independent state request queue. While this coupling is minimal, a better solution would be for the instantiated management module object to run its attribute manager server in a separate thread from the main radio control thread. The benefit of this separation would mean that once created, there would be no further interaction or acknowledgment of the secondary thread by LLSR for the remainder of the process lifetime. The only implicit interaction that would be necessary in this case is the use of locking non thread-safe state for updates by the system. Without adequate locks in place, the manager could end up corrupting internal state by modifying it as it is being accessed in parallel. Similarly, the manager may return corrupted state to the service administrator when issued a GET request for the state of a variable being modified by the system's main thread. However, the use of a single lock to mutually exclude the two threads would result in great loss of performance due to over generalizing the scope of the exclusive state. One solution to mitigate this might be to define more specific locks, thus only blocking limited portions of the program's execution during state management. The monolithic state server approach was chosen for this project to avoid diverting work effort from the primary focus of a working management moduel, to modifying the LLSR code for supporting this type of state locking.

Another feature excluded in this initial work of integrating the technologies, is support for the SNMP TRAP message type. This feature would allow asynchronous messages from the managed system back to a client without the need for polling or a client issued request.[1] Using the TRAP message, a system administrator is not required to manually check in on the events of the system in order to become aware of a problem. Instead, they are able to define thresholds for specific variables and be notified as soon as such a situation arises. With the TRAP functionality in place, there is little need to check on the system manually if the only use case is mitigating downtime. However, there remains the use case of monitoring regular, non service impacting states for the purpose of metric gathering, debugging, or general interest in the current situation. For these use cases, the currently supported functionality is sufficient without the extra message handler. The TRAP functionality is currently supported in the pySNMP library used by the custom grSnmpAgent.[2] Therefore the difficulty in the implementation resides rather in the registering of events and thresholds and monitoring them passively via the grStateManager module. What brings this out of the scope of this project description is the fact that TRAP logic differs significantly from GET and SET messages since it is initiated by the managed server to the client.[3] In our case, handling the message itself would be fairly straightforward, by defining the TRAP details in the MIB for the specific objects and delegating the polling of the state to the pySNMP library's internal logic, shown to work in a demo of the API which serves as the foundation for this project's SNMP Agent.[4][5] Getting the state from this point would simply require re-using the GET request of the local grStateRequester to determine if the value of the attribute's value matches the event described in the MIB's TRAP description. However, at this point, special handling would be required

[1] "Net-SNMP," Net-SNMP, accessed April 3, 2016, http://www.net-snmp.org/tutorial/tutorial-5/commands/snmptrap.html.

[2] "PySNMP Examples: High-level API to Standard SNMP Applications: Notification Originator," SNMP Library for Python PySNMP, accessed April 3, 2016, http://pysnmp.sourceforge.net/examples/current/v3arch/oneliner/agent/ntforg/trap-v2c-with-mib-lookup.html.

[3] "Understanding Simple Network Management Protocol (SNMP) Traps," Cisco, accessed April 3, 2016, http://www.cisco.com/c/en/us/support/docs/ip/simple-network-management-protocol-snmp/7244-snmp-trap.html.

[4] Neal Charbonneau, www.nealc.com/blog/blog/2013/02/23/writing-an-snmp-agent-with-a-custom-mib-using-pysnmp/.

[5] "TUT:snmptrap Wiki," Net-SNMP, accessed April 3, 2016, http://www.net-snmp.org/wiki/index.php/TUT:snmptrap.

to initiate an outgoing connection to the client in order to notify of the occurrence. Otherwise, the event would not be known outside of the SNMP Agent's host.

While this project leaves room for future development of features and improvements, the current implementation proves rather successful in accomplishing the original goals of monitoring the inner workings of a Python GNU Radio routing extension through the use of Simple Network Management Protocol. At this point, support for the most common actions of retrieving and setting variable state is supported, as well as the prerequisites of a functional Management Information Base module. The hope is that this work will serve as a good foundation of network management for the LLSR project, and be adapted over time to become a worthy contributor to the success of the remote system in the field.

3.2 References

"17 Common MIB Design Errors - MIB Smithy User's Guide — Muonics, Inc." Network Management Software, Services & Consulting — Muonics, Inc. Accessed April 3, 2016. http://www.muonics.com/Docs/MIBSmithy/UserGuide/commonerrors.php#underscore.

"20.17. SocketServer A Framework for Network Servers Python 2.7.11 Documentation." Overview Python 3.5.1 Documentation. Accessed April 3, 2016. https://docs.python.org/2/library/socketserver.html.

Barbeau, Michel, Stephane Blouin, Gimer Cervera, Joaquin Garcia-Alfaro, and Evangelos Kranakis. "Michelbarbeau/gr-llsr: Location-free Link State Routing (LLSR) Implementation for GNU Radio." GitHub. Accessed April 3, 2016. https://github.com/michelbarbeau/gr-llsr.

Charbonneau, Neal. www.nealc.com/blog/blog/2013/02/23/writing-an-snmp-agent-with-a-custom-mib-using-pysnmp/.

"GNURadioCompanion - GNU Radio - Gnuradio.org." GNU Radio. Accessed April 3, 2016. https://gnuradio.org/redmine/projects/gnuradio/wiki/GNURadioCompanion#Adding-Custom-Blocks.

http://www.net-snmp.org/wiki/index.php/TUT:snmptrap.

"Library Reference PySNMP." SNMP Library for Python PySNMP. Accessed April 3, 2016. http://pysnmp.sourceforge.net/docs/api-reference.html.

Mauro, Douglas R., and Kevin J. Schmidt. Essential SNMP. Beijing: O'Reilly, 2005.

Net-SNMP. Accessed April 3, 2016. http://www.net-snmp.org/.

Net-SNMP. Accessed April 3, 2016. http://www.net-snmp.org/docs/mibs/NET-SNMP-MIB.txt.

“Net-SNMP.” Net-SNMP. Accessed April 3, 2016. http://www.net-snmp.org/tutorial/tutorial-5/commands/snmptrap.html.

“Network Management / Network Monitoring / SNMP Monitoring / MIB Browser.” Network Management / Network Monitoring / SNMP Monitoring / MIB Browser. Accessed April 3, 2016. http://ireasoning.com/mibbrowser.shtml.

“OutOfTreeModules - GNU Radio - Gnuradio.org.” GNU Radio. Accessed April 3, 2016. http://gnuradio.org/redmine/projects/gnuradio/wiki/OutOfTreeModules#Making-your-blocks-available-in-GRC.

“OutOfTreeModules - GNU Radio - Gnuradio.org.” GNU Radio. Accessed April 3, 2016. http://gnuradio.org/redmine/projects/gnuradio/wiki/OutOfTreeModules.

“Overview - GNU Radio - Gnuradio.org.” GNU Radio. Accessed April 3, 2016. https://gnuradio.org/redmine/projects/gnuradio.

“PySNMP Examples: High-level API to Standard SNMP Applications: Notification Originator.” SNMP Library for Python PySNMP. Accessed April 3, 2016. http://pysnmp.sourceforge.net/examples/current/v3arch/oneliner/agent/ntforg/trap-v2c-with-mib-lookup.html.

Rose, Marshall T. The Simple Book: An Introduction to Management of TCP/IP-Based Internets. Englewood Cliffs, N.J.: Prentice Hall, 1991.

“Single Responsibility Principle — Object Oriented Design.” Design Patterns — Object Oriented Design. Accessed April 3, 2016. http://www.oodesign.com/single-responsibility-principle.html.

“Smilint.” Institute of Operating Systems and Computer Networks. Accessed April 3, 2016. https://www.ibr.cs.tu-bs.de/projects/libsmi/smilint.html.

“SNMP Management: OIDs, MIBs, Traps, Notifications & More.” Network Management Software - Reviews & Network Monitoring Tools. Accessed April 3, 2016. http://www.networkmanagementsoftware.com/snmp-tutorial-part-2-rounding-out-the-basics/.

Tseitlin, Meir. “Developing PySNMP Based Agent with Custom MIB-Cloud Rocket.” Cloud Rocket. Accessed April 3, 2016. http://www.cloud-rocket.com/2013/08/writing-custom-mib-for-pysnmp-agent/.

“TUT:snmptrap Wiki.” Net-SNMP. Accessed April 3, 2016. http://www.net-snmp.org/wiki/index.php/TUT:snmptrap.

“Understanding Simple Network Management Protocol (SNMP) Traps.” Cisco. Accessed April 3, 2016. http://www.cisco.com/c/en/us/support/docs/ip/simple-network-management-protocol-snmp/7244-snmp-trap.html.