5-2014

The Space Plug-and-Play Architecture Services Manager and Its Relevance in Diverse Plug-and-Play Software Systems

Brandon Holdaway
Utah State University

THE SPACE PLUG-AND-PLAY ARCHITECTURE SERVICES MANAGER AND ITS RELEVANCE IN DIVERSE PLUG-AND-PLAY SOFTWARE SYSTEMS

by

Brandon Holdaway

A report submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

Approved:

Dr. Daniel Watson Major Professor	Dr. Scott Cannon Committee Member
Dr. Jacob Christensen Committee Member	

UTAH STATE UNIVERSITY
Logan, Utah

2014

ABSTRACT

The Space Plug-and-Play Architecture Services Manager and its Relevance in Diverse Plug-and-Play Software Systems

by

Brandon Holdaway, Master of Science

Utah State University, 2014

Major Professor: Dr. Daniel Watson
Department: Computer Science

The Space Plug-and-Play Archichecture System Manager (SSM) is a system of Space Plug-and-Play Architecture (SPA) hardware, software components, and services for rapid integration of satellite systems. It was originally designed to help decrease the amount of time and money required to integrate payloads onto a satellite. The features of the SSM make it useful in non-satellite systems.

In this report, I demonstrate diverse uses of the SSM in a multi-use payload management and underwater vehicle software solution which was designed and implemented using the SSM. Further I give a detailed explanation of the successes and failures of the software when used in a robotics application. Finally, I conclude with recommended additions to the SSM that would help solve the deficiencies found in the underwater vehicle software.

(52 pages)

PUBLIC ABSTRACT

Brandon L. Holdaway

The world of robotics software systems is vast, but much of the research and development happen on software systems that have little to no allowance for reconfiguraton or automatic management of redundent hardware. Satellites have similar problems with reconfiguration. Much of the software on satellites is written for a single satellite and does not handle reconfiguration. The Space Plug-and-Play Architecture System Manager (SSM) is a set of software that creates a plug-and-play system of hardware and software modules. It uses services to mask hardware communication buses and allow software modules to communicate without knowledge of the hardware path used. The SSM was written to solve the problems faced on satellite systems. In this report, I show a satellite and robitics software example that uses the SSM to solve each problem.

CONTENTS

LIST OF FIGURES

ACRONYMS

SPA Space Plug-and-Play Architecture

SSM The Space Plug-and-Play Architecture Services Manager

I2C Inter-Intergrated Circuit

SPA-S SPA-SpaceWire. The SPA SpaceWire subnet

SPA-1 SPA-I2C. The SPA I2C subnet

SPA-L SPA-Local. The SPA local subnet

xTEDS Extensible Transducer Electronic Data Sheet

ROS Robot Operating System

IMU Inertial Measurement Unit

API Application Programming Interface

GPS Global Positioning Satellites

IPC Inter-Process Communication

CHAPTER 1

INTRODUCTION

In this fast paced world economy, many people rely on satellites in their everyday lives. The uses can range from Global Positioning Satellites (GPS) for accurate positioning and directions to communication satellites for global telecommunications. In extreme cases, these satellites are used by governments for military operations. In a budget document by the Department of Defense, it was shown that they planned to set aside 653 million dollars to strengthen the Nations secure communication links in space [1]. Much of the money spent on satellites is used to create highly specific software designed for a single satellite. This expense is neither needed nor warranted. To fight the high costs of building satellites, the SSM was created as a reusable, generic, and open source alternative to single use satellite software.

The SSM is a Plug-and-Play system that requires any participating device to express its capabilities through an XML document known as an xTEDS [2]. Using the xTEDS information, any device can request information. It is light weight and runs on multiple systems including Linux, Windows, and VxWorks. Since all communication is handled by the SSM, a device receives any requested information regardless of what type of device it came from or whether the producing device lay on a different hardware bus. The SSM can also handle redundant devices by switching over to alternative devices on failure. Due to the light weight size and natural redundancy handling, it can be used in a diverse problem set such as satellite payload management, unmanned underwater vehicles, and home automation.

This paper will further explain the SSM and its subsystems. Next, it will talk in detail about some diverse uses of the SSM including multi-use payload management and unmanned underwater vehicles. Then, the paper will conclude with a discussion of the

results and some needed future capabilities.

CHAPTER 2

SSM

The SSM is a system of Space Plug-and-Play Architecture (SPA) hardware, software components, and services for rapid integration. It was originally designed to help decrease the amount of time and money required to integrate payloads onto a satellite.

For example, if a company or government body wanted to produce and fly a SPA satellite, then they would pull from a list of processors, components, and communication mediums and put them onto a satellite frame. Since all of the components would be SPA components, there would be no configuration or coding required to find or communicate with the hardware.

The information in this section should not be considered complete nor used as a standards document. Only basic information needed to understand the two usage scenarios is included. For a detailed explanation of the SPA protocols and standards, please read the standards documents.

2.1 Services

The SSM contains a set of three services: Lookup Service, Central Address Service, and Time Service. Each of the services is responsible for handling a part of the feature set offered by the SSM. This section will discuss the responsibilities of Lookup Service.

2.1.1 Lookup Service

The Lookup Service is one of the most important services in the SSM. The Lookup Services main responsibility is to manage all manager, service, and component Extensible Transducer Electronic Data Sheets (xTEDS) on the system. This means that if any device is active on the SPA network, then the Lookup Service will know about it. This forces the

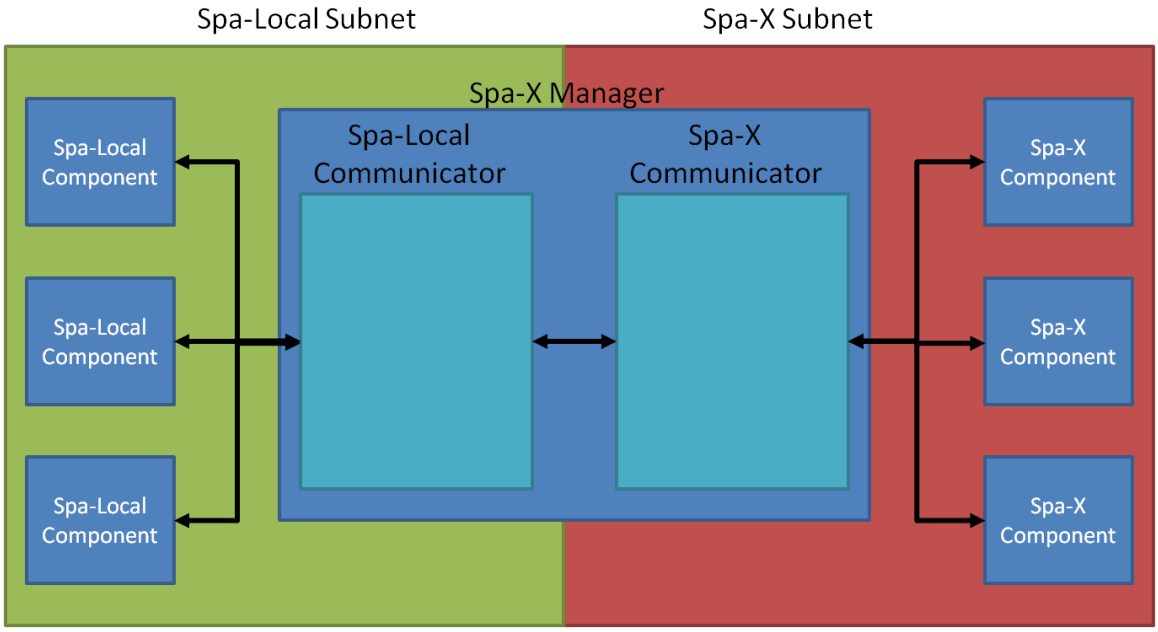

Figure 2.1: A figure showing the layout of a generic subnet manager.

Lookup Service to act as a pseudo database manager of capabilities.

Most of the xTEDS management involves handling queries from other components. If at any time a SPA participant wishes to find another participant on the network, then it may query the Lookup Service. The querying mechanism is the most complex feature of the Lookup Service. A query consists of an XML string describing what information the inquisitor is looking for and in what format. A more detailed explanation of the query mechanism will be covered in section 2.3. Once a query is received, the Lookup Service will send all matches to the originator.

2.2 Subnet Managers

The SSM is broken up in to many subnets. A subet is an area of communication, usually hardware, where all components communicate using the same protocols. For example a network device can use the UDP protocol to communicate with other network devices. The SSM contains one subnet manager for every supported SPA subnet. The list of currently supported subnets is: Inter-Process Communication (IPC) via UDP, SpaceWire, and I2C. These subnets represent the SPA-Local, SPA-SpaceWire, and SPA-1 subnets respectively.

Managers play an important role in the SSM. Subnet managers have the responsibility of component management and discovery, communication management, and bridging traffic from its subnet on to the SPA-Local subnet. Figure 2.1 shows a graphical representation of a generic subnet manager.

Subnet managers are responsible for finding, registering, deregistering, and maintaining the health status of all components on the subnet. Although each manager must find all components, the protocols required to do so are subnet specific. The subnet protocol for finding components will always end with the manager having a subnet address for each component. These addresses are required for basic communication on the subnet. As part of the registration process, the assignment of a SPA address to each component and manager is required. Each component receives a 32-bit SPA address which is split into two 16-bit unsigned values. The two most significant bytes represent the managers block address. The least significant bytes represent a subnet address which is handled by that manager. A zero value in the least significant bytes is specifically owned by the manager. Thus any components on the subnet would begin at 0x0001. For example: manager X could receive the address 0x00010000 where 0x0001 represents the managers block, 0x000010000 represents the managers address, and 0x0001xxxx represents the component addresses. After the manager receives its block, it will assign an address to each of the components on its subnet. This address allows any SPA component, whether it is located on the same subnet or not, to communicate with other SPA components by sending messages to a SPA address. Once the manager has a subnet address and SPA address, it will be able to route message on to and off of its subnet. The final stage of registration involves notifying the Lookup Service of each component. The process requires the registering component to send their xTEDS to the Lookup Service. Once received, the Lookup Service will check the new xTEDS against any existing queries and send out responses as needed. Finally, the subnet manager is responsible for maintaining a simple health status of its subnet components. This is performed on most subnets via a ping and reply message. If the component does not respond to a set number of health requests, then the manager will deregister the component

with the Lookup Service.

The SSM allows any device to communicate with any other devices independent of the subnets. Unfortunately, most communication medium, be it UDP, USB, serial, I2C, or SpaceWire, use an independent addressing schema. To get around this, the SSM uses a global SPA address and a hardware specific address held by the subnet manager. The ability of the managers to handle traffic coming in and out of the subnet lies in the routing table. Since the routing table contains the subnet and SPA addresses of each component, the manger can use this table to look up needed addresses. The managers bridging ability also stems from a requirement that every subnet manager, excluding the SPA-Local Manager, also reside on the SPA-Local subnet. This allows the managers to send messages on both the subnet specific and shared SPA-Local subnets. Also, since every manager is required to reside on the SPA-Local subnet, this puts the SPA-Local Manger in charge of all managers.

2.2.1 SPA-Local Manager

The SPA-Local Manager is responsible for handling communication over the SPA-Local subnet. This subnet uses Inter-Process Communication (IPC) via UDP sockets. The SPA-Local Manager is one of the few managers that resides at a known location. The SPA-Local Manager must always bind itself to port 3500. This is evidenced in the protocol for component discovery.

The SPA-Local subnet discovery is based off of the SPA-Local Managers known port. Every SPA-Local component is required to indicate existence and participation on the SPA-Local subnet by sending a message to the SPA-Local Managers known port. Once the manager receives a message, it will have the components UDP port number needed for further communication.

2.2.2 SPA-S Manager

The SPA-SpaceWire (SPA-S) manager is responsible for handling communication over the SpaceWire subnet. The communication medium is a standard packet switching SpaceWire bus.

The SPA-S subnet discovery is different from SPA-Local because all components reside on an unknown packet switching network. The network may contain many routers and components in an unknown configuration. The SPA-S Manager uses a breadth first search probing of the network to find all components. The SPA-S Manager recursively searches each router and its ports for other SPA-S components or managers. Once the message is received by a participating component, the component sends a response. After the manager receives the reply message, the components SpaceWire route is documented in the routing table.

Of a particularly interesting note, the SPA-S Manager is the only manager capable of supporting multiple managers on a single bus. This ability allows for multiple SPA-Local configurations to be bridged via the SPA-S subnet.

2.2.3 SPA-1 Manager

The SPA-I2C (SPA-1) manager is responsible for handling communication over the I2C subnet. The communication medium is a standard 5 Volt I2C bus running at a clock speed of 100KHz. Aside from the master arbitration component of the address resolution protocol, the bus runs with a single master during normal operations.

The SPA-1 subnet discovery differs from the other protocols due to the limited nature of I2C. The normal operation of a TWI or I2C bus involves a single master communicating with one or many slaves that reside at specific 7-bit or 10-bit addresses. All communication is performed by a master who addresses a slave and specifically requests that the slave either read or write. This leads to two problems. Since the slave has no control over the bus, there is no acceptable way for slaves to initiate communication. Also, it is possible for multiple slaves to reside on the same address. This would corrupt messages since multiple slaves would attempt to read and write at the same time.

The first issue was solved by having the SPA-1 manager use a RoundRobin approach for reading from and writing to each component. This gives each component one chance to read and write per loop. The addressing issue was solved using Multi-Master Arbitration. During the initialization phase, a slave device acts as master and attempts to write a

Globally Unique Identifier (GUID) to a specific address. If this operation is performed simultaneously by multiple masters, then only one device will finish writing the message and will take the slave address as its own and switch to slave mode. The other SPA-1 components will fail arbitration and try again on the next address. This pattern will continue until each SPA-1 component has a unique slave address. Once the SPA-1 components have a slave address they will be discovered by the SPA-1 Manager.

2.3 Querying

Querying is an important part of the SPA system. Querying is the mechanism by which a SPA component can find and consume data from another component. In querying, a SPA component creates a message that expresses what type of data it is looking for and sends it to the Lookup Service. For querying to work there are two requirements that must be met. The first requirement is the use of an XML document called an Extensible Transducer Electronic Data Sheet (xTEDS). The xTEDS is used to describe what a component can produce. The second requirement is that the consumer must know something about the data it wishes to consume.

2.3.1 xTEDS

The xTEDS is a document that uses XML syntax to express what data a component is capable of producing and what data a component requires in order to be commanded. It is shared with the Lookup Service during a components registration process. Since the xTEDS is very customizable, this report should not be considered a complete guide. Only the basics needed to understand the software in this report will be discussed. For a more detailed guide, please see the SSM documentation. The following sections will discuss the main elements of the xTEDS document.

```
<?xml version='1.0' encoding='utf-8'?>
<xTEDS xmlns='https://pnpsoftware.sdl.usu.edu/redmine/projects/xtedsschema'
       xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
       xsi:schemaLocation='https://pnpsoftware.sdl.usu.edu/redmine/projects/xtedsschema
           https://pnpsoftware.sdl.usu.edu/spa/xteds/current.xsd'
```

```
      name='SensorXteds' version='1.0'>
  <Application name='SensorController' kind='application' programMemoryRequired='1'
      dataMemoryRequired='1'/>
  <Interface name='StatusInterface' id='1'>
    <Notification>
      <DataMsg name='CurrentDepth' id='1' msgArrival='EVENT'>
        <Variable name='Depth' kind='depth' dataType='FLOAT32' units='feet'/>
      </DataMsg>
    </Notification>
    <Notification>
      <DataMsg name='RawDepth' id='2' msgArrival='EVENT'>
        <Variable name='Depth' kind='psi' dataType='FLOAT32' units='psi'/>
      </DataMsg>
    </Notification>
    <Notification>
      <DataMsg name='GetTemperature' id='3' msgArrival='EVENT'>
        <Variable name='Temperature' kind='degrees' dataType='FLOAT32' units='Farenheit'
            numberOfArrayElements='3'/>
      </DataMsg>
    </Notification>
    <Notification>
      <DataMsg name='MotorKillState' id='4' msgArrival='EVENT'>
        <Variable name='KillState' kind='state' dataType='UINT08' units='bool' description='
            1 means killed'/>
      </DataMsg>
    </Notification>
    <Notification>
      <DataMsg name='GetSystemPower' id='5' msgArrival='EVENT'>
        <Variable name='SystemVoltage' kind='volts' dataType='FLOAT32' units='volts'/>
        <Variable name='SystemCurrent' kind='current' dataType='FLOAT32' units='amps'/>
      </DataMsg>
    </Notification>
  </Interface>
  <Interface name='CalibrationInterface' id='2'>
    <Command>
      <CommandMsg name='CalibrateDepthSensor' id='1'>
        <Variable name='ActualDepth' kind='depth' dataType='FLOAT32' units='feet'/>
      </CommandMsg>
    </Command>
  </Interface>
</xTEDS>
```

xTEDS Element

The hierarchical structure of an xTEDS consists of a root xTEDS element. The xTEDS element can contain only three child element types: Device, Application, and Interface. The xTEDS attributes are used to describe the schema and its location, the XML version, the xTEDS name, and the xTEDS version.

```
<xTEDS xmlns='https://pnpsoftware.sdl.usu.edu/redmine/projects/xtedsschema'
       xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
       xsi:schemaLocation='https://pnpsoftware.sdl.usu.edu/redmine/projects/xtedsschema
           https://pnpsoftware.sdl.usu.edu/spa/xteds/current.xsd'
       name='SensorXteds' version='1.0'>
<\xTEDS>
```

Application and Device Elements

The first child of the xTEDS element must be either the Application or Device element. Both of these elements are similar in function but define that the xTEDS describes an application or device. The purpose of the first child element is to define the name and kind attribute. The Application differes from the Device element by its required inclusion of the 'programMemoryRequired' and 'dataMemoryRequired' attributes. Currently, neither of the two attributes are used.

```
<Application name='SensorController' kind='application'
programMemoryRequired='1' dataMemoryRequired='1'/>
```

Interface Element

Generically, an Interface element is used to contain a set of messages, but generally it is used to group related messages together. The set of message elements an Interface can contain are Notification, Command, and Request elements. The name and id attributes are required for the Interface element. Although many interfaces can be used in a single xTEDS, the id and name attributes of each interface must be unique through out the document. Similarly, the child message elements must also contain unique id attributes in their respective Interface.

```
<Interface name='CalibrationInterface' id='2'>
```

```
  <Command>
    <CommandMsg name='CalibrateDepthSensor' id='1'>
      <Variable name='ActualDepth' kind='depth' dataType='FLOAT32' units='feet'/>
    </CommandMsg>
  </Command>
  <Notification>
    <DataMsg name='CurrentDepth' id='2' msgArrival='EVENT'>
      <Variable name='Depth' kind='depth' dataType='FLOAT32' units='feet'/>
    </DataMsg>
  </Notification>
</Interface>
```

Notification Element

The Notification element is used to describe messages that are associated with data subscriptions. Notifications are produced periodically or on an event. The Notification element contains a single DataMsg child element. The name and id attribute inside of the DataMsg should be unique among other message elements in the interface.The DataMsg contains a list of message data variables described with Variable child elements.

```
<Notification>
  <DataMsg name='CurrentDepth' id='1' msgArrival='EVENT'>
    <Variable name='Depth' kind='depth' dataType='FLOAT32' units='feet'/>
  </DataMsg>
</Notification>
```

Command Element

The Command element defines a required set of variables that must be filled in order to command a given component. Similar to the Notification element, the Command element has no attributes and contains a single child element. The CommandMsg element defines the name and id of a given command.

```
<Command>
  <CommandMsg name='CalibrateDepthSensor' id='1'>
    <Variable name='ActualDepth' kind='depth' dataType='FLOAT32' units='feet'/>
  </CommandMsg>
</Command>
```

Request Element

The Request element is the final possible message definition. It defines a required set of variables which define a command and reply message. The Request element contains exactly two children. The first child is the CommandMsg element. This element does not differ from the CommandMsg element described in section 2.3.1. The second child element is the DataReplyMsg. It is used to describe the reply message. The DataReplyMsg attributes and child elements are the same as the CommandMsg and NotificationMsg. The name and id attributes in the CommandMsg and DataReplyMsg elements must be unique among the interface.

```
<Request>
  <CommandMsg name='GetCurrentIncrement' id='2' />
  <DataReplyMsg name='CurrentIncrementReply' id='3' >
    <Variable name='curIncrement' kind='count' dataType='FLOAT32' units='count'/>
  </DataReplyMsg>
</Request>
```

Variable Element

The Variable element represents a chunk of data. The Variable element's most important, and required, attributes are the name, kind, dataType, and units. When an xTEDS has these attributes properly defined, a component can look for specific types of messages and variables. In fact, a component can even use the precision attribute to make decisions on which producer, if many are present, to subscribe to. The information even allows the consumer component to fail over to a less precise device in the event that the better devices is deregistered.

```
<Variable name='curIncrement' kind='count' dataType='FLOAT32' units='count'/>
```

2.3.2 Querying Mechanism

The querying mechanism is an important part of the SSM. It is based completely off of the Lookup Service's ability to search every components xTEDS and return the results to the requester. Every component must have an xTEDS, although not all xTEDs will contain

messages. Thus, every component can be found given an appropriate query string. The querying mechanism allows a consumer to search based off of any element in the xTEDS. This means that a consumer could query for an application, interface, message, or any attribute.

Although the querying mechanism is powerful, a consuming component must know something about the data it is trying to find. A common approach is for a consumer to know the exact name and variable list of the message it is searching for. This allows the consumer to find a very specific message no matter when the producer may show up or on what subnet. This has the potential for problems if xTEDS or whole components are changed.

A better approach is to query for a type of data. For example, a consumer may be looking for some orientation variables. This is useful when the consumer does not care what device is actually producing the data. Thus, if the system contained more than one producer that had orientation variables in its xTEDS, the consumer would be able to select any of the available producers. Also in this instance, if one of the producers went offline, the consumer could easily switch to another producer even if the message formatting was different.

2.4 Virtual SPA Components

The SSM is a very useful system, but it can only handle a small set of subnets. Many hardware devices require the use of USB, serial, or other non supported subnets. Unfortunately, there is currently no way for a third party to add a subnet to the SSM. Thankfully, the SSM does allow a way to add a specific piece of hardware to the SPA system. The software that bridges between the unsupported hardware and the SPA-Local subnet is called a Virtual Component. The basic idea of a virtual component is to translate an unsupported hardware device to the SPA-Local subnet by creating a translation layer. Figure 2.2 gives a visual representation of the virtual component layout.

The virtual component is broken up into two parts, the SPA API and hardware specific communicator. The SPA API layer represents the SPA side of the virtual component. It

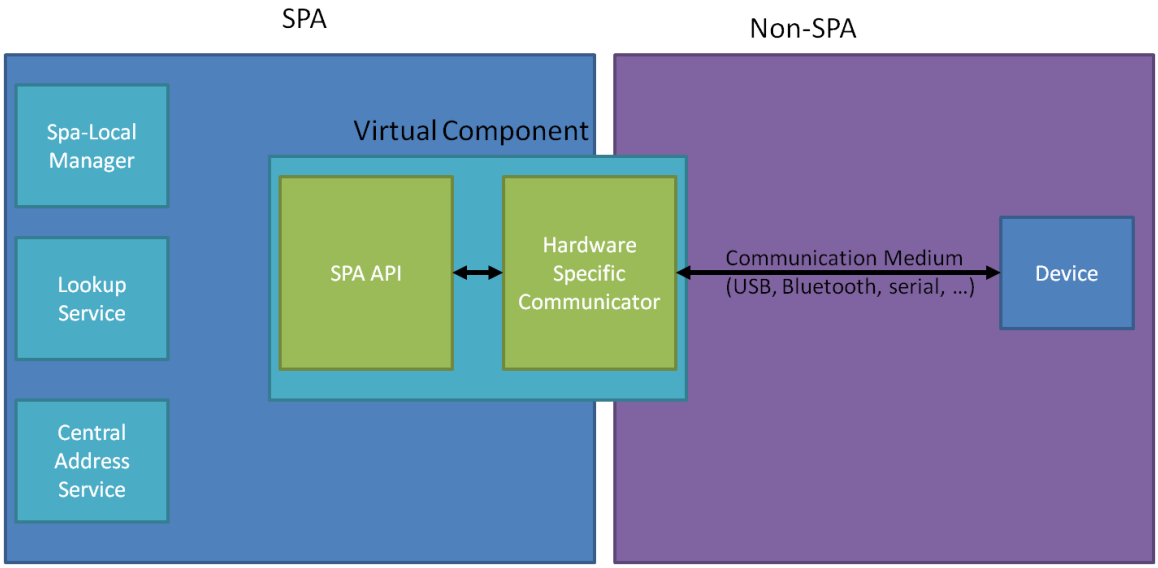

Figure 2.2: A figure showing the layout of a virtual SPA component.

will include an xTEDS describing how the component should be handled. For example, a virtual component controlling a camera may have one message for changing resolutions and another message for frame subscriptions. The SPA API layer is responsible for handling all communication corresponding to the message outlined in the xTEDS. It also uses the custom commands defined in the hardware communicator to forward requests and pull data from the hardware.

The hardware specific communicator is the heart of the virtual component. It must handle all communication with the hardware. Generally, this means that it has a set of threads reading from and writing to the hardware. The hardware communicator is also responsible for defining an interface for the SPA API. The interface must allow the SPA API layer to pull data from and forward commands to the hardware device.

When the two layers work together, they create a true virtual component that is indistinguishable from a physical component. Although virtual components allow a developer to put any hardware onto a SPA subnet, it is not a replacement for adding new SPA subnets to the SSM.

CHAPTER 3

MULTI-USE SATELLITE PAYLOAD MANAGEMENT

3.1 Purpose

The ability for a satellite to host payloads is a big incentive because it helps minimize the cost of satellite development. Unfortunately, communicating with a large number of hosted payloads may require satellite developers to spend extra time on each hosted payload to guarantee that the payloads data is telemetered to the ground. In the traditional world, this problem would be even more difficult because each satellite may have a different protocol. This creates a problem with payload developers who wish to fly a payload on multiple satellites and multiple protocols. Fortunately, the SSM can be used to minimize this problem, because it allows for very powerful searching and subscribing of components regardless of which subnet a component resides on. Thus, if a payload is designed to interface with a satellite running the SSM, then it will be usable on any satellite running the SSM.

A problem still exists however. The SSM may be good at handling searching and subscribing to components that exist anywhere on a SPA subnet, but difficulties arise due to a lack of structure on how the payloads should communicate in order to have data telemetered to the ground. Thus, the Satellite Payload Management software was designed and written to act as a standard for hosted payloads to communicate on an SSM satellite.

3.1.1 The Requirements

During the development of the Satellite Payload Management software, the hosting of I2C components was considered the most important subnet. Because of this, much of the standards are designed specifically for simple devices.

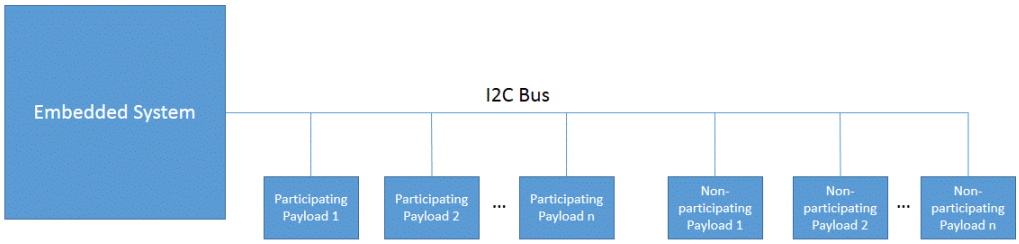

Figure 3.1: The configuration of participating and non-participating payloads on the same I2C bus.

The requirements were created to facilitate a specific satellite that is currently being developed. The satellite uses an I2C bus to host up to 24 payloads. The satellite also uses the same I2C bus to communicate with its own internal hardware.

The first requirement was that Payload Manager must only manage participating components. This means that participating and non-participating components may exist on a single subnet. Figure 3.1 gives an example of such a subnet.

The second requirement was that each payload was commandable by the Payload Manager. Each payload must accept a set of commands and handle them. Some of the commands can be used by the Payload Manager to give the payload important information, and others are used to turn the payload off and on.

The next requirement was that the Payload Manager must be able to telemeter payload and status data down to the ground.

The final requirement was that the Payload Manager must accept a schedule for handling payload command and data handling.

3.2 The Standards

The standards represent a list of items that each payload developer must follow. Mostly, the standards require modifications to an xTEDS, but they also require some implementation modifications. The following list represents the required standards.

The first standard was implemented to allow participating and non-participating components to exist on the same bus. It involves a minor modification to an xTEDS by adding a qualifier tag to the Device or Application element.

```
<Qualifier name='PayloadId' qualifierValue='1'/>
```

The tag consists of a Qualifier named 'PayloadId' with an integer value representing the actual identification number. The integer value should be considered globally unique such that no other participating component may have the same value. The qualifier is used by the Payload Manager to find and uniquely address each participating payload. The PayloadId is also used in the schedule to identify commands for a specific payload. Since non-participating payloads will not have the PayloadId, they will be ignored by the Payload Manager. This ability, along with the SSM's ability to mask, or hide, an applications originating bus, will allow the payload to exist anywhere on the SSM system. Thus, the participating payloads may be on the SPA-SpaceWire, SPA-I2C, SPA-Local, or any other subnets added in the future.

The second standard involves defining two standard interfaces for use in a participating components xTEDS. The standard interfaces must be implemented by all participating components. The following consists of the two standard interfaces:

```
<Interface name='ManagerControl' id='1'>
  <Command>
    <CommandMsg name='PayloadCommand' id='1' description='Send the payload a command'>
      <Variable name='Command' kind='byte' dataType='UINT08' units='byte'/>
      <DynamicArray name='CommandParameters' kind='bytes' dataType='UINT08' units='bytes'
          maxArrayElements='1'/>
    </CommandMsg>
  </Command>
  <Notification>
    <DataMsg name='PayloadData' id='2' msgArrival='EVENT'>
```

```
      <DynamicArray name='Data' kind='bytes' dataType='UINT08' units='bytes'
          maxArrayElements='255'/>
    </DataMsg>
  </Notification>
</Interface>
<Interface name='InformationInterface' id='2'>
  <Notification>
    <DataMsg name='PayloadStatus' id='1' msgArrival='EVENT'>
      <Variable name='PayloadStatus' kind='enum' dataType='UINT08' units='enum'>
        <Enumeration name='PayloadStatusEnum'>
          <Option name='STATUS_UNSET' optionValue='0'/>
        </Enumeration>
      </Variable>
    </DataMsg>
  </Notification>
</Interface>
```

The first standard interface is the ManagerControl interface. This interface expresses two messages. The first message, named PayloadCommand, is a command message. The PayloadCommand message helps fulfil the requirement that a participating component must be commandable. The standard message means that the Payload Manager can safely send a command to a participating payload and have it properly handled. The second message, named PayloadData, is a notification message. The PayloadData message helps fulfill the requirement that a payloads data must be telemetered to the ground. The Payload Manager subscribes to all devices that produce data using the PayloadData message and telemeters the data to the ground.

The second standard interface is the InformationInterface interface. This interface expresses a single notification message. The message, named PayloadStatus, is used to telemeter a component's specific status to the ground.

The final standard involves a schedule file expressed using XML. The schedule file directly solves the scheduling requirement. The schedule gives the satellite developer a format for describing different modes for an orbit and power and command timing for payloads. Due to its highly customizable format, many schedules can be described. For a further explanation of the schedule, please see section 3.5.

3.3 Software

The Payload Management software is designed to act as a payload manager for a satellite. The Payload Management software is designed to manage, execute commands on, and even download data from I2C payloads. This section gives a brief explanation of the purpose and responsibilities of the different Payload Management software components.

3.3.1 Payload Manager

The Payload Manager is the main piece of software. It is responsible for finding, managing, commanding, and reading all participating payloads. It is designed to ignore all non-participating payloads. The Payload Manager reads from a schedule and implements it. All other software pieces are there to support the Payload Manager.

3.3.2 Power Manager

The Power Manager is responsible for powering on and off the physical payloads. In a real use scenario, this software would be replaced or altered to control the physical power management board.

3.3.3 File Manager

The File Manager is responsible for creating, appending, reading, and deleting from a file according to requests. Depending on a satellites position, it may not be in constant contact with the groud. The File Manager is used to store all payload data while the satellite is out of contact with the ground. The current File Manager creates circular files on the file system. In a real use scenario, the software would be replaced or altered to use a physical storage location.

3.3.4 Transmitter Application

The Transmitter Application is responsible for telemetering data to and from the Satellite Payload Management software and the ground station. Currently, it uses UDP sockets

to telemeter data. In a real use scenario, the software would be replaced or altered to use a physical radio.

3.3.5 Payload Application

The Payload Application is an example payload. It adheres to the xTEDS template for participating payloads and can be used to test the functionality of the Payload Manager. As far as the Payload Manager is concerned, the Payload Application is no different than any other participating payload. In a real use scenario, this functionality would be moved into a physical device.

3.3.6 Cli

The Command Line Interface (CLI) is an application designed to act as a simple ground station. It takes parameters from the command line and relays them to the Payload Management software via the Transmitter Application. It uses a UDP socket to communicate directly with the Transmitter Application. In a real use scenario, this functionality would be written as part of a full feature ground station.

3.4 Hardware

The Payload Management software is very flexible in its ability to handle different hardware configurations. This is because of the flexibility inherent in the SSM architecture. The Payload Management software, namely the Payload Manager, was designed to manage participating payloads on an I2C bus. As mentioned earlier, the flexibility of the SSM and standards allow for the participation payloads to exist on any supported subnet. Due to this flexibility, this section will only cover the originally intended hardware configuration.

The original configuration is fairly simple. It consists of a processor and up to 24 I2C payloads. The processor is a BeagleBone Black running embedded Linux and the SSM. The processor uses I2C to connect to both participating and non-participating payloads as shown in figure 3.1. The participating payloads are supplied by different payload developers. The payload hardware represents experiments created by other organizations.

The non-participating payloads represent hardware required by the satellite itself. The non-participating payload hardware may consist of, but is not limited to, reaction wheels, Inertial Measurement Units, or Global Positioning Systems.

3.5 Schedule

The schedule file is the primary method used to control the Payload Manager. It expresses what, when, and where to send commands. It is also used to determine when and if a payload should be powered on and allowed to operate. The file uses XML syntax to express the schedule. The following example represents a simple schedule file.

```
<OrbitalSchedule>
  <OrbitPlanList orbitDuration='10' orbitUnits='minutes' >
    <OrbitPlan name='FirstOrbit' timeSliceDuration='10' timeSliceUnits='seconds' id='1' />
    <OrbitPlan name='SAFE_MODE' timeSliceDuration='10' timeSliceUnits='seconds' id='4' />
  </OrbitPlanList>
  <PayloadList>
    <Payload name='PayloadDemo1' PayloadId='1' PowerBusAddress='1' >
      <PayloadCommandList>
        <Command name='Cmd1' id='1' commandByte='1' params='1' />
        <Command name='Cmd2' id='2' commandByte='2' params='2,2' />
        <Command name='Cmd3' id='3' commandByte='3' params='3,3,3' />
        <Command name='Cmd4' id='4' commandByte='4' params='4,4,4,4' />
        <Command name='Cmd5' id='5' commandByte='5' params='5,5,5,5,5' />
        <Command name='Cmd6' id='6' commandByte='6' params='6,6,6,6,6,6' />
        <Command name='Cmd7' id='7' commandByte='7' params='7,7,7,7,7,7,7' />
        <Command name='POWER_ON' id='255' commandByte='255' />
        <Command name='POWER_OFF' id='254' commandByte='254' />
        <Command name='NO_OP' id='253' commandByte='253' />
        <Command name='READ_STATUS' id='252' commandByte='252' />
        <Command name='RESET' id='251' commandByte='251' />
        <Command name='LOW_POWER' id='250' commandByte='250' />
        <Command name='DATA_STOP' id='249' commandByte='249' />
        <Command name='DATA_RESUM' id='248' commandByte='248' />
        <Command name='NORMAL_POWER' id='247' commandByte='247' />
      </PayloadCommandList>
      <RunList>
        <Run name='FirstRun' orbitRef='FirstOrbit' >
          <CommandRef name='POWER_ON' />
          <CommandRef name='NORMAL_POWER' timeSliceIndex='1' />
```

```
        <CommandRef name='READ_STATUS' timeSliceIndex='2' />
        <CommandRef name='Cmd1' timeSliceIndex='3' reissueCount='9' />
        <CommandRef name='LOW_POWER' timeSliceIndex='13' />
        <CommandRef name='READ_STATUS' timeSliceIndex='14' />
      </Run>
      <Run name='SafeRun' orbitRef='SAFE_MODE' >
        <CommandRef name='LOW_POWER' timeSliceIndex='0' reissueCount='13' />
        <CommandRef name='READ_STATUS' timeSliceIndex='14' />
      </Run>
    </RunList>
  </Payload>
 </PayloadList>
 <PayloadSequence>
  <PayloadRef name='PayloadDemo1' />
 </PayloadSequence>
 <OrbitPlanSequence>
  <OrbitPlanRef name='FirstOrbit' />
 </OrbitPlanSequence>
</OrbitalSchedule>
```

To really understand the schedule, one must fully understand how the schedule expresses real world functionality. The first thing that should be understood is the idea of multiple orbit plans. Physically speaking, an orbit is the path a satellite will take as it circumnavigates the earth. In most satellites, this path does not change. Although the orbit path does not change, developers may wish that operations change on a per orbit basis. For example, on the initial earth orbit, the satellite may schedule commands that tests the hardware. Then, on the next orbit, normal operations may begin. The ability to change operations on a per orbit basis correlates with the definition of an Orbit Plan. An Orbit Plan describes a set of operations to be performed during an orbit. Many Obit Plans may be described in a single schedule. Also, a sequence of different Orbit Plans may be described through the OrbitSchedule element.

During an Orbit Plan, time is broken up in to N programmable slices. These slices are referred to as time slices. During a single time slice, every active payload will receive a piece of the time slice. During a payloads time, the Payload Manager may send a command. Thus, every payload has the potential to receive a command every time slice. The

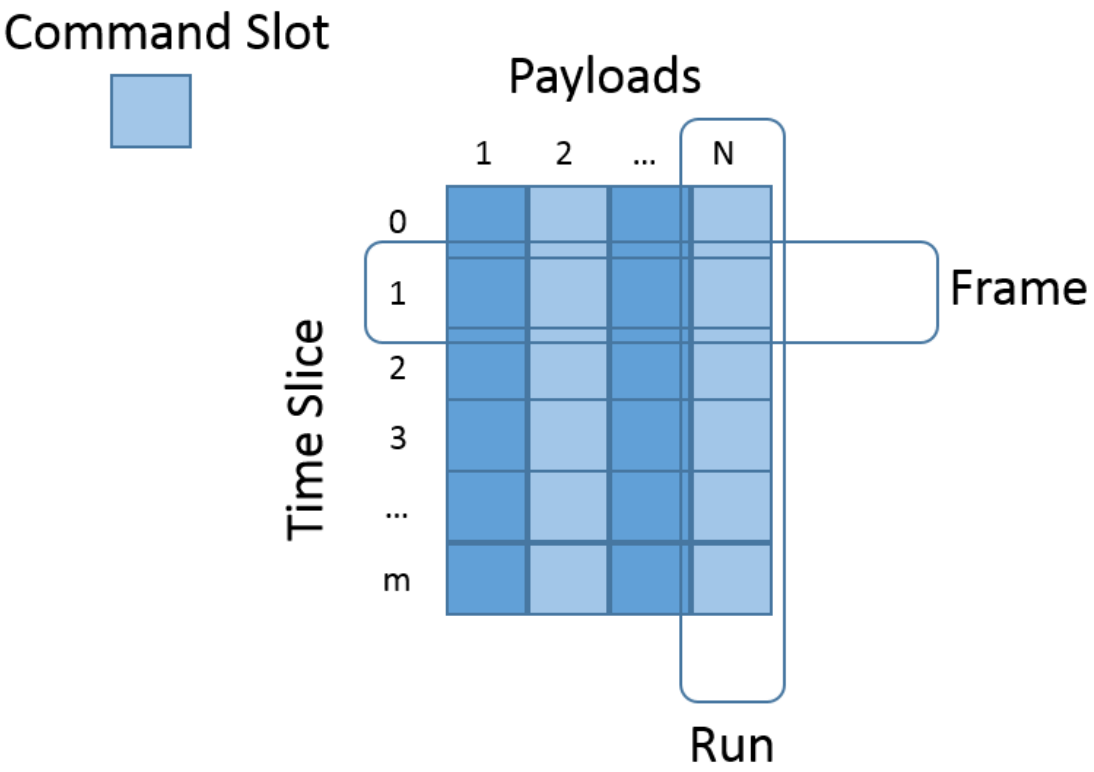

Figure 3.2: A figure describing the relationship between time slices, payloads, frames, and runs.

Payload Manager decides what commands to send to a payload based on a list of commands associated with a particular Orbit Plan. A payloads list of commands is called a Run. A good example of the relationship between time slices, payloads, frames, and runs is shown in figure 3.1.

3.5.1 The XML Elements

This section of the document will explain each of the XML elements in detail.

OrbitalSchedule Element

The main body of the XML is the OrbitalSchedule element. It is the root element and encapsulates everything needed for the schedule. The following is a simple example of the element.

```
<OrbitalSchedule>
</OrbitalSchedule>
```

The OrbitalSchedule element must have the following elements: OrbitalPlanList, PayloadList, PayloadSequence, and OrbitalPlanSequence. For example:

```
<OrbitalSchedule>
  <OrbitalPlanList />
  <PayloadList />
  <PayloadSequence />
  <OrbitPlanSequence />
</OrbitalSchedule>
```

The OrbitPlanList Element

The OrbitPlanList element is responsible for expressing how long a physical orbit and a scheduled Orbit Plan should take. It has two values that should be given: orbitDuration and orbitUnits. The orbitDuration must be an integer value and orbitUnits should be an indication of length. The possible units are seconds, minutes, hours, and days. For example:

```
<OrbitPlanList orbitDuration='10' orbitUnits='minutes' >
</OrbitPlanList>
```

The above example expresses an orbit that will take 10 minutes to finish. The OrbitPlanList must contain a set of OrbitPlan's. These plans are used to express how long a scheduled orbit will take. A full OrbitPlanList element might look like the following:

```
<OrbitPlanList orbitDuration='10' orbitUnits='minutes' >
   <OrbitPlan name='FirstOrbit' timeSliceDuration='10' timeSliceUnits='seconds' id='1' />
   <OrbitPlan name='SAFE_MODE' timeSliceDuration='10' timeSliceUnits='seconds' id='4' />
</OrbitPlanList>
```

The above example expresses an orbit that will take 10 minutes. It also expresses two distinct Orbit Plans FirstOrbit and SAFE_MODE.

The OrbitPlan Element

The OrbitPlan element is responsible for expressing how long an orbit schedule will take. An OrbitPlan must have a unique name and ID. The OrbitPlan also uses the timeSliceDuration and timeSliceUnits to describe how long a Time Slice will be. The timeSlice-

Duration must be an integer. The timeSliceUnits must be seconds, minutes, hours, or days. An OrbitPlan element might look like the following:

```
<OrbitPlan name='FirstOrbit' timeSliceDuration='10' timeSliceUnits='seconds' id='1' />
```

The above example expresses an OrbitPlan named FirstOrbit with id 1 that has a time slice duration of 10 seconds. If the OrbitPlanList from above was used with FirstOrbit, then the OrbitPlan would have room for 10 minutes / 10 seconds = 60 time slices.

The PayloadList Element

The PayloadList element is responsible for enumerating every payload what will managed by the Payload Manager. The PayloadList element contains a list of Payload elements. For example:

```
<PayloadList>
  <Payload name='Payload1' PayloadId='1' PowerBusAddress='1' />
</PayloadList>
```

The above example expresses a single payload named Payload1.

3.5.2 The Payload Element

The Payload element is responsible for expressing a payload including name, ID, payload ID, power bus address, command list, and run list. The Payload element requires the name, PayloadId, and PowerBusAddress attributes. The Payload element also requires the PayloadCommandList and RunList elements. For example:

```
<Payload name='Payload1' PayloadId='1' PowerBusAddress='1' >
  <PayloadCommandList>
    <Command name='Cmd1' id='1' commandByte='1' params='1' />
    <Command name='Cmd2' id='2' commandByte='2' params='2,2' />
    <Command name='Cmd3' id='3' commandByte='3' params='3,3,3' />
    <Command name='Cmd4' id='4' commandByte='4' params='4,4,4,4' />
    <Command name='Cmd5' id='5' commandByte='5' params='5,5,5,5,5' />
    <Command name='Cmd6' id='6' commandByte='6' params='6,6,6,6,6,6' />
    <Command name='Cmd7' id='7' commandByte='7' params='7,7,7,7,7,7,7' />
    <Command name='POWER_ON' id='255' commandByte='255' />
    <Command name='POWER_OFF' id='254' commandByte='254' />
    <Command name='NO_OP' id='253' commandByte='253' />
```

```
    <Command name='READ_STATUS' id='252' commandByte='252' />
    <Command name='RESET' id='251' commandByte='251' />
    <Command name='LOW_POWER' id='250' commandByte='250' />
    <Command name='DATA_STOP' id='249' commandByte='249' />
    <Command name='DATA_RESUM' id='248' commandByte='248' />
    <Command name='NORMAL_POWER' id='247' commandByte='247' />
</PayloadCommandList>
<RunList>
  <Run name='FirstRun' orbitRef='FirstOrbit' >
    <CommandRef name='POWER_ON' />
    <CommandRef name='NORMAL_POWER' timeSliceIndex='1' />
    <CommandRef name='READ_STATUS' timeSliceIndex='2' />
    <CommandRef name='Cmd1' timeSliceIndex='3' reissueCount='9' />
    <CommandRef name='LOW_POWER' timeSliceIndex='13' />
    <CommandRef name='READ_STATUS' timeSliceIndex='14' />
  </Run>
  <Run name='SafeRun' orbitRef='SAFE_MODE' >
    <CommandRef name='LOW_POWER' timeSliceIndex='0' reissueCount='13' />
    <CommandRef name='READ_STATUS' timeSliceIndex='14' />
  </Run>
</Payload>
```

The above example expresses a payload named Payload1 with payload ID 1 and power bus address 1. It goes on to show the PayloadCommandList and RunList.

The PayloadCommandlist Element

The PayloadCommandList element is used to enumerate all of the commands a Payload supports. The PayloadCommandList element contains a list of Command elements. For example:

```
<PayloadCommandList>
    <Command name='Cmd1' id='1' commandByte='1' params='1' />
    <Command name='Cmd2' id='2' commandByte='2' params='2,2' />
    <Command name='Cmd3' id='3' commandByte='3' params='3,3,3' />
    <Command name='Cmd4' id='4' commandByte='4' params='4,4,4,4' />
    <Command name='Cmd5' id='5' commandByte='5' params='5,5,5,5,5' />
    <Command name='Cmd6' id='6' commandByte='6' params='6,6,6,6,6,6' />
    <Command name='Cmd7' id='7' commandByte='7' params='7,7,7,7,7,7,7' />
    <Command name='POWER_ON' id='255' commandByte='255' />
    <Command name='POWER_OFF' id='254' commandByte='254' />
    <Command name='NO_OP' id='253' commandByte='253' />
```

```
    <Command name='READ_STATUS' id='252' commandByte='252' />
    <Command name='RESET' id='251' commandByte='251' />
    <Command name='LOW_POWER' id='250' commandByte='250' />
    <Command name='DATA_STOP' id='249' commandByte='249' />
    <Command name='DATA_RESUM' id='248' commandByte='248' />
    <Command name='NORMAL_POWER' id='247' commandByte='247' />
</PayloadCommandList>
```

The above example shows a PayloadCommandList that enumerates 16 commands supported by the Payload.

The Command Element

The Command element is used to express a payload command. It describes the name, id, command byte, and parameters to the command. The Command element requires the use of the name, id, and commandByte. The name and id must be unique. The commandByte must be a value from 0 to 255 and must ignore values used for system commands, i.e. 247 to 255. The params attribute is optional. It contains a comma separated set of parameter values with a value ranging from 0 to 255. Currently, the PayloadManager only supports a single parameter value, so any extra parameters will be ignored. For example:

```
<Command name='Cmd1' id='1' commandByte='1' params='1' />
```

The above example expresses a command named Cmd1 with an ID of 1, command of 1, and parameter value of 1.

3.5.3 The RunList Element

The RunList element contains a list of Run elements. The RunList has no attributes. For example:

```
<RunList>
  <Run name='FirstRun' orbitRef='FirstOrbit' >
    <CommandRef name='POWER_ON' />
    <CommandRef name='NORMAL_POWER' timeSliceIndex='1' />
    <CommandRef name='READ_STATUS' timeSliceIndex='2' />
    <CommandRef name='Cmd1' timeSliceIndex='3' reissueCount='9' />
    <CommandRef name='LOW_POWER' timeSliceIndex='13' />
```

```
    <CommandRef name='READ_STATUS' timeSliceIndex='14' />
  </Run>
</RunList>
```

The above example shows a RunList with a single Run element.

The Run Element

The Run element is responsible for defining how commands should be issued to a Payload according to an OrbitPlan. The Run elements contain and require the name and orbitRef attributes. The orbitRef attribute is used to associate the Run with a previously defined OrbitPlan where the value is the name of the OrbitPlan. The Run element also contains a list of CommandRef elements. For example:

```
<Run name='FirstRun' orbitRef='FirstOrbit' >
  <CommandRef name='POWER_ON' />
  <CommandRef name='NORMAL_POWER' timeSliceIndex='1' />
  <CommandRef name='READ_STATUS' timeSliceIndex='2' />
  <CommandRef name='Cmd1' timeSliceIndex='3' reissueCount='9' />
  <CommandRef name='LOW_POWER' timeSliceIndex='13' />
  <CommandRef name='READ_STATUS' timeSliceIndex='14' />
</Run>
```

The above example expresses a Run named FirstRun that defines the OrbitPlan named FirstOrbit. The Run uses the CommandRefs to express the commands that will be executed at specific times.

The CommandRef Element

The ComandRef element is used to reference a Payload's Command and express when the Command should be sent to the Payload. The CommandRef element requires the name attribute to reference a previously defined Command element for that Payload. The name value must exactly match the name of the Command. The CommandRef uses the optional timeSliceIndex and reissueCount attributes to express when the command should be sent. The timeSliceIndex and reissueCount attributes must be integer values. The timeSliceIndex references the zero based position of the command in the Run. The reissueCount attribute

expresses how many more times the command will be sent to the Payload. If the timeSliceIndex is not used, then the relative position in the list will be used. The reissueCount defaults to 0. A CommandRef example:

```
<CommandRef name='Cmd1' timeSliceIndex='3' reissueCount='9' />
```

The above example expresses that the Command, named Cmd1, will be executed in the 4th time slice and be repeated 9 times. That means the command will be sent 10 times starting at index 3.

The PayloadSequence Element

The PayloadSequence element contains a list of PayloadRefs. The PayloadSequence represent the order in which Payloads will receive commands. If a payload is not represented in the PayloadSequence then it will not receive commands. For example:

```
<PayloadSequence>
  <PayloadRef name='PayloadDemo1' />
  <PayloadRef name='PayloadDemo2' />
</PayloadSequence>
```

The above example expresses an ordering for payloads 1 and 2. It means that PayloadDemo1 will always receive a command before PayloadDemo2.

The PayloadRef Element

The PayloadRef element is used to reference a previously defined Payload. The PayloadRef element contains only a required name attribute. The name attribute must exactly match the name of a previously defined Payload. For example:

```
<PayloadRef name='PayloadDemo1' />
```

The above example references the PayloadDemo1 payload. Repetition of PayloadRefs is allowed.

The OrbitPlanSequence Element

The OrbitPlanSequence element is used to express an ordering of OrbitPlans. The OrbitPlanSequence contains a list of OrbitPlanRef elements. For example:

```
<OrbitPlanSequence>
  <OrbitPlanRef name='FirstOrbit' />
  <OrbitPlanRef name='SecondOrbit' />
</OrbitPlanSequence>
```

The above example expresses an ordering of OrbitPlans such that FirstOrbit will be followed by SecondOrbit. Since the OrbitPlanSequence will be repeated when the end of the list is reached, the pattern will always be FirstOrbit, SecondOrbit, etc. Repetition of OrbitPlanRefs is allowed.

The OrbitPlanRef Element

The OrbitPlanRef element is used to reference a previously defined OrbitPlan. The OrbitPlanRef contains a single required name attribute. The name attribute is used as the reference key and must exactly match an OrbitPlan name. For example:

```
<OrbitPlanRef name='FirstOrbit' />
```

The above example references an OrbitalPlan named FirstOrbit.

3.5.4 Software Setup

This section will cover the software setup for the Payload Management software and the SSM. It will not cover all possible setups, but it will cover the minimum requirements to get a system up and running. For an example, please see busat-scdh/scdh/testUtil/scripts/runAll.sh.

Basic Setup

The Payload Management software requires that the SSM be setup and running. At a minimum, the SSM requires the SpaLocalManager, LookupService, and CentralAddressService to be running. These represent the services required for other applications to join the SSM system. An example execution of these services would be as follows:

```
# busat-scdh/ssm/bin/SpaLocalManager -f=sml.log -g=1234 &

# busat-scdh/ssm/bin/LookupService -f=ls.log -g=1234 &
```

```
# busat-scdh/ssm/bin/CentralAddressService -f=cas.log -g=1234 &
```

These commands would start the services and create log files. The log files would not contain everything logged, but the files would contain fatal, error, warning, and info messages. Since the Payload Management software was designed to manage I2C payloads, the Spa1Manager will also need to run. The following command will start the Spa1Manager listening on the /dev/i2c-1 bus:

```
# busat-scdh/ssm/bin/Spa1Manager -f=sm1.log -g=1234 -i=/dev/i2c-1 &
```

Since the Payload Management software is based off of the SSM libraries, many of the parameters from the SSM software can be found in the Payload Management software. For example, the logging parameters will work for all Payload Management software.

The PayloadManager is the main piece of software, so it will be the first one discussed. At a minimum, it requires a safe mode schedule. The following command will start the PayloadManager:

```
# busat-scdh/scdh/bin/PayloadManager -S=schedule-SAFE.xml -f=pm.log -g=1234 &
```

This command instructs the Payload Manager to use the schedule-SAFE.xml file as the safe schedule. Since the safe schedule should be considered a safe mode schedule, the safe schedule will never be modified. The Payload Manager can be instructed to modify its running schedule, but all changes will be saved to the non-safe mode schedule. The next piece of software required is the File Manager. Although it has no required parameters, there are a few that should be used. The following command starts the File Manager and instructs it to store all created and used files in the ‘pwd‘/dataFiles/ directory.

```
# busat-scdh/scdh/bin/FileManager -f=fm.log -g=1234 -F=dataFiles/ &
```

The above mentioned commands should be enough to start a basic setup. All payloads will be managed by the Payload Manager according to the specified schedule and the data will be stored by the File Manager. This setup does not allow for ground based interaction with the Payload Manager nor the telemetering of payload data. In a real use scenario, the other software pieces would be needed.

CHAPTER 4
UNMANNED UNDERWATER VEHICLES

4.1 Purpose

Today, unmanned machines and vehicles are used extensively. The military uses unmanned drones for reconnaissance and decisive action in places that would put a pilot at risk. Unmanned robots are used in countless factories around the world. Even Google has gotten involved in unmanned robotics by developing an autonomous car for consumers. In all of these cases, a software system needs to communicate with a set of sensors and control hardware.

In the research community, the communication and services required for software, sensors, and control hardware are often handled with the Robot Operating System (ROS). Software based off of ROS uses topic strings for communication. The data is produced and consumed based off of the topic string. Unfortunately, if a component or software piece is unsure of the topic, there is no method provided for searching based off of types or contents of data messages. Also, if the topic strings are changed on the producer, then every consumer module will have to be modified. These problems are solvable with the use of the SSM.

The SSM allows unmanned vehicle developers and researchers to create a software solution capable of rapid reconfiguration. A positive side effect of the SSM's query mechanism means that redundant hardware can be kept active and only used when needed. In the event of a failure, the consumer software can automatically switch over to another producer.

In an attempt to prove this ability, the unmanned underwater vehicle software was written as a proof of concept. The unmanned underwater vehicle software was written using many different services. It was written only as a proof of concept for USUs existing

submarine platform. The software was designed to give a user basic joystick control of a submarine. The following sections will describe the details of the hardware and software.

4.2 Hardware

The Utah State Universitys first revision submarine, Gilligan, was used to test the software. The submarine consists of an acrylic rectangular frame with two decks. There are six thrusters positioned in strategic places on the submarines frame. There are two thrusters mounted center mass on each side of the frame. These thrusters are used for forward and reverse movement. There are also two thrusters mounted between the decks and facing up for depth. The last two thrusters are used for turning. They are mounted next to the depth thrusters on either end of the submarine. Although the thrusters are reversible, they are much more powerful moving in the forward direction. For this reason, the turn thrusters were placed aiming opposite directions. This gives a similar handicap to both turning directions.

The top deck is used to hold two pelican cases filled with electronics. The bottom deck is designed to hold either one or two pelican cases with each containing a large 12V 22Ah battery. On the top deck, the forward pelican case is used to house the main computer components along with a few secondary items. The mother board and hard drive are mounted on the inside top of the case. This helps the motherboard stay dry in case of a minor leak. The forward case also contains the Inertial Measurement Unit (IMU). The IMU is placed at the bottom of the case but on top of a riser. This riser is meant to help keep the IMU dry in case of a minor flood. The last component in the forward case is an Arduino. The Arduino uses a few water sensors to detect if any flooding has occurred.

The top back case is the most complicated case and is generally labeled the controller box. It houses the motor controllers, sensor boards, pressure sensor, and kill button controller. The submarine uses three dual channel motor controllers called Wild Thumpers. The Wild Thumper is an Arduino based motor controller that contains built in voltage and current sensors. These controllers were primarily chosen because of the easy C like programming language, existing open source API, high 15 Amp current rating, and USB

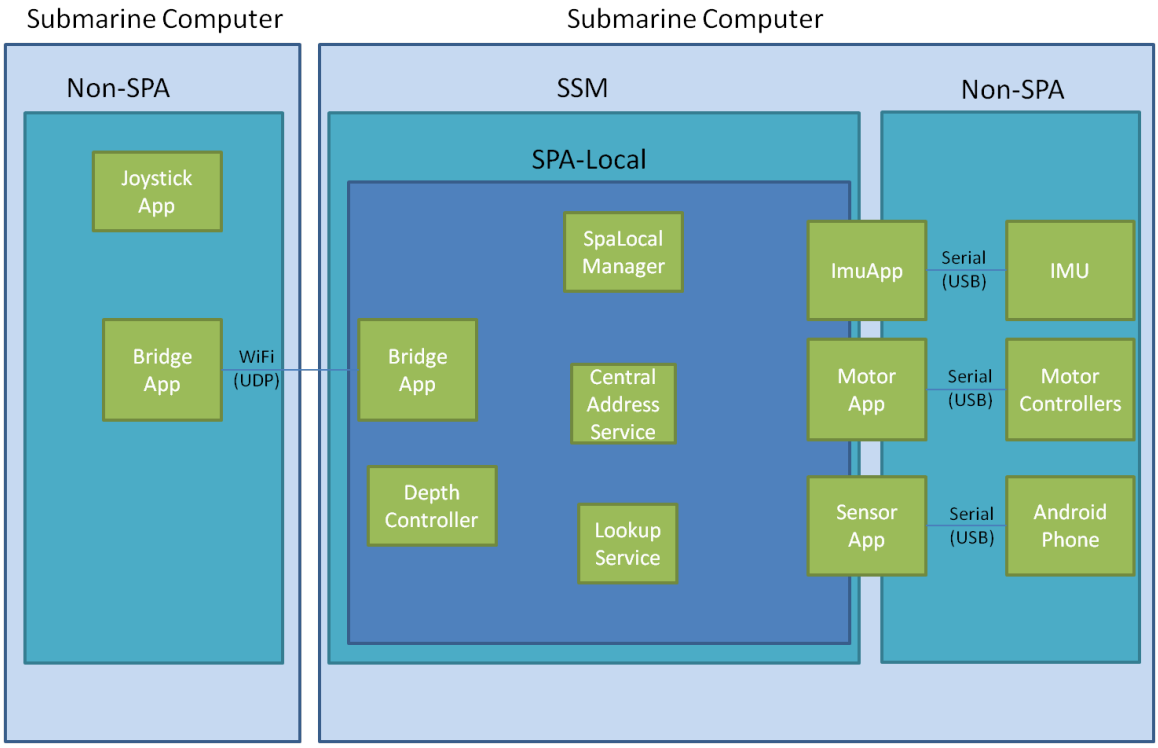

Figure 4.1: A figure showing the layout of the submarine software.

connectivity. The controller box also contains the sensor board. The sensor board consists of an Arduino Nano and a custom built board for connecting the kill switch, pressure sensor, system wide voltage and current sensor, and a torpedo launching solenoid. Given the cramped nature of the case, the sensor board also acts as a flood sensor for the controller box.

4.3 Software

The software was designed as a proof of concept for an already existing platform. A full software set would include software for positioning, grabber controls, artificial intelligence, and much more. Most of the existing software was designed to manage a specific piece of the submarines control or sensor hardware. The rest of the software is used to process sensor data and allow for human control of the submarine. A graphical description of the software is given in Figure 4.1. This section will describe all the software used in the project.

4.3.1 ImuApp

The ImuApp is a virtual component responsible for handling IMU related data. It uses a serial interface to communicate with a MicroStrain 3DM-GX3 IMU. The IMU is calibrated to give Euler Angels, raw acceleration, and raw gyro readings. The ImuApp reads the orientation data from the IMU and shares it using the following xTEDS:

```
<?xml version='1.0' encoding='utf-8'?>
<xTEDS xmlns='https://pnpsoftware.sdl.usu.edu/redmine/projects/xtedsschema'
       xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
       xsi:schemaLocation='https://pnpsoftware.sdl.usu.edu/redmine/projects/xtedsschema
           https://pnpsoftware.sdl.usu.edu/spa/xteds/current.xsd'
       name='ImuxTeds' version='1.0'>
  <Application name='ImuApp' kind='application' programMemoryRequired='1'
      dataMemoryRequired='1'/>
  <Interface name='InfoInterface' id='1'>
    <Notification>
      <DataMsg name='TelemetryEuler' id='1' msgArrival='EVENT'>
        <Variable name='Yaw' kind='rotation' dataType='FLOAT32' units='degrees'/>
        <Variable name='Pitch' kind='rotation' dataType='FLOAT32' units='degrees'/>
        <Variable name='Roll' kind='rotation' dataType='FLOAT32' units='degrees'/>
      </DataMsg>
    </Notification>
  </Interface>
</xTEDS>
```

4.3.2 FloodSensorApp

The FloodSensorApp is arguably one of the most important safety features. It is the virtual component responsible for reading from an Arduino and indicating when water is detected. The Arduino is running a simple application that reads from a set of water sensors and outputs the voltage for each sensor over the serial bus. The FloodSensorApp uses a serial interface to communicate with the Arduino, make a decision about the current voltage, and indicate to the system if a flood has occurred. The xTEDS is as follows:

```
<?xml version='1.0' encoding='utf-8'?>
<xTEDS xmlns='https://pnpsoftware.sdl.usu.edu/redmine/projects/xtedsschema'
       xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
       xsi:schemaLocation='https://pnpsoftware.sdl.usu.edu/redmine/projects/xtedsschema
           https://pnpsoftware.sdl.usu.edu/spa/xteds/current.xsd'
```

```
      name='FloodSensorApp' version='1.0' description='Flood Sensor App'>
  <Application name='FloodSensorApp' kind='appliation' programMemoryRequired='1'
      dataMemoryRequired='1'/>
  <Interface name='StatusInterface' id='1'>
    <Notification>
      <DataMsg name='FloodStatus' id='1' msgArrival='EVENT'>
        <DynamicArray name='SensorPositionString' kind='string' dataType='INT08' units='char
            ' maxArrayElements='1300'/>
        <Variable name='SensorValue' kind='adc' dataType='UINT32' units='millivolts'/>
        <Variable name='SensorPosition' kind='position' dataType='UINT32' units='position'/>
      </DataMsg>
    </Notification>
  </Interface>
</xTEDS>
```

4.3.3 MotorApp

The MotorApp is the virtual component application responsible for managing the motor controllers and moving the submarine. There will be one instance of the MotorApp for each motor controller. The Utah State University's submarine required an instance of the MotorApp for the drive, dive, and turn motor controllers. The MotorApp has more complexity than any other application. It has to keep track of each motors current speed and perspective speed for ramping purposes. The ramping is used to prevent damage to the motors and motor controllers caused by rapid speed and direction changes. The MotorApp is also responsible for reading current, voltage, and error states from the motor controllers. All of the motor controller status messages, along with the ability to control the motors, are all exposed by the xTEDS. The xTEDS is as follows:

```
<?xml version='1.0' encoding='utf-8'?>
<xTEDS xmlns='https://pnpsoftware.sdl.usu.edu/redmine/projects/xtedsschema'
      xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
      xsi:schemaLocation='https://pnpsoftware.sdl.usu.edu/redmine/projects/xtedsschema
          https://pnpsoftware.sdl.usu.edu/spa/xteds/current.xsd'
      name='MotorApp' version='1.0' description='The MotorApp xTEDS'>
  <Application name='MotorApp' kind='appliation' programMemoryRequired='1'
      dataMemoryRequired='1'>
    <Qualifier name='MotorType' qualifierValue=''/>
  </Application>
```

```
  <Interface name='CommandInterface' id='1'>
    <Command>
      <CommandMsg name='SetMotors' id='1' description='Set the motor speeds'>
        <Variable name='LeftChannelSpeed' kind='speed' dataType='INT32' units='speed'
            rangeMin='-1023' rangeMax='1023'/>
        <Variable name='RightChannelSpeed' kind='speed' dataType='INT32' units='speed'
            rangeMin='-1023' rangeMax='1023'/>
      </CommandMsg>
    </Command>
  </Interface>
  <Interface name='StatusInterface' id='2'>
    <Notification>
      <DataMsg name='GetControllerVoltage' id='1' msgArrival='EVENT'>
        <Variable name='Voltage' kind='voltage' dataType='FLOAT32' units='millivolts'/>
      </DataMsg>
    </Notification>
    <Notification>
      <DataMsg name='GetControllerCurrent' id='2' msgArrival='EVENT'>
        <Variable name='LeftChannelCurrent' kind='current' dataType='FLOAT32' units='unknown
            '/>
        <Variable name='RightChannelCurrent' kind='current' dataType='FLOAT32' units='
            unknown'/>
      </DataMsg>
    </Notification>
  </Interface>
</xTEDS>
```

4.3.4 JoystickApp

The JoystickApp is a very simple control application. This virtual component reads from a standard joystick and converts the different axis into appropriate commands for the MotorApp components. The application must use the built in querying mechanism to search for the turn and drive controllers along with the DepthController. The JoystickApp uses the three queries to forward commands to the appropriate component. Also, the JoystickApp consumes the data for the submarine voltage and current, orientation, depth, and individual motor currents. This information is sent directly to the command line for user visibility. Since the JoystickApp does not produce any data, it does not contain an xTEDS.

4.3.5 DepthController

The DepthController is a software component devoted to the maintaining of depth. It allows for the setting of a target depth through its xTEDS. The DepthController queries the SensorApp for depth information and the dive specific DepthController for physical depth control. The controller also uses a ramping algorithm to help minimize the possibility of overshooting the target depth. The DepthController's xTEDS is as follows:

```
<?xml version='1.0' encoding='utf-8'?>
<xTEDS xmlns='https://pnpsoftware.sdl.usu.edu/redmine/projects/xtedsschema'
       xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
       xsi:schemaLocation='https://pnpsoftware.sdl.usu.edu/redmine/projects/xtedsschema
           https://pnpsoftware.sdl.usu.edu/spa/xteds/current.xsd'
       name='DepthControllerXteds' version='1.0'>
  <Application name='DepthController' kind='application' programMemoryRequired='1'
      dataMemoryRequired='1'/>
  <Interface name='ControlInterface' id='1'>
    <Command>
      <CommandMsg name='SetDepth' id='1'>
        <Variable name='Depth' kind='depth' dataType='FLOAT32' units='feet'/>
      </CommandMsg>
    </Command>
  </Interface>
</xTEDS>
```

4.3.6 BridgeApp

The purpose of the BridgeApp is to allow a SPA-Local component that does not reside on a SPA-Local subnet to communicate as if it were on a given SPA-Local subnet. Although the BridgeApp does not register as a SPA component, it does send SPA messages. The BridgeApp works like a virtual tunnel and forwards traffic between instances on both the local and remote computer.

The remote BridgeApp instance communicates with the off side component by simulating a SPA-Local Manager. Normally, a SPA-Local component sends a message to the SPA-Local Manager on UDP port 3500 when it begins. The remote BridgeApp takes advantage of this by binding to port 3500. From this port, the remote instance will listen to messages from the component and forward them to the local BridgeApp instance.

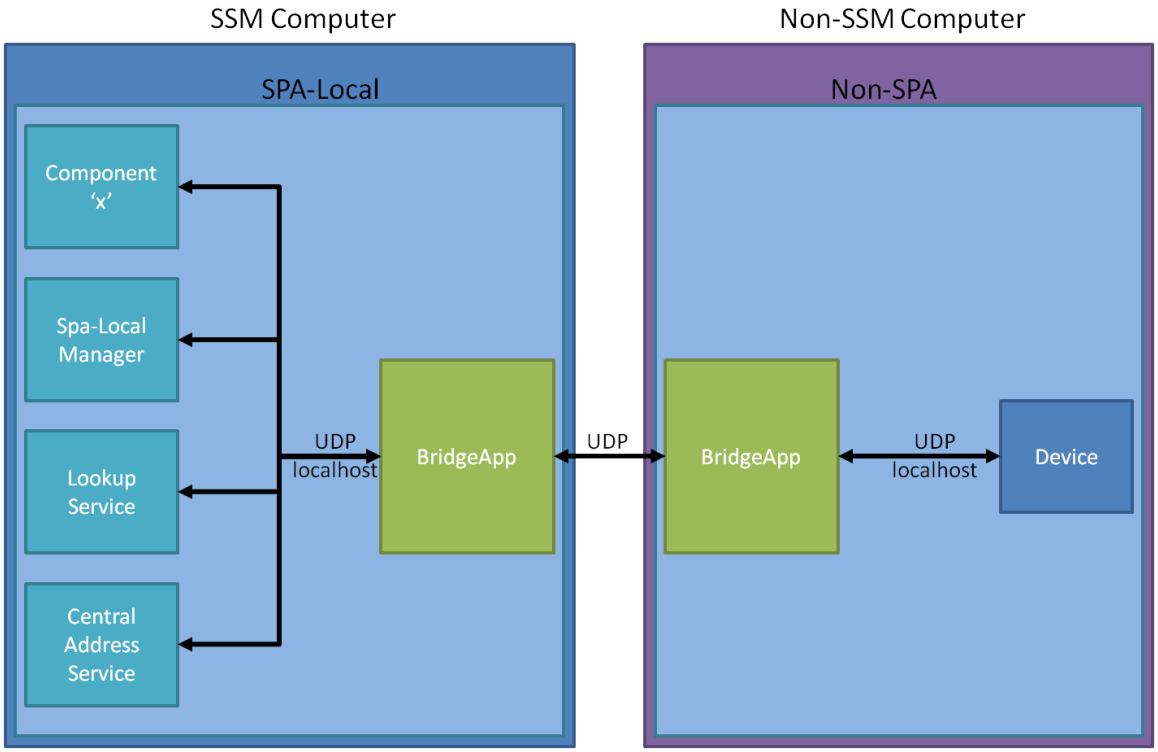

Figure 4.2: A figure describing layout of the BridgeApp.

The purpose of the local BridgeApp instance is to simulate the remote SPA-Local component. The local instance communicates directly with all components on the SPA side including the Lookup Service, SPA-Local Manager, and other components. The local BridgeApp receives message from the remote instance and forwards them to the appropriate recipient. It also forwards messages from the local side to the remote instance. Figure 4.2 gives a graphical view of the local and remote BridgeApp instances.

4.3.7 SensorApp

The SensorApp is one of the most important components in the submarine software. This virtual component is responsible for communicating with the main sensor board. The sensor board reads the pressure, current, voltage, and temperature sensors and relays the information over the serial bus. The SensorApp is responsible for converting the raw readings and sharing the data. Since the depth calculation is derived from pressure, a calibration

feature was added to compensate for surface air pressure and water density. The calibration feature is exposed in the xTEDS as a command message. The SensorApp's xTEDS is as follows:

```
<?xml version='1.0' encoding='utf-8'?>
<xTEDS xmlns='https://pnpsoftware.sdl.usu.edu/redmine/projects/xtedsschema'
       xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
       xsi:schemaLocation='https://pnpsoftware.sdl.usu.edu/redmine/projects/xtedsschema
           https://pnpsoftware.sdl.usu.edu/spa/xteds/current.xsd'
       name='SensorXteds' version='1.0'>
  <Application name='SensorController' kind='application' programMemoryRequired='1'
      dataMemoryRequired='1'/>
  <Interface name='StatusInterface' id='1'>
    <Notification>
      <DataMsg name='CurrentDepth' id='1' msgArrival='EVENT'>
        <Variable name='Depth' kind='depth' dataType='FLOAT32' units='feet'/>
      </DataMsg>
    </Notification>
    <Notification>
      <DataMsg name='RawDepth' id='2' msgArrival='EVENT'>
        <Variable name='Depth' kind='psi' dataType='FLOAT32' units='psi'/>
      </DataMsg>
    </Notification>
    <Notification>
      <DataMsg name='GetTemperature' id='3' msgArrival='EVENT'>
        <Variable name='Temperature' kind='degrees' dataType='FLOAT32' units='Farenheit'
            numberOfArrayElements='3'/>
      </DataMsg>
    </Notification>
    <Notification>
      <DataMsg name='MotorKillState' id='4' msgArrival='EVENT'>
        <Variable name='KillState' kind='state' dataType='UINT08' units='bool' description='
            1 means killed'/>
      </DataMsg>
    </Notification>
    <Notification>
      <DataMsg name='GetSystemPower' id='5' msgArrival='EVENT'>
        <Variable name='SystemVoltage' kind='volts' dataType='FLOAT32' units='volts'/>
        <Variable name='SystemCurrent' kind='current' dataType='FLOAT32' units='amps'/>
      </DataMsg>
    </Notification>
  </Interface>
```

```
  <Interface name='CalibrationInterface' id='2'>
    <Command>
      <CommandMsg name='CalibrateDepthSensor' id='1'>
        <Variable name='ActualDepth' kind='depth' dataType='FLOAT32' units='feet'/>
      </CommandMsg>
    </Command>
  </Interface>
</xTEDS>
```

CHAPTER 5

CONCLUSION

5.1 Benefits

The SSM is a very powerful system. It allows particular software designs to function well together. The Satellite Payload Management software is a perfect example of a software design that worked well with the SSM. It was able to take advantage of the SSM's ability to mask subnets. This allowed the payload management software to manage any number of payloads that were scattered across all of the supported subnets. Similarly, the payload management software was able to take full advantage of the SSM's querying ability to distinguish between participating and non-participating components.

Despite the fact that the SSM was not intended for use in robotics applications, the submarine software had some good results. The communication between the components worked well. Since the SSM handles notifications when components go offline, every consumer was able to compensate. This was most useful during development. A component could be taken offline and modified with out disturbing the other software components. The consumers would simply stop receiving messages during the down time. This is good functionality to have while developing a robotic software system.

Another important aspect of the SSM for use in robotics applications was the ability to handle redundant hardware. Although the example proof of concept did not use this feature, it exists and is very important. For example, if the IMUApp xTEDS was used on multiple IMUs then the existing code would work with out modification. Also, if any of the IMUs failed, then the SSM would handle an automatic switchover when the failed IMU went offline and was deregistered. Along the same line, if the IMU consumer wanted to receive the data from all IMUs, then the code would only need to be slightly modified.

This would give a robotic design a simple method to help it leverage and coalesce the data from all available IMUs. In fact, as IMUs were added or taken away, no code modifications would have to be made.

The ease of adding test code is a benefit to the SSM. At the time of this writing, the SSM contained over 1700 unit tests in 202 files. Those unit tests accounted for an average code coverage of 90 percent. The design for unit tests allows the development code to be completely unchanged for testing. Unfortunately, a more hardware oriented environment would make unit testing more difficult. One approach that has work for the SSM is to create a set of simulated hardware through SPA software only components.

5.2 Problems

Unfortunately, the SSM does not work great for everything. As with a lot of experiments, the negative results tell us the most. The biggest negative result was the need to create a virtual component for every hardware device. Every component in the submarine software, except for one, acted as a virtual component. This doubled the amount of development time and created more points of failure. In a commercial robotics environment, the extra development time would put an increased expense on the use of the SSM and potentially hinder acceptance.

Another problem with using the SSM in a robotic application is the lack of network support. In the proof of concept, a bridging application, called the BridgeApp, was written to allow a single SPA-Local application to communicate with a remote SSM system. This solution works but would not be scalable. Many robotics development teams rely heavily on remote computer interfaces during development, testing, and deployment for real time status information. This would not be practical with the given network limitation.

5.3 Future Changes

There are a few changes or additions that can be made in order to improve the results for unmanned vehicle software. The first change that could be made is the addition of the SPA-USB subnet. A properly vetted SPA-USB subnet would reduce the coding requirements for

unmanned vehicle software by almost half. Instead of a virtual component and firmware for each piece of hardware, each firmware could be made to follow the SPA-USB subnet standards. This would remove all virtual component code and replace them with a single SPA-USB manager.

The next change that should be made is the addition of a SPA-Internet subnet. This subnet could allow automatic bridging from one SPA-Local subnet to another on a remote machine via a network. By adding this, it would not only increase the number of supported subnets, but it would allow a more versatile and broad use of the SSM in robotics. With the SPA-Internet subnet, robotics developers would be able to create software that kept tabs on the state and output of each component. Also, the software required to maintain a live status stream would not have to modify the exiting system at all. The robot local software would be unaware of the status software.

As stated above, the SSM is a powerful tool. It can mean the difference between a system of highly depended software modules and a dynamic plug-and-play system that adapts on the fly. With the above mentioned additions to the SSM, robotics developers will have the tools they need to begin development using the software. This will also open the door to using the SSM in even more areas such as home automation and sensor networks.

REFERENCES

[1] DoD, http://www.whitehouse.gov/sites/default/files/omb/budget/fy2014/assets/defense.pdf, 2013, [Online; accessed January 09, 2014].

[2] J. H. Christensen, "Space plug-and-play architecture networking: A self-configuring heterogeneous network architecture," 2012.