Space Plug-and-Play Architecture Networking: A Self-Configuring Heterogeneous Network Architecture

Jacob Holt Christensen
Utah State University

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

Part of the Computer Sciences Commons

Recommended Citation

This Dissertation is brought to you for free and open access by the Graduate Studies at DigitalCommons@USU. It has been accepted for inclusion in All Graduate Theses and Dissertations by an authorized administrator of DigitalCommons@USU. For more information, please contact digitalcommons@usu.edu.
SPACE PLUG-AND-PLAY ARCHITECTURE NETWORKING: A SELF-CONFIGURING HETEROGENEOUS NETWORK ARCHITECTURE

by

Jacob Holt Christensen

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

DOCTOR OF PHILOSOPHY

in

Computer Science

Approved:

Dr. Scott Cannon
Major Professor

Dr. Stephen Clyde
Committee Member

Dr. Vladimir Kulyukin
Committee Member

Dr. Patric Patterson
Committee Member

Dr. Dan Watson
Committee Member

Dr. Mark R. McLellan
Vice President for Research and Dean of the School of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2012
ABSTRACT

Space Plug-and-Play Architecture Networking: A Self-Configuring Heterogeneous Network Architecture

by

Jacob Holt Christensen, Doctor of Philosophy
Utah State University, 2012

Major Professor: Dr. Scott Cannon
Department: Computer Science

The Space Plug-and-Play Architecture (SPA) networking approach outlined in this dissertation is an improvement over the previous approach used by the Satellite Data Model (SDM). The first improvement is the introduction of a SPA network model based on the Open Systems Interconnection (OSI) model. Second, a new addressing and routing scheme is presented, which places the burden of routing on the network infrastructure instead of the network endpoints. These improvements have been implemented in a software infrastructure called the SPA Services Manager (SSM). The SSM was developed under an International Organization for Standardization (ISO) 9001 certified development process, the details of which are presented. A collection of network timing graphs that measure latency and jitter of the SPA network is contained in this dissertation, as well as a runtime memory footprint. The maturity of the development process and these initial performance measurements demonstrate that the SSM is qualified for spaceflight.
In spacecraft engineering, the time and money involved in satellite construction is largely spent on design and integration of custom hardware and software. These efforts are duplicated for nearly every satellite with little to no reuse between spacecraft. There is a huge potential for cost savings in removing the duplication of work. However, there is a lack of standardization in the spaceflight community, causing soaring costs and delayed schedules as each component of a spacecraft is individually designed and custom built.

The Air Force Research Laboratory (AFRL) has developed the Space Plug-and-Play Architecture (SPA) to address this problem. SPA provides the ability to reuse spacecraft hardware and software components by creating a standard set of protocols used to discover and exchange data between spacecraft components. The first software infrastructure to implement these protocols was called the Satellite Data Model (SDM). After several years of research and development, SPA and the SDM were presented to a team of six industry contractors. Their feedback was that SPA lacked a unified model and the SDM lacked an elegant and scalable networking approach.

This dissertation presents a SPA network model based on the standard Open Systems Interconnection (OSI) networking model and a new addressing and routing scheme for the SPA network. In order to achieve this new networking approach, the standard SPA networking protocols were redesigned and reimplemented in a new software infrastructure called the SPA Services Manager (SSM). In the SSM, the burden of routing data is handled by the network internals and not by the hardware or software endpoints.
ACKNOWLEDGMENTS

It has been said that no great work is done alone. I can attest to the truthfulness of this statement. I would feel ungrateful if I did not acknowledge those who have helped me reach this goal.

- The Space Dynamics Laboratory, for the PhD Fellowship and the use of the many facilities.

- Jim Lyke, for funding the majority of my research, creating SPA, and letting me have an impact.

- Dave Anderson, for sitting through all those APT meetings with me and being my greatest champion with our customers.

- Mark Greenman, for mentoring me and building my confidence.

- Bryan Hansen, for always finding the flaws in my ideas before anyone else could and for being a friend.

- Scott Cannon, for accepting me and pointing me in the right direction.

- Dad, for teaching me to not quit until the job is done (even if you have to work by the light of the riding lawnmower's headlights).

- Mom, for letting me stay up late and watch Star Trek and always encouraging me to do my best.

- Dessie, Andrew, and Hannah for keeping my life in balance with bike rides, wrestling matches, forts made out of blankets, and story time.

- Melinda, for believing in me, and always encouraging me to follow my dreams no matter where they take her. No one has given more to this work than she.
This material is based upon work supported by the United States Air Force under Contract No. FA9453-08-C-0244. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the United States Air Force.

Jacob H. Christensen
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>iii</td>
</tr>
<tr>
<td>PUBLIC ABSTRACT</td>
<td>iv</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>v</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>ix</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>xi</td>
</tr>
<tr>
<td>ACRONYMS</td>
<td>xx</td>
</tr>
<tr>
<td>CHAPTER</td>
<td></td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Space Environment Design Constraints</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Self-Configuring Networks</td>
<td>4</td>
</tr>
<tr>
<td>1.4 The Plug-and-Play Promise</td>
<td>5</td>
</tr>
<tr>
<td>1.5 Space Plug-and-Play Architecture History</td>
<td>6</td>
</tr>
<tr>
<td>1.6 Research Scope</td>
<td>8</td>
</tr>
<tr>
<td>2 RELATED WORK</td>
<td>9</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>9</td>
</tr>
<tr>
<td>2.2 Historical Progress</td>
<td>9</td>
</tr>
<tr>
<td>2.3 Onboard Spacecraft Networking</td>
<td>10</td>
</tr>
<tr>
<td>2.4 Self-Configuring Onboard Spacecraft Networking</td>
<td>11</td>
</tr>
<tr>
<td>2.5 Summary</td>
<td>18</td>
</tr>
<tr>
<td>3 SPA NETWORK ARCHITECTURE</td>
<td>19</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>19</td>
</tr>
<tr>
<td>3.2 The SPA Stack</td>
<td>20</td>
</tr>
<tr>
<td>3.3 The Physical Layer</td>
<td>22</td>
</tr>
<tr>
<td>3.4 The Data Link Layer</td>
<td>25</td>
</tr>
<tr>
<td>3.5 The Network Layer</td>
<td>36</td>
</tr>
<tr>
<td>3.6 The Transport Layer</td>
<td>45</td>
</tr>
<tr>
<td>3.7 The Application Layer</td>
<td>51</td>
</tr>
<tr>
<td>3.8 Summary</td>
<td>57</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.1</td>
<td>Message definition for the LocalHeader message</td>
</tr>
<tr>
<td>B.2</td>
<td>Message definition for the LocalHello message</td>
</tr>
<tr>
<td>B.3</td>
<td>Message definition for the LocalAck message</td>
</tr>
<tr>
<td>B.4</td>
<td>Message definition for the LocalRouteRequest message</td>
</tr>
<tr>
<td>B.5</td>
<td>Message definition for the LocalRoute message</td>
</tr>
<tr>
<td>B.6</td>
<td>Message definition for the OneArp message</td>
</tr>
<tr>
<td>B.7</td>
<td>Message definition for the OneHello message</td>
</tr>
<tr>
<td>B.8</td>
<td>Message definition for the OneAck message</td>
</tr>
<tr>
<td>B.9</td>
<td>Message definition for the UsbHello message</td>
</tr>
<tr>
<td>B.10</td>
<td>Message definition for the UsbAck message</td>
</tr>
<tr>
<td>B.11</td>
<td>Message definition for the SpwHeader message</td>
</tr>
<tr>
<td>B.12</td>
<td>Message definition for the SpwRouterProbe message</td>
</tr>
<tr>
<td>B.13</td>
<td>Message definition for the SpwEndpointPing message</td>
</tr>
<tr>
<td>B.14</td>
<td>Message definition for the SpwEndpointPingReply message</td>
</tr>
<tr>
<td>B.15</td>
<td>Message definition for the SpwConfigureTopologyDiscovery message</td>
</tr>
<tr>
<td>B.16</td>
<td>Message definition for the SpwRouteRequest message</td>
</tr>
<tr>
<td>B.17</td>
<td>Message definition for the SpwRoute message</td>
</tr>
<tr>
<td>B.18</td>
<td>Message definition for the SpaHeader message</td>
</tr>
<tr>
<td>B.19</td>
<td>Message definition for the SpaAck message</td>
</tr>
<tr>
<td>B.20</td>
<td>Message definition for the SpaXtedsRequest message</td>
</tr>
<tr>
<td>B.21</td>
<td>Message definition for the SpaXtedsReply message</td>
</tr>
</tbody>
</table>
B.22 Message definition for the SpaSubscriptionRequest message . . . . . . . . . 130
B.23 Message definition for the SpaSubscriptionReply message . . . . . . . . . . 131
B.24 Message definition for the SpaQueryRequest message . . . . . . . . . . . . . 131
B.25 Message definition for the SpaQueryReply message . . . . . . . . . . . . . . 132
B.26 Message definition for the SpaRequestAddressBlock message . . . . . . . . 133
B.27 Message definition for the SpaAssignAddressBlock message . . . . . . . . . 133
B.28 Message definition for the SpaDistributeRoute message . . . . . . . . . . . 134
B.29 Message definition for the SpaRequestLookupServiceProbe message . . . . 134
B.30 Message definition for the SpaData message . . . . . . . . . . . . . . . . . . 135
B.31 Message definition for the SpaServiceRequest message . . . . . . . . . . . 135
B.32 Message definition for the SpaServiceReply message . . . . . . . . . . . . . 136
B.33 Message definition for the SpaProbeRequest message . . . . . . . . . . . . . 136
B.34 Message definition for the SpaProbeReply message . . . . . . . . . . . . . . 137
B.35 Message definition for the SpaCommand message . . . . . . . . . . . . . . . 137
B.36 Message definition for the SpaAssignAddress message . . . . . . . . . . . . 138
LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Comparing SPA to the OSI model</td>
<td>21</td>
</tr>
<tr>
<td>3.2</td>
<td>The SPA subnet hierarchy</td>
<td>23</td>
</tr>
<tr>
<td>3.3</td>
<td>Probe protocol</td>
<td>27</td>
</tr>
<tr>
<td>3.4</td>
<td>Local discovery protocol</td>
<td>28</td>
</tr>
<tr>
<td>3.5</td>
<td>I²C address resolution protocol sequence diagram</td>
<td>31</td>
</tr>
<tr>
<td>3.6</td>
<td>SPA-1 round-robin</td>
<td>32</td>
</tr>
<tr>
<td>3.7</td>
<td>I²C discovery protocol</td>
<td>33</td>
</tr>
<tr>
<td>3.8</td>
<td>USB discovery protocol</td>
<td>34</td>
</tr>
<tr>
<td>3.9</td>
<td>Reflecting a packet back to oneself in a SpaceWire network</td>
<td>35</td>
</tr>
<tr>
<td>3.10</td>
<td>SpW discovery protocol</td>
<td>36</td>
</tr>
<tr>
<td>3.11</td>
<td>SPA logical address and subdivision of the subnet ID and component ID</td>
<td>39</td>
</tr>
<tr>
<td>3.12</td>
<td>Bootstrapping the SPA address assignment process</td>
<td>41</td>
</tr>
<tr>
<td>3.13</td>
<td>A sample routing table</td>
<td>43</td>
</tr>
<tr>
<td>3.14</td>
<td>Routing a message through a SPA network</td>
<td>45</td>
</tr>
<tr>
<td>3.15</td>
<td>Sample portion of simple xTEDS</td>
<td>46</td>
</tr>
<tr>
<td>3.16</td>
<td>Registration protocol</td>
<td>47</td>
</tr>
<tr>
<td>3.17</td>
<td>A very simple query</td>
<td>48</td>
</tr>
<tr>
<td>3.18</td>
<td>Query protocol</td>
<td>49</td>
</tr>
<tr>
<td>3.19</td>
<td>Subscription Protocol depicting direct subscription and subscription via the Lookup Service</td>
<td>50</td>
</tr>
<tr>
<td>3.20</td>
<td>xTEDS notification protocol</td>
<td>52</td>
</tr>
</tbody>
</table>
C.21 Endpoint 1 histogram - Three routers, message size: 1000 bytes . . . . . . 150
C.22 Endpoint 2 - Summary of all three message sizes on one router . . . . . . . 151
C.23 Endpoint 2 - Summary of all three message sizes on two routers . . . . . . 151
C.24 Endpoint 2 - Summary of all three message sizes on three routers . . . . . 152
C.25 Endpoint 2 scatter plot - One router, message size: 100 bytes . . . . . . . . 153
C.26 Endpoint 2 histogram - One router, message size: 100 bytes . . . . . . . . 153
C.27 Endpoint 2 scatter plot - Two routers, message size: 100 bytes . . . . . . . 154
C.28 Endpoint 2 histogram - Two routers, message size: 100 bytes . . . . . . . . 154
C.29 Endpoint 2 scatter plot - Three routers, message size: 100 bytes . . . . . . 155
C.30 Endpoint 2 histogram - Three routers, message size: 100 bytes . . . . . . . 155
C.31 Endpoint 2 scatter plot - One router, message size: 500 bytes . . . . . . . . 156
C.32 Endpoint 2 histogram - One router, message size: 500 bytes . . . . . . . . 156
C.33 Endpoint 2 scatter plot - Two routers, message size: 500 bytes . . . . . . . 157
C.34 Endpoint 2 histogram - Two routers, message size: 500 bytes . . . . . . . . 157
C.35 Endpoint 2 scatter plot - Three routers, message size: 500 bytes . . . . . . 158
C.36 Endpoint 2 histogram - Three routers, message size: 500 bytes . . . . . . . 158
C.37 Endpoint 2 scatter plot - One router, message size: 1000 bytes . . . . . . . 159
C.38 Endpoint 2 histogram - One router, message size: 1000 bytes . . . . . . . . 159
C.39 Endpoint 2 scatter plot - Two routers, message size: 1000 bytes . . . . . . 160
C.40 Endpoint 2 histogram - Two routers, message size: 1000 bytes . . . . . . . 160
C.41 Endpoint 2 scatter plot - Three routers, message size: 1000 bytes . . . . . 161
C.42 Endpoint 2 histogram - Three routers, message size: 1000 bytes . . . . . . . 161
C.43 Endpoint 3 - Summary of all three message sizes on one router . . . . . . . 162
C.44 Endpoint 3 - Summary of all three message sizes on two routers . . . . . . 162
C.45 Endpoint 3 - Summary of all three message sizes on three routers . . . . . 163
C.46 Endpoint 3 scatter plot - One router, message size: 100 bytes . . . . . . . . 164
C.47 Endpoint 3 histogram - One router, message size: 100 bytes . . . . . . . . 164
C.48 Endpoint 3 scatter plot - Two routers, message size: 100 bytes . . . . . . . 165
C.49 Endpoint 3 histogram - Two routers, message size: 100 bytes . . . . . . . . 165
C.50 Endpoint 3 scatter plot - Three routers, message size: 100 bytes . . . . . . 166
C.51 Endpoint 3 histogram - Three routers, message size: 100 bytes . . . . . . . . 166
C.52 Endpoint 3 scatter plot - One router, message size: 500 bytes . . . . . . . . 167
C.53 Endpoint 3 histogram - One router, message size: 500 bytes . . . . . . . . 167
C.54 Endpoint 3 scatter plot - Two routers, message size: 500 bytes . . . . . . . 168
C.55 Endpoint 3 histogram - Two routers, message size: 500 bytes . . . . . . . . 168
C.56 Endpoint 3 scatter plot - Three routers, message size: 500 bytes . . . . . . 169
C.57 Endpoint 3 histogram - Three routers, message size: 500 bytes . . . . . . . . 169
C.58 Endpoint 3 scatter plot - One router, message size: 1000 bytes . . . . . . . . 170
C.59 Endpoint 3 histogram - One router, message size: 1000 bytes . . . . . . . . 170
C.60 Endpoint 3 scatter plot - Two routers, message size: 1000 bytes . . . . . . . 171
C.61 Endpoint 3 histogram - Two routers, message size: 1000 bytes . . . . . . . . 171
C.62 Endpoint 3 scatter plot - Three routers, message size: 1000 bytes . . . . . . 172
C.63 Endpoint 3 histogram - Three routers, message size: 1000 bytes . . . . . . . . 172
C.64 Endpoint 4 - Summary of all three message sizes on one router . . . . . . . . 173
C.65 Endpoint 4 - Summary of all three message sizes on two routers . . . . . . . . 173
C.66 Endpoint 4 - Summary of all three message sizes on three routers . . . . . . . . 174
C.67 Endpoint 4 scatter plot - One router, message size: 100 bytes . . . . . . . . 175
C.68 Endpoint 4 histogram - One router, message size: 100 bytes . . . . . . . . 175
C.69 Endpoint 4 scatter plot - Two routers, message size: 100 bytes . . . . . . . . 176
C.70 Endpoint 4 histogram - Two routers, message size: 100 bytes . . . . . . . . 176
## ACRONYMS

<table>
<thead>
<tr>
<th>ACRONYM</th>
<th>FULL FORM</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACK</td>
<td>Acknowledge</td>
</tr>
<tr>
<td>AFRL</td>
<td>Air Force Research Laboratory</td>
</tr>
<tr>
<td>AFRL/RV</td>
<td>Air Force Research Laboratory Space / Vehicles Directorate</td>
</tr>
<tr>
<td>AIAA</td>
<td>American Institute of Aeronautics and Astronautics</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>APT</td>
<td>Advanced Plug-and-Play Technology</td>
</tr>
<tr>
<td>ARP</td>
<td>Address Resolution Protocol</td>
</tr>
<tr>
<td>ASIM</td>
<td>Applied Sensor Interface Module</td>
</tr>
<tr>
<td>CCSDS</td>
<td>Consultative Committee for Space Data Systems</td>
</tr>
<tr>
<td>CDH</td>
<td>Command and Data Handling</td>
</tr>
<tr>
<td>CORBA</td>
<td>Common Object Request Broker Architecture</td>
</tr>
<tr>
<td>CAS</td>
<td>Central Address Service</td>
</tr>
<tr>
<td>CDC</td>
<td>Communications Device Class</td>
</tr>
<tr>
<td>CDD</td>
<td>Common Data Dictionary</td>
</tr>
<tr>
<td>CDT</td>
<td>C++ Development Toolkit</td>
</tr>
<tr>
<td>CFS</td>
<td>Core Flight System</td>
</tr>
<tr>
<td>CVS</td>
<td>Concurrent Versions System</td>
</tr>
<tr>
<td>DLR</td>
<td>German Aerospace Center</td>
</tr>
<tr>
<td>ESA</td>
<td>European Space Agency</td>
</tr>
<tr>
<td>I²C</td>
<td>Inter-Integrated Circuit</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electrical and Electronics Engineers</td>
</tr>
<tr>
<td>IPC</td>
<td>Inter-Process Communication</td>
</tr>
<tr>
<td>ISO</td>
<td>International Organization for Standardization</td>
</tr>
<tr>
<td>ITAR</td>
<td>International Traffic in Arms Regulations</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>IV&amp;V</td>
<td>Independent Validation and Verification</td>
</tr>
<tr>
<td>JAXA</td>
<td>Japan Aerospace Exploration Agency</td>
</tr>
<tr>
<td>LRO</td>
<td>Lunar Reconnaissance Orbiter</td>
</tr>
<tr>
<td>LVDS</td>
<td>Low Voltage Differential Signalling</td>
</tr>
<tr>
<td>MODAS</td>
<td>Modular Avionics System</td>
</tr>
<tr>
<td>MSV</td>
<td>Modular Space Vehicle</td>
</tr>
<tr>
<td>NASA</td>
<td>National Aeronautics and Space Administration</td>
</tr>
<tr>
<td>NRE</td>
<td>Non-Recurring Engineering</td>
</tr>
<tr>
<td>OSI</td>
<td>Open Systems Interconnection</td>
</tr>
<tr>
<td>ORS</td>
<td>Operationally Responsive Space</td>
</tr>
<tr>
<td>PAL</td>
<td>Platform Abstraction Layer</td>
</tr>
<tr>
<td>PnP</td>
<td>Plug-and-Play</td>
</tr>
<tr>
<td>RST</td>
<td>Responsive Space Testbed</td>
</tr>
<tr>
<td>RKA</td>
<td>Russian Federal Space Agency</td>
</tr>
<tr>
<td>SCM</td>
<td>Software Configuration Management</td>
</tr>
<tr>
<td>SDD</td>
<td>Software Design Description</td>
</tr>
<tr>
<td>SDP</td>
<td>Software Development Plan</td>
</tr>
<tr>
<td>SDL</td>
<td>Space Dynamics Laboratory</td>
</tr>
<tr>
<td>SDM</td>
<td>Satellite Data Model</td>
</tr>
<tr>
<td>SEU</td>
<td>Single Event Upset</td>
</tr>
<tr>
<td>SHA</td>
<td>Secure Hash Algorithm</td>
</tr>
<tr>
<td>SM</td>
<td>Subnet Manager</td>
</tr>
<tr>
<td>SM-1</td>
<td>SPA-1 Subnet Manager</td>
</tr>
<tr>
<td>SM-L</td>
<td>SPA-Local Subnet Manager</td>
</tr>
<tr>
<td>SM-S</td>
<td>SPA-S Subnet Manager</td>
</tr>
<tr>
<td>SM-U</td>
<td>SPA-U Subnet Manager</td>
</tr>
<tr>
<td>SOIS</td>
<td>Spacecraft Onboard Interface Services</td>
</tr>
<tr>
<td>SPA</td>
<td>Space Plug-and-Play Architecture</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>SpW</td>
<td>SpaceWire</td>
</tr>
<tr>
<td>SRS</td>
<td>Software Requirements Specification</td>
</tr>
<tr>
<td>SSM</td>
<td>SPA Services Manager</td>
</tr>
<tr>
<td>STL</td>
<td>Standard Template Library</td>
</tr>
<tr>
<td>TEDS</td>
<td>Transducer Electronic Data Sheet</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>USB</td>
<td>Universal Serial Bus</td>
</tr>
<tr>
<td>UUID</td>
<td>Universally Unique Identifier</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>xTEDS</td>
<td>Extensible Transducer Electronic Data Sheet</td>
</tr>
</tbody>
</table>
CHAPTER 1
INTRODUCTION

1.1 Introduction

Building a satellite is a tricky thing to do. A traditional satellite takes from 2 to 10 years to build on a budget from $10 million to $8.7 billion (the current cost of NASA’s James Webb Space Telescope [89]). The time and money involved in satellite construction is largely spent on design and integration of custom hardware and software. These efforts are duplicated for nearly every satellite with little to no reuse between spacecraft. The reason for this is the custom nature of the hardware and software. New hardware typically does not use a standard logical interface, therefore new software is required to communicate with it. There is a lack of standardization in the spaceflight community, which causes soaring costs and long schedules as nearly every component of a spacecraft is individually designed and custom built [109].

The spaceflight hardware and software markets are very small compared to other industries. Most spaceflight hardware and software are built and used only once and are very different from typical terrestrial hardware and software in that the intended environments are dramatically different. The space environment poses different challenges to hardware designers from radiation effects to electrostatic charging to micro-meteoroid impacts. Similarly, spaceflight software is different than terrestrial software [55]. Radiation effects can cause single event upsets (SEU) and permanent latch ups forcing spaceflight software to be more robust in order to handle these types of errors that do not occur in terrestrial software. These radiation effects constrain the processing power of spaceflight processors making efficiency and performance a software issue.

A self-configuring network of sensors, processing units, and flight software could greatly
reduce the efforts required to build a spacecraft by reducing duplicated engineering efforts. Standard interfaces could replace custom proprietary interfaces in both hardware and software. A dramatic reduction in time and money could be achieved if hardware and software components were reused across a broad range of satellites.

The Air Force Research Laboratory (AFRL) is looking for ways to reduce the time and money required to build a satellite by using concepts from the modern desktop PC and implementing them in a spacecraft environment. This effort led to the creation of the Space Plug-and-Play Architecture (SPA) [96]. SPA uses well known physical bus and network technologies to create a network for all spacecraft hardware and software. As it was originally developed, SPA lacked a strong network model to guide the addition and development of new networking technologies. SPA did not have any concept of a layered network architecture and the networking approach had devolved into a piecemeal solution.

This dissertation defines a network model based on the Open Systems Interconnection (OSI) model [74] and a layered network approach that fits within the conceptual architecture of the proposed model. The goal of this approach is to provide a network architecture that is transport agnostic and does not place the burden of routing on the network endpoints. To be clear, this dissertation's contribution to SPA is the creation of the SPA network model and the design of its data link and network layers.

1.2 Space Environment Design Constraints

If a typical engineer was asked to build a large robot that was required to function under water, the engineer would immediately understand some of the very basic design constraints. For example, everything would have to be waterproof, the structure would have to be able to handle extreme pressure, and there would be little need for cooling. If the same engineer was asked to build a robot that could fly, again, the engineer would understand the very basic design constraints. The robot would need some mechanism to provide lift, the structure would have be capable of handling impact, and it would all have to be very lightweight. However, when the same engineer is asked to build a spacecraft, there are no natural experiences that this engineer can draw from to help in the
understanding of the space environment. Only engineers that have specifically studied the space environment really understand the constraints that space imposes on the design on electronics and associated software. One possible explanation for this innate understanding of the water or air environment compared to the space environment is the simple experiences the engineer has had in water or air. The engineer has most likely been swimming and has probably flown a kite. However, it is highly unlikely the engineer has ever been in an environment that is even remotely close to the space environment.

The space environment is a very harsh environment for hardware and, by extension, software. The effects of the space environment can be broken down into five categories: vacuum, neutral, plasma, radiation, and micrometeoroid orbital debris. Of these five categories, the radiation environment has the largest effect on space flight micro-electronics and software. “Very energetic (MeV - GeV) charged particles can be found in the trapped radiation belts, solar flare protons, and galactic cosmic rays. The total dose effects of this high-energy radiation can degrade microelectronic devices, solar arrays, and sensors. A single energetic particle can also cause single-event phenomena within microelectronic devices which can temporarily disrupt or permanently damage components.” Single-event phenomena are perhaps the more upsetting to spaceflight software. Highly charged radiation particles pass through almost any matter wrecking havoc along the way. When a charged particle passes through a transistor, it can either add or subtract energy, which can cause a bit to flip in memory, or in the processor itself. With the potential for random bit flips, spaceflight software has to be engineered differently from the ground up. Examples of this are handling error cases that normally cannot be reached, always checking for valid input, and running the same code segment multiple times to verify it produces the same result each time.

Spaceflight software is generally unique, built to be reliable, robust, power efficient, and, where possible, rooted in past space successes. Spaceflight software has many constraints placed on the techniques and technologies that can be used. For example, spaceflight software written in C++ is generally not allowed to use dynamic memory allocation,
templates, or the Standard Template Library (STL). It has not been until the last 10 years that operating systems have started to become accepted on spacecraft. This change has largely been precipitated by the success of the twin Mars rovers, Spirit and Opportunity, which run the VxWorks operating system [30].

Because of the design constraints of the space environment, the space community is highly critical of software. Using terrestrial software onboard a spacecraft is discouraged and looked upon very negatively. Spaceflight software only gets one chance to run correctly and failures are highly publicized [54]. Software built for spaceflight must undergo a very rigorous verification and validation process that most terrestrial software projects never see. The work done in for this dissertation was held to these higher standards.

1.3 Self-Configuring Networks

When each component of a spacecraft has a custom and proprietary interface a lot of time and money is spent integrating all the components together. This includes a hardware effort to develop wiring manifolds, electrical protocols, and even connectors. It also includes a software effort to develop drivers for the custom hardware. Since the hardware and software are new developments, they must both be thoroughly tested. This testing is not just on each single component, but also as components are assembled into a subsystem, each subsystem has to be retested, and then again as the whole system is assembled. This explosion in testing leads to cost growth all because a single component manufacturer uses a non-standard interface.

A self-configuring network is a network that automates the configuration process and removes the need for human expertise, instead replacing it with a well-defined protocol that informs the network hosts of all the information needed to build the routing infrastructure [84]. This infrastructure includes routing tables, logical addresses, and physical addresses. A typical self-configuring network approach contains two phases; the first is a discovery phase where components are found by either probing the network or by the components reporting to a well-known network location. The second phase is registration, when the newly discovered components get assigned a logical address and the information about the
components, including the logical address and routing table updates, are propagated to the rest of the network. A robust self-configuring network will maximize the ability to discover new components and minimize the amount of registration information and the distance that it has to be sent in the network.

Spacecraft development stands to greatly benefit from a self-configuring network. It could provide a significant reduction in the time and cost of integrating new components onto a spacecraft. For a spacecraft to use a self-configuring network each component manufacturer has to use a common communication technology and speak a standard network protocol. The common communication technology does not have to be reinvented. However, in existing communication technologies that are accepted and used in modern spacecraft, there is not a one size fits all solution. A self-configuring network for a spacecraft system should allow for a heterogeneity of communication technology. While many communication technologies do provide self-configuration mechanisms there still exists a need for a unifying network protocol that can combine the heterogeneous communication technologies into a single spacecraft-wide network. This standard unifying network protocol is the core of the self-configuring heterogeneous network.

1.4 The Plug-and-Play Promise

A plug-and-play (PnP) network takes the self-configuring network one step further. The network’s self-configuration addresses the “plug” part of a PnP system. The “play” part is not only setting up communication paths between components, but also allowing the components to find other components that can fulfill their data and processing needs. A self-configuring network can easily be used to build a PnP system by extending the registration phase of self-configuration. Each component provides a description of their data products and processing capabilities to a central indexing service. Components can then issue queries to this service to find the network address of another component. Part of the standard network protocol can then include a standard command and data exchange scheme that will allow any component connected to the PnP network to communicate with all other components on the same network. With this addition of a simple central indexing
service the self-configuring network has become a plug-and-play network.

An onboard PnP spacecraft network enables many possibilities for time and cost savings. Hardware can be built in parallel without having to wait for other components to be completed. Software can be developed to command and control the spacecraft without having to wait for custom drivers or even specifications of the hardware that will actually perform the control. The PnP network can also provide the spacecraft with new capabilities such as some inherent fault tolerance. An example of inherent fault tolerance can occur when a query for data or a capability returns multiple results. If one source fails, the querying component can use one of the other sources.

The biggest promise of a plug-and-play onboard spacecraft network is the cost savings during spacecraft development. These savings are to be found by the eliminating the non-recurring engineering (NRE) costs that occur when every spacecraft component is a one-off solution, reducing the amount of time required for assembly, integration and test (AI&T), and minimizing the amount of independent validation and verification (IV&V) required for new components. In summary, a PnP spacecraft takes less time and money to build because:

- the initial engineering of communication protocols has already been done
- the spacecraft is easier to assemble and test
- the quality assurance process takes less time

All three of these benefits come directly from software and hardware reuse in the spacecraft architecture, design, and implementation.

1.5 Space Plug-and-Play Architecture History

The Space Plug-and-Play Architecture (SPA) was first introduced in 2005 as Space Plug-and-Play Avionics at the AIAA 3rd Responsive Space Conference. “The SPA effort sought to achieve a PnP technology capable of rapidly forming a system, even dynamically, exploiting machine-negotiated interfaces to, in effect, self organize that system. Recognizing
this as a goal, it is clear that the random citation of standards would not be enough. Rather, it was necessary for SPA to follow a different tact, one that drew from the considerable base of terrestrial standards in a way to enforce the vision of PnP needed to make Operationally Responsive Space (ORS) a reality. Though interconnect standards, including USB, SpaceWire, and Ethernet have been chosen, they are themselves not sufficient to achieve PnP. It has been found necessary to supplement the commercial interconnections with other provisions for power and synchronization. More importantly, a software infrastructure was developed to make possible a deeper idea for PnP, one not just capable of supporting automatic component identification, but one capable of device independent interchange, robustness, and flexibility to meet the diverse needs of ORS mission concepts” [96].

From 2005 to 2008 the SPA software infrastructure, called the Satellite Data Model (SDM), was developed at Utah State University by Dr. Scott R. Cannon et al. In May of 2008 I joined Dr. Cannon's team and helped maintain the SDM as it was being prepared to fly on PnPSat [23], AFRL’s first plug-and-play satellite [114]. PnPSat was the first choice to launch on third Space Exploration Technologies’ Falcon 1 rocket [129]. However, PnPSat was not ready in time to catch its ride to space. The PnPSat program was canceled due to trouble during final integration and test. For their next PnP satellite effort AFRL decided to involve industry contractors to help refine the SPA technology. In October 2009 six industry contractors were selected for AFRL’s Advanced PnP Technologies (APT) program. These six contractors (Comtech AeroAstro, Broad Reach Engineering, Miltec Corporation, Northrop Grumman, SEAKR Engineering Inc., and Sierra Nevada Corporation) were tasked with refining the SPA concepts for future commercialization of the SPA technology.

When the six APT contractors started using the SDM there quickly arose a concern with the networking scheme. The scheme worked by using IP addresses as logical addresses for the network. The IP address was parsed into different sections with different semantic meanings. When a processor received a packet with an address, it looked at certain bits in the address and decided where to forward the packet. It then changed the contents of the address for the next hop in the network. This caused confusion because the algorithm for
changing the address was different depending on how far the packet had to travel, which
bus technologies it had to traverse, and where it had come from. This problem was caused
by the lack of a strong architectural model for the SPA network. New bus technologies were
added to the network in an ad-hoc and evolutionary manner without following any sort of
design.

There were many teleconferences where the six APT contractors would debate the need
to change the networking scheme and the desired strategy for doing so. It was after one such
teleconference that I decided I could solve this problem. After several days I had refined my
ideas into a 190 slide PowerPoint presentation. In June of 2010 I traveled to Albuquerque,
New Mexico and presented my ideas to the APT group (see Appendix A). This was the
first time that a concrete idea had been put in front of the group, and it spawned a lot
of competing ideas. Over the following month a debate ensued. In the end the debate
was brought to a vote by the APT committee. Several other proposals were considered,
however, in the end my idea was selected by the committee and approved by AFRL.

1.6 Research Scope

After the APT committee selected my idea as the SPA Networking standard, I was
given the opportunity to develop a working implementation. This dissertation presents
the body of research demonstrating that I have created a network infrastructure that is
transport agnostic and does not place the burden of routing on the network endpoints. I
present a SPA model which organizes the network into a layered architecture that loosely
follows the OSI model \[74\]. The designs for the data link and network layers are presented.
In order to give the data link and network layers context, the other layers of the network are
also presented. It covers the related work, design, implementation, experimental setup, and
some testing results from the SPA network implementation. The focus of this dissertation is
not the implementation, but instead the network model, addressing, and routing approach.
However, one contribution of this work is the working implementation of a self-configuring
heterogeneous network suitable for spacecraft flight and operation.
CHAPTER 2
RELATED WORK

2.1 Introduction

There is a large volume of research related to self-configuring networks. There is also a significant amount of research and real world implementations of heterogeneous networking (i.e. the Internet). There is a much smaller amount of research related to the application of these two technologies on a spacecraft system. The research in the area of self-configuring heterogeneous spacecraft networks largely covers the requirements of the spacecraft system itself. Due to the space industry’s aversion to risk and the cost of launching a spacecraft, it is difficult to find an actual flight with any technology that is considered to still be in a research phase. This creates a classic chicken and egg problem. It is difficult to fly research and it is difficult to move out of the research phase without flight heritage. This creates a research environment where it is very difficult to make progress. While there are several organizations that have researched self-configuring spacecraft networking, only three are still continuing the research. This chapter gives a summary of the state of the research in applying these two technologies to spacecraft systems.

2.2 Historical Progress

The concept of using a modular spacecraft design to reduce cost is not new. Modularity has been researched in every part of spacecraft design including the physical structure [69]. The Multimission Modular Spacecraft standard was developed by NASA as early as 1978 [31]. The Multimission Modular Spacecraft standard was used for six spacecraft from 1980 to 1992. In 1995 IEEE published the IEEE 1355-1995 standard for Heterogeneous Interconnect [71]. In 2003, IEEE 1355 was reworked for spaceflight application, which gave
birth to SpaceWire [53, 56]. In 1997 IEEE published the IEEE 1451.2-1997 standard for Transducer Electronic Data Sheets (TEDS) which allows a device to describe its capabilities and data products [72]. In the early 2000’s software architectures for configuring transducer networks and creating modular spacecraft started to appear [104, 119]. In 2005 one of these new ideas was for a plug-and-play spacecraft using the USB and SpaceWire standards to create an onboard spacecraft plug-and-play network. This was the beginning of the Space Plug-and-Play Architecture [96].

2.3 Onboard Spacecraft Networking

Spacecraft systems do not typically use any kind of network technology, instead utilizing point to point data connections between components. This is due to the inherent lack of determinism in network technology compared to the solid determinism of point to point connections. Over the last decade there has been an increase in the research for onboard spacecraft networking. The first widely adopted network technology developed for spacecraft use was SpaceWire [3, 115]. SpaceWire is a version of the IEEE 1355 standard that has been enhanced to use low voltage differential signalling (LVDS), which is more reliable in the space environment. SpaceWire also uses path-based wormhole routing, which makes it acceptable for real-time use [116]. The first draft of the SpaceWire standard was released in 2003 [56] by the European Space Agency (ESA) through their European Cooperation on Space Standardization (ECSS) group. SpaceWire has seen wide adoption from the United States National Aeronautics and Space Administration (NASA), the Japan Aerospace Exploration Agency (JAXA), and the Russian Federal Space Agency (RKA). In the later half of the 2000’s the literature starts to mention SpaceWire for plug-and-play applications [52, 105]. Today SpaceWire is still under active development and is used on many spacecraft [58, 117].

Research was started on using Ethernet onboard a spacecraft [145], but due to the maturity and adoption of SpaceWire, which has a higher bandwidth and better real-time performance, the research did not progress. Ethernet for spaceflight applications is still in a research phase.
2.4 Self-Configuring Onboard Spacecraft Networking

Several organizations have participated in the research of self-configuring heterogeneous networking for spacecraft systems. These organizations can be divided into three groups: academia, industry, and government. Academia is focused on the educational aspects of developing a spacecraft. Industry is focused on getting contracts from government organizations. It is in the government sector that we find the pursuit (and the money) to develop the capabilities that are needed to achieve actual progress in the research.

2.4.1 Academia

Academia has proposed several approaches and architectures for self-configuring onboard spacecraft networks. In 2005 the Adaptive Network Architecture [138] was proposed as a software architecture designed for real-time operations of multiple spacecraft grouped in a constellation or formation. In 2006 it was extended to serve as a mechanism for managing onboard science processing [139]. The modifications allowed for runtime reconfiguration and redeployment of software components across a set of processors. This work was discontinued after the presentation of the 2006 paper [139].

In 2010 the SpaceCraft Area Network (SCAN) was introduced [111,112]. SCAN uses a collection of hardware switches [120] to translate between different communication protocols. SCAN uses the Realtime Onboard Dependable Operating System (RODOS) [110] as its underlying software architecture for message passing and data publishing and subscription. There is very little information in the literature about RODOS as it is a proprietary system from the German Aerospace Center (DLR).

In 2011 a new plug-and-play architecture was proposed [122] that focuses on plug-and-play at the satellite subsystem level. It uses microcontrollers to implement the intelligence required for self configuration. In order to accomplish self discovery, standardized interface IDs are used. When the onboard computer receives an interface ID, it knows which subsystem it is communicating with and what data it provides as well as what commands it can receive. This approach is similar to how USB uses its device classes to create plug-and-play.

It should be noted that after 2006 the only research being done by academia takes place
outside of the United States. The majority of the research being done in the US is related to AFRL’s Space Plug-and-Play Architecture (SPA). Due to the large amount of research and resulting literature, SPA is discussed in section 2.4.3. This split between the domestic and foreign research was caused in 2008 when the US Congress decided that the SPA effort fell under the International Traffic in Arms Regulations (ITAR) restrictions, and could not be shared outside the United States.

2.4.2 Industry

The industry call for spacecraft hardware and software interface standardization started as early as 1996 [34]. Research for self-configuring onboard spacecraft networks started five years later in a paper published at the IEEE Aerospace Conference [67]. In this paper the author called for a network based approach instead of the individual wiring harnesses. The author also suggested using the IEEE-1451 TEDS standard to achieve a small measure of self-configuration. This paper can be seen as a founding paper for the work done in this dissertation.

In 2003 a paper was presented on self-configuring networks for satellite avionics at the GOMACTech conference [51]. This paper called out the need for more self-configuration in the increasing research of onboard spacecraft networking. However, it was short on details to accomplish the self-configuration.

In 2004 the SCOUT program was presented [125] as a development program to create a modular multimission spacecraft architecture. SCOUT proposed the term “Plug-and-Sense” and defined it as the ability to create “smart” systems by expanding the electrical and functional connectivity advantage to include physical properties, orientation, location, and synergistic aspects of a device. This allowed SCOUT modules to convey extra meta data to the spacecraft system such as dimensions, mass properties, position in the spacecraft, and dynamic attributes. In addition a SCOUT module was able to transfer necessary software drivers, functional code modules, or even new software frameworks to the rest of the spacecraft system. The architecture developed in the SCOUT program was presented again in 2005 [76] under the name of SMARTBus as the name of the overall architecture.
and Astrologic as the name of the communication layer's software implementation. AstroLogic used User Datagram Protocol (UDP) communication over local loopback, Inter-
Integrated Circuit (I²C), and serial physical connections. Instead of using a self-discover
and self-configuration approach, SMARTBus uses statically assigned Internet Protocol (IP)
ports to address subsystems in the system. All messages exchanged in the system were
done in Extensible Markup Language (XML). Development of SMARTBus and Astrologic
has not been seen in the literature since 2006. Many of the concepts and design principles
found in SMARTBus and Astrologic have been considered, augmented, and implemented
in the SPA Network architecture.

In 2007 the Space Plug-and-play Architecture (SPA) became the dominate platform
and almost all other US industry research united behind the SPA effort. The few exceptions
are the research efforts that are being continued by NASA and its contractors.

2.4.3 Government

All of the research done by academia and industry can trace its motivation and funding
back to the government sector. Typically, a government customer has an idea and seeds re-
search money to industry and academia to do the research. All of the industry and academic
research has been adopted by government and distilled down to 3 efforts lead by the US
National Aeronautics and Space Administration (NASA), the Consultative Committee for
Space Data Systems (CCSDS) group, and the US Air Force Research Laboratory (AFRL).

Core Flight System

NASA started research on plug-and-play spacecraft systems as early as 1997. The
Essential Services Node (ESN) is a multi-chip module that uses the standard MIL-STD-
1553B and MIL-STD-1773 data link protocols to interface with the other spacecraft sub-
systems and bus. The ESN reads commands and data from the spacecraft instruments
or subsystems, processes the information, and sends it to the command and data handling
(CDH) system. It also receives commands from the CDH system, processes the information,
and sends it to an instrument or a subsystem. The ESN is a hardware unit that provides
the plug-and-play capability to the spacecraft system.

In 1999 the Space Object Technology Group (SOTG) was created. The purpose of SOTG is to establish industry standards for object-oriented software to be used in the space industry to enable plug-and-play interoperability among software such as orbital analysis, mission planning, maneuver planning and scheduling tools [1]. The reference plug-and-play architecture that came out of the SOTG efforts was intended to be deployed on the Common Object Request Broker Architecture (CORBA) [35,66]. The spaceflight community felt that CORBA was too much for a spacecraft to handle [41]. The SOTG was disbanded in the early 2000’s.

By 2005 NASA had started working on the Core Flight System (CFS) [148–151]. The Core Flight System is a mission independent, platform independent, flight software environment integrating a reusable core flight executive. The CFS provides a inter-task message router called the software bus. The software bus allows software to communicate with other software tasks on a flight processor. A software task is not a separate thread or a separate process, but instead a part of the same process and same thread that is executing a different conceptual task. This is a common architecture in spaceflight software. The software bus allows tasks to post messages to a named pipe. Other tasks can then take messages from the same named pipe. This software bus accomplished “software plug-and-play” because the code to communicate between tasks did not have to be re-developed. To integrate hardware devices CFS requires device drivers to be developed for each hardware device attached to the spacecraft. A software application is then developed to communicate with one or more hardware devices and make the devices data available on the software bus. In 2009, CFS was flown on the Lunar Reconnaissance Orbiter (LRO) [4]. In 2010 a group from the Applied Physics Laboratory at John Hopkins University based their proposed plug-and-play architecture on NASA’s Core Flight System (CFS).

A 2006 paper [64] again proposes using CORBA to create a plug-and-play software environment allowing additions and subtractions of data sources and command recipients. This works was performed on a drilling station using modern personal computers as the
computing platform. Again, it has been shown that CORBA is too processor intensive for typical spaceflight processors [41].

Most recently in 2011, a group out of NASA’s Jet Propulsion Laboratory has put together a plug-and-play environmental monitoring spacecraft subsystem [118].

**Spacecraft Onboard Interface Services**

The Consultative Committee for Space Data Systems (CCSDS) was founded in 1982 by the major space agencies of the world. The CCSDS is a multi-national forum for the development of communications and data systems standards for spaceflight. In the mid 2000s the CCSDS started to work on standardization of spacecraft onboard interfaces. By 2007 the group had created a new architecture called Spacecraft Onboard Interface Services (SOIS). The CCSDS report concerning SOIS [42:p. 99] states

The SOIS approach is to standardise the interfaces between items of spacecraft equipment by specifying well-defined standard service interfaces and protocols which allow standardised access to sensors, actuators, and generic spacecraft functions, allowing spacecraft applications to be developed independently of the mechanisms that provide these services. Applications are thus insulated from the specifics of a particular spacecraft implementation and may be reused across different spacecraft platforms with little regard of implementation details. Service interface standardisation allows hardware interfaces to be accessed by flight software such that core spacecraft software may be reused on different underlying communications infrastructures with little or no change. The standard services could be implemented using a standard Application Programming Interface (API) that would enable portability and re-use of application software, and of service implementations.

One part of the SOIS architecture is the Device Enumeration Service which is intended to be used to support future plug-and-play applications. This service is still to be defined. In December 2011 I was invited to the CCSDS SOIS meeting to help define the Device
Enumeration Service. Since SOIS is an architecture and not an implementation, the CSSDS SOIS group wanted to make the plug-and-play architecture compliant with AFRL’s Space Plug-and-play Architecture (SPA) so that both groups could unite and move forward in the face of space industry budget cuts.

**Space Plug-and-play Architecture**

The Air Force Research Laboratory started researching self-configuring networks to support the Operationally Responsive Space (ORS) effort. Although early ideas that led to the creation of the Space Plug-and-play Architecture (SPA) can be found in 2001 [40,90,98], and 2004 [86,113], SPA first appears in the literature by name in 2005 at the Responsive Space conference [96]. These efforts were led by Dr. James Lyke of AFRL’s Space Vehicles division and Dr. Scott Cannon of the Utah State University Computer Science Department [94,97]. Also in 2005 a plug-and-play testbed was created at Kirtland Air Force Base called the Responsive Space Testbed (RST) [135]. The RST would become the center of SPA research and activity for the next 7 years.

In 2006, the Satellite Data Model was introduced [39,136,137]. The Satellite Data Model is the underlying software system that enables the self-configuration and self-discover of the spacecraft systems. Since its introduction the SDM has been used on several cubesats and on two international spacecraft [36,37,85]. It is upon the work of the Satellite Data Model that this dissertation builds.

The AFRL’s Space Vehicles Directorate (AFRL/RV) made Responsive Space one of its six core thrusts and the TacSat-3 mission integrated a small SPA experiment onboard [146]. Due to the large amount of funding and time being spent on SPA, many progress report style papers and Air Force news articles were published [68,91,128]. There was also a first attempt at standardizing SPA in a set of documents published by the AIAA, however, this effort did not produce the needed documentation [130]. One of the areas where standardization was attempted and has never fully been solved is the Common Data Dictionary (CDD) or SPA Ontology [87,131].

SPA has been proposed for use in advanced spacecraft concepts [92], such as satellite
constellations [100], PC-like satellite construction [77], and the metric for an advanced modularity measurement for spacecraft system [134]. One advanced spacecraft concept was the first fully SPA spacecraft, PnPSat-1 [23,60,62,63,65,101]. PnPSat-1 was never launched due, in part, to problems with the SDM’s networking approach. This is one of the reasons that led to the creation of the Advanced Plug-and-play Technology (APT) program, which was tasked with maturing the SPA technology. The PnPSat-2 technology testbed was also created to perform advanced characterization of the SPA network [61]. Another advanced spacecraft concept that has been demonstrated with SPA is the automatic generation of device firmware and flight software from the electronic data sheet used by SPA [47,48].

The literature contains several papers about the various SPA subnet types. SPA-SpaceWire was first published in 2007 [106]. SPA-Optical was first published in 2010 [59]. SPA-1 based on the I^2C standard was also first published in 2010 [95,142]. SPA-1 was selected for the TrailBlazer cubesat mission [81,83]. SPA-1 was also selected as an interface for a laboratory experiment sent to the International Space Station [75]. SPA-USB and SPA-SpaceWire were flown on the TacSat-3 satellite as an experimental bus technology [102].

The interface for hardware devices and a SPA network can be encapsulated into the device itself [99]. For legacy devices that do not already have this capability a sensor interface called the Appliqué Sensor Interface Module (ASIM) was developed [127] and has gone through several iterations [33]. ASIMs have been developed for SPA-1, SPA-U, SPA-S, and SPA-Optical [107,126,144]. Along with the hardware development, the software development has been seen in the literature. Dr. Ken Center and Mr. Robert Vick have published multiple papers on the tools, software strategy, and complexity of developing SPA software modules [43,45,143].

SPA has been adapted for the CubeSat platform [108,123]. Several CubeSats have been developed using SPA [81,83]. In 2010, a SPA training course was developed around CubeSat technologies. Over 500 individuals representing more than 100 companies, universities, and government agencies have participated in this training [79,80].

In 2010, the Advanced Plug-and-play Technology (APT) program was initiated. The
APT program was a group of six industry contractor teams given the assignment to move SPA from an AFRL-funded research project to an industry adopted spacecraft standard. It was during this program that the events outlined in section 1.5 took place. One of the outcomes from the APT program was the SPA Standards \[6,7,9,10,28\] which were released in draft form in February 2011. There has already been literature published on the SPA network architecture as it is presented in this dissertation \[49,50\]. The reader will note that the primary author of these papers is the same author as this dissertation.

2.5 Summary

In summary, there are three primary platforms that have come out of the last 20 years of research. NASA’s Core Flight System, CCSDS’s SOIS architecture, and AFRL’s Space Plug-and-play Architecture. Of these three platforms, SPA has been the most published and perhaps the best funded. In all the research that has been done on SPA, the most difficult problems have never been technical. Dr. James Lyke described the most difficult challenge best when he said, “Coming up with the satellite architecture was pretty hard, but convincing a risk-averse aerospace industry to even consider our approach has been even harder” \[93\].
CHAPTER 3

SPA NETWORK ARCHITECTURE

3.1 Introduction

In a PnP system, machine-negotiated interfaces are used to enable components to interoperate. The PnP process of electronic self-discovery and self-configuration eliminates the need to develop specialized hardware and software interfaces for each spacecraft. The Space Plug-and-Play Architecture (SPA) supports a method of constructing arbitrarily complex arrangements of components, and is a networked data exchange model. One of the premises of SPA is that there is no distinction between a hardware device that supports a data interface and a software application that does the same.

A typical spacecraft system involves many different components that vary in bandwidth demand. Sensors that require a very low data rate may reside on a simple two-wire interface such as I²C. Complex sensors, such as an advanced imager, that require a much high data rate may reside on a SpaceWire or optical interface. A spacecraft system is also likely to have a number of components of intermediate performance, with a data rate greater than the simpler sensors, but lower than high-performance payloads. A SPA system needs to be able to support multiple types of interconnection networks, for both hardware and software components, that are dramatically different in their addressing schemes and routing capabilities.

The SPA networking infrastructure is a transport agnostic approach which allows a SPA component to communicate with any other SPA component without a prior knowledge of where the component is physically located on the network or what type of interconnection network it uses. This approach has been reviewed and accepted by the SPA standards committee. The SPA network provides interoperability across existing heterogeneous inter-
connection networks and a methodology for adding any number of future network technologies without affecting existing SPA components. This chapter presents the SPA network architecture in relation to the standard five layers of the Open System Interconnect (OSI) model \[74\].

### 3.2 The SPA Stack

The OSI model is a layered model of abstraction levels for communications \[74\]. The OSI model is a broadly recognized and industry standard means of providing an ordered, flexible, and extensible communications system architecture. Each layer of the OSI networking model encapsulates and addresses a different aspect of the communication system requirements.

Use of a layered architecture provides many benefits. In a layered architecture, each layer provides services to higher level layers and receives services from the layer below. The modularity of the layers promotes ease of understanding the architecture. The design of each layer can be addressed individually, reducing the complexity of the associated communication system. Each layer can be developed independently, can be tested at the layer boundaries, and is easier to maintain. Since the responsibilities of the layer are well categorized, swapping out a layer with an alternate implementation has minimal or no impact on other layers.

A simplified form of the OSI abstraction layers can be composed into a hierarchy of five abstraction levels, where the application layer includes the traditional session and presentation layers as well. Similar levels of abstraction have been applied to the design of the SPA architecture. The layers in the architecture model for SPA are the physical, data link, network, transport, and application levels (see Figure \[3.1\]).

In the OSI model, the physical layer defines the electrical and physical specifications for devices and the relationship between a device and a transmission medium. The physical layer provides for conversion between the physical transmission medium and digital data, the establishment and termination of connections, and provides for flow control and contention resolution. The physical layer of the SPA model defines standards for the physical
Figure 3.1. Comparing SPA to the OSI model

interconnect between devices in a specific physical subnet type, and includes designs for local UDP sockets, SpaceWire, I²C, and USB.

The data link layer provides the means to transfer data between the network participants, including discovery of physical addressing, and for correction of errors that may occur in the physical layer. The SPA model provides these capabilities utilizing subnet-specific protocols, denoted generically as SPA-X with the X indicating the physical subnet type (i.e. SPA-U for the SPA USB subnet). The data link layer is responsible for making up any shortfalls in the physical layer’s ability for self-discovery or asynchronous communication.

The network layer provides for the transfer of variable length data messages from a source host on one network to a destination host on a different network. The network layer provides routing, and utilizes a logical addressing scheme. Where required, fragmentation and reassembly of messages occurs at this layer. The network layer handles the convergence of messaging traffic from subnet-specific transfer to network independent messaging. The network layer also handles reliability requirements such as message acknowledgement and retransmission. The SPA model provides these capabilities through the use of the SPA Logical Messaging protocols. More information on the data link and network layers can be
found in the SPA Logical Standard \[7\].

The transport layer enables the connection and discovery between network endpoints. In the SPA model, the discovery and connection is made between clients and services based upon specifications of the endpoint interfaces in an XML specification known as the xTEDS. All network component’s xTEDS are collected and indexed in a central lookup service. Network components can issue queries to the lookup service to discover other components in the network.

The application layer is the layer which interacts directly with the software applications that implement a communicating component. In the SPA model, network components can be software applications executing on a general processor resource or a device on the network. The communication and interoperation is independent of the nature of the resource, and consumers of services and data are unaware of physical type or physical network location of their producers.

3.3 The Physical Layer

A SPA network is composed of several different communication technologies. Each of these are referred to as a SPA subnet. The SPA standards currently support four different subnets. They are: I\(^2\)C, USB, SpaceWire, and local UDP sockets. In order to understand the overall approach of a SPA network it is important to understand the requirements for a SPA subnet and how each networking technology measures up to those requirements. Each subnet has different capabilities and none of them meet all of the requirements for a SPA subnet; therefore, each must be augmented in its own way.

It has often been asked why there are so many different SPA subnets. The answer to that question is best explained by Figure 3.2. A single SPA subnet is not the solution to all problems. An example of this is a simple temperature sensor. A systems engineer would not appreciate the power requirements of the SpaceWire interface that would be required to have that single temperature reside on the SpaceWire network. There would also be a huge waste of bandwidth. An I\(^2\)C bus would be sufficient and has lower power requirements. The inverse of this could be a high-resolution imaging device that has many gigabytes of data to
transfer. Even though the low power requirements of the I²C may be attractive, the higher bandwidth of a SpaceWire interface would be more desirable. A system engineer should choose the SPA subnet that consumes the least amount of power and still meets bandwidth requirements.

There are two primary capabilities that are required of all SPA subnets:

1. Discover currently attached components
2. Send and receive messages asynchronously

When a subnet is not capable of performing one of these two tasks, it has to be augmented in the data link layer. These necessary augmentations are described in Section 3.4. The rest of this section compares I²C, USB, SpaceWire, and local UDP sockets and discusses how they compare to the required capabilities enumerated above.

3.3.1 I²C

I²C [2] (Inter-Integrated Circuit) (generically referred to as “two-wire interface”) is a multi-master, serial, single-ended bus invented by Philips. An I²C physical address is a 7-bit address. Being a two-wire bus creates a single broadcast domain. This means all
devices on the bus receive all communications from all other devices. Each communication is preceded by the seven bit address of the device that is being spoken to. Only the device which the corresponding address responds to the communication. I²C does not have the native capability to discover the devices on the bus. Due to the master-slave communication paradigm, an I²C bus does not allow asynchronous sending and receiving of messages. This makes the I²C bus the least capable subnet and requires the greatest augmentation of its capabilities.

### 3.3.2 USB

USB [16] is a two-wire bus network that is found on many terrestrial systems including PCs, mobile devices, consumer electronics, and even automobiles. USB was developed by Ajay Bhatt while working for Intel. USB is a master-slave bus, in which communication is managed by a host controller. USB devices cannot initiate data transfers, but instead only respond to requests given by the host controller. As with I²C, the master-slave nature of USB inhibits devices from asynchronous sending and receiving of messages. However, standard USB host controllers implement a round-robin polling loop that allows this to be overcome. This same round-robin polling loop allows USB to discover the devices that are currently attached to the bus.

### 3.3.3 SpaceWire

SpaceWire [3] is a communication network designed specifically for spacecraft. It is coordinated by the European Space Agency (ESA) in collaboration with international space agencies including NASA, JAXA and RKA. A SpaceWire network consists of nodes that are connected through low-cost, low-latency, full-duplex, point-to-point serial links, and packet-switching wormhole routers. Because of the full-duplex serial links, SpaceWire components can send and receive message asynchronously. However, SpaceWire uses path-based routing, and a physical address is determined by the path between two components on the network. The physical address used to send data from one component to another is dependent on the topology of the network and is unique from each components perspective. This makes
discovery of the components that are attached to a SpaceWire network very difficult.

3.3.4 Inter-Process Communication Using UDP Sockets

The last subnet type discussed in this section is not like the others. Applications running on a processor are treated as though they live in a sub-network. This subnet is the inter-process communication (IPC) network. In SPA, the IPC network uses sockets as the transmission media. Sockets were chosen for their portability, pervasiveness, and ease of use. Because there are no physical wires to communicate on, the sockets can send and receive messages asynchronously quite easily. Each socket is assigned a port to communicate on. The port functions as a 16-bit address for the software process. This creates a possibility of 65536 possible addresses making it difficult for an application to detect which ports are used by a SPA software process.

3.4 The Data Link Layer

The purpose of the data link layer is to provide the means to transfer data between the network participants. To enable data transfer, this layer includes network discovery at a physical addressing level. In a SPA network, facilitation of data transfer and discovery is handled by components known as SPA subnet managers. The subnet managers are the routing elements in the SPA network architecture. They are responsible for making up any shortfalls in the physical layers ability for self discovery or asynchronous communication.

3.4.1 SPA Subnet Managers

Each physical network type requires a certain degree of management in order to perform physical address discovery and asynchronous communication as required by the SPA network. This management is handled by software components known as SPA subnet managers. A SPA subnet manager has several key responsibilities at the data link layer in a SPA network. It is responsible for discovering the components within its subnet, routing messages in and out of its subnet, and monitoring the health and state of those discovered components. Each SPA subnet being managed by a SPA subnet manager is associated with
a different physical transport medium: SPA-Local uses UDP sockets, SPA-U uses USB, SPA-1 uses I²C, and SPA-S uses SpaceWire. This set is expandable as different physical transport networks are brought into SPA. It is also important to note that because a SPA subnet manager is a software process, it is capable of communicating on both the SPA-Local subnet of the processor on which it executes as well as on the physical transport subnet that it manages.

A SPA subnet manager is responsible for discovering the components that reside on the subnet it manages. The discovery protocols differ for each subnet type due to the different capabilities each provides. An example of these differences can be seen when comparing SPA-Local and SPA-1. A software component utilizing UDP sockets can proactively alert the SPA-Local subnet manager of its presence on the subnet when it starts execution. In contrast, a device on an I²C bus is a slave and cannot initiate communication with the SPA-1 subnet manager. Therefore, a SPA-1 device must wait to be discovered by the SPA-1 subnet manager. The SPA subnet manager is responsible for bridging the gap between its subnets inherent discovery capability and the discovery requirements for a SPA subnet. In order to bridge this gap, subnet specific data link layer discovery protocols are defined. These protocols contain physical address information to enable discovery. Because the protocols contain physical address information they never leave their respective subnets. This makes sense because a socket based subnet does not care about SpaceWire routing paths.

Another responsibility of the SPA subnet manager that is related to discovery is monitoring for components when they become unavailable. This is accomplished through a simple heartbeat mechanism, where the subnet manager sends a SpaProbeRequest (Table B.33) message to each component in its managed subnet and expects a SpaProbeReply (Table B.34) message in return. If the SpaProbeReply does not come back after several retries, the subnet manager assumes the component is no longer available and takes the necessary steps to remove the component from the network. This simple protocol is outlined in Figure 3.3.

The SPA subnet manager is also responsible for enabling asynchronous communication
in its managed subnet. Not every subnet possesses the inherent capability to send and receive messages asynchronously. Again this can be seen by contrasting SPA-Local and SPA-1. UDP sockets can send packets at anytime to any other local UDP socket, with no need to wait for permission from any kind of master or host. However, I²C is a master-slave bus and therefore a slave device cannot initiate communication, but must instead wait for the master to initiate communication. Again it is the SPA subnet manager’s responsibility to bridge the gap between the subnet’s inherent message transmission capability and the message transmission requirements for a SPA subnet.

The following sections discuss the native discovery and message transmission capabilities of each SPA subnet and the strategy the associated subnet manager uses to augment the inherent capabilities of the subnet.

### 3.4.2 SPA-Local

The SPA-Local subnet consists of applications running on a single processing node in the network. A distinct SPA-Local subnet exists for each processing node in the network. These applications consist of user software, SPA subnet managers, and any other software process that participates on the SPA network. A SPA-Local subnet utilizes sockets as
an IPC mechanism to allow applications running on the same processor to communicate with each other. More information on SPA-Local can be found in the SPA Local Subnet Adaptation Standard [6].

**Discovery**

The SPA-Local discovery protocol (Figure 3.4) consists of two messages in a simple request-reply type of protocol. The SPA-Local component sends a LocalHello message (Table B.2) to the SPA-Local subnet manager. The SPA-Local subnet manager replies with a LocalAck (Table B.3). Once the LocalAck has been received, the SPA-Local subnet manager has discovered the SPA-Local component. Until the LocalAck has been received the SPA-Local component continues to resend the LocalHello. The SPA-Local discovery protocol is unique because it is the only discovery protocol where the component actually notifies the subnet manager of its presence. This is done because of the nature of a local socket. The SPA-Local subnet manager listens on the well-known port of 3500 and the SPA-Local components get their port numbers randomly assigned by the operating system. Hence, the SPA-Local components are able to find the SPA-Local subnet manager easier than the SPA-Local subnet manager can find the SPA-Local components.
Asynchronous Messaging

Because the SPA-Local network uses sockets for a communication medium, it is fully capable of sending and receiving messages asynchronously and requires no augmentation by the SPA-Local subnet manager.

3.4.3 SPA-1

A SPA-1 subnet consists of devices connected to an I^2^C bus attached to a processing node in the SPA network. As previously discussed, I^2^C components are addressed with a seven-bit address, which must be unique on the bus. A seven-bit address allows for 128 unique addresses. Because of the small address space, attaching multiple devices produced by different vendors does not provide a guarantee that all the I^2^C addresses are unique. Because there is no bus arbitration for slaves on I^2^C, if two components attempt to utilize the same address, their data will become mixed and corrupted as both attempt to transmit on the bus at the same time. Generally, avoiding duplicated slave addresses is accomplished by either modifying the firmware configuration of a device or by applying power or ground to a set of address select pins on the device. This type of manual bus configuration does not meet the SPA requirements for self-configuration. To solve this problem, SPA-1 defines an Address Resolution Protocol (ARP) which functions on standard I^2^C. This allows the SPA-1 components on an I^2^C bus to self-organize their physical address space.

Address Resolution Protocol

When transmitting data on an I^2^C bus, each byte is acknowledged with a 9th bit, known as the ACK bit. The SPA-1 Address Resolution Protocol (ARP) works by taking advantage of this ACK bit. Each SPA-1 component starts at the common address of 0x11 and tries to become the bus master and send a OneArp (Table B.6) message to that address. The discovery address contains the SPA-1 components UUID. Due to I^2^C master arbitration each component will drop out when they detect that the bits they are placing on the bus are being overridden by another device and switch to slave mode. Once in slave mode the device will acknowledge the discovery message with the I^2^C ACK bit. If an ACK bit is detected,
then the component knows there is another device on that address and therefore increments its own address. This continues until the component does not get an ACK bit, at which point it knows that no other SPA-1 component is on that address and it claims the address as its own. After several iterations, the SPA-1 components self-organize the I\textsuperscript{2}C address space with all components having unique addresses beginning at 0x11 and completely filling the address space above that. This efficient use of the address space ensures that a maximum number of components may exist on the network. This organized physical address space makes it very simple to for the discovery algorithm to find all components in the I\textsuperscript{2}C address space. Note that the SPA-1 subnet manager does not need to participate in the ARP process since it acts as a master on the bus and has no need of a slave address.

A sequence diagram of this algorithm performed by three SPA-1 components is shown in Figure 3.5 and pseudocode for the SPA-1 Address Resolution Protocol is given in Algorithm 3.1.

**Algorithm 3.1: SPA-1 address resolution protocol**

1. `currentAddress := 0x11;`
2. `foundAddress := false;`
3. `while !foundAddress do` 
4. `result := sendAsMaster(currentAddress, OneArp);`
5. `if result != lostArbitration then` 
6. `if result == NACK then` 
7. `foundAddress := true;`
8. `else` 
9. `currentAddress++;`
10. `end`
11. `else` 
12. `ACK message;`
13. `end`
14. `end`

**Asynchronous Messaging**

The SPA-1 discovery process is easiest understood by first discussing the asynchronous message transmission strategy. I\textsuperscript{2}C is a master-slave bus, where only the master can initiate communication. To give the appearance of asynchronous message transmission, the SPA-1
subnet manager takes advantage of the organized I²C address space and a simple round robin loop (Figure 3.6). Starting at the first address of 0x11, the SPA-1 subnet manager does a read followed by a write. The read action allows the SPA-1 component to write 256 bytes to the SPA-1 subnet manager. If the component has more data to write than that, it will be read during the next iteration of the round-robin loop. During the write action, the SPA-1 subnet manager writes any data addressed to the SPA-1 component to that component. The SPA-1 subnet manager continues to perform the read/write cycle while incrementing through the address space. Once the SPA-1 subnet manager completes the read/write cycle for the last known discovered SPA-1 component, it is ready to perform the discovery step.
Discovery

At the end of the round-robin communication loop, the SPA-1 subnet manager is ready to perform discovery again. This is accomplished by simply trying to send a OneHello message to the address one increment above the address of the last known discovered component. In the event that a new SPA-1 component was added to the I2C bus, it would start at address 0x11 and increment its way to the next available address, ending up with the address just after the last known SPA-1 component. If a new component is discovered, the discovery step is repeated in the round-robin loop. If a new component is not discovered, then the round-robin loop is restarted. This discovery step can be seen at the end of the round-robin loop in Figure 3.6. The SPA-1 discovery protocol itself is a simple request-reply protocol shown in Figure 3.7.

3.4.4 SPA-U

A SPA-U subnet consists of the set of components and hubs of a USB network attached to a processing node. USB uses a two wire serial bus network similar to I2C. Most modern operating systems including Windows, Linux, and VxWorks, already have USB self-discovery support built in. SPA-U components are required to implement the USB communications device class (CDC), which is commonly used for asynchronous bulk data transfer, and use a vendor ID of 0xfffe and product ID of 0x1110. This vendor and product
ID are used by the SPA-U driver installed on the operating system to recognize when a
SPA-U component is attached to the USB subnet. This common support makes discovery
on the SPA-U subnet very simple.

**Discovery**

USB has wide support in most operating system including Windows, Linux, and Vx-
Works. The operating system provides a mechanism for registering a program to handle
USB plug-in events. The SPA-U subnet manager registers for the plug-in events from de-
vices with the vendor ID of 0xfffe and product ID 0x1110. When this event is handled by
the SPA-U subnet manager, a callback function is invoked. When this function is invoked,
a file descriptor is part of the argument list. This file descriptor is a handle that can be used
to perform standard read and write operations to the SPA-U component. Verification is
still performed by sending a UsbHello (Table B.9) to the SPA-U component and expecting
a UsbAck (Table B.10) in return (Figure 3.8).

**Asynchronous Messaging**

The USB communications device class is used for SPA-U components because it is
commonly used to perform asynchronous bulk data transfer. The SPA-U subnet manager can perform read and writes to the SPA-U component asynchronously and the underlying USB support of the operating system handles the complexity the actual communication with the USB device.

### 3.4.5 SPA-S

A SPA-S subnet uses SpaceWire (SpW) as its underlying transport medium. A SPA-S subnet consists of SPA-S components interconnected through a series of routers, which are connected to at least one or more processing nodes. SpaceWire is used to interconnect processing nodes because it is fully routable and has suitable bandwidth. Each SpaceWire network is viewed as a single subnet. If multiple processing nodes are connected to a SpaceWire network, each node will run a SPA-S subnet manager. In this scenario, all the SPA-S subnet managers are viewed as a single distributed subnet manager, with one functioning as the primary subnet manager, and the rest functioning as secondary subnet managers. The only difference between the primary and secondary subnet managers is that the primary subnet manager is responsible for assigning logical addresses and the secondary subnet managers are not. Each SPA-S subnet manager must complete discovery on the subnet. This is due to the SpaceWire routing paths being unique from each perspective in
the network.

**Discovery**

A flooding algorithm is used to perform discovery on a SPA-S subnet. The basic idea behind the algorithm is to utilize the properties of the path-based routing scheme to discover routers. The SPA-S subnet manager accomplishes this by reflecting packets back to itself. An example of this process can be seen in Figure 3.9 where the SPA-S subnet manager sends a packet back to itself through the network by sending the packet with the path of \{2,1,1\}. The packet leaves the first SpW router out port 2, then leaves the second SpW router out port 1, thus returning to the first router, and then leaving the first router out port 1, returning the packet back to the SPA-S subnet manager. When the SPA-S subnet manager receives the returned packet it knows that it has discovered a SpW router with the forward path of \{2\}. Because the SPA-S subnet manager does not know which ports have actual connections on them, the discovery algorithm will try every possible path combination and only the packets with real connections make it back. After finding a router, communication is attempted with all possible ports on that router to discover connections to other routers. To discover SPA-S components on the SpaceWire network the SPA-S subnet manager sends SpwEndpointPing (Table B.13) messages to each possible port on the newly discovered router. When a SPA-S component receives the SpwEndpointPing, it will reply with a SpwEndPointPingReply (Table B.14). This simple request-reply protocol is the

![Diagram](image)

**Figure 3.9. Reflecting a packet back to oneself in a SpaceWire network**
discovery protocol for the SPA-S subnet. A sequence diagram for the discovery protocol is shown in Figure 3.10 and the pseudocode for the complete router and SPA-S component discovery is given in Algorithm 3.2.

The SPA-S subnet manager also plays a special role in discovering processors on the SPA network. When a SPA-S subnet manager is assigned an address it tries to discover a SPA-Local subnet manager on its processor by sending a LocalRoute (Table B.5) message with the AckRequired field set to true. If there is a SPA-Local subnet manager present, it will respond to the LocalRoute message with a LocalAck (Table B.3) message. If the SPA-S subnet manager receives the LocalAck, it requests a new address block from the Central Address Service for the newly discovered SPA-Local subnet manager. It is through this process that discovery passes onto other processes from the SPA-S subnet.

**Asynchronous Messaging**

SpaceWire networks are fully capable of sending and receiving messages asynchronously and require no augmentation by the SPA-S subnet manager.

### 3.5 The Network Layer

The network layer of the the SPA network model is responsible for logical addressing
Algorithm 3.2: SPA-S SpaceWire topology discovery algorithm

1 begin function init
2   for i from 1 to 31 do
3       send SpwRouterProbe to (i);
4   end
5 end

6 begin function handleReceivedMessages {SpwMessage message}
7   switch message.Opcode do
8     case SpwRouterProbe
9         knownForwardPath := message.ForwardPath;
10        knownReturnPath := message.ReturnPath;
11        for proposedForwardPath from 1 to 31 do
12            for proposedReturnPath from 1 to 31 do
13                send SpwRouterProbe to (knownForwardPath +
14                                       proposedForwardPath +
15                                       proposedReturnPath +
16                                       knownReturnPath);
17            end
18        end
19     for proposedForwardPath from 1 to 31 do
20                send SpwEndpointPing to (knownForwardPath +
21                                               proposedForwardPath);
22        end
23     endsw
24     case SpwEndpointPingReply
25         RoutingTable.add(UUID, message.ForwardPath, message.ReturnPath);
26         send SpaAssignAddress to (message.ReturnPath);
27     endsw
28   end
29 end

and message routing. It is the network layers responsibility to aggregate all of the different
SPA subnets into a single, unified, and transport agnostic network. The data link layer
abstracts the differences between the different transport mediums. This allows the network
layer to be transport agnostic. The SPA logical address is a transport agnostic address that
SPA components can use to address components anywhere in the SPA network regardless
of the SPA subnet in which they reside. However, message routing still requires physical
communication with the underlying SPA subnets. The translation between the transport
agnostic SPA logical address and the physical addresses required to route the message
is encapsulated into a routing table. The routing tables are stored in the SPA subnet
managers, making the SPA subnet managers the routing entities in the SPA network. This
section will explain the format of the SPA logical address, how it is generated and assigned,
and how it is used in conjunction with the routing table to route messages in a SPA network.

3.5.1 SPA Logical Address

The first step in combining each SPA subnet into a single SPA network is to define a
SPA logical address that can be used to address any component in the SPA network. The
definition of the SPA logical address has a reaching impact on the SPA network and many
different schemes were considered. To understand the SPA logical address and its design, it
is important to understand the underlying design decisions and goals that were considered.

1. The SPA logical address is transport agnostic. The SPA logical address does not
have any connection to a physical address. This is important because the SPA logical
address is carried across all SPA subnets and it does not make sense to carry a subnet’s
physical address into another subnet. An example of this would be trying to address
an envelope with an email address; it just does not make sense.

2. The SPA logical address is not large. If the SPA logical address is large, it creates
overhead on message size because the majority of SPA message traffic is less than 100
bytes. Initially the idea of using the component Universally Unique Identifier (UUID)
was considered, however, at a size of 16 bytes the SPA message header would have
been larger than 32 bytes when carrying the source and destination address. On a
message payload of 100 bytes this would have created an overhead of 33% just to
address the message. The size of the SPA logical address was chosen in consideration
of the impact to message overhead.

3. The SPA logical address does not change while traversing the network. In the old SPA
network approach, the address fields of a SPA message were changed and updated as
the message traveled through the network. This caused confusion when trying to
trace message traffic. It also forced extra knowledge of the SPA network topology to be spread around so the address could be interpreted correctly at each hop in the network. A consistent and immutable logical address greatly simplifies the amount of knowledge that each routing element must have in order to route the message correctly. It also makes tracing message traffic much simpler.

4. The SPA logical address requires no extra information to route a message. Again, in the old SPA network approach, the address sometimes required the routing element to inspect part of the internals of the message in order to determine where the message should be sent. The SPA logical address is the only thing that is needed to route a message through the network.

Given these considerations, the SPA logical address is a 4 byte field with two parts. The upper 2 bytes of the address are the SPA subnet ID, and the lower 2 bytes of the address are the component ID (Figure 3.11). This two field approach lends itself to representing SPA logical addresses in an ordered-pair notation (i.e. (3,2)). The two fields in the SPA logical address reflect the how the address is assigned.

3.5.2 Logical Address Assignment and the Central Address Service

SPA logical address assignment is coordinated by a Central Address Service (CAS)
and the SPA subnet managers. SPA subnet managers are discovered much like any other SPA component. When a SPA subnet manager is discovered, the SPA subnet manager that discovered it requests a block of addresses from the Central Address Service for the newly discovered SPA subnet manager. Once the newly discovered SPA subnet manager has received its address block, it can freely make SPA logical address assignments to SPA components in its subnet without having to coordinate those addresses with any other subnet manager.

SPA logical address is divided into two fields: the subnet ID and the component ID. The subnet ID is unique for each SPA subnet manager in the network. The Central Address Service was designed to coordinate subnet IDs. The role of the Central Address Service is to assign subnet IDs to SPA subnet managers. An address block comes in the form of a SPA logical address that has a 0 for the component ID (i.e. (1,0), (2,0), (3,0), etc.). This means that a SPA logical address with a 0 for a component ID belongs to a SPA subnet manager. The Central Address Service keeps a simple count and assigns address blocks by incrementing the count after each new address block is assigned.

The entire address assignment process is bootstrapped when the Central Address Service is discovered by a SPA-Local subnet manager. This happens like all other SPA-Local components with the Central Address Service sending a LocalHello (Table B.2) to the SPA-Local subnet manager. The ComponentType field of the LocalHello is set to distinguish the sender as the Central Address Service. When the SPA-Local subnet manager sees that it has discovered the Central Address Service, it sends a SpaRequestAddressBlock (Table B.26) message to the Central Address Service, which sends a SpaAssignAddressBlock (Table B.27) message in return. One important thing to note about the SPA-Local subnet is that all SPA subnet managers are SPA-Local components. This means that SPA subnet managers also send a LocalHello (Table B.2) to the SPA-Local subnet manager on the processor on which they reside. The ComponentType field of the LocalHello (Table B.2) message allows the SPA subnet manager to tell the SPA-Local subnet manager they are subnet managers. When the SPA-Local subnet manager finds a new SPA subnet manager it does not assign
an address from within in its address block, but instead requests a new address block on behalf of the newly discovered SPA subnet manager by sending a SpaRequestAddressBlock (Table B.26) to the Central Address Service. This is done because the new SPA subnet manager does not know how to speak to the Central Address Service yet. The SPA-Local subnet manager then forwards the SpaAssignAddressBlock (Table B.27) to the newly discovered SPA subnet manager. After the new SPA subnet manager has received its address block, the SPA-Local subnet manager sends it the logical address of all other known SPA core components using a SpaDistributeRoute (Table B.28) message. A sequence diagram for the bootstrap of the address assignment process is given in Figure 3.12.

Once a SPA subnet manager has received an address block it can freely make SPA logical address assignments from that block. SPA logical address assignment is accomplished
by sending a SpaAssignAddress (Table B.36) message to each SPA component. Each address assignment is sequentially assigned. The SPA subnet manager that has been assigned the address block (4,0) would assign the address (4,1) to the first discovered component in its subnet. The subsequent address assignments would be (4,2), (4,3), (4,4), …, etc. Figure 3.14 shows an example network with SPA logical address assignments and a routing path. Because of the two 2 byte subnet ID and the 2 byte component ID, a SPA network can contain a maximum of 65536 different subnets, each containing 65536 individual components.

3.5.3 Routing Tables and Routing

SPA subnet managers also act as routing entities in the SPA network. Each subnet manager is responsible for maintaining a routing table. The routing table contains the information necessary to translate a logical address into the appropriate physical address to move a message on to its final destination. A routing table contains two different sets of entries. The first set contains entries for all of the other SPA subnet managers in the SPA network. The second set contains entries for all of the components attached to the subnet manager’s subnet. When a message is received by a SPA subnet manager, it follows the routing algorithm shown in Algorithm 3.3.

Algorithm 3.3: SPA routing algorithm

1 if dest[subnetId][0] == myLogicalAddress then
2    send to component on my subnet;
3 else
4    send to subnet manager at dest[subnetId][0];
5 end

In order for the routing table to contain all the needed entries, when a SPA subnet manager finds another SPA subnet manager, it sends a SpaDistributeRoute (Table B.28) message to all other known SPA subnet managers telling them, “If you want to talk to this new subnet manager, talk to me.” This behavior generates an entry for every subnet manager in the routing table, creating a route from any subnet manager to every other subnet.
Each routing table entry contains four items: 1) a SPA logical address, 2) the component’s UUID, 3) the physical address type, and 4) the actual physical address. An example of a routing table is shown in Figure 3.13. The first four entries are the individual components on the manager’s subnet. It is apparent that this routing table belongs to a SPA-S subnet manager. It can also be seen that the SpaceWire network only has one router. This is evident by the fact that the SpaceWire routes stored in the routing table only have a single byte for the SpaceWire path. The remaining routes in the routing table are for the Central Address Service and the other SPA subnet managers. From the routes for the addresses (4,0) and (5,0) it is apparent that these subnet managers are not on the same processing node as this SPA-S subnet manager. In fact, if the reader looks closely, it can be seen that this is the routing table for the SM-S with the address (3,0) from Figure 3.14.

When routing messages between different SPA subnet types, each SPA subnet manager must have a way to communicate with each other. Because each SPA subnet manager is a software process, the SPA-Local subnet is used as the common language for all SPA subnet managers. Each SPA subnet manager speaks two protocols: SPA-Local and their own SPA subnet protocol. This allows a message originating from the SPA-1 subnet to get
to the SPA-S subnet by traversing the SPA-Local subnet. A SPA subnet manager could function on a different subnet than SPA-Local. An example of this would be if a SPA-S component hosted a SPA-U network. The SPA-U subnet manager would still have to be capable of speaking a protocol other than SPA-U, which in this case would be SPA-S instead of SPA-Local. While this is technologically possible, it has not been done in practice.

Figure 3.14 shows an example of a SPA network. The large blue squares represent processing nodes. The small colored squares represent SPA core components, including: the Central Address Service, two SPA-Local subnet managers, a SPA-1 subnet manager, a SPA-U subnet manager, and a primary and secondary SPA-S subnet manager. The colored triangles represent SPA components on their respective SPA subnets. There are five distinct subnets on this SPA network with each subnet using a different color: red, green, orange, purple, and blue.

As an example of how routing works in a SPA network, the SPA-1 component with address (2,2) is going to send a message to the SPA-U component with address (5,1). First, the SPA-1 component sets the message destination to (5,1) and the message source to (2,2). However, since it cannot initiate communication with the SPA-1 subnet manager, it has to wait for its turn in the round-robin polling loop. Once it has been contacted, it sends the message to the SPA-1 subnet manager. The SPA-1 subnet manager sees the destination subnet ID is 5 and looks up in its routing table the physical address for a message destined to the SPA subnet manager with the subnet ID of 5. It sends the message to the SPA-S subnet manager with the address (3,0). The SPA-S subnet manager looks up the next address in the path and sends the message through the SpaceWire network to the secondary SPA-S subnet manager with address (3,3). The secondary SPA-S subnet manager with address (3,3) looks up the next address and sends the message to the SPA-U subnet manager over the SPA-Local network. The SPA-U subnet manager sees that the message is destined for a component on its subnet and sends the message to its final destination, the component with address (5,1).
3.6 The Transport Layer

The purpose of the transport layer is to enable end-to-end connectivity and discovery between SPA components. In a SPA network, this involves resolving a component’s data dependencies with those items currently available in the network. This is accomplished through registering descriptive data sheets embedded within each component with a centralized repository and then issuing queries to that repository to locate the desired dependency.

3.6.1 xTEDS

Each component on a SPA network describes itself and its capabilities through an embedded document known as an eXtsible Transducer Electronic DataSheet (xTEDS). xTEDS extend the concepts of the IEEE 1451 TEDS [72] standard by not only including identification, calibration, and manufacturer information, but also by describing the component’s data inputs and outputs. All of this information is stored in XML format. Using XML allows the xTEDS to be human-readable, but also easily machine-parseable. Any xTEDS document can also be verified for correctness by validating it against the xTEDS XML schema definition [28]. The purpose of these xTEDS in a SPA network is to allow components to dynamically discover their data needs or dependencies. Figure 3.15 contains a portion of an xTEDS. Also associated with the xTEDS is an UUID. The xTEDS UUID,
or XUUID, is a type one Secure Hash Algorithm (SHA-1) hash of the xTEDS truncated to 128 bits. It is used to uniquely identify an xTEDS without having to send the xTEDS over the network.

3.6.2 Lookup Service

The Lookup Service is a critical component on a SPA network and acts as a repository and query engine for all xTEDS on the network. After a component has successfully been discovered and assigned a logical address, it must then register with the Lookup Service. This process involves having the SPA subnet manager that discovered the component inform the Lookup Service of the discovery by sending a SpaRequestLookupServiceProbe message (Table B.29) to the Lookup Service. Upon receiving the SpaRequestLookupServiceProbe, the Lookup Service sends a SpaProbeRequest message (Table B.33) to the SPA component, which responds with a SpaProbeReply message (Table B.34). From the SpaProbeReply, the Lookup Service inspects the component’s xTEDS UUID, or XUUID, to check if it has already cataloged a copy of the components xTEDS. The Lookup Service will then decide if that component’s xTEDS should be requested. This allows the Lookup Service to not request an xTEDS it already has in its repository, saving network bandwidth and processing time. An xTEDS might already be present if multiple components present the same xTEDS to the system or if the xTEDS was cached from a previous run. If the XUUID is unknown to the Lookup Service, then it sends a SpaX TEDSRequest message (Table B.20) to the SPA network to request the component’s xTEDS.
component and receives a copy of the xTEDS in a SpaXtedsReply message (Table B.21).
Figure 3.16 shows a sequence diagram for the registration protocol.

3.6.3 Query

Any component in the system may issue queries to the Lookup Service to find data or components to satisfy their dependencies by sending a SpaQueryRequest message (Table B.24). These queries are issued using the common terms defined in the SPA ontology and associating a desired value with those terms. These terms are represented as XML attributes in the xTEDS. If the values associated with the attributes are numeric, the query may use standard arithmetic operations to specify desired ranges for certain quantities. An example would be precision \( \geq 2 \) (Figure 3.17). This query would return any value in the system with a precision greater than or equal to two. When multiple specifications are present

\footnote{Queries are one area in the transport layer where the author did make changes. The author created a new query syntax in XML that allows the components to query any field in the xTEDS using matching, logic, and arithmetic operations. This is an update from using regular expressions.}
in a query, they combine conjunctively, so the result is the intersection of the results that would be returned if each specification were present individually. A very specific query will return fewer results, whereas a more generic query will return more results. The Lookup Service responds to queries by sending a SpaQueryReply message (Table B.25). A query reply contains the SPA logical address and other information for a SPA component whose xTEDS matches the query. If there are multiple matches, then multiple responses are sent. An empty SpaQueryReply message is sent last to signify the end of the responses. This protocol is shown in Figure 3.18.

It is possible that a SPA component could issue a query before the needed component has registered its xTEDS. In order to eliminate these race conditions among registrations, queries can also be made for future registrations. This allows the component that issued the query to be notified of any future additions to the network that may better meet its needs. Queries can also be made for cancellations where the SPA component issuing the query will be notified if a SPA component associated with an xTEDS that had previously matched a query is de-registered from the system. This happens if a component is turned off or damaged. A detailed description of the SpaQuery syntax can be found in [7].

Each SpaQueryReply (Table B.25) message contains the logical address of the component, the byte-level format of the provided message, and the section of xTEDS that defines the message. This information allows the issuer of the query to make an intelligent selection of which source to utilize. At this point, the issuer of the query has sufficient information to issue the selected commands to the component or request a subscription to the provided data.
3.6.4 Publish/Subscribe

Data transfer within a SPA network is done primary using the publish/subscribe paradigm. After issuing a query and selecting a data provider, components may issue subscription requests by sending a SpaSubscriptionRequest message (Table B.22) to the provider of that message. Along with requesting a subscription to a piece of data, the requester may also request a specific lease period for which the subscription will remain valid. Lease periods provide a degree of fault tolerance, as the subscriber can detect a failed data provider and select a different data source to subscribe to. The subscriber should renew this subscription before the lease period expires to ensure continued data flow. If a component produces the data at a rate higher than the subscriber desires, the subscriber may also specify a delivery rate divisor to indicate that it should only publish one of every $N$ messages to the subscriber. Data producers or publishers can reject any requested subscription to their data based on priority, resources, or data availability. The data producer alerts the subscriber to the acceptance or rejection of their subscription by sending a SpaSubscriptionReply message (Table B.23) to the subscriber.

Subscriptions can also be performed through the Lookup Service. In this scenario, the SpaSubscriptionRequest is sent to the Lookup Service. The Lookup Service then forwards
the request to the SPA component. The SpaSubscriptionReply is returned to the Lookup Service and then redirected to the original subscriber. The benefit of subscription via the Lookup Service is when the producing SPA component disconnects from the network (i.e. powered off or damaged) the Lookup Service will automatically notify all subscribers that the component is gone. The Lookup Service can only do this if it is aware of the subscription. If a SPA component subscribes via the Lookup Service, the data still flows directly from producer to consumer and the Lookup Service is not in the loop. The subscription protocol can be seen in Figure 3.19.

Figure 3.19. Subscription Protocol depicting direct subscription and subscription via the Lookup Service

3.6.5 xTEDS Message Protocols

The xTEDS schema definition [28] outlines the proper syntax for a SPA component’s
eXtensible Electronic Data Sheet. The schema defines three types of message formats: 1) Notification, 2) Command, and 3) Request. The xTEDS defines the contents of each type of message. In this section, each message type is explained and the sequence diagram for each type is given. A more detailed description of xTEDS and their associated protocols can be found in the SPA Logical and SPA Ontology specifications [7,9].

**Notification**

The xTEDS Notification message is used for data products that can accept subscriptions. It is subscribed to using the SpaSubscriptionRequest message (Table B.22). The SPA component that is being subscribed to responds with a SpaSubscriptionReply message (Table B.23). The reply will alert the subscriber if the subscription was accepted or rejected. The data producer can reject a subscription based on priority, resources, or data availability. If the subscription was accepted, then the data products are delivered in a SpaData message (Table B.30). The notification protocol can be seen in Figure 3.20.

**Command**

The xTEDS Command message is used to send commands to a SPA component. A xTEDS command is sent to a SPA component by sending a SpaCommand message (Table B.35). The command message has no reply. Sometimes the xTEDS Command message is used to just push data to a SPA component. The command protocol can be seen in Figure 3.21.

**Request**

The xTEDS Request message is used to send a command that requires a reply. The command is sent using the SpaServiceRequest message (Table B.31) and the reply is received as a SpaServiceReply message (Table B.32). The request protocol can be seen in Figure 3.22.

### 3.7 The Application Layer
Figure 3.20. xTEDS notification protocol

Figure 3.21. xTEDS command protocol
All the layers of the SPA model culminate in the application layer. The application layer supports SPA components interoperating in a plug-and-play manner. This section will describe the life cycle of a SPA component. The SPA components have been discovered, assigned a logical address, registered with the Lookup Service, queried for their data needs, made their subscriptions, and can now perform the function they were designed to accomplish. In the SPA model, a SPA component is an endpoint whose interface conforms to the SPA standards. SPA components can be software applications executing on a general processor resource, or a hardware device physically attached to the network. These may include applications for guidance, navigation and control, power management, payload management and operation, system health and status, etc. The communication and interoperation of the SPA components is independent of the nature of the components. Consumers of data and services are unaware of the physical type or physical network location of their producers.

A core concept of SPA is that components register their capabilities with the Lookup Service when they are added to the system. Once this information is captured, any component with a data need may query the Lookup Service for available sources and receive matches to that query. A SPA component may contact any or all matching components directly and subscribe to the data that it provides or utilize its data services. To make all of this possible, a SPA component goes through a five-phase life cycle. The phases in the
life cycle are:

1. Component discovery
2. Component registration
3. Data source query
4. Data subscription
5. Normal operations

### 3.7.1 Discovery

Component discovery is the process by which a component is found and assigned a logical address on the network. Discovery has previously been discussed as part of the Data Link layer in section 3.4 and SPA logical address assignment was previously discussed as part of the Network layer in section 3.5. Each subnet discovers the SPA components on its subnet using the process (probing, listening, etc.) appropriate for the subnet topology and transport medium. Upon assignment of a logical address, a component is ready to participate in a SPA network and transitions to the registration phase.

### 3.7.2 Registration

In the component registration phase, the component informs the Lookup Service of its available interfaces. These interfaces include commands available to configure and control the component, available data products for subscription, and services which include a request and response from the component. This interface description is called an eXtensible Electronic Data Sheet or xTEDS. It is provided to the Lookup Service and any other SPA component that requests it. Upon completion of registration with the Lookup Service, the SPA component and its xTEDS are now available to be found by other SPA components that query for compatible services. xTEDS and component registration were previously discussed as part of the Transport layer in section 3.6.
Figure 3.23. SPA component life cycle
3.7.3 Query

After registration, the component enters the query phase and searches for SPA components that can meet its service and data requirements. By following the query protocol outlined in Figure 3.18, a component issues a query to the Lookup Service and receives responses. The query does not search for components physical location. Instead a query searches by the type of service, the name of the component, the name of the interface, or for specific data. The query mechanism uses an XML schema very similar to that used for the xTEDS. The Lookup Service returns a response to the query, including a list of SPA logical addresses for the components that provide the requested data services. In addition, the query may remain in effect in the Lookup Service so that any matches to the query that become available after the initial query response will also be forwarded to the querying component. Some components may only be producers of data. Not all components are required to issue queries. For example, a controller of a set of temperature monitors may only publish data to subscribers and not require issuance of any queries to locate other services. Simple hardware devices typically fall into this category. Once query responses have been received, SPA components are ready to select and subscribe to services. Queries were previously discussed as part of the Transport layer in section 3.6.

3.7.4 Subscription

After receiving a list of query matches, a SPA component can select and subscribe to data and service providers. A SPA component can utilize whichever selection algorithm that is most appropriate. It is left up to the SPA component developer to decide how to select a best query response. The SPA component can request the use of the service of the providing component in an ad hoc manner, or it may use SPA protocols to establish a subscription to the service or data for a period of time. Similarly, a SPA component that provides data services may receive requests for those services, as well as subscriptions. The SPA component can decide whether the request or subscription will be honored and alert the requester. Subscription was previously discussed as part of the Transport layer in section 3.6.
3.7.5 Normal Operation

After establishment of subscriptions, the component is prepared to enter normal operations. Services that are required by a SPA component have been matched to providers and requests and subscriptions received from other SPA components can now be serviced. It is at this point that the system is considered stable and each SPA component performs the function it was created for. However, as a dynamic system, providers may become unavailable. The subscription protocols and Lookup Service provide for notification if a data provider drops out of the network. This can be caused by a SPA component being powered off or damaged. Similarly, a SPA component may be powered on late and become available. SPA components which have queried previously for services that a newly registered component provides will be notified and can respond if the new source is better than an existing provider or fulfills an unmet need.

3.8 Summary

The SPA networking approach described in this chapter provides a unified methodology for self-discovery and self-configuration of heterogeneous PnP networks. The network infrastructure is transport agnostic and does not place the burden of network routing on the SPA components. It provides a method for components to publish and subscribe to resources and interoperate regardless of their physical location on the system or the type of interconnection network they use. It allows spacecraft component providers to design and develop SPA-compliant components without any prior knowledge of how or where their component will be utilized in the system. It also provides a well-defined methodology for adding new and future network technologies without affecting existing SPA components. Support for a new or future interconnection network can be added by simply including the appropriate subnet manager software module at the data link layer. Likewise, removing support for an interconnection network that is not needed simply requires removing the appropriate subnet manager software module. The current SSM implementation supports SPA Local (SPA-L) and SpaceWire (SPA-S) subnets. Efforts are currently underway to incorporate the SPA-1 (I²C) and SPA-U (USB) subnets as well.
CHAPTER 4
IMPLEMENTATION

4.1 Introduction

An important outcome of this research is an actual working implementation of the SPA network architecture. A working implementation is an important product to demonstrate that the ideas presented in this research are actually possible. In a simulation it is too easy to produce an implementation that will not work on real hardware in a real system. An example of this can be found in timing. In a typical simulation every node in the network runs in a lock step manner, while in reality the network nodes all run independent and totally asynchronous. A real world implementation demonstrates that the proposed protocols and messages function in a real system. This chapter presents a successful implementation of the SPA Network as an operational system.

The SPA Services Manager (SSM) is an AFRL-funded implementation of the AIAA SPA standards including the SPA networking architecture presented in chapter 3. The SSM was developed at the Space Dynamics Laboratory (SDL) using an ISO 9001 certified spaceflight software development process. The SSM provides the core services required in order to support self-configuration on a heterogeneous network, including a mechanism for SPA hardware and software components to publish their data and capabilities with the system. SPA components capabilities are described in an XML document called an extensible Transducer Electronic Data Sheet (xTEDS). As part of the self-discovery and self-configuration process, a component provides its xTEDS to the SSM. The SSM parses and stores the information contained in the xTEDS. After this process is complete, any component can query the SSM for data or capabilities that it needs. The SSM sends a list of components that can provide the needed data and capabilities. The requesting component
can then subscribe to the required data or capability from the providing SPA component of its choice, regardless of where it is physically located or what type of interconnection network it uses.

Spaceflight software is held to a high standard when it comes to implementation. There are rarely second chances when your software is installed on a rocket and shot into space. Part of the purpose of this chapter is to demonstrate that the SSM is not a student project, but is a professional-grade, and flight-ready implementation. In no way should this dissertation be used as evidence that the SSM should not be flown on a spacecraft. This chapter presents the ISO 9001 certified development process used to implement the SSM, the high level SSM software architecture, some important implementation details, and the environment that is used to test the SSM.

4.2 Development Process

The SSM was developed at the Space Dynamics Laboratory (SDL) using an ISO 9001 certified flight software development process. The aerospace community has been locked into the the traditional waterfall approach to software development for the last couple of decades. A normal software project life cycle goes through the standard phases of planning, requirements, design, implementation, test, and finishes in a maintenance phase. There is a growing movement to incorporate newer agile methodologies into the traditional process [14]. SDL has done this by including common agile practices into the implementation phase of the traditional waterfall model. A dramatic departure from the traditional approach is not acceptable because the new approach still has to fit with the rest of the spacecraft development process which still follows the traditional approach. The agile practices that have been added to the implementation phase include feature driven development, judicious use of pair programming, code inspections and peer reviews, software unit testing, and a continuous integration approach to software verification and validation.

4.2.1 Planning

During the SSM planning phase, important trades in the software development process
were studied out, and goals were set. These goals and decisions are captured in the SSM Software Development Plan (SDP) [13]. The key topics from the SDP are issue tracking and reporting, software configuration management, and quality assurance.

Deficiency Tracking and Reporting

As with all software projects, defects are often found by users. It is important to provide a method for users to report the software defects to the development team. It is equally important after a report has been made that the user can track the status of the software defect. This process improves the overall quality of the software. Defect tracking is one of the hallmarks of a good software team [132]. If the defect list is not written down, defects are quickly (and sometimes conveniently) forgotten. If the defect list is not accessible by all developers and all users then its usefulness is limited. Defect tracking and reporting helps to improve the quality of the software and the satisfaction of the customer.

The SSM uses Redmine [22] to track software defects and report status of the defects. Redmine is an open source tool designed for this purpose. The Redmine instance for the SSM can be found at https://pnpsoftware.sdl.usu.edu/redmine. Redmine enables users to enter new bugs with the software and then see their status. One benefit of this process is the increased communication between developer and user. According to Eric S. Raymond’s version of Linus’ Law, “given enough eyeballs, all bugs are shallow” [124]. In spaceflight software there is no room for a software defect. Defect tracking and reporting is essential to eliminating software defects.

Software Configuration Management

Software configuration management (SCM) is the task of tracking and controlling changes in the source code. SCM enables the integration of several developer’s code into a single working code base. SCM is important because it ensures that no code is lost during development, there is always a path back to a stable state, and team members can work independently without fear of overwriting each other’s code. SCM is so common today, that there are entire companies built on just hosting code and performing SCM (i.e. github,
sourceforge). It is a sign of process immaturity if there is no SCM in place on a software project.

A software project’s SCM strategy is closely related to the SCM tool that is used for the project. Many SCM tools exist at the time of this writing (i.e. Subversion, git, CVS, Mercurial, Bazaar). The SSM uses git for software configuration management. Git uses a distributed model for source management. Each software developer has a local copy of the source repository. Any repository can be synchronized with any other repository. Typically there is one repository designated as the single point of synchronization. This creates a single point of integration for the entire code base while still maintaining the capability to synchronize across peers [46]. The SSM Software Configuration Management Plan [1] describes how git is used for the SSM project.

Quality Assurance

Software quality must be built in during development. Quality is not something that can be achieved after the code has been written. It is important to have a software quality assurance plan that gives the software developers good metrics and methods for measuring software quality, declares appropriate responsibilities, and outlines the use of proper tools. Good metrics and methods are hard to define, but should still be sought out. The SSM uses a standard metric of code coverage to provide a measure of how much effort has been put into quality assurance. Responsibilities are given to each developer so that each person knows their roles and tasks. Tools usage is outlined and standardized so that everyone collects the same metric the same way. All these items and more are defined in the SSM Software Quality Assurance Plan [14].

4.2.2 Requirements

The SPA Standards documents were produced by the Advanced Plug-and-play Technology (APT) program. The APT program consisted of six industry contractor teams divided into different committees. Each committee produced a standards document for their area of SPA. The standards that contain software specific requirements include Logical Inter-
face [7], Network [8], Local Subnet Adaptation [6], SpaceWire Subnet Adaptation [10], and Ontology [9]. Each of these documents includes a section with a list of requirements for a SPA implementation. The SSM draws its requirements from these documents. The requirements were captured in a Software Requirements Specification (SRS) [5] that follows the IEEE SRS recommended practices [70].

4.2.3 Design

The design of the SSM had the benefit of reimplementing the older Satellite Data Model (SDM). The developers that had maintained the SDM were the same developers that worked on the SSM. Many of the good parts of the SDM design were reused, while the worst parts were redesigned. After an initial design, the SSM was rapidly prototyped to further refine the design. The design was captured in a Software Design Description (SDD) [12] that follows the IEEE SDP recommended practices [73]. A high-level overview of the software design is presented in section 4.3.

4.2.4 Implementation

Agile practices were used during the SSM’s implementation phase. These practices include feature driven development, judicious use of pair programming, software unit testing, code inspections and peer reviews, and a continuous integration approach to software verification and validation.

Feature Driven Development

Software projects are broken down into a collection of requirements and features that meet these requirements. The features are then tracked through design, implementation, peer review, and test phases. Each feature is assigned to a software developer. The developer is responsible for reporting status on feature they have been assigned. All requirements are met when all features have been completed. The overall status of the project can then be thought of as sum of the status of each of the features. Features are grouped together to create a software release. Several releases are planned out in advance and the customer is
During the implementation of the SSM, a web-based project management tool called Redmine was used to plan releases. Redmine is also used as an issue tracker. In the SSM project there are three issue types: Bug, Feature, and Improvement. The issues are grouped together into versions and planned out on a roadmap. An example roadmap for the SSM 0.9.5 release can be seen in Figure 4.1. Again, several releases are planned out in advance and the customer has full access to the Redmine site. This allows the customer to use Redmine to participate in the planning process.

**Judicious Use of Pair Programming**

Pair programming has been a controversial idea since it was introduced. During the SSM implementation sections of the code were developed using pair programming. The majority of code can easily be written by a single developer with little fear of grievous mistakes. This code is quickly validated through the use of code review. However, there are certain sections of the code that are especially complex and benefit greatly from having two software developers write it at the same time. One example of this is the xTEDS indexing system in the Lookup Service. The author and Bryan Hansen sat down together and wrote the indexer. It is a very complicated data structure with an array of hash tables.
that contain polymorphic nodes which contain more hash tables or linked lists containing other linked lists or another layer of hash tables. Although development spanned several days, the code worked correctly on the first test run. This can be attributed to the many bugs which were caught during pair programming. Just because it is good for some code, does not mean pair programming is good for all code. Pair programming was judiciously used during the development of the SSM.

**Software Unit Testing**

Unit testing is the practice of exercising an individual unit of code to ensure it functions correctly. A software unit can be defined at almost any level. In the SSM the basic unit is a class, however, entire applications are also tested in integration-level unit tests. The software developer writes unit tests before, during, or after a unit of code is written. The practice of writing tests before and during code development is known as Test Driven Development [25]. The developed code and its associated unit tests are committed to the source code repository at the same time.

Code coverage is a common software development metric for how thoroughly the code has been tested. Code coverage is a function of the number of lines executed during the test execution over the total number of lines in the code base. The SSM has a standing goal of 80% code coverage. As of this moment, the code coverage is at 93%. High code coverage is important because it helps detect dependency failures in the code base. If a software developer changes the functionality of a piece of code and another part of the system depends on that functionality, the unit tests will detect a failure in the unchanged code. This gives the developers high confidence that the entire system is functioning as intended.

The SSM uses the Google C++ unit testing framework [20]. Unit tests are written at the method, class, and system level. The collection of unit tests are executed in the Linux, Windows, and VxWorks environments.
Code Inspection and Peer Reviews

When a software feature and its associated tests are completed, they undergo a peer review process. During a peer review one or more software developers who did not produce the software feature review the implemented code. Emphasis is placed on finding performance and functional defects, as well as more minor issues such as code style and sufficient commenting. No code is committed to the software configuration management repository until it has been reviewed by at least one other software developer. This process increases the stability of the overall code base. In general is causes all developers to be held to a higher standard when they are developing code because others will actually see their code. It also has a bonus side effect of sharing the collective knowledge of the project architecture and implementation details with all who are involved in the code review. It has been my experience on this project that code reviews have the greatest impact on the quality of code that is produced. I personally feel that the quality of code I produce has improved greatly because of the peer code reviews.

To facilitate a painless and efficient peer code review process, the SSM development team uses a web based tool called Code Collaborator which was developed by SmartBear [17]. Traditionally peer code review has been done by getting a group of developers in a meeting while the code author scrolls through the code and tries to explain what has been developed and how existing code has been changed. Code Collaborator enables software developers to participate in peer code review without having to be in the same physical location. The review process becomes asynchronous and distributed. When a developer posts code for review they can select reviewers from their team. Those team members receive an email alerting them to the new review. A reviewer logs in to Code Collaborator and can view the code online. Code Collaborator shows diffs so the reviewer only has to review code that has been changed. The reviewer can make comments and even mark defects in the code. Once the reviewer has finished an email is sent to the code author who can then address the comments and defects. The SSM development team has found that this distributed and asynchronous approach to peer code review has improved the efficiency
Continuous Integration

Unit testing is of great value to software developers while developing a large system. However, the utility of the unit tests is only found in the execution of said tests. Continuous integration is the practice of applying quality checks throughout the software development life cycle. This differs from the traditional practice of applying quality control after completing development. Continuous integration is a common practice in agile software development.

A continuous integration server is used to build the software each time code is committed to the software repository and on a nightly basis. The continuous integration server then executes the body of unit tests and collects other code health statistics. In the event an unhealthy code base (broken or unstable) is detected, emails are sent to the entire software development team. The cause of the unhealthy code base is immediately investigated and corrected.

The SSM uses Jenkins as its continuous integration server. Jenkins is an extensible open source continuous integration server. There are many plugins available for Jenkins. When Jenkins runs the SSM project it performs the unit tests and runs Valgrind to collect information about the memory usage, including memory leaks, read or write errors, and invalid memory accesses. Jenkins runs a static analysis tool called cppcheck which looks for common bugs that compilers do not detect such as out of bounds checking, exception safety, memory leaks, obsolete functions, unused code, and uninitialized variables. Jenkins also compiles the source code documentation using Doxygen. During the Doxygen documentation build, Jenkins validates that there are no errors or warnings in the Doxygen comments.

In the event that any one of the checks that Jenkins performs reports a warning, the build is reported as unstable. If one of the checks fails, the build is reported as broken. When a build is unstable or failed, the responsible software developer can examine the Jenkins build log output and correct the code. Due to the continuous nature of Jenkins, most of
the quality control checks have been automated. This enables the software development team to focus on code production and stability instead of spending a lot of time trying to integrate and debug new code.

### 4.2.5 System Test

The test plan for the SSM is outlined in a Software Test Plan [15]. SSM system testing was performed at the Space Dynamics Laboratory using a Windows PC, a Linux PC, and SDL’s Modular Avionics System (MODAS) [21]. SpaceWire endpoints consisted of a collection of 8051 base SPA-S sensor simulators. A example test network configuration can be seen in Figure 4.2 During a system test, the network is configured with the Central Address Service, Lookup Service, a data producer, and a data consumer application. The test is executed multiple times with different configurations of where each component resides in the network. The test is successful when the data flows from sensor simulators and data producer to the data consumer. The focus of these tests are to validate that discovery, registration, query, subscription, and data delivery all function correctly regardless of the network configuration. These tests are not intended to evaluate network characterization, but instead their focus is on functionality.
4.2.6 Maintenance

SSM releases are tagged with their appropriate release version and Jenkins build number, and then posted to the Redmine site [24]. Users can post bug reports and see status of reported bugs, as well as planned release dates for the version that contain the bug fixes. Each bug report goes through an inspection process to reproduce the bug. Once the bug has been reproduced, a unit test is written that captures the reproduction. Initially this unit test is failing. When the bug is fixed, the unit test passes. This test is then added to the body of unit tests and protects the code base from introduction of the same bug. After the bug is corrected, the unit test and the fixed code goes through the peer review and continuous integration processes before the bug is marked as resolved.

Other code improvements are also part of the maintenance process including refactoring existing code to use new techniques and algorithms, or updating functionality of already completed features to be easier to use or access. These types of improvements also go through a code review, unit test, and continuous integration process. Code improvements are also recorded in software releases and can be seen in the Roadmap (Figure 4.1) on the SSM Redmine site [24].

4.3 Software Architecture

Each core component in the SSM was implemented in an object-oriented, modular fashion. A layered software architecture was used to abstract and encapsulate the code into functional modular units in order to decrease coupling and increase internal class cohesion. The layers of the software architecture include a platform abstraction layer, a collection of common utilities, several core components, and a SPA Application Programming Interface (API). A layered software architecture diagram can be seen in Figure 4.3.

4.3.1 Platform Abstraction Layer

The Platform Abstraction Layer (PAL) abstracts operating system and architecture differences from all higher-level code. Because of the PAL design, all code modifications needed to port the SSM codebase to a specific operating system and architecture will only
Figure 4.3. Layered software architecture of the SSM

take place inside the PAL. When the SSM was first ported from Linux to VxWorks, it took a total of 20 minutes and no files were modified outside the PAL. When porting from Linux to Windows, a single software engineer made the port in a couple of days as parts of the PAL had to be adapted for a non-Posix compliant operating system. However, no files were modified outside of the PAL.

The PAL is a set of classes that encapsulate common operating system and architecture specific functionality and implementations. The public interface to these classes will be common across all architectures, while the underlying implementation can vary widely from one platform to the next. An example of an object in the PAL is the SpaThread class, which handles threading abstraction. SpaThead exposes methods for the common threading operations of start, stop, join, and detach. The SpaThread class internally has two implementations, one of which is selected via compile time #ifdef statements. For Linux and VxWorks there is an implementation using the pthread library and the other implementation uses the standard Windows threading API. A class outline for SpaThread can be seen in Figure 4.4.
4.3.2 Common Utilities

The collection of common utility classes are reused by the SPA core components and the SPA API. The idea is to store reoccurring functional needs into classes that are easy to use. This enables rapid development of SPA capable software. This section presents some of the common utilities that are most used and encapsulate vital SPA functionality.

**Generic Node, List, Queue, Vector, and Priority Queue**

Normally a C++ code base will use the Standard Template Library (STL) for common data structures such as list, queue, vector, and a priority queue. However, the STL is avoided in spaceflight software because of the lack of heritage. It is seen as a risk because of its dynamic memory allocation. Because these basic data structures are common to all software and the inability to use the the existing STL implementations, the SSM provides an implementation of each of these data structures.

The GenericNode is used as the common data object in all of the generic data structures. It is used by subclassing and then adding the private data members that need to be stored. The subclass is also required to implement the assignment, equality, and less than
operators. The GenericNode provides a getNext function that is used in the linked lists that backs the GenericList and GenericQueue data structures.

The GenericList encapsulates a linked list of GenericNodes. Common list operations like getting an iterator, inserting, removing, sorting, etc are all exposed through the GenericList class. A list class derives from GenericList to inherit this functionality. Using inheritance limits the linked list logic to one central location to increase maintainability.

The GenericQueue also encapsulates a linked list of GenericNode derived objects. The common enqueue and dequeue operations are defined in the base class. GenericQueue and its derived classes are inherently thread-safe as all operations are explicitly locked.

The GenericVector uses an underlying array for its implementation. GenericVector will automatically resize itself when the array starts to reach capacity. The default allocation strategy doubles the size of the array each time it is resized. This is the same as the STL’s vector class and is a typical allocation strategy for a vector.

The GenericPriorityQueue uses a GenericVector as a data store and implements a min heap. This was found to be the fastest implementation and is close to the speed of the STL priority queue class. Class diagrams for the entire Generic family can be seen in Figure 4.5.

Message Classes

The SSM middleware uses a message passing paradigm, where components communicate by sending distinct messages to each other. Each message includes a standard header, a payload, and a standard footer (Figure 4.6). Each message is encapsulated in its own class. A base class called SpaMessage implements the standard header and footer. A new message can then be created by subclassing SpaMessage and implementing the payload portion of the message. A portion of the SpaMessage class hierarchy can be seen in Figure 4.7. Each message object implements the marshal and unmarshal functions. The marshal function will take the object and turn it into a byte array, ready for transmission on the network. The unmarshal function takes a byte array and populates all of the private data members of the object.
Figure 4.5. The family of Generics: Node, List, Queue, Vector, and Priority Queue

<table>
<thead>
<tr>
<th>Standard Header</th>
<th>Payload</th>
<th>Standard Footer</th>
</tr>
</thead>
</table>

Figure 4.6. Generic message format for a SPA message
Threading and Inter-thread Communication

Threading is common in the SSM. It is used to monitor I/O channels and logging. All communication channels have a dedicated blocking thread for sending and a thread for receiving messages. This approach ensures that messages are not dropped due to full I/O buffers. Thread safe message queues are used for all inter-thread communication. The main application thread pulls incoming messages from the listener thread’s message queue and appends outgoing message to the sender thread’s message queue. When logging, the main thread simply appends the log message to the logger thread’s message queue. This approach allows the main thread of an application to process incoming messages and send new messages out without interruption. Each thread is either blocking on an I/O interface or on a thread safe queue. This keeps processor utilization low even though there are an average of 4 to 6 threads in each application.

Communicators

The communicator classes were designed to encapsulate all communication on the SPA network. There are two types of communicator classes: 1) PhysicalCommunicators (Figure 4.8), and 2) SpaCommunicators (Figure 4.9). PhysicalCommunicators encapsulate the sending and receiving of byte arrays on a SPA subnet. There is one PhysicalCommunicator for each SPA subnet type. The SpaCommunicators encapsulate the logical sending, receiving, and routing of SpaMessages. A SpaCommunicator contains at least one PhysicalCommunicator. Each derived SpaCommunicator serves a special purpose. The SpaApplicationCommunicator is designed to facilitate communication from an applications...
perspective. The SpaNetworkCommunicator encapsulates two SpaPhysicalCommunicators to allow for subnet-to-subnet routing to occur at a lower level and freeing the SSM core components from this processing. Last, the SpaApiCommunicator is specially designed to facilitate communication on the SPA network from within the API.

**Routing Table**

The RoutingTable class is used by all SSM Core Components. It provides translation between a SPA logical address and the physical address needed to transmit the message to the next component along the path to the final destination. For efficiency purposes, the RoutingTable is implemented as a hash table. The hash function is specifically crafted to minimize collisions in the hash table and thereby keep lookups to constant or near-constant time. This optimization is done through a knowledge of the size of address blocks distributed by the Central Address Service. The routing table hash function will not have a
collision until there are more than 16 components in a single subnet, or there are more than TABLE_SIZE subnets on the SPA network. The RoutingTable hash function is as follows:

```c
UInt16 hash (UInt32 addr)
{
    UInt16 result = 0;
    result = (((addr >> 16) * OFFSET);
    result += (addr & 0x0000FFFF) & (TABLE_SIZE − 1);
    return result;
}
```

Memory Pool

Allocating dynamic memory throughout run-time on a space system is a risk because any memory leaks can lead to the eventual failure of the software system and could cause an entire space craft failure. The SSM uses memory pools to control dynamic allocations. A memory pool makes an initial memory allocation as an application starts and does not allocate any more memory during the lifetime of the application. The application’s memory allocations are then directed to the memory pool instead of the operating system. Memory allocations for the memory pool come in a block size, which is configured for each memory pool. This discourages segmentation within the pool. These memory pool allocations are faster than standard operating system allocations because the program does not have to make a system call and switch into kernel mode to allocate the memory. This increased performance is an added bonus to the benefit of never being able to allocate more memory than the memory pool has already allocated. Memory pools are a standard design artifact in spaceflight software.

The SSM implementation includes a MemoryPool class which encapsulates a single memory pool. It is responsible for allocating, deallocating, and managing a single memory block size. The allocation and deallocation functions are not called directly by the user, but rather the global new/delete C++ operations are overloaded to use a memory pool.
Users will replace calls to `new` with `new(MemoryPool* pPool)` and calls to `new[]` with `new[](MemoryPool* pPool)`. An example to allocate a 1024 buffer from a memory pool:

```c++
UInt8* pBuf = new(pMyPool) [1024];
```

Deleting memory allocated from a MemoryPool remains syntactically unchanged from standard C++.

Applications often allocate memory of varying sizes throughout their execution. In a single memory pool with a small sized block, this can cause segmentation and under-utilization. The MemoryPoolController class controls multiple MemoryPools, each having a different size and block size. The MemoryPoolController can automatically determine which MemoryPool will best fit the requested allocation. As with the MemoryPool, to allocate memory from a MemoryPoolController the global `new/delete` operations have been overloaded. Users replace calls to `new` with `new(MemoryPoolController* pPoolController)` and calls to `new[]` with `new[](MemoryPoolController* pPoolController)`. An example to allocate a 1024-byte buffer from a memory pool controller:

```c++
UInt8* pBuf = new(pMyPoolController) [1024];
```

Deleting memory allocated from a MemoryPoolController remains syntactically unchanged from standard C++. Appendix D contains a further discussion on SSM memory usage that is outside the scope of this dissertation, but still remains relevant as reference material.

**Logger**

The SSM implementation includes a Logger class utility. The Logger is used to write formatted, time-stamped log message to stdout and/or to file. It is implemented as a singleton per application. The Logger supports seven levels of logging: FATAL, ERROR, WARN, INFO, DEBUG, TRACE, USER. During an application’s execution, command line parameters can be used to enable any or all levels and the Logger output destination. Global macros are defined in the Logger class which enable easy use of the Logger. These macros are: LOG_FATAL, LOG_ERROR, LOG_WARN, LOG_INFO, LOG_DEBUG,
LOG_TRACE, and LOG_USER. These macros use variadic variable statements so that processing is not done to build the log message strings if that log level is not enabled. The Logger also uses its own thread. Each logging macro simply constructs the log message, and then enqueues the message onto a thread safe queue. The Logger thread is blocking on the queue and writes the log messages to the enabled destinations. Between the variadic macros and the separate thread for I/O, the Logger has been designed to cause a minimal impact on the performance of any application that uses the Logger.

4.3.3 Core Components

The SSM core components are those that are required for the system to function correctly. They include the Central Address Service, SPA-Local Subnet Manager, the SPA-SpaceWire Subnet Manager, and the Lookup Service. Other SPA subnet managers would go in this list if they are implemented and used in the SPA network. The following sections outline a brief design and provide a high level UML diagram for each SPA core component.

Central Address Service

The Central Address Service is responsible for making address block assignments for SPA subnet managers. The Central Address Service is responsible for ensuring there are no duplicate address blocks assigned. The Central Address Service does this by using a SpaLocalComminicicator for communicating on the SPA-Local subnet and a routing table to store address assignment information. The Central Address Service is the simplest core component in the system.

SPA-Local Subnet Manager

The SPA-Local Manager (SM-L) is the software process which manages SPA-Local subnets. It handles discovery, subnet component address assignments, and registration for SPA-Local components. It also acts as the routing entity for SPA-Local components and routes messages within its subnet. The SPA-Local Manager also performs component monitoring. The SPA-Local Manager uses a RoutingTable to track address assignments as well
as routes to other SPA subnet managers. A SpaNetworkCommunicator is used to automatically handle the routing of SPA messages. This SpaNetworkCommunicator only uses a single SpaLocalCommunicator because the SPA-Local Manager does not communicate with any other SPA subnet. A list implementation called the ComponentList is used to store information for each component on the SPA-Local subnet over which this SPA-Local Manager is in charge. This list includes information about the state of health of each component, such as the last time a heartbeat was received from the component. The final class used by the SPA-Local Manager is the DiscoveryQueue. The DiscoveryQueue is common among SPA subnet mangers. It is used when a new SPA component has been discovered, but the subnet manager has not yet received an address block. Each newly discovered SPA component is enqueued into the DiscoveryQueue. When the SPA subnet manager receives its address block, then the DiscoveryQueue is processed and address assignments are made for each discovered SPA component.

**SPA-SpaceWire Subnet Manager**

The SPA-SpaceWire Manager (SM-S) is the software process which manages SPA-SpaceWire subnets. It handles discovery, subnet address assignment, and registration for SPA-SpaceWire components. It also acts as the routing entity for SPA messages between a SPA-Local subnet and a SPA-SpaceWire subnet. The SPA-SpaceWire Manager uses a RoutingTable to track address assignments as well as routes to other SPA subnet man-
Figure 4.11. SPA-Local subnet manager class architecture

A SpaNetworkCommunicator is used to automatically handle the routing of SPA messages. This SpaNetworkCommunicator uses a SpaLocalCommunicator and a SpaSpaceWireCommunicator because the SPA-SpaceWire Manager communicates with both the SPA-SpaceWire subnet and the SPA-Local subnet. A list implementation called the ComponentList is used to store information for each SPA-SpaceWire component on the subnet. A second ComponentList is used to store information for any other SPA-SpaceWire Manager that are attached to the same SPA-SpaceWire subnet. This list includes information about the state of health of each component, such as the last time a heartbeat was received from the component. The final class used by the SPA-SpaceWire Manager is the DiscoveryQueue. The DiscoveryQueue is common among SPA subnet managers. It is used when a new SPA component has been discovered, but the subnet manager has not yet received an address block. Each newly discovered SPA component is enqueued into the DiscoveryQueue. When the SPA subnet manager receives its address block, then the DiscoveryQueue is processed and address assignments are made for each discovered SPA component.
Figure 4.12. SPA-SpaceWire subnet manager class architecture

**Lookup Service**

The Lookup Service is the software process that collects and indexes the xTEDS of every SPA component on the SPA network. There is no requirement for where the Lookup Service runs on the SPA network, however only one Lookup Service is allowed to run on a SPA network. The Lookup Service is the most complex application in SSM due to its xTEDS indexing and query functionality. Queries are executed on xTEDS once they have been collected and indexed by the Lookup Service. Queries can be stored and run on each new xTEDS or when an xTEDS is canceled. If a new xTEDS is registered and it matches a stored query, the component that issued the query will receive a new query result. When an xTEDS is canceled all stored cancellation queries are run against the canceling xTEDS and query results are sent to each component. Query results for a cancellation query are marked as such so the SPA components know that the xTEDS has been removed from the Lookup Service.

The software architecture of the Lookup Service can be divided into three logical groups: 1) xTEDS, 2) Communication, and 3) miscellaneous. The xTEDS group is the collection of objects that make up the xTEDS parsing and indexing, the Communication group is
the SpaApplicationCommunicator and a SpaLocalCommunicator, and the miscellaneous group is a collection of data structures used to store registered component information, subscriptions, and queries.

xTEDS  The xTEDS handling in the Lookup Service consists of two main objects, 1) the XtedsParser, and 2) the XtedsRepository. The XtedsParser takes an XML version of an xTEDS and deserializes it into an object. The Xteds object is then passed to the XtedsRepository, which indexes every element and attribute and their objects into a large hash table. This hash table is then used during queries to find xTEDS that match the query.

Communication  The Lookup Service uses the SpaApplicationCommunicator for sending and receiving SPA messages on the SPA network. The SpaApplicationCommunicator handles the discovery and registration process automatically, so that no extra code has to be added to the Lookup Service to accomplish these tasks. The SpaApplicationCommunicator also contains a RoutingTable used to send messages along optimal paths. The SpaApplicationCommunicator internally uses a SpaLocalCommunicator to do the actual socket-based communication.

Miscellaneous  The miscellaneous group consists of the RegisteredComponentList, SubscriptionTable, and QueryList. The RegisteredComponentList is used to store the current state of each component that has registered its xTEDS. When a component has deregistered its xTEDS, the xTEDS is not removed from the XtedsRepository. Instead the component’s information is updated in the RegisteredComponentList to show that the xTEDS is now inactive. All query results are checked again the RegisteredComponentList before being finalized.

The SubscriptionTable stores the subscription information that the Lookup Service is currently managing. In the SpaSubscriptionRequest (Table B.22) message there is a field marked for subscription manager. This gives the Lookup Service knowledge of subscription
relationships in the network. In the event that a component deregisters its xTEDS, the Lookup Service checks the SubscriptionTable for any components that were currently subscribed to the deregistering component and sends them an alert that the component is no longer available.

The QueryList is responsible for storing persistent queries registered with the Lookup Service. This list is used when xTEDS are registered or deregistered. The QueryList contains FUTURE and CANCELLATION queries. When new xTEDS are registered, the FUTURE queries are executed on the newly registered xTEDS. When an xTEDS is deregistered, all CANCELLATION queries are executed against it. Appropriate SpaQueryReply (Table [B.25]) messages are sent out of the owner to the stored persistent query.

4.3.4 SPA Application Programming Interface

The SSM Application Programming Interface (API) is designed to be a high-level framework to simplify the development of SPA software components. The simplification is done through abstracting all the SPA messaging and protocols within the API and allowing the programmer to write code to only interface at the xTEDS level. It is important to note that while it is called the SSM API, it actually functions more like a framework. The
key difference being that when using an API your application maintains control of the application’s life cycle and execution flow and only loses control when invoking functions in the API. In a framework, it is the framework that maintains control of the application’s life cycle and execution flow and it only surrenders control to your function at distinct hook-in points. The SSM API works by deriving a class from the SpaApplication class and implementing several functions. These functions are invoked by the API when it is ready to execute them. The API also provides the user with a simple set of functions to use for setting up their application to issue queries, send commands, produce data periodically, etc. Again, program execution control is maintained by the API and only passed back to the user in callbacks. This control flow is shown in Figure 4.14. The public interface of the SpaApplication class is shown in Figure 4.15.

The SSM API architecture is similar to most of the other SSM applications. It uses a SpaApiCommunicator, SubscriptionManager, XtedsMsgTable, and TimerTable. The SpaApiCommunicator functions like the other PhysicalCommunicators by handling communication with the SPA-Local subnet. The API itself handles discovery, logical address assignment, and xTEDS registration. The XtedsMsgTable provides a mapping from xTEDS
message to callbacks that are invoked when the SPA message containing the xTEDS messages arrive. The TimerTable stores the individual timers that the derived user application uses to perform its own local processing. An overview of the SSM API class architecture is shown in Figure 4.16.

4.4 Software Implementation

The SSM is an actual implementation and not just a software design. This section gives a few details about the software implementation that are not part of the software architecture.

4.4.1 Programming Language

The SSM is implemented using object oriented ANSI standard C++. It was written using the 2003 version of the C++ language.
4.4.2 Open Source Libraries

The SSM uses two open source libraries in its implementation. The first is RapidXml \[78\]. RapidXml is used in the XtedsParser class to perform the syntactic XML parsing. RapidXml does not perform the semantic parsing the xTEDS. RapidXml is a fast and portable in-situ parser written in ANSI standard C++. RapidXml’s parsing speed is approximately the same as that of the strlen() function executed on the same data. The entire library is contained in a single header file. RapidXml also uses memory pools for its memory allocation.

The second open source library used in the SSM implementation is the Google C++ Testing Framework (Gtest) \[20\]. Gtest is used in the SSM unit tests. Roughly 45% of the SSM codebase is unit tests. Gtest is based on the xUnit architecture \[29\] and supports automatic test discovery, a rich set of assertions, user-defined assertions, death tests, fatal and non-fatal failures, value- and type-parametrized tests, various options for running the tests, and XML test report generation.
4.4.3 Operating System Support

Using the Platform Abstraction Layer (Section 4.3.1), the SSM natively supports three different platforms 1) Linux, 2) Windows, and 3) VxWorks 6.7.

4.4.4 Development Environment

On Linux, SSM uses a Makefile to manage the build process. Software developers have their choice of text editor. The SSM was primarily developed on Linux using both the Eclipse C++ Development Toolkit (CDT) and QT Creator as text editors. For Windows, SSM comes with Visual Studio 2008 projects. VxWorks development is done using WindRiver Workbench 3.1.

4.5 Summary

The SSM has been developed using an ISO-9001 approved software development process and is ready for flight. The SSM currently consists of 80,831 lines of code: 43,465 (54%) are executable code and 37,366 (46%) are test code. Currently the Operationally Responsive Space (ORS) office is using the SSM for its Modular Space Vehicle (MSV) bus. MSV is being designed and built by Northrop Grumman Aerospace Systems. MSV is currently scheduled for launch on the ORS-2 mission in May of 2013.
CHAPTER 5
EXPERIMENTAL SETUP

5.1 Introduction

This chapter outlines the experimental setup used to characterize, measure performance, and evaluate spaceflight suitability of the SPA network as described in Chapters 3 and 4. In order to facilitate network characterization and capturing performance data, a simulation environment was set up at the Space Dynamics Laboratory (SDL). The simulation environment consists of a collection of endpoints and a Windows 7 PC connected to a SpaceWire router. The Windows 7 machine runs a graphical SPA application that is used to configure the other network endpoints. A screen shot of the graphical test application can be see in Figure 5.1. Examples of this configuration can be seen in Figures 5.2, 5.3, and 5.4. The Windows based test graphical SPA application enables the dynamic configuration of the test endpoints. The endpoint simulators can be configured to produce data at a specific size and rate. A test agent also running the PC is configured to collect a set number of messages before ending the test. Once the test has ended the test agent reports the resulting data to the test application, which then does post processing on the data and displays graphs. This testing architecture enables the test agent application run anywhere in the network, such as on a flight processor.

5.2 Performance Measures

In characterizing the SPA network the metrics that are being collected are latency, jitter, and packet loss percentage.

Latency is the time that it takes for a message to traverse the network from the source to destination. Latency is measured by synchronizing time on both the endpoints and the
Figure 5.1. Graphical network configuration interface

processor, then placing a time stamp in the data just before it is transmitted on the network and then taking a time stamp just after it is received at the destination.

\[ \text{latency} = \text{received time} - \text{sent time} \]

**Jitter** is the variability over time of the packet latency across the network. Jitter is found by performing a statistical analysis of packet latencies and finding the standard deviation.

\[ \sigma = \sqrt{\frac{\sum (x - \bar{x})^2}{N}} \]

**Packet loss percentage** is the number of packets lost from of the total number sent. Packet loss percentage is a simple computation of one subtract the number of packets received over the total packets sent.
Figure 5.2. Single router network configuration

\[
\text{packet loss} = 1 - \frac{\text{packets received}}{\text{packets sent}}
\]

5.3 Tests Configurations

Three different network configurations are used with each configuration intended to exercise a different typical network scenario. The first configuration is the single router configuration (Figure 5.2). In the single router configuration the test PC and six endpoints are all connected to a single router. This is a base case network configuration to provide a comparison with the other network configurations.

The second network configuration is the string of routers configuration (Figure 5.3). In this configuration three routers are connected in a serial fashion. The test PC is connected to the first router and two endpoint are connect to each router including the first. In order for the data from the data from endpoints 5 & 6 to reach the test PC, it must traverse through each of the other routers. This network configuration tests the effects of serially connect routers.
The third and last network configuration is the split router network configuration (Figure 5.4). In this configuration the test PC is attached to the first router. Two other routers are connected to the first router on different ports. Two endpoints are connected to each of the three routers. This network configuration tests the effects of a diverging path in router connections.
5.4 Test Parameters

Each endpoint is characterized by running a series of individual tests on the network. Three tests are conducted by sending 100, 500, and 1000 byte messages at a data rate of 10 hertz and collecting 5000 messages. Each of these three tests is run on a single, double, and triple router configuration (Figure 5.6). These tests serve as a measure of optimal performance.

The effect of the network topology is measured by using all the endpoints in the configurations given in section 5.3. First all endpoints produce the same size of message, again using the 100, 500, and 1000 byte messages. Finally, during the complete network tests, two endpoints produce 100 byte messages, two endpoints produce 500 byte messages, and two endpoints produce 1000 byte messages. These assignments are then rotated among
Figure 5.6. Characterizing each endpoint in each configuration

the endpoints to test the message size production at each location in the network (Figure 5.7). All messages are sent using a 10 hertz production rate and sample sizes of 5000 messages are collected.

5.5 Test Procedure and Data Collection

After a network is powered on and each application is started and configured, the network is configured by entering the test parameters into the graphical test SPA application. After the test is then started, the TestAgent application has an idle period where it ignores all incoming messages. After it reaches the end of its idle period it starts to record data. This is done to allow the SPA network time to get up to speed in its data production. Data is collected for a fixed number of messages and then the test agent unsubscribes from the endpoints and alerts the graphical test SPA application that it is done collecting data. The
Figure 5.7. Complete network test message size rotations

<table>
<thead>
<tr>
<th>Config</th>
<th>100</th>
<th>500</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>1,2</td>
<td>3,4</td>
<td>5,6</td>
</tr>
<tr>
<td>String</td>
<td>1,2</td>
<td>3,4</td>
<td>5,6</td>
</tr>
<tr>
<td></td>
<td>6,1</td>
<td>2,3</td>
<td>4,5</td>
</tr>
<tr>
<td></td>
<td>5,6</td>
<td>1,2</td>
<td>3,4</td>
</tr>
<tr>
<td></td>
<td>4,5</td>
<td>6,1</td>
<td>2,3</td>
</tr>
<tr>
<td></td>
<td>3,4</td>
<td>5,6</td>
<td>1,2</td>
</tr>
<tr>
<td></td>
<td>2,3</td>
<td>4,5</td>
<td>6,1</td>
</tr>
<tr>
<td>Split</td>
<td>1,2</td>
<td>3,4</td>
<td>5,6</td>
</tr>
<tr>
<td></td>
<td>6,1</td>
<td>2,3</td>
<td>4,5</td>
</tr>
<tr>
<td></td>
<td>5,6</td>
<td>1,2</td>
<td>3,4</td>
</tr>
<tr>
<td></td>
<td>4,5</td>
<td>6,1</td>
<td>2,3</td>
</tr>
<tr>
<td></td>
<td>3,4</td>
<td>5,6</td>
<td>1,2</td>
</tr>
<tr>
<td></td>
<td>2,3</td>
<td>4,5</td>
<td>6,1</td>
</tr>
</tbody>
</table>

graphical application then requests all the data from the TestAgent. After receiving all the test data, the graphical test application post processes the data. All of the test metrics are calculated and graphs are generated. The test was designed to run this way so that the graphical test application would not interfere with the network while the test was running. Only the endpoints and the test agent participate during the network test. A sequence diagram of the test procedures can be see in Figure 5.8.
Figure 5.8. Test procedure sequence
CHAPTER 6
RESULTS

6.1 Introduction

It is not the intention of this dissertation to characterize and measure the SPA network performance, however, some data was collected to demonstrate that the network is sufficiently performant for use on a spacecraft. This chapter presents the results from the SSM implementation, as described in Chapter 4 using the experimental setup detailed in Chapter 5. It has been found that message latency is dependent on three factors, ordered from greatest to smallest impact:

1. message size

2. the number of endpoints and their data production

3. network topology

A selection of the collected data is presented and discussed, and the comprehensive set of data is available in Appendix C.

6.2 Measured Performance

In the tests that ran with all the endpoints it was found that a 100 byte message has a latency of 4.5 milliseconds and a jitter of 100 microseconds. A 500 byte message has a latency of 13.06 milliseconds and a jitter of 100 microseconds. A 1000 byte message has a latency of 24.22 milliseconds and a jitter of 100 microseconds. The packet loss percentage during the tests was zero.
Figure 6.1. All 100 byte message data points collected from endpoint 1

Figure 6.2. All 500 byte message data points collected from endpoint 1
6.3 Selected Data and Aggregation

Figures 6.1, 6.2, and 6.3 represent the aggregation of all the 100, 500, and 1000 byte message latency data collected from endpoint 1. These three graphs provide a summary of the findings from the experiments with each test represented in a different color. It is to be noted in Figure 6.1 that the majority of the 100 byte message latencies is just below the 4.5 millisecond mark. The message latencies recorded around 5 milliseconds and above are from the network experiments with other endpoints sending 500 and 1000 byte messages. The increased latency caused by the other network traffic has a greater impact in the 500 and 1000 byte aggregated latency scatter plots shown in Figures 6.2 and 6.3.

It can be seen that the majority of message latencies are statistically equal to the data collected during the original characterization of the endpoint (see Section 5.4). The bands that are clearly outside the normal distribution are collected from the tests with all of the endpoints connected to the network. Those that have the most delay are from the tests where the endpoints in the network are producing messages with a size of 1000 bytes. This
causes the longest wait for the network links to become available.

6.4 Conclusion

According to the data that has been collected, it has been found that message latency is dependent on three factors, ordered from greatest to least impact:

1. message size
2. the number of endpoints and their data production
3. network topology

In all of the experiments the effects of factors 1 and 2 are apparent, however, factor 3, the effect of the network topology seems so insignificant that there is little evidence of it. As for the effect of message size, it is logical that longer messages take longer to send. It is also logical that the number of endpoints on the network cause increased message latency due to higher wait times for network links to become available. As for the lack of evidence of the delay caused by increasing the complexity of the network topology, it is hypothesized that this effect is minimal due to the worm hole nature of SpaceWire routers. Essentially, once a link is established, the only added delay is that of the propagation of an electron on the added length of physical wire. High precision equipment would be required to measure these effects and determine the correctness of this hypothesis.
CHAPTER 7
CONCLUSIONS

7.1 Introduction

The research and work presented in this dissertation provides the basis for a complete self-configuring heterogeneous network suitable for spaceflight. The SPA network model presented here has been demonstrated in a working implementation called the SPA Services Manager (SSM). While the SSM is a working implementation there are still many improvements that can be made to the SPA network. As the concepts and designs are further developed, this dissertation provides a strong model to guide the evolution of the SPA network.

7.2 Contributions

I set out to create a SPA network model and a transport agnostic networking infrastructure that does not place the burden of routing on the network endpoints. These two goals have been met by the SPA network model and the design of the data link and network layers. The complete SPA network stack as presented in Chapter 3 has been implemented in a real world software package called the SPA Services Manager (SSM). The SSM is a flight ready, ISO-9001 certified implementation of the research done in this dissertation. Specifically, the research contributions of this dissertation include:

- Presentation of the SPA network model, enabling future development to fit properly within the rest of the SPA network. This is one of the first instances of the OSI model being applied to a spacecraft system.

- Design of the data link and network layers, which do not place the burden of routing on the network endpoints.
• The realization that the collection of software application running on a processor substantiate a new SPA subnet type, SPA-Local.

• Definition of a new query syntax for xTEDS queries to the Lookup Service.

• The flight ready implementation of this research, the SSM.

• A test platform for measuring SPA network performance.

7.3 Future Work

The research presented in this dissertation presents several opportunities for further study to increase the scalability and robustness of the network. Further study is also merited on the performance and characterization of the SPA network. Other research topics could include:

• Handling dynamic discovery and topology mapping of a SpaceWire network with loops

• Fault tolerance and fail over of core network components including subnet mangers and the Lookup Service

• Implementation and characterization of the SPA-1 and SPA-U subnets

• Further characterization of the SPA-Local and SPA-SpaceWire
REFERENCES


[47] Christensen, J., Cannon, S., and Hansen, B. Automatic software generation of asim program code from an xteds. In AIAA Infotech@Aerospace Conference (Atlanta, GA, April 2010), AIAA.


[57] Falkenhayn, E. J. Multimission modular spacecraft. In *Space Programs and Technologies Conference* (Houston, TX, June 1988), AIAA.


[60] Fronterhouse, D., Center, K., and Preble, J. Building spa pnp satellites. In *7th Responsive Space Conference* (Los Angeles, CA, April 2009), AIAA.


[75] Jacobson, Z., Rawashdeh, S., and Lumpp, J. A spa-1 enabled plug-and-play cubelab for iss payloads. In AIAA Infotech@Aerospace Conference (Garden Grove, CA, June 2012), AIAA.


[87] Lanza, D., Vick, R., and Lyke, J. The space plug-and-play avionics common data dictionary – constructing the language of spa. In AIAA Infotech@Aerospace Conference (Atlanta, GA, April 2010), AIAA.


[92] Lyke, J. Plug-and-play as an enabler for future systems. In SPACE 2010 Conference and Exposition (Anaheim, CA, August 2010), AIAA.


[99] Lynaugh, K., Dossey, S., Davis, M., Middlestead, R., Lyke, J., Crane, J., and Roman, C. Space plug-and-play avionics application programmers interface for radio devices. In AIAA Infotech@Aerospace Conference (St. Louis, MO, March 2011), AIAA.


[123] Puig-Suari, J., Turner, C., and Twiggs, R. J. Cubesat: The development and launch support infrastructure for eighteen different satellite customers on one launch. In


Romero, J., Bourdu, H., Lyke, J., Czajkowski, D., Collier, C., and Avery, K. Design of an optical applique sensor interface module (o-asim). In *AIAA Infotech@Aerospace Conference* (St. Louis, MO, March 2011), AIAA.


Slane, F., and Hooke, A. Space plug-and-play avionics standards: Progress, problems and a view of the future. In *AIAA Infotech@Aerospace Conference* (Rohnert Park, CA, May 2007), AIAA.

Some, R., Neff, J., and Lyke, J. Lessons learned in building a spacecraft xml taxonomy and ontology. In *AIAA Infotech@Aerospace Conference* (Rohnert Park, CA, May 2007), AIAA.


[143] Vick, R. W. Challenges in designing reusable flight software modules. In *AIAA Infotech@Aerospace Conference* (St. Louis, MO, March 2011), AIAA.


[149] Wilmot, J. A reusable and adaptable software architecture for embedded space flight system: The core flight software system (cfs). In *International Conference on Hardware/Software Codesign and System Synthesis* (Jersey City, NJ, September 2005), NASA.


APPENDICES
Appendix A

First Recorded Instance of the SPA Network

The text below is the first record of the SPA network as it is presented in this dissertation in Chapter 3. The following is taken from the online discussion at https://pnpsoftware.sdl.usu.edu/redmine/issues/163 (see entry #11).

– July 08, 2010 at 3:51 PM –

Updated by Jacob Christensen almost 2 years ago

USU Logical Addressing Proposal

Description:
The networking is handled by the SPA Managers. Each SPA subnet, (SPA-1, SPA-U, SPA-S, SPA-O, etc.) has a SPA Manager. The SPA Managers map their respective subnets, request logical addresses from an addressing authority, and assign them to the endpoints. Each SPA Manager knows how to talk to each other and the endpoints in their subnet. When routing a message the SPA Manager takes the message and encapsulates it in a packet that is native to their subnet. Simple routing tables are mapped during configuration. These tables are mostly static and require very little updating.

Address format:
SPA Version - 1 byte
Message Opcode - 1 byte
QoS - 1 byte
Source - 2 bytes (Logical ID)
Destination - 2 bytes (Logical ID)
Length - 2 bytes
CRC - 2 bytes
**Router Behavior:**

Routers do not require any special routing tables. The router does not look at the address fields in the SPA header. On SpW, path routing is used. Ethernet uses IP routing. Each SPA frame is encapsulated in a frame native to the transport medium it is traversing. The routers make no changes to the SPA frame. Nothing needs to be updated in the router during reconfiguration.

**Endpoint behavior:**

The endpoint only needs the 2 byte destination address to send a message to an endpoint. Because we are using logical addressing, in the event of reconfiguration the logical addresses stay the same and only the physical addresses would change. This is transparent to the endpoints. (In other words, no change on reconfiguration.)

**The pros of your system as you see it.**

- Very flexible approach that abstracts the networking details away from the endpoints.
  No custom hardware requirements.
- COTS networking components can route encapsulated SPA messages.
- SPA Message is intact/unmodified from source to destination
- Scales to new SPA subnets and network sizes.
- Ability to nest subnets.
- Works for all SPA subnet types.
- Works for future SPA subnet types.

**The cons of your system as you see it.**

- Requires a centralized addressing authority in the system (although redundancy is an option).
- Quick table lookup required at each hop in the network.
Appendix B
Message Definitions

B.1 Introduction

This appendix provides a complete detailed listing of the SPA Network messages. This appendix is given as a reference. Many of the message tables defined in this appendix are referenced throughout this dissertation. It is important to list these to allow for easy reproduction of the work.

B.2 SPA-L Messages

<table>
<thead>
<tr>
<th>Message Name</th>
<th>LocalHeader</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Opcode</td>
<td>n/a</td>
</tr>
<tr>
<td>Summary Description</td>
<td>Standard header included on every Local message</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SourcePort</td>
<td>UInt16</td>
<td>n/a</td>
<td>Port of the sending component</td>
</tr>
<tr>
<td>Length</td>
<td>UInt16</td>
<td>n/a</td>
<td>Length of the message payload</td>
</tr>
<tr>
<td>Opcode</td>
<td>UInt8</td>
<td>n/a</td>
<td>Unique opcode of the message</td>
</tr>
</tbody>
</table>

Table B.1. Message definition for the LocalHeader message
<table>
<thead>
<tr>
<th>Message Name</th>
<th>LocalHello</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Opcode</td>
<td>0x20</td>
</tr>
<tr>
<td>Summary Description</td>
<td>Sent by a component to the SM-L in order to alert the SM-L of its presence</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUUID</td>
<td>UInt128</td>
<td>UUID</td>
<td>Universally Unique Id of the SPA component</td>
</tr>
<tr>
<td>ComponentType</td>
<td>UInt8</td>
<td>n/a</td>
<td>The type of SPA component</td>
</tr>
</tbody>
</table>

Table B.2. Message definition for the LocalHello message

<table>
<thead>
<tr>
<th>Message Name</th>
<th>LocalAck</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Opcode</td>
<td>0x21</td>
</tr>
<tr>
<td>Summary Description</td>
<td>Sent in order to acknowledge a message</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status</td>
<td>UInt8</td>
<td>n/a</td>
<td>The status of the acknowledgement</td>
</tr>
</tbody>
</table>

Table B.3. Message definition for the LocalAck message

<table>
<thead>
<tr>
<th>Message Name</th>
<th>LocalRouteRequest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Opcode</td>
<td>0x23</td>
</tr>
<tr>
<td>Summary Description</td>
<td>Sent by a component to the SM-L in order to request a direct route to another component</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical Address</td>
<td>UInt32</td>
<td>n/a</td>
<td>Logical address for which the sender desires a direct path</td>
</tr>
</tbody>
</table>

Table B.4. Message definition for the LocalRouteRequest message
### LocalRoute Message

**Message Opcode**
0x24

**Summary Description**
Sent by the SPA-L to a component in order to provide a direct route from the component to another component without having to go through the SM-L.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUUID</td>
<td>UInt128</td>
<td>UUID</td>
<td>Universally Unique Id of the SPA component</td>
</tr>
<tr>
<td>Logical Address</td>
<td>UInt32</td>
<td>n/a</td>
<td>Logical address of requested route</td>
</tr>
<tr>
<td>Port</td>
<td>UInt16</td>
<td>n/a</td>
<td>Port of the most direct route</td>
</tr>
<tr>
<td>ComponentType</td>
<td>UInt8</td>
<td>n/a</td>
<td>The type of SPA component</td>
</tr>
<tr>
<td>AckRequired</td>
<td>UInt8</td>
<td>n/a</td>
<td>Acknowledge required flag</td>
</tr>
</tbody>
</table>

Table B.5. Message definition for the LocalRoute message
## B.3 SPA-1 Messages

<table>
<thead>
<tr>
<th>Message Name</th>
<th>OneArp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Opcode</td>
<td>0x30</td>
</tr>
<tr>
<td>Summary Description</td>
<td>Sent by a SPA-1 component to discover if anyone is on the given address</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUUID</td>
<td>UInt128</td>
<td>UUID</td>
<td>Universally Unique Id of the SPA component</td>
</tr>
</tbody>
</table>

Table B.6. Message definition for the OneArp message

<table>
<thead>
<tr>
<th>Message Name</th>
<th>OneHello</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Opcode</td>
<td>0x31</td>
</tr>
<tr>
<td>Summary Description</td>
<td>Sent by the SPA-1 subnet manager to discover SPA-1 components</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUUID</td>
<td>UInt128</td>
<td>UUID</td>
<td>Universally Unique Id of the SPA-1 subnet manager</td>
</tr>
</tbody>
</table>

Table B.7. Message definition for the OneHello message

<table>
<thead>
<tr>
<th>Message Name</th>
<th>OneAck</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Opcode</td>
<td>0x32</td>
</tr>
<tr>
<td>Summary Description</td>
<td>Sent in order to acknowledge a message</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUUID</td>
<td>UInt128</td>
<td>UUID</td>
<td>Universally Unique Id of the SPA-1 component</td>
</tr>
</tbody>
</table>

Table B.8. Message definition for the OneAck message
### B.4 SPA-U Messages

<table>
<thead>
<tr>
<th>Message Name</th>
<th>UsbHello</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Opcode</td>
<td>0x90</td>
</tr>
<tr>
<td>Summary Description</td>
<td>Sent by a SPA-U subnet manager to discover SPA-U components</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUUID</td>
<td>UInt128</td>
<td>UUID</td>
<td>Universally Unique Id of the SPA-U subnet manager</td>
</tr>
</tbody>
</table>

Table B.9. Message definition for the UsbHello message

<table>
<thead>
<tr>
<th>Message Name</th>
<th>UsbAck</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Opcode</td>
<td>0x91</td>
</tr>
<tr>
<td>Summary Description</td>
<td>Sent in order to acknowledge a message</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUUID</td>
<td>UInt128</td>
<td>UUID</td>
<td>Universally Unique Id of the SPA-U component</td>
</tr>
</tbody>
</table>

Table B.10. Message definition for the UsbAck message
### B.5 SPA-S Messages

<table>
<thead>
<tr>
<th>Message Name</th>
<th>SpwHeader</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Opcode</td>
<td>n/a</td>
</tr>
<tr>
<td>Summary Description</td>
<td>SPA-S header</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Route</td>
<td>UInt8</td>
<td>bytes</td>
<td>SpaceWire route to an SpaceWire endpoint</td>
</tr>
<tr>
<td>Protocol ID</td>
<td>UInt8</td>
<td>n/a</td>
<td>SpaceWire protocol ID</td>
</tr>
<tr>
<td>Return Route</td>
<td>UInt8</td>
<td>bytes</td>
<td>Null terminated SpaceWire route to return to the sending endpoint from the receiving endpoint</td>
</tr>
<tr>
<td>Forward Route</td>
<td>UInt8</td>
<td>bytes</td>
<td>Null terminated SpaceWire route from the sending endpoint to the receiving endpoint</td>
</tr>
<tr>
<td>MessageLength</td>
<td>UInt16</td>
<td>bytes</td>
<td>Number of bytes in the message</td>
</tr>
<tr>
<td>Opcode</td>
<td>UInt8</td>
<td>n/a</td>
<td>Unique opcode of the message</td>
</tr>
</tbody>
</table>

Table B.11. Message definition for the SpwHeader message

<table>
<thead>
<tr>
<th>Message Name</th>
<th>SpwRouterProbe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Opcode</td>
<td>0x6A</td>
</tr>
<tr>
<td>Summary Description</td>
<td>SpaceWire Router Probe used to discover routers via reflection</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUUID</td>
<td>UInt128</td>
<td>UUID</td>
<td>Universally Unique Id of the SPA-S subnet manager that sent the message</td>
</tr>
</tbody>
</table>

Table B.12. Message definition for the SpwRouterProbe message
<table>
<thead>
<tr>
<th>Message Name</th>
<th>SpwEndpointPing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Opcode</td>
<td>0x6B</td>
</tr>
<tr>
<td>Summary Description</td>
<td>SpaceWire Endpoint Ping used to discover SPA-S components attached to a SpaceWire router</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUUID</td>
<td>UInt128</td>
<td>UUID</td>
<td>Universally Unique Id of the SPA-S subnet manager that sent the message</td>
</tr>
</tbody>
</table>

Table B.13. Message definition for the SpwEndpointPing message

<table>
<thead>
<tr>
<th>Message Name</th>
<th>SpwEndpointPingReply</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Opcode</td>
<td>0x6C</td>
</tr>
<tr>
<td>Summary Description</td>
<td>SpaceWire Endpoint Ping Reply returned to the SPA-S subnet manager by a SPA-S component in response to a SpwEndpointPing</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUUID</td>
<td>UInt128</td>
<td>UUID</td>
<td>Universally Unique Id of the SPA-S subnet manager that sent the message</td>
</tr>
<tr>
<td>ComponentType</td>
<td>UInt8</td>
<td>n/a</td>
<td>Type of SPA component receiving address</td>
</tr>
</tbody>
</table>

Table B.14. Message definition for the SpwEndpointPingReply message
<table>
<thead>
<tr>
<th>Message Name</th>
<th>SpwConfigureTopologyDiscovery</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Opcode</td>
<td>0x6D</td>
</tr>
<tr>
<td>Summary Description</td>
<td>Message used to configure the SPA-S subnet manager rediscovery rate of the SPA-S subnet</td>
</tr>
<tr>
<td>Field Name</td>
<td>Type</td>
</tr>
<tr>
<td>Period</td>
<td>UInt32</td>
</tr>
</tbody>
</table>

Table B.15. Message definition for the SpwConfigureTopologyDiscovery message

<table>
<thead>
<tr>
<th>Message Name</th>
<th>SpwRouteRequest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Opcode</td>
<td>0x6F</td>
</tr>
<tr>
<td>Summary Description</td>
<td>Message used to request a route between two SPA logical addresses</td>
</tr>
<tr>
<td>Field Name</td>
<td>Type</td>
</tr>
<tr>
<td>FromLogicalAddress</td>
<td>UInt32</td>
</tr>
<tr>
<td>ToLogicalAddress</td>
<td>UInt32</td>
</tr>
</tbody>
</table>

Table B.16. Message definition for the SpwRouteRequest message
<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUUID</td>
<td>UInt128</td>
<td>UUID</td>
<td>Universally Unique Id for the SPA-S destination component</td>
</tr>
<tr>
<td>LogicalAddress</td>
<td>UInt32</td>
<td>n/a</td>
<td>The logical address of the destination component</td>
</tr>
<tr>
<td>ComponentType</td>
<td>UInt8</td>
<td>n/a</td>
<td>Type of SPA component receiving address</td>
</tr>
<tr>
<td>PathLength</td>
<td>UInt8</td>
<td>bytes</td>
<td>Length of the SpaceWire route</td>
</tr>
<tr>
<td>PathRoute</td>
<td>UInt8 [ ]</td>
<td>bytes</td>
<td>The SpaceWire route</td>
</tr>
</tbody>
</table>

Table B.17. Message definition for the SpwRoute message
### B.6 SPA Messages

<table>
<thead>
<tr>
<th>Message Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpaHeader</td>
<td></td>
<td>n/a</td>
<td>Standard header included on every SPA message</td>
</tr>
<tr>
<td>Message Opcode</td>
<td>n/a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Summary Description</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>UInt8</td>
<td>n/a</td>
<td>SPA version number</td>
</tr>
<tr>
<td>Priority</td>
<td>UInt8</td>
<td>n/a</td>
<td>Message priority</td>
</tr>
<tr>
<td>Length</td>
<td>UInt16</td>
<td>bytes</td>
<td>Length of the message payload</td>
</tr>
<tr>
<td>Destination</td>
<td>UInt32</td>
<td>n/a</td>
<td>Destination logical address</td>
</tr>
<tr>
<td>Source</td>
<td>UInt32</td>
<td>n/a</td>
<td>Source logical address</td>
</tr>
<tr>
<td>Flags</td>
<td>UInt16</td>
<td>n/a</td>
<td>Special message flags</td>
</tr>
<tr>
<td>Opcode</td>
<td>UInt8</td>
<td>n/a</td>
<td>Unique opcode of the message</td>
</tr>
<tr>
<td>Extended Header Length</td>
<td>UInt8</td>
<td>bytes</td>
<td>Length of the extended headers</td>
</tr>
</tbody>
</table>

Table B.18. Message definition for the SpaHeader message

<table>
<thead>
<tr>
<th>Message Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpaAck</td>
<td>0x41</td>
<td></td>
</tr>
<tr>
<td>Message Opcode</td>
<td></td>
<td>SPA message guaranteed delivery acknowledgement</td>
</tr>
<tr>
<td>Summary Description</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AckId</td>
<td>UInt8</td>
<td>n/a</td>
<td>Id to associate ack with original message</td>
</tr>
</tbody>
</table>

Table B.19. Message definition for the SpaAck message
<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUUID</td>
<td>UInt128</td>
<td>UUID</td>
<td>Universally Unique Id of the SPA component</td>
</tr>
<tr>
<td>XUUID</td>
<td>UInt128</td>
<td>UUID</td>
<td>Universally Unique Id of the xTEDS</td>
</tr>
<tr>
<td>Address</td>
<td>UInt32</td>
<td>n/a</td>
<td>Address of a component</td>
</tr>
<tr>
<td>DialogId</td>
<td>UInt16</td>
<td>n/a</td>
<td>Dialog identifier set by the requester</td>
</tr>
<tr>
<td>RequestType</td>
<td>UInt16</td>
<td>n/a</td>
<td>Indicator for which field to use in the request</td>
</tr>
</tbody>
</table>

Table B.20. Message definition for the SpaXtedsRequest message

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XUUID</td>
<td>UInt128</td>
<td>UUID</td>
<td>Universally Unique Id of the xTEDS</td>
</tr>
<tr>
<td>Address</td>
<td>UInt32</td>
<td>n/a</td>
<td>Address of the component</td>
</tr>
<tr>
<td>DialogId</td>
<td>UInt16</td>
<td>n/a</td>
<td>Dialog identifier set by the requester</td>
</tr>
<tr>
<td>ReplyStatus</td>
<td>UInt16</td>
<td>n/a</td>
<td>Status code of the message</td>
</tr>
<tr>
<td>xTEDS</td>
<td>String</td>
<td>n/a</td>
<td>xTEDS of the requested component</td>
</tr>
</tbody>
</table>

Table B.21. Message definition for the SpaXtedsReply message
<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProducerAddress</td>
<td>UInt32</td>
<td>n/a</td>
<td>Address of the producer component</td>
</tr>
<tr>
<td>ConsumerAddress</td>
<td>UInt32</td>
<td>n/a</td>
<td>Address of the consumer component</td>
</tr>
<tr>
<td>ManagerAddress</td>
<td>UInt32</td>
<td>n/a</td>
<td>Address of the subscriptions manager component</td>
</tr>
<tr>
<td>LeasePeriod</td>
<td>UInt32</td>
<td>seconds</td>
<td>Duration of the subscription, 0 = unlimited</td>
</tr>
<tr>
<td>DialogId</td>
<td>UInt16</td>
<td>n/a</td>
<td>Dialog identifier set by the requester</td>
</tr>
<tr>
<td>DeliveryRateDivisor</td>
<td>UInt16</td>
<td>n/a</td>
<td>Subscribe to every nth message</td>
</tr>
<tr>
<td>InterfaceId</td>
<td>UInt8</td>
<td>n/a</td>
<td>xTEDS interface Id</td>
</tr>
<tr>
<td>MessageId</td>
<td>UInt8</td>
<td>n/a</td>
<td>xTEDS message Id</td>
</tr>
<tr>
<td>SubscriptionPriority</td>
<td>UInt8</td>
<td>n/a</td>
<td>subscription priority, 0=highest, 255=lowest</td>
</tr>
<tr>
<td>Type</td>
<td>UInt8</td>
<td>n/a</td>
<td>message type, subscription or unsubscription</td>
</tr>
</tbody>
</table>

Table B.22. Message definition for the SpaSubscriptionRequest message
<table>
<thead>
<tr>
<th>Message Name</th>
<th>SpaSubscriptionReply</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Opcode</td>
<td>0x47</td>
</tr>
<tr>
<td>Summary Description</td>
<td>Reply to SpaSubscriptionRequest message</td>
</tr>
<tr>
<td>Field Name</td>
<td>Type</td>
</tr>
<tr>
<td>DialogId</td>
<td>UInt16</td>
</tr>
<tr>
<td>ReplyType</td>
<td>UInt8</td>
</tr>
</tbody>
</table>

Table B.23. Message definition for the SpaSubscriptionReply message

<table>
<thead>
<tr>
<th>Message Name</th>
<th>SpaQueryRequest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Opcode</td>
<td>0x48</td>
</tr>
<tr>
<td>Summary Description</td>
<td>Request information about currently registered providers</td>
</tr>
<tr>
<td>Field Name</td>
<td>Type</td>
</tr>
<tr>
<td>DialogId</td>
<td>UInt16</td>
</tr>
<tr>
<td>QueryType</td>
<td>UInt8</td>
</tr>
<tr>
<td>Reserved</td>
<td>UInt8</td>
</tr>
<tr>
<td>Query</td>
<td>String</td>
</tr>
</tbody>
</table>

Table B.24. Message definition for the SpaQueryRequest message
<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProviderCuuid</td>
<td>UInt128</td>
<td>UUID</td>
<td>Component UUID of the message provider</td>
</tr>
<tr>
<td>ProviderXuuid</td>
<td>UInt128</td>
<td>UUID</td>
<td>xTEDS UUID of the message provider</td>
</tr>
<tr>
<td>ProviderAddress</td>
<td>UInt32</td>
<td>n/a</td>
<td>Logical address of the message provider</td>
</tr>
<tr>
<td>DialogId</td>
<td>UInt16</td>
<td>n/a</td>
<td>Dialog identifier set by the requester</td>
</tr>
<tr>
<td>InterfaceId</td>
<td>UInt8</td>
<td>n/a</td>
<td>xTEDS interface Id</td>
</tr>
<tr>
<td>MessageId</td>
<td>UInt8</td>
<td>n/a</td>
<td>xTEDS message Id</td>
</tr>
<tr>
<td>MessageType</td>
<td>UInt8</td>
<td>n/a</td>
<td>Message type: Notification(0), Command(1), Request(2)</td>
</tr>
<tr>
<td>ReplyType</td>
<td>UInt8</td>
<td>n/a</td>
<td>Indicates registration or cancellation of the query</td>
</tr>
<tr>
<td>VariableIdList</td>
<td>n1</td>
<td>n/a</td>
<td>Null terminated, comma separated list of variable IDs</td>
</tr>
<tr>
<td>MessageDefinition</td>
<td>n2</td>
<td>n/a</td>
<td>Null-terminated format string of the xTEDS message</td>
</tr>
<tr>
<td>XtedsSection</td>
<td>n3</td>
<td>n/a</td>
<td>Null terminated section of the xTEDS message definition from the xTEDS</td>
</tr>
</tbody>
</table>

Table B.25. Message definition for the SpaQueryReply message
### Table B.26. Message definition for the SpaRequestAddressBlock message

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUUID</td>
<td>UInt128</td>
<td>UUID</td>
<td>Universally Unique SPA Identifier of requesting component</td>
</tr>
</tbody>
</table>

### Table B.27. Message definition for the SpaAssignAddressBlock message

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUUID</td>
<td>UInt128</td>
<td>UUID</td>
<td>Component UUID of the destination component</td>
</tr>
<tr>
<td>AddressBlock</td>
<td>UInt32</td>
<td>n/a</td>
<td>The base address of the address block</td>
</tr>
<tr>
<td>ResponseType</td>
<td>UInt8</td>
<td>n/a</td>
<td>Type of address response: valid (0) or invalid (1)</td>
</tr>
<tr>
<td>Field Name</td>
<td>Type</td>
<td>Units</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>---------</td>
<td>-------</td>
<td>--------------------------------------------------------------------</td>
</tr>
<tr>
<td>CUUID</td>
<td>UInt128</td>
<td>UUID</td>
<td>Universally Unique Id of the component</td>
</tr>
<tr>
<td>Address</td>
<td>UInt32</td>
<td>n/a</td>
<td>Logical address</td>
</tr>
<tr>
<td>ComponentType</td>
<td>UInt8</td>
<td>n/a</td>
<td>Type of SPA component for which the route is being advertised</td>
</tr>
</tbody>
</table>

Table B.28. Message definition for the SpaDistributeRoute message

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>UInt32</td>
<td>n/a</td>
<td>Logical address</td>
</tr>
</tbody>
</table>

Table B.29. Message definition for the SpaRequestLookupServiceProbe message
<table>
<thead>
<tr>
<th>Message Name</th>
<th>SpaData</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Opcode</td>
<td>0x74</td>
</tr>
<tr>
<td>Summary Description</td>
<td>xTEDS level message for a Notification message</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DialogId</td>
<td>UInt16</td>
<td>n/a</td>
<td>Dialog identifier set by the requester</td>
</tr>
<tr>
<td>PayloadLength</td>
<td>UInt16</td>
<td>bytes</td>
<td>Length of the payload data</td>
</tr>
<tr>
<td>SequenceIndex</td>
<td>UInt16</td>
<td>n/a</td>
<td>The 1-indexed number of index of the message within the sequence, the 'i' in 'i' of 'j'</td>
</tr>
<tr>
<td>SequenceCount</td>
<td>UInt16</td>
<td>n/a</td>
<td>The total number of messages within the sequence, the 'j' in 'i' of 'j'</td>
</tr>
<tr>
<td>InterfaceId</td>
<td>UInt8</td>
<td>n/a</td>
<td>xTEDS interface Id</td>
</tr>
<tr>
<td>MessageId</td>
<td>UInt8</td>
<td>n/a</td>
<td>xTEDS message Id</td>
</tr>
<tr>
<td>Payload</td>
<td>n</td>
<td>n/a</td>
<td>Message data payload</td>
</tr>
</tbody>
</table>

Table B.30. Message definition for the SpaData message

<table>
<thead>
<tr>
<th>Message Name</th>
<th>SpaServiceRequest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Opcode</td>
<td>0x75</td>
</tr>
<tr>
<td>Summary Description</td>
<td>xTEDS level message for the CommandMsg portion of a Request message</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DialogId</td>
<td>UInt16</td>
<td>n/a</td>
<td>Dialog identifier set by the requester</td>
</tr>
<tr>
<td>PayloadLength</td>
<td>UInt16</td>
<td>bytes</td>
<td>Length of the payload data</td>
</tr>
<tr>
<td>InterfaceId</td>
<td>UInt8</td>
<td>n/a</td>
<td>xTEDS interface Id</td>
</tr>
<tr>
<td>MessageId</td>
<td>UInt8</td>
<td>n/a</td>
<td>xTEDS message Id</td>
</tr>
<tr>
<td>Payload</td>
<td>n</td>
<td>n/a</td>
<td>Message data payload</td>
</tr>
</tbody>
</table>

Table B.31. Message definition for the SpaServiceRequest message
### Table B.32. Message definition for the SpaServiceReply message

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DialogId</td>
<td>UInt16</td>
<td>n/a</td>
<td>Dialog identifier set by the requester</td>
</tr>
<tr>
<td>PayloadLength</td>
<td>UInt16</td>
<td>bytes</td>
<td>Length of the payload data</td>
</tr>
<tr>
<td>SequenceIndex</td>
<td>UInt16</td>
<td>n/a</td>
<td>The 1-indexed number of index of the message within the sequence, the 'i' in 'i' of 'j'</td>
</tr>
<tr>
<td>SequenceCount</td>
<td>UInt16</td>
<td>n/a</td>
<td>The total number of messages within the sequence, the 'j' in 'i' of 'j'</td>
</tr>
<tr>
<td>InterfaceId</td>
<td>UInt8</td>
<td>n/a</td>
<td>xTEDS interface Id</td>
</tr>
<tr>
<td>MessageId</td>
<td>UInt8</td>
<td>n/a</td>
<td>xTEDS message Id</td>
</tr>
<tr>
<td>Payload</td>
<td>n</td>
<td>n/a</td>
<td>Message data payload</td>
</tr>
</tbody>
</table>

### Table B.33. Message definition for the SpaProbeRequest message

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DialogId</td>
<td>UInt16</td>
<td>n/a</td>
<td>Dialog identifier set by the requester</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUUUID</td>
<td>UInt128</td>
<td>UUID</td>
<td>Universally Unique Id of the SPA compo-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>nent</td>
</tr>
<tr>
<td>XUUID</td>
<td>UInt128</td>
<td>UUID</td>
<td>Universally Unique Id of the xTEDS</td>
</tr>
<tr>
<td>FaultIndicator</td>
<td>UInt32</td>
<td>Bit field</td>
<td>Indicates fault conditions</td>
</tr>
<tr>
<td>Uptime</td>
<td>UInt32</td>
<td>seconds</td>
<td>Seconds since power on</td>
</tr>
<tr>
<td>DialogId</td>
<td>UInt16</td>
<td>n/a</td>
<td>Dialog identifier set by the requester</td>
</tr>
</tbody>
</table>

Table B.34. Message definition for the SpaProbeReply message

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PayloadLength</td>
<td>Uint16</td>
<td>bytes</td>
<td>Length of the payload data</td>
</tr>
<tr>
<td>InterfaceId</td>
<td>Uint8</td>
<td>n/a</td>
<td>xTEDS interface Id</td>
</tr>
<tr>
<td>MessageId</td>
<td>Uint8</td>
<td>n/a</td>
<td>xTEDS message Id</td>
</tr>
<tr>
<td>Payload</td>
<td>n</td>
<td>n/a</td>
<td>Message data payload</td>
</tr>
</tbody>
</table>

Table B.35. Message definition for the SpaCommand message
<table>
<thead>
<tr>
<th>Message Name</th>
<th>SpaAssignAddress</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Opcode</td>
<td>0x7b</td>
</tr>
<tr>
<td>Summary Description</td>
<td>Sent to assign a logical address to a component</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUUID</td>
<td>UInt128</td>
<td>UUID</td>
<td>Universally Unique Id of the SPA component</td>
</tr>
<tr>
<td>Address</td>
<td>UInt32</td>
<td>n/a</td>
<td>The logical address being assigned</td>
</tr>
</tbody>
</table>

Table B.36. Message definition for the SpaAssignAddress message
Appendix C

Comprehensive Results
C.1 Endpoint Characterization

Endpoint 1

Figure C.1. Endpoint 1 - Summary of all three message sizes on one router
Figure C.2. Endpoint 1 - Summary of all three message sizes on two routers

Figure C.3. Endpoint 1 - Summary of all three message sizes on three routers
Figure C.4. Endpoint 1 scatter plot - One router, message size: 100 bytes

Figure C.5. Endpoint 1 histogram - One router, message size: 100 bytes
Figure C.6. Endpoint 1 scatter plot - Two routers, message size: 100 bytes

Figure C.7. Endpoint 1 histogram - Two routers, message size: 100 bytes
Figure C.8. Endpoint 1 scatter plot - Three routers, message size: 100 bytes

Figure C.9. Endpoint 1 histogram - Three routers, message size: 100 bytes
Figure C.10. Endpoint 1 scatter plot - One router, message size: 500 bytes

Figure C.11. Endpoint 1 histogram - One router, message size: 500 bytes
Figure C.12. Endpoint 1 scatter plot - Two routers, message size: 500 bytes

Figure C.13. Endpoint 1 histogram - Two routers, message size: 500 bytes
Figure C.14. Endpoint 1 scatter plot - Three routers, message size: 500 bytes

Figure C.15. Endpoint 1 histogram - Three routers, message size: 500 bytes
Figure C.16. Endpoint 1 scatter plot - One router, message size: 1000 bytes

Figure C.17. Endpoint 1 histogram - One router, message size: 1000 bytes
Figure C.18. Endpoint 1 scatter plot - Two routers, message size: 1000 bytes

Figure C.19. Endpoint 1 histogram - Two routers, message size: 1000 bytes
Figure C.20. Endpoint 1 scatter plot - Two routers, message size: 1000 bytes

Figure C.21. Endpoint 1 histogram - Three routers, message size: 1000 bytes
Endpoint 2

Figure C.22. Endpoint 2 - Summary of all three message sizes on one router

Figure C.23. Endpoint 2 - Summary of all three message sizes on two routers
Figure C.24. Endpoint 2 - Summary of all three message sizes on three routers
Figure C.25. Endpoint 2 scatter plot - One router, message size: 100 bytes

Figure C.26. Endpoint 2 histogram - One router, message size: 100 bytes
Figure C.27. Endpoint 2 scatter plot - Two routers, message size: 100 bytes

Figure C.28. Endpoint 2 histogram - Two routers, message size: 100 bytes
Figure C.29. Endpoint 2 scatter plot - Three routers, message size: 100 bytes

Figure C.30. Endpoint 2 histogram - Three routers, message size: 100 bytes
Figure C.31. Endpoint 2 scatter plot - One router, message size: 500 bytes

Figure C.32. Endpoint 2 histogram - One router, message size: 500 bytes
Figure C.33. Endpoint 2 scatter plot - Two routers, message size: 500 bytes

Figure C.34. Endpoint 2 histogram - Two routers, message size: 500 bytes
Figure C.35. Endpoint 2 scatter plot - Three routers, message size: 500 bytes

Figure C.36. Endpoint 2 histogram - Three routers, message size: 500 bytes
Figure C.37. Endpoint 2 scatter plot - One router, message size: 1000 bytes

Max: 23.677 ms
Min: 22.809 ms
Mean: 23.168 ms
Std Dev: 0.078 ms

Figure C.38. Endpoint 2 histogram - One router, message size: 1000 bytes
Figure C.39. Endpoint 2 scatter plot - Two routers, message size: 1000 bytes

Figure C.40. Endpoint 2 histogram - Two routers, message size: 1000 bytes
Figure C.41. Endpoint 2 scatter plot - Three routers, message size: 1000 bytes

Figure C.42. Endpoint 2 histogram - Three routers, message size: 1000 bytes
Endpoint 3

Figure C.43. Endpoint 3 - Summary of all three message sizes on one router

Figure C.44. Endpoint 3 - Summary of all three message sizes on two routers
Figure C.45. Endpoint 3 - Summary of all three message sizes on three routers
Figure C.46. Endpoint 3 scatter plot - One router, message size: 100 bytes

Figure C.47. Endpoint 3 histogram - One router, message size: 100 bytes
Figure C.48. Endpoint 3 scatter plot - Two routers, message size: 100 bytes

Figure C.49. Endpoint 3 histogram - Two routers, message size: 100 bytes
Figure C.50. Endpoint 3 scatter plot - Three routers, message size: 100 bytes

Figure C.51. Endpoint 3 histogram - Three routers, message size: 100 bytes
Figure C.52. Endpoint 3 scatter plot - One router, message size: 500 bytes

Figure C.53. Endpoint 3 histogram - One router, message size: 500 bytes
Figure C.54. Endpoint 3 scatter plot - Two routers, message size: 500 bytes

Figure C.55. Endpoint 3 histogram - Two routers, message size: 500 bytes
Figure C.56. Endpoint 3 scatter plot - Three routers, message size: 500 bytes

Figure C.57. Endpoint 3 histogram - Three routers, message size: 500 bytes
Figure C.58. Endpoint 3 scatter plot - One router, message size: 1000 bytes

Figure C.59. Endpoint 3 histogram - One router, message size: 1000 bytes
Figure C.60. Endpoint 3 scatter plot - Two routers, message size: 1000 bytes

Figure C.61. Endpoint 3 histogram - Two routers, message size: 1000 bytes
Figure C.62. Endpoint 3 scatter plot - Three routers, message size: 1000 bytes

Max: 23.470 ms  
Min: 22.833 ms  
Mean: 23.192 ms  
Std Dev: 0.067 ms

Figure C.63. Endpoint 3 histogram - Three routers, message size: 1000 bytes
Endpoint 4

Figure C.64. Endpoint 4 - Summary of all three message sizes on one router

Figure C.65. Endpoint 4 - Summary of all three message sizes on two routers
Figure C.66. Endpoint 4 - Summary of all three message sizes on three routers
Figure C.67. Endpoint 4 scatter plot - One router, message size: 100 bytes

Figure C.68. Endpoint 4 histogram - One router, message size: 100 bytes
Figure C.69. Endpoint 4 scatter plot - Two routers, message size: 100 bytes

Figure C.70. Endpoint 4 histogram - Two routers, message size: 100 bytes
Figure C.71. Endpoint 4 scatter plot - Three routers, message size: 100 bytes

Figure C.72. Endpoint 4 histogram - Three routers, message size: 100 bytes
Figure C.73. Endpoint 4 scatter plot - One router, message size: 500 bytes

Figure C.74. Endpoint 4 histogram - One router, message size: 500 bytes
Figure C.75. Endpoint 4 scatter plot - Two routers, message size: 500 bytes

Figure C.76. Endpoint 4 histogram - Two routers, message size: 500 bytes
Figure C.77. Endpoint 4 scatter plot - Three routers, message size: 500 bytes

Figure C.78. Endpoint 4 histogram - Three routers, message size: 500 bytes
Figure C.79. Endpoint 4 scatter plot - One router, message size: 1000 bytes

Figure C.80. Endpoint 4 histogram - One router, message size: 1000 bytes
Figure C.81. Endpoint 4 scatter plot - Two routers, message size: 1000 bytes

Figure C.82. Endpoint 4 histogram - Two routers, message size: 1000 bytes
Figure C.83. Endpoint 4 scatter plot - Three routers, message size: 1000 bytes

Figure C.84. Endpoint 4 histogram - Three routers, message size: 1000 bytes
Endpoint 5

Figure C.85. Endpoint 5 - Summary of all three message sizes on one router

Figure C.86. Endpoint 5 - Summary of all three message sizes on two routers
Figure C.87. Endpoint 5 - Summary of all three message sizes on three routers
Figure C.88. Endpoint 5 scatter plot - One router, message size: 100 bytes

Figure C.89. Endpoint 5 histogram - One router, message size: 100 bytes
Figure C.90. Endpoint 5 scatter plot - Two routers, message size: 100 bytes

Figure C.91. Endpoint 5 histogram - Two routers, message size: 100 bytes
Figure C.92. Endpoint 5 scatter plot - Three routers, message size: 100 bytes

Figure C.93. Endpoint 5 histogram - Three routers, message size: 100 bytes
Figure C.94. Endpoint 5 scatter plot - One router, message size: 500 bytes

Figure C.95. Endpoint 5 histogram - One router, message size: 500 bytes
Figure C.96. Endpoint 5 scatter plot - Two routers, message size: 500 bytes

Figure C.97. Endpoint 5 histogram - Two routers, message size: 500 bytes
Figure C.98. Endpoint 5 scatter plot - Three routers, message size: 500 bytes

Figure C.99. Endpoint 5 histogram - Three routers, message size: 500 bytes
Figure C.100. Endpoint 5 scatter plot - One router, message size: 1000 bytes

Figure C.101. Endpoint 5 histogram - One router, message size: 1000 bytes
Figure C.102. Endpoint 5 scatter plot - Two routers, message size: 1000 bytes

Figure C.103. Endpoint 5 histogram - Two routers, message size: 1000 bytes
Figure C.104. Endpoint 5 scatter plot - Three routers, message size: 1000 bytes

Figure C.105. Endpoint 5 histogram - Three routers, message size: 1000 bytes
Endpoint 6

Figure C.106. Endpoint 6 - Summary of all three message sizes on one router

Figure C.107. Endpoint 6 - Summary of all three message sizes on two routers
Figure C.108. Endpoint 6 - Summary of all three message sizes on two routers
Figure C.109. Endpoint 6 scatter plot - One router, message size: 100 bytes

Figure C.110. Endpoint 6 histogram - One router, message size: 100 bytes
Figure C.111. Endpoint 6 scatter plot - Two routers, message size: 100 bytes

Figure C.112. Endpoint 6 histogram - Two routers, message size: 100 bytes
Figure C.113. Endpoint 6 scatter plot - Three routers, message size: 100 bytes

Figure C.114. Endpoint 6 histogram - Three routers, message size: 100 bytes
Figure C.115. Endpoint 6 scatter plot - One router, message size: 500 bytes

Figure C.116. Endpoint 6 histogram - One router, message size: 500 bytes
Figure C.117. Endpoint 6 scatter plot - Two routers, message size: 500 bytes

Figure C.118. Endpoint 6 histogram - Two routers, message size: 500 bytes
Figure C.119. Endpoint 6 scatter plot - Three routers, message size: 500 bytes

Figure C.120. Endpoint 6 histogram - Three routers, message size: 500 bytes
Figure C.121. Endpoint 6 scatter plot - One router, message size: 1000 bytes

Figure C.122. Endpoint 6 histogram - One router, message size: 1000 bytes
Figure C.123. Endpoint 6 scatter plot - Two routers, message size: 1000 bytes

Figure C.124. Endpoint 6 histogram - Two routers, message size: 1000 bytes
Figure C.125. Endpoint 6 scatter plot - Three routers, message size: 1000 bytes

Figure C.126. Endpoint 6 histogram - Three routers, message size: 1000 bytes
C.2 Topology Characterization

All Endpoints on a Single Router

![All endpoints scatter plot - One router, message size: 100 bytes](image)

Figure C.127. All endpoints scatter plot - One router, message size: 100 bytes
Figure C.128. All endpoints histogram - One router, message size: 100 bytes

Figure C.129. All endpoints scatter plot - One router, message size: 500 bytes
Figure C.130. All endpoints histogram - One router, message size: 500 bytes

Figure C.131. All endpoints scatter plot - One router, message size: 1000 bytes
Figure C.132. All endpoints histogram - One router, message size: 1000 bytes

Figure C.133. All endpoints scatter plot - One router, all message sizes
All Endpoints on a String of Three Routers

Figure C.134. All endpoints scatter plot - String of three routers, message size: 100 bytes

Figure C.135. All endpoints histogram - String of three routers, message size: 100 bytes
Figure C.136. All endpoints scatter plot - String of three routers, message size: 500 bytes

Figure C.137. All endpoints histogram - String of three routers, message size: 500 bytes
Figure C.138. All endpoints scatter plot - String of three routers, message size: 1000 bytes

Figure C.139. All endpoints histogram - String of three routers, message size: 1000 bytes
Figure C.140. All endpoints scatter plot - String of three routers, all message sizes, configuration 1

Figure C.141. All endpoints scatter plot - String of three routers, all message sizes, configuration 2
Figure C.142. All endpoints scatter plot - String of three routers, all message sizes, configuration 3

Figure C.143. All endpoints scatter plot - String of three routers, all message sizes, configuration 4
Figure C.144. All endpoints scatter plot - String of three routers, all message sizes, configuration 5

Figure C.145. All endpoints scatter plot - String of three routers, all message sizes, configuration 6
All Endpoints on a Split of Three Routers

Figure C.146. All endpoints scatter plot - Split of three routers, message size: 100 bytes

Figure C.147. All endpoints histogram - Split of three routers, message size: 100 bytes
Figure C.148. All endpoints scatter plot - Split of three routers, message size: 500 bytes

Figure C.149. All endpoints histogram - Split of three routers, message size: 500 bytes
Figure C.150. All endpoints scatter plot - Split of three routers, message size: 1000 bytes

Figure C.151. All endpoints histogram - Split of three routers, message size: 1000 bytes
Figure C.152. All endpoints scatter plot - Split of three routers, all message sizes, configuration 1

Figure C.153. All endpoints scatter plot - Split of three routers, all message sizes, configuration 2
Figure C.154. All endpoints scatter plot - Split of three routers, all message sizes, configuration 3

Figure C.155. All endpoints scatter plot - Split of three routers, all message sizes, configuration 4
Figure C.156. All endpoints scatter plot - Split of three routers, all message sizes, configuration 5

Figure C.157. All endpoints scatter plot - Split of three routers, all message sizes, configuration 6
Appendix D

Memory Usage
Figure D.1. Entire SSM minimal memory footprint

Figure D.2. Central Address Service minimal memory footprint
Figure D.3. Central Address Service memory usage, including SM-X address assignments

Figure D.4. Lookup Service minimal memory footprint
Figure D.5. Lookup Service memory usage during query processing

Figure D.6. Lookup Service memory usage during registration and deregistration
Figure D.7. SPA-Local Manager minimal memory footprint

Figure D.8. SPA-SpaceWire minimal memory footprint
CURRICULUM VITAE

Jacob Holt Christensen

EDUCATION

Ph.D., Computer Science. Utah State University, Logan, UT. 2012.


RESEARCH INTERESTS

Distributed and parallel systems, self-configuring network, spaceflight software, and concurrency

CONFERENCE PUBLICATIONS


INDUSTRY EXPERIENCE

Software Engineer. Space Dynamics Laboratory. 2008 to Present.

Software Developer. LDS Church. 2007.

Software Development Intern. Stennis Space Center, NASA. 2006.

TEACHING EXPERIENCE


SERVICE

Invited presentation on “Space Plug-and-Play Architecture”, CCSDS Fall meeting. 2011.

Panelist, Plug-and-Play panel, ReSpace/MAPLD. 2010.

Invited presentation on “SSM, How it works, and rapid application development”, ReSpace/MAPLD. 2010.