2014

# PROPOSED MIDDLEWARE SOLUTION FOR RESOURCE-CONSTRAINED DISTRIBUTED EMBEDDED NETWORKS

Jason T. Rexroat
*University of Kentucky*, jtrexr2@gmail.com

STUDENT AGREEMENT:

I represent that my thesis or dissertation and abstract are my original work. Proper attribution has been given to all outside sources. I understand that I am solely responsible for obtaining any needed copyright permissions. I have obtained needed written permission statement(s) from the owner(s) of each third-party copyrighted matter to be included in my work, allowing electronic distribution (if such use is not permitted by the fair use doctrine) which will be submitted to UKnowledge as Additional File.

I hereby grant to The University of Kentucky and its agents the irrevocable, non-exclusive, and royalty-free license to archive and make accessible my work in whole or in part in all forms of media, now or hereafter known. I agree that the document mentioned above may be made available immediately for worldwide access unless an embargo applies.

I retain all other ownership rights to the copyright of my work. I also retain the right to use in future works (such as articles or books) all or part of my work. I understand that I am free to register the copyright to my work.

REVIEW, APPROVAL AND ACCEPTANCE

The document mentioned above has been reviewed and accepted by the student's advisor, on behalf of the advisory committee, and by the Director of Graduate Studies (DGS), on behalf of the program; we verify that this is the final, approved version of the student's thesis including all changes required by the advisory committee. The undersigned agree to abide by the statements above.

Jason T. Rexroat, Student

Dr. James Lumpp Jr., Major Professor

Dr. Cai-Cheng Lu, Director of Graduate Studies

# PROPOSED MIDDLEWARE SOLUTION FOR RESOURCE-CONSTRAINED DISTRIBUTED EMBEDDED NETWORKS

---

THESIS

---

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science in
Electrical Engineering from the College of Engineering
at the University of Kentucky


By

Jason Timothy Rexroat

Lexington, Kentucky

Director: Dr. James Lumpp Jr, Professor of Electrical Engineering

Lexington, Kentucky


2014

ABSTRACT OF THESIS

PROPOSED MIDDLEWARE SOLUTION FOR RESOURCE-CONSTRAINTED DISTRIBUTED EMBEDDED NETWORKS

The explosion in processing power of embedded systems has enabled distributed embedded networks to perform more complicated tasks. Middleware are sets of encapsulations of common and network/operating system-specific functionality into generic, reusable frameworks to manage such distributed networks. This thesis will survey and categorize popular middleware implementations into three adapted layers: host-infrastructure, distribution, and common services. This thesis will then apply a quantitative approach to grading and proposing a single middleware solution from all layers for two target platforms: CubeSats and autonomous unmanned aerial vehicles (UAVs). CubeSats are 10x10x10cm nanosatellites that are popular university-level space missions, and impose power and volume constraints. Autonomous UAVs are similarly-popular hobbyist-level vehicles that exhibit similar power and volume constraints. The MAVLink middleware from the host-infrastructure layer is proposed as the middleware to manage the distributed embedded networks powering these platforms in future projects. Finally, this thesis presents a performance analysis on MAVLink managing the ARM Cortex-M 32-bit processors that power the target platforms.

KEYWORDS: Middleware, CubeSat, Distributed Computing, UAV, MAVLink

Jason T. Rexroat .

December 2, 2014 .

# PROPOSED MIDDLEWARE SOLUTION FOR RESOURCE-CONSTRAINED DISTRIBUTED EMBEDDED NETWORKS

By

Jason Timothy Rexroat

James E Lumpp, Jr, PhD .
Director of Thesis

Cai-Cheng Lu PhD .
Director of Graduate Studies

December 2, 2014 .

To my Parents: Curtis and Mary Jo Rexroat
To my Brother: Michael Rexroat
To my Wife: Brittany Rexroat

## Acknowledgements

I would like to acknowledge the invaluable support and advice of Dr. James Lumpp, who invited an untested freshman to build space hardware. This invitation changed the course of my life, and without him, I would not be the person I am today. I would also like to thank the other members of my thesis committee, Dr. Larry Holloway and Dr. Hank Deitz, for their insight and support.

I would also like to thank my colleagues from the Space Systems Laboratory. Under your tutelage, I grew from a kid who didn't know how to solder to a confident and competent engineer, and I consider our time spent together among the most enjoyable experiences of my life.

I would also like to thank my wife, Brittany Rexroat. I'll never forget you telling me to "just do it" on our walk to class; without your constant support and love, I would not have finished this thesis.

Finally I would like to thank my wonderful family. My parents have shown me nothing but love and support, and have been a never ending source of encouragement. My brother has been there for me through every season, and is my dearest friend. It gives me indescribable joy to see him pursue engineering.

## Table of Contents

## List of Tables

## List of Figures

# 1 Introduction

In this chapter, a target set of distributed embedded systems is defined, a taxonomy for the comparison of middleware for managing these distributed embedded systems is presented, and the proposed set of layers for classifying these middleware is defined. Finally, the problem statement for this thesis is defined.

## 1.1 Distributed Embedded Systems

Embedded systems are systems managed by embedded computational units with specialized functions. Mirroring the parallelization trend in home- and enterprise- computing, distributed embedded systems split tasks between multiple processing units for more efficient processing with less power consumption. This thesis will overview middleware techniques that manage such distributed embedded networks, and will adopt a classification hierarchy in order to categorize and compare these middleware. Finally, this thesis will suggest and will suggest a recommended middleware technique targeted toward application in two popular embedded systems: CubeSats and autonomous unmanned aerial vehicles (UAVs).

### 1.1.1 CubeSats

CubeSats are target systems for the middleware reviewed and recommended by this thesis. CubeSats are nanosatellite-class satellites that measure 10 cm x 10 cm x 10 cm and weigh 1.33kg per unit [1]. This form-factor creates a volume and cost savings that is ideal for university and small-scale research, with over 80 CubeSats launched to date and nearly 1,000 forecast in the next decade [2] [3]. As autonomous, intelligent systems, CubeSats contain many of the same systems as their larger satellite cousins, including a command and data handling (C&DH) system, an electrical power system (EPS), a communications system, and typically a science or technology demonstration payload.

As processing units become cheaper, more powerful, and less power-hungry, such small satellites can support missions of increasing complexity. Powerful microcontrollers can manage satellite systems, support complex scientific measurement, and account for the harsh space environment through advanced fault-tolerance schemes. Distributed networks of microcontrollers have been demonstrated on several CubeSat missions, parallelizing processing tasks and dividing the satellite management workload between multiple discrete processing units, further extending the capabilities of CubeSats [4] [5]. The middleware methods of managing such distributed embedded networks are the subject of this thesis.

### 1.1.2 Unmanned Aerial Vehicles

Autonomous UAVs, particularly quadcopters, are target systems for the middleware reviewed and recommended by this thesis. Stretching further back into history than man-made satellites, UAVs have recently experienced a resurgence thanks to the cell-phone industry. The miniaturization in form-factor and power consumption, along with the parallel explosion in processing power, of processing units and sensors for cell phones have translated to UAVs. Autonomous autopilots and complex vision systems are feasible on small-scale quadcopter UAVs, prompting a dramatic rise of hobbyist and research quadcopters. These systems, with their restricted power budgets and small form-factors, form a terrestrial analog to CubeSats and can be serviced by many of the same distributed embedded networks. The middleware methods of managing these quadcopter systems are also the subject of this thesis.

## 1.2 Survey Taxonomy

The goal of this thesis is to recommend a distributed computing middleware for low power distributed computing platforms; this middleware will be based upon a survey on the current state of distributed computing frameworks and middleware currently employed or theorized for the target systems.

There are four widely accepted categories that describe how middleware handles distributed interaction: transactional, message-oriented, procedural, and object/component [6]. These classifications are detailed in Chapter 2. Due to the goal of supporting a system with generic subsystem interfaces and plug-and-play capabilities in a resource-constrained and strict application environment, this thesis targets the object/component classification of middleware.

In order to organize the survey of distributed computing frameworks, an established taxonomy within the object/component classification will be adapted for classification and comparison of these frameworks. This taxonomy defines middleware as the encapsulation of common and network/operating system-specific functionality into generic, reusable frameworks for software modules (SMs) running on processing elements (PEs). Due to the range and scale of abstraction that different middleware provide, they can be separated and categorized into layers: host-infrastructure middleware, distribution middleware, common middleware services, and domain-specific middleware services [7]. The specifics of these layers are defined in Chapter 2, and are targeted toward distributed object computing (DOC) systems. A goal of this thesis is to refine and adapt the taxonomy used to describe DOC systems in order to create a middleware

taxonomy for low-power, distributed embedded systems, specifically those found in CubeSats and small-scale UAVs.

The taxonomy to be introduced adopts Schmidt's definition of middleware for DOC systems: middleware is the reusable set of abstractions and services that encapsulate lower level and error-prone functionality by providing a generic application programming interface (API) for end-SM developers [8]. This new taxonomy, while targeted toward CubeSats and UAVs as end systems, applies in general to low-power, distributed embedded systems. The end goal of this thesis is to score the middleware in each category of this taxonomy and choose the best middleware for the target distributed embedded systems. The choice of middleware is not based on the layer that the middleware is categorized in; rather, the layering exists to classify similar middleware and allow them to be compared.

### 1.2.1 Host-Infrastructure Layer Middleware

Chapter 3 surveys the lowest layer of object/component middleware: the host-infrastructure layer. Historically, this layer is immediately above the operating system and protocols for transferring data. However, in order to update this taxonomy to better classify middleware within the target system range and to account for more middleware complexity closer to the hardware level, this category has been extended and categorized into two levels: hardware and network.

#### 1.2.1.1 Host-Infrastructure (HI)-Hardware

The HI-hardware layer of host-infrastructure middleware includes the hardware protocols used to transfer bytes over physical media in a distributed system. Middleware at this layer is not concerned with the meaning or ordering of bytes to be transferred; rather, it provides encapsulated sending and receiving functions. This layer is akin to the physical and data link layers in the Open Systems Interconnection (OSI) network model [9]. The hardware layer blends these two network layers in that it encompasses protocols that transmit bits over physical media and provide some form of synchronization or error detection; however, it is concerned only with the transmission of bits from a physical hardware PE, rather than the meaning of the bits. Examples of middleware in this sub-layer are serial communications drivers, such as Inter-integrated Circuit ($I^2C$) or controller area network (CAN) drivers.

#### 1.2.1.2 Host Infrastructure (HI)-Network Middleware

The HI-network middleware layer is above the hardware layer; middleware in this layer actively routes messages. This is akin to the network layer in the OSI model [9], and uses the

functions provided by the HI-hardware layer to transmit streams of bits called packets to the desired recipient(s). While the OSI model network layer assumes some form of addressing to route packets, the HI-network layer encompasses middleware that both uses literal addressing in the packet and addressing on the client side of a broadcast/multicast topology. Examples of middleware in this sub-layer are Micro-Aerial Vehicle-Link (MAVLink) and Space Plug-and-play Avionics-1 Lite (SPA-1L).

### 1.2.2 Distribution Layer Middleware

Chapter 4 surveys the layer immediately above host-infrastructure middleware in object/component middleware: the distribution layer. Middleware in this layer is responsible for extending the encapsulations provided by the host-infrastructure layer. This layer allows for standalone applications that harness the networking APIs that mask object locations, addresses, hardware, etc. While the host infrastructure layer is not concerned with the meaning of bytes, distribution layer middleware is.

#### 1.2.2.1 Distribution (D)-Transport Middleware

The D-transport layer of distribution middleware extends the encapsulations provided by the host infrastructure layer to remove node location and implementation dependence, and provides fault-tolerance and message transportation functionality not found in the host infrastructure layer. The D-transport layer includes middleware that actively routes messages transparently to end SMs, but does not follow the publish/subscribe model.

#### 1.2.2.2 Presentation, Encoding, Protocol, transport (PEPt) Middleware

Presentation, Encoding, Protocol and transport (PEPt) is a framework that describes service-oriented architectures (SOAs), which can specifically be remote procedure call (RPC) and object/component middleware. SOAs offer abstracted descriptions of applications and components, adopting a direct object-oriented model that hides the programming models (presentation), encodings of data, protocols used to frame messages, and the transport mechanisms to deliver/route the frames [10]. The middleware reviewed in this layer follows the publish/subscribe model and represents applications and components on the distributed network as services with object-oriented syntax. Examples of such middleware include Common Object Request Broker Architecture (CORBA) and micro-object request broker (uORB).

### 1.2.3 Common Services Layer Middleware

Chapter 5 surveys middleware in the layer immediately above distribution layer middleware: the common services layer. Middleware in this layer extend the APIs present in

distribution and host-infrastructure layers to provide reusable components that are common in the computational environment. These can include security, threading, transactions, and logging, as well as many more. This allows application developers to focus more on the logic of their specific SM, instead of needing to write these common and reused components. Examples of middleware in this layer include Spacecraft Onboard Interface Services (SOIS) and Core Flight System (CFS).

Each of the above layers contains middleware that must address five key requirements of middleware. These requirements are network communication, which defines how the middleware manages different hosts communicating with each other and defines node-oriented versus message-oriented messaging as a comparison metric; coordination, which defines how the middleware manages synchronizes communicating PEs and defines synchronous or asynchronous communications as a comparison metric; reliability, which defines what guarantees the middleware makes about the integrity of inter-PE communication and uses at-most-once, at-least-once, and exactly-once as comparison metrics; scalability, which defines the extent to which the middleware can accommodate the addition or subtraction of hosts and defines transparency levels as a comparison metric; and heterogeneity, which defines the differences in architectures, programming languages, operating systems, and network mechanics that the middleware can handle between PEs, and defines hardware heterogeneity, network heterogeneity, and software heterogeneity as comparison metrics.

## 1.3 Problem Statement

This thesis categorizes a set of middleware that are candidates for or already in use to manage distributed embedded networks on the target platforms of CubeSats and UAVs. These middleware approaches are each detailed for functionality and analyzed for how they address the set of middleware requirements: network communication, coordination, reliability, scalability, and heterogeneity. Based upon this analysis and comparison, MAVLink from the HI-Network layer is proposed for managing future CubeSat and UAV projects. Through a performance analysis testing the throughput, latency, and central processing unit (CPU) cycle usage on a demonstration Advanced RISC Machines (ARM) Cortex-M microcontroller, it will be shown that MAVLink offers the performance and handling of the above set of middleware requirements for these target platforms.

# 2 Background

This chapter discusses the history of the UAV and CubeSat target embedded systems, as well as the current trends being experienced in the processing units that support them. This chapter also introduces and defines the terminology used to characterize the middleware reviewed in later sections, as well as the classifications of such middleware. Finally, a history of the Space Systems Laboratory is given, where much of this work originated.

## 2.1 Unmanned Aerial Vehicles

UAVs have a history beginning well before the demonstration of the first functional piloted aircraft. One of the earliest recorded uses of UAVs was on 22 August, 1849, when an Austrian army besieging Venice launched balloons against the city defenders. Despite conflicting reports as to whether there were two or 200 such balloons, whether the balloons dropped bombs or exploded in shrapnel, and whether the devices were timed or actuated via a trailing copper wire to the ground, the fact remains that while no great material damage was done, pilotless aerial platforms had made their debut [11] [12]. Similar unmanned platform patents followed in 1862 [13] and 1863 [14], though no apparent demonstrations or constructions of these patents exist. In 1898, Nikola Tesla demonstrated wireless control of a vehicle at an exhibition in Madison Square Garden in New York City, using radio signals to guide and flash lights aboard a small iron boat. Tesla foresaw the practical application of such a capability, envisioning “mechanical men which will do the laborious work of the human race” [15] [16].

The 20$^{th}$ century, often dubbed the “Age of Flight”, saw the glamorous and dramatic rise of piloted aircraft, beloved by militaries, stuntmen, and world travelers alike. UAVs have seen a similarly dramatic yet less glamorous rise. The United States Navy experimented with UAVs during World War I by developing a flying torpedo, conducting over 100 tests showing range and radio control. The Army followed suit and was assisted by Orville Wright in building the Kettering Bug, an unmanned flying bomb. Despite only eight successful flights out of 36, a total of 25 Bugs were ordered. The war ended, however, before either of these systems could be further improved and deployed [17]. The British Royal Navy invested in both pilotless and radio-controlled (RC) technology to develop the Queen Bee in the 1930s [18]. The Queen Bee was a reusable RC aircraft used for aerial target practice for naval pilots; similar land versions were developed for target practice for antiaircraft gunners. On the eve of World War II, the U.S. Navy routinely used pilotless drones as target practice for naval warship gunners, proving effective in training operations and unmasking air defense weaknesses [17]. These weaknesses led to greater emphasis on developing wartime attack drones, culminating in converted (obsolete) Devastator

torpedo bombers, controlled by following aircraft, sinking a beached Japanese merchantman in July 1944 [17]. Development on such systems stalled, however, as Navy leadership canceled these programs as the tide of the Pacific War turned and enough piloted aircraft and aircraft carriers were available.

Further developments to present-day UAVs followed two paths: military and hobbyist. The Cold War saw UAVs mature into reliable pilotless reconnaissance platforms, due to political and military fallout from shot-down piloted U-2 aircraft over the Soviet Union and during the Cuban Missile Crisis; the venerable Teledyne-Ryan AQM-34 drones resulted from these pre-Vietnam events. During the Vietnam War, over 3000 UAV operations were flown, leading to further expansion of offensive capabilities. These capabilities were realized during the 1973 Yom Kippur War and the 1990-1991 Persian Gulf War, with further UAV-borne offensive operations in Bosnia, Haiti, Somalia, and in the Second Persian Gulf war [17].

Hobbyist UAV development started with personal RC aircraft, historically perceived as toys. The 2000's, however, marked the beginning of the "personal drone movement". In large part, this movement has been driven by the mobile computing industry, spearheaded by smartphones. These devices see increasing computing power, more precise Global Positioning System (GPS) units, and more powerful environmental sensors, all under tight power and physical volume constraints; the microcontrollers, sensors, and antennas that make such trends possible have been ideally suited to similarly increase the capabilities of UAVs, particularly in autonomous flight. For autonomous flight control, UAVs use autopilots, which are systems that utilize knowledge about the vehicle's environment, capabilities, and preprogrammed goals to control the vehicle without intervention from a human. These range in complexity, from the first gyroscope-stabilized flights in 1913 [17] to autonomous passenger aircraft utilizing hundreds of sensors. While there are many commercial autopilots available, these can be prohibitively expensive for hobbyists (the Piccolo autopilot, for example, costs over $1,000). Utilizing the high-performance, low-power, small form-factor microprocessors, GPS chips, and environmental sensors running today's smartphones, online communities and companies sprung up to service the hobbyist community, all at much lower prices. One such company, 3D Robotics, estimates that it alone has shipped over 10,000 autopilots and assorted drone components, totaling more than the entire U.S. military operates [19]. There are many open-source autopilots available, including ArduPilotMega, pxIMU Autopilot, Santa Cruz Low-cost UAV Guidance Navigation Control (GNC) System (SLUGS) Autopilot, SmartAP Autopilot, and AutoQuad 6. These autopilots service a range of platforms, from familiar fixed-wing aircraft to tri-, quad-, and hexa-copters.

The Federal Aviation Administration (FAA) has recognized the rapid rise of hobbyist UAVs, incorporating control and provision for such platforms into regulations. Current regulations specify that hobbyist UAVs always fly within line-of-sight of the operator at launch, maintain altitudes below 400 feet, maintain a five-mile distance away from airports, and avoid commercial uses; however, many practical applications of these regulations fall into gray areas [20]. The FAA plans to further revise these and other regulations by 2015, allowing expanded use and further integration of government, commercial, and hobbyist UAVs in the United States [21]. Online shopping giants such as Amazon are investing in autonomous UAV technology in preparation for the relaxing of regulations, proposing Amazon Prime Air to deliver five pound packages to consumers in a matter of minutes using quad- and hexa-copter platforms [22]. Google acquired Titan Aerospace, maker of jet-sized, solar-powered UAVs built to fly for years, for Earth-imaging and Internet-delivery [23]. Facebook's Connectivity Lab was created to build 747-sized drones that deliver internet to the billions still without [24]. As mentioned, such systems require advanced and complex computing platforms.

### 2.2 CubeSats

Despite academic descriptions of geosynchronous satellites by German physicists in the 1920's [25] [26], Arthur C. Clarke first popularized the concept of telecommunications satellites in his 1945 paper, "Extra-Terrestrial Relays: Can Rocket Stations Give World-wide Radio Coverage?" In this paper, Clarke suggests stations in orbit that exhibit an orbital period of exactly 24 hours, servicing a very large area for radio and television signals, and requiring only three such stations for global coverage [27]. Clarke's early drawing is shown in Figure 1.

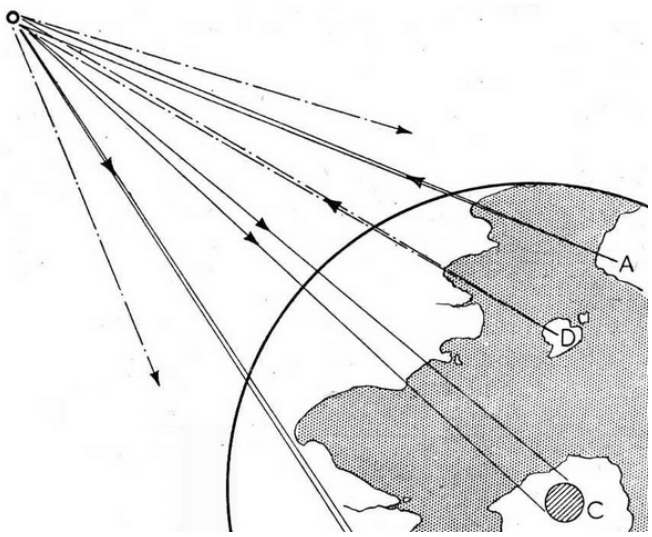


**Figure 1: Clarke's illustration of a telecommunications satellite [27]**

With the souring of U.S.-Soviet relations at the end of World War II and the beginning of the atomic age, the launch of Sputnik-1 in 1957 began the so-called “space race”, ushering in a period of rapid development and launch of a variety of space systems. The year 1958 saw the launch of six satellites; in 1962, 72 satellites were launched, and over 7,000 have been launched since then [28]. Those satellites have varied widely in function and size, and fulfill roles from communications, navigation, and remote sensing to scientific research, weather, and education. The first man orbited the Earth in 1961, men walked on the moon in 1969, and in 2014 an orbiting space station holds six humans and has been continuously occupied for fourteen years [29]. Access to space, despite the number of space-capable launch vehicles and volume of launches per year, is still exorbitantly expensive. Currently estimated as costing around $10,000 per pound to get to orbit for a launch vehicle, access to space has historically been reserved for and granted by governments with the resources to fund such ventures [30]. While progress is being made to lower this cost, both by private companies – SpaceX claims the Falcon Heavy will drive the cost down to under $1000 per pound [31] – and through NASA’s Advanced Space Transportation Program, which targets next-generation technologies to lower the cost to under $100 per pound in the coming decades [32], launch access to space is still a high-cost activity. Building the actual payloads is also expensive. A 2008 NASA study of historical missions categorized these costs by mission type: through 33 surveyed unscrewed Earth orbit satellites, the average cost was $100 million per mission; through 16 surveyed unscrewed planetary satellites, the average cost was $370 million per mission; and through nine crewed missions, the average cost was $4.6 billion per mission [33]. The cost in terms of money is high, but the cost in terms of

time is also high, with an early-2000's average of 30 months required to construct a commercial satellite, with the time typically longer for NASA missions [34].

While large commercial spacecraft and launches present challenges to cash-strapped groups wanting cheaper access to space, satellite miniaturization has somewhat alleviated the above obstacles. Several classes of small satellites exist, categorized by their mass ranges: small satellites describe the 100-500kg range, microsatellites describe the 10-100kg range, nanosatellites describe the 1-10kg range, picosatellites describe the 100g-1kg range, and femtosatellites describe the 10-100g range [3].

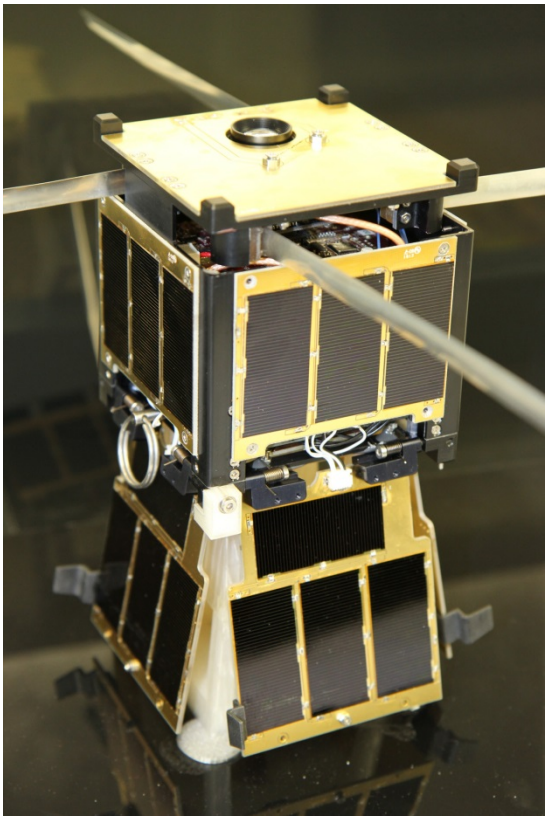

**Figure 2: KySat-2, a CubeSat launched in November 2013 [5]**

Various standards exist for each class, but perhaps the most visible class over the last decade is the nanosatellite, particularly the CubeSat. Developed by Dr. Jordi Puig-Sauri at California Polytechnic State University (Cal Poly) at San Luis Obispo and Professor Bob Twiggs at Stanford University in 1999, the CubeSat is a 1.33 kg, 10 cm cube. This 10 cm cube is the one-unit ("1U") version, but can be extended to 2U, 3U, etc. for more volume. KySat-2, pictured in Figure 2 and built by University of Kentucky and Morehead State University students, is a 1U stowed and 2U deployed CubeSat.

In addition to the CubeSat concept and specification, Dr. Puig-Sauri and Professor Twiggs also created the Poly Picosatellite Orbital Deployer (P-POD), which is a CubeSat deployment system. While the CubeSat addresses the issues of the money- and time-cost of satellite construction, the P-POD attempts to address the issue of launch cost per pound by serving as an add-on interface to existing launch vehicles. The standard P-POD holds 3U-worth of CubeSats, and operates as a "jack-in-the-box", with a spring back plate forcing the satellites out into orbit once the door opens. By adhering to the P-POD and CubeSat design specifications and requirements, CubeSats can safely "piggyback" on existing flights to space, dramatically reducing the money-cost of getting satellites to space. Furthermore, NASA's Educational Launch of Nanosatellites (ELaNa) program partners with universities to provide the cost of the actual launch, leaving only the construction and environmental testing costs for the satellite builders. The CubeSat standard has enjoyed worldwide success in getting to space, with nearly 100 launched between 2003 and 2012; 2013 alone was a banner year for CubeSats, with over 80 launched [2]; market projections indicate that over 2,000 nanosatellite- and microsatellite-class spacecraft will need launch opportunities between 2014 and 2020 [3].

With the popularity of CubeSats, the technology powering them has also advanced. CubeSats are comprised of many of the same subsystems as their larger cousin spacecraft, typically including: a radio communications system, an EPS, an attitude determination and control system (ADCS), a science or technology payload, and a C&DH. While each of these subsystems have undergone significant maturation and are the subjects of extensive academic research and commercial development, the C&DH will be studied in more detail. The mobile computing industry has had a significant impact on the computational capabilities of satellite C&DHs, in fact much the same as with UAVs. The miniaturization of sensor and processing components, along with the reduction in power requirements despite an upward trend in computational power, have both made the C&DH capable of enhancing CubeSats far beyond their Sputnik-like origins. The ELaNa IV launch in November 2013 from Wallops Flight Facility carried 11 university- and high school-constructed satellites; these satellites performed a variety of complex missions, ranging from technology demonstration (stellar gyroscope, pyramidal control moment gyroscopes, open-source satellite bus architectures, and Android-powered spacecraft) and educational outreach to space science (radiation dissipation during auroras and infrared Earth imaging) [35].

## 2.3 Distributed Processing

From the abacus and Napier's bones to modern billion-transistor processors, the power of computing hardware has experienced an exponential growth, particularly during the last 60 years. Charles Babbage's Turing-complete Analytical Engine was designed in 1834 [36], table-sized punched card machines processed Social Security records in the 1930's [37], and exponentially miniaturizing transistor sizes and power consumption have enabled powerful computational units to proliferate every modern industry.

### 2.3.1 8051

In 1980, Intel released the venerable MCS-51 "8051" microcontroller series. A microcontroller is a similar computational unit to a microprocessor, but contains the processor, memory, and input/output peripherals on the chip for standalone use in embedded environments. The 8-bit 8051 implements a Harvard architecture, meaning the instruction and data memories are independent, and is typically implemented with universal asynchronous receiver/transmitter (UART), $I^2C$, serial peripheral interface (SPI), and other peripheral modules. The 8051 is a popular microcontroller that sees adoption in many industries, including aerospace, automotive, home appliances, and even the music industry due to its small size, low power consumption, and standardized architecture [38]. Though Intel no longer produces the 8051, many companies still develop and sell 8051-architecture chips, including Atmel, NXP, Silicon Labs, and Texas Instruments. The 8051 is a target architecture for the distributed embedded middleware recommended by this thesis.

### 2.3.2 ARM

Until the late 2000's, 8-bit microcontroller families such as the 8051 filled the low-power embedded niche. However, the Advanced Reduced Instruction Set Computer (RISC) Machine (ARM) core has achieved low enough power consumption to begin to fill this niche, with over 98 percent of smartphones sold per year containing at least one ARM-core processor [39]. Similar to the rise in intelligent hobbyist UAVs, the smartphone revolution has driven the power consumption and processing power of ARM core processors, specifically the ARM Cortex-M family, to the point where they can be integrated into low power systems [40]. The ARM Cortex-M is a target architecture for the distributed embedded middleware recommended by this thesis.

## 2.4 Distributed Middleware

### 2.4.1 Terminology

In order to present the survey and classification of middleware for distributed, low-power embedded systems, the terminology must be reviewed. Embedded systems are standalone computers with very specific functions, and are often integrated into larger systems. These larger systems are growing to encompass nearly every industry, and notably include automotive systems, such as braking systems and engine control; consumer appliances, such as microwaves, refrigerators, and washer/dryers; and aerospace, including flight avionics [41]. This thesis specifically surveys the middleware available to flight avionics.

#### 2.4.1.1 Distributed Embedded Systems

The first major distinction involves that between distributed and centralized embedded systems. Embedded systems are essentially standalone computers with very specific functions, and they are often integrated into larger systems. Centralized embedded systems integrate all processing functionality onto a single processing element. An advantage of this is a shared memory space for all functionality, requiring no lossy network communication between processes or services; however, a centralized architecture requires more resources from the processing element, necessitating more processing power and power consumption of the processing element. Distributed embedded systems, however, split processing functionality onto multiple discrete processing elements. The advantage is that less processing power and power consumption are required from any single processing element, and individual processing elements can be less complex. However, a distributed embedded network requires an additional layer of physical communications, formed over lossy connections and introducing latency between communicating processes. This thesis examines middleware intended for distributed embedded systems, and ignores centralized architectures.

#### 2.4.1.2 Fault-Tolerance

Within distributed embedded systems, another major distinction is in the system's level of fault-tolerance, which is a system's behavior in response to a fault. Two general terms enumerate the state of a system: safe, which means the system preserves state and system data, and causes no harm to itself or environment; and live, which means the system is running and is not in a stopped or shutdown state. Using these definitions for a system, a system can be in one of four possible fault-tolerant states: fault masking, where the system preserves system liveness and safety and is most desirable; fail safe, where the system preserves system safety at the cost of

liveness; non-masking, where the system preserves liveness at the cost of safety, and none, where the system guarantees neither system liveness nor safety, and is least desirable [42]. Table 1 summarizes these states.

**Table 1: Matrix of System States**

| | Live | Not live |
|---|---|---|
| **Safe** | Fault Masking | Fail Safe |
| **Not safe** | Non-masking | None |

Two phases are required to handle faults: detection and correction. Fault detection begins at the system design level, where predictable faults are grouped into fault classes that may be handled differently [43]. Once fault classes are created, there are four broad categories of fault detection once the system is deployed: n-version redundancy and voting, state estimation and monitoring, system feedback monitoring, and software wrapping and monitoring. Fault detection essentially addresses the system safety aspect of fault tolerance. Once the fault has been detected, the most essential fault correction method is redundancy, the forms of which can be categorized into three approaches: n-version redundancy with voting, redundant estimation, and redundant resource allocation. Fault correction essentially addresses the system liveness aspect of fault tolerance [42] [44]. All middleware surveyed in this thesis incorporate some level of fault-tolerance; those that do not are not considered.

#### 2.4.1.3 Real-Time Embedded Systems

Within fault-tolerant, distributed embedded systems, a further distinction can be made about the timeliness required of the system. A real-time embedded system is one that must meet timing requirements, else severe consequences for the system will result. These real-time systems can be further subdivided into two categories: hard real-time, where specific timing requirements absolutely must be met, and soft real-time, where the tasks running just need to be performed as quickly as possible [41]. Hard real-time embedded systems typically require a real-time operating system (RTOS), which is outside the scope of this thesis; the middleware surveyed in this thesis call all be described as soft real-time.

#### 2.4.1.4 ISO/OSI Network Stack

This thesis is studying distributed embedded system middleware, and the classifications will be described by their OSI network layer equivalents. The purpose for the OSI model is to

provide a standardized layering where each layer successively encapsulates lower layers while contributing its own value. Each layer is independent of the layer above and below, and scales appropriately with the complexity of the host system [45]. The OSI model is classically divided into seven layers: physical, data link, network, transport, session, presentation, and SM. While popularly restricted to networking for personal computers, the OSI model also describes network interactions between the components of distributed embedded systems. A summary of these layers is pictured in Table 2, and the roles of each layer are detailed below [45] [9].

**Table 2: Summary of OSI Model Layers [45] [9]**

| Layer | Data | Function |
|---|---|---|
| **SM** | Data | SM interface for networking |
| **Presentation** | Data | Data representation |
| **Session** | Data | Host-to-host connection |
| **Transport** | Segment | End-to-end data transportation |
| **Network** | Packet | Routing and logical addressing |
| **Data Link** | Frame | Physical addressing and error detection |
| **Physical** | Bit | Signal transmission and hardware protocol |

#### 2.4.1.4.1 Physical

The physical layer is the lowest layer in the OSI model, and is the base layer for distributed SMs. This layer encompasses the electrical protocols and specifications used to wire the components of the distributed system together. This can include Institute of Electrical and Electronics Engineers (IEEE) 802.11 (wireless), Universal Serial Bus (USB), Bluetooth, and RS-232 serial. This layer is by nature an unreliable physical link.

#### 2.4.1.4.2 Data Link

The data link layer is above the physical layer, and implements a reliable link between physically connected components. This reliability is achieved through error detection and correction and synchronization between components. An example of a program in this layer is the Point-to-Point Protocol (PPP) that splits packets from higher layers into frames for transmission onto the Internet.

#### 2.4.1.4.3 Network

The network layer is above the data link layer, and implements the routing of packets for the distributed system. Routing is accomplished via addressing, where each node on the same network has a unique address. This layer does not guarantee reliability, however; packets may be dropped or reordered. An example of a program in this layer is the Internet Protocol (IP), which routes packets based on IP address.

#### 2.4.1.4.4 Transport

The transport layer is above the network layer, and provides for the end-to-end transportation of data from SMs. The purpose of this layer is to provide an encapsulation of all network transportation to higher layers. An example of a program in this layer is Transmission Control Protocol (TCP), which provides a guarantee of packet delivery by establishing a connection between distributed components and retransmitting any dropped or corrupted packets.

#### 2.4.1.4.5 Session

The session layer is above the transport layer, and provides connection setup and closing between distributed components. This layer binds the transportation of data between distributed components into a logical relationship. An example of a program in this layer is Net-Basic Input/Output System (BIOS), which establishes connections and provides an API for exchanging data between connected systems.

#### 2.4.1.4.6 Presentation

The presentation layer is above the session layer, and provides translation between application formats and the network format required to transport data. This layer both formats data from applications on the transmitting node to be sent over the network, and translates received data to be consumed by applications on the receiving node. An example of a program in this layer is Multipurpose Internal Mail Extensions (MIME), which is used to format hypertext-transfer protocol (HTTP) into its required email-like format for transmission.

#### 2.4.1.4.7 Application

The application layer is above the presentation layer, and is the final layer that is directly called by applications in systems implementing the full OSI model. An example of a program in this layer is Network File System (NFS), which implements a distributed file system across a network.

Not all distributed components implement all layers of the OSI model, nor are they required to. Some systems, particularly distributed embedded systems, are less complex than general-purpose computers, such as personal computers (PCs). These less complex systems, with fewer computational resources and applications that are closer to the system's hardware layer, typically only implement the physical, data link, network, and transport layers.

#### 2.4.1.5 Middleware

Middleware is a software layer between applications and an underlying network that provides generic abstractions and services applications [6]. The motivations for such middleware are several fold: they provide layers of abstraction between application developers and low-level details that are often tedious and prone to errors; they reduce development time by providing previously-tested and reusable code; and the abstractions they provide can mimic network- and object-oriented strategies that are closer to application-level programming [46]. This thesis will explore a host of middleware implementations, and a common lexicon is needed for comparisons between them. Since this thesis is restricted to middleware for distributed systems, software will be running on multiple physical computational units. Adopting the naming convention of the D-Transport middleware Automatically Reconfigurable Distributed Embedded Architecture (Ardea) [47], a middleware that is investigated in Chapter 4, these physical computational units will be henceforth referred to as processing elements (PEs); the software running on them will be henceforth referred to as software modules (SMs). Multiple SMs can run on one PE.

While middleware can be simply defined as an abstraction layer between the tedious details of a distributed network and SMs, Emmerich defines a set of five requirements that middleware must in some way address. These requirements allow for middleware to be classified and evaluated, and are: network communication, coordination, reliability, scalability, and heterogeneity [6].

A distributed network is a set of PEs with some combination of SMs running on them. There are two kinds of architectures that govern when messages will be exchanged between these PEs: event-triggered architecture and time-triggered architecture.

#### 2.4.1.6 Event-Triggered Architecture

An event-triggered architecture (ETP) describes a distributed network where messages are only generated and exchanged between PEs when they are needed. Messages are based on events, and the network is idle if no event requiring such transactions is occurring. There are several strengths and weaknesses of ETP. ETP allows for more dynamic topologies by only

requiring connection to the physical bus, instead of complex and predetermined messaging schedules and algorithms. Additionally, ETP may be more efficient in certain scenarios, such as systems where messaging is sparse or where the data exchanged is large. However, ETP relies on events to trigger communication. The occurrence of multiple events simultaneous or during another event-induced transaction could cause bus contention, potentially starving PEs or rendering the data stale. The failure of any PE disables whatever data exchange that PE normally initiates. Also, message latency is not constant since there is no temporal limit or restriction to the occurrence of events. An example of an ETP is Ethernet, where packets are only exchanged when a PE wants to supply or request data from another PE [48].

#### 2.4.1.7 Time-Triggered Architecture

A time-triggered architecture (TTP) describes a distributed network where each PE can only transmit data during a predetermined, specified time interval. This time interval is based on a global time base, and each PE is allocated a finite slot during which it can transmit or request information. Responding PEs would then use their allocated slots to respond. Each PE is given its slot, and the process is repeated, yielding a predictable, periodic communication time for each PE. The global time base can either be sourced from a master PE, providing a clock synchronization message to each PE, or a combination of PE clock sources to form a “masterless” network, where the failure of any single PE doesn’t destroy the global time base.

There are several strengths and weaknesses of TTP. TTP offer constant and known message latency, since each PE is given a predetermined period of time to transmit. TTP also offers known and optimizable bus loading, since the periodicity and sequencing of messages from PEs is precisely known. Finally, there is no bus contention, since each PE can only transmit during its specified interval. This helps ensure hard real-time compliance of the network. However, TTP require heavy upfront design and a static network, allowing the addition of no new PEs without changing the messaging schedule. With TTP, large data needs to be segmented into chunks that are transmitted when the sending PE’s time arrives, inducing latency and delaying delivery of the file. This could be unacceptable for the system, such as one that delivers video. Likewise, PEs with no new data leads to wasted slots, introducing unnecessary latency for other PEs needing to transmit [48].

#### 2.4.1.8 Network Communication

Distributed middleware must facilitate components on different hosts communicating with each other. Classically this relies on the ISO/OSI reference model, where the physical layer,

data link layer, network layer, and transport layer are all handled by a network operating system and the session and presentation layers are handled by the middleware. However, the lower power consumption and computational power of embedded systems are bringing middleware closer to the hardware; this thesis allows the network communication requirement to extend down to the data link layer. In satisfaction of this requirement, the network communication of middleware will be classified as node-oriented, where each node in the network is addressed uniquely, or message-oriented, where the messages themselves are addressed and nodes on the network must know whether to process the message.

#### 2.4.1.9 Coordination

Distributed middleware must be cognizant of the many points of control in a distributed system. With different hosts responding to requests and running a heterogeneous mixture of components, some form of synchronization is required. This can include blocking while waiting for a requested service to execute, polling while waiting for executing services to complete, or server-client asynchronous requests. Additionally, the coordination requirement implies that middleware must reflect the necessities of a large number of hosts. At any given point, different numbers and combinations of hosts may disappear from the network. In satisfaction of this requirement, the coordination of middleware will be classified as synchronous, where transmitting nodes must wait for receiving nodes to acknowledge the message, or asynchronous, where the transmitting node sends the message and continues execution without waiting for the receiver.

#### 2.4.1.10 Reliability

Middleware used in safety-critical SMs must offer some level of reliability in communication between distributed PEs; and middleware in non-safety critical SMs should offer a level of reliability. This reliability is measured in terms of successful message delivery and message duplication, and can be categorized as: at-most-once, at-least-once, and exactly-once. At-most-once means that the message will not be duplicated, but still may not be successfully delivered. At-least-once means that the message will be successfully delivered, but will possibly be duplicated. Finally, exactly-once is a combination of the previous two, meaning that the message will be successfully delivered and will not be duplicated [6] [49]. These results are summarized in Table 3.

**Table 3: Comparison of reliability measures**

| | Delivery | Duplication |
|---|---|---|
| **At-most-once** | Not guaranteed | Guaranteed |
| **At-least-once** | Guaranteed | Not guaranteed |
| **Exactly-once** | Guaranteed | Guaranteed |

#### 2.4.1.11 Scalability

Middleware for distributed systems must be able to accommodate the addition or subtraction of hosts without changing the architecture or code. In order to accomplish this, various levels of transparency must be provided by the middleware. These transparency levels are defined by the ISO Open Distributed Processing (ODP) Reference Model; only the subset defined by Emmerich [6] will be used here; these are summarized in Table 4. Access transparency means that SMs have no knowledge whether services it uses are local or remote; location transparency means that SMs have no knowledge of the physical location of other services; migration transparency means that services or components can be transferred between hosts for load balancing or fault tolerance with no knowledge to SMs; and replication transparency means that multiple copies of a service can exist on many hosts, again for load balancing or fault tolerance, with no knowledge to the SM. It is these transparency services that middleware should provide.

**Table 4: Comparison of scalability measures**

| Transparency | |
|---|---|
| **Access** | Components do not know if services are local or remote |
| **Location** | Components do not know physical location of services |
| **Migration** | Components do not know if service has been migrated |
| **Replication** | Components do not know which redundant copy service is using |

#### 2.4.1.12 Heterogeneity

Distributed systems are composed of multiple discrete processing elements, and these processing elements are not necessarily homogeneous. Distributed middleware should handle differences in programming languages, operating systems, and hardware implementations between hosts in a distributed network. This can also include the different ways that heterogeneous hosts encode data: Unicode vs. American Standard Code for Information

Interchange (ASCII) vs. Extended Binary Coded Decimal Interchange Code (EBCDIC), 8-bit vs. 16-bit numbers, and little-endian vs. big-endian representation. Such heterogeneity is measured by what levels of abstraction of middleware are required at each PE: hardware heterogeneity, where the instruction set and data representation architectures can differ between PEs; network heterogeneity, where transmission media, signaling, and protocols can differ between PEs; and software heterogeneity, where the operating systems, programming languages, and SMs can differ between PEs . These are summarized in Table 5.

**Table 5: Comparison of heterogeneity measures**

| Heterogeneity | |
|---|---|
| **Hardware** | Different computer architectures |
| **Network** | Different signaling and protocols |
| **Software** | Different operating systems, programming languages, and SMs |

### 2.4.2 Classifications

Middleware has been in the computing lexicon since the 1980's [8], and has grown to support a wide range of distributed systems with varying degrees of complexity and deployment. Two early reasons for the packaging of explicit middleware stand out: first, the consolidation of the information technology (IT) industry and merging of companies brought together disparate systems and services that would have required too much time and effort to build from the ground up. As a result, these services were integrated and combined using middleware to deliver IT computing services as quickly as possible to customers. Secondly, the exponential growth of the Internet in the 1990's and 2000's made scalability a requirement for web services to survive; utilizing distributed systems and the middleware that knit them together allowed for websites and service providers to keep pace with growing demand. With the minimization of embedded systems in both size and power requirements, the use of distributed computing has continued to drive middleware development and deployment. In the time since these early days of middleware, distributed systems have grown into ubiquitous elements of technological infrastructure.

The distributed methodology, of dividing the typical resources available in a central computer (memory, processing power, power consumption, etc.), offers many advantages over a centralized methodology. Distributed systems can integrate legacy devices and components, can incorporate component and service redundancy and fault tolerance, and are more scalable than centralized systems [6]. This classification restricts middleware to distributed systems.

Middleware encompasses the software services and “plumbing” that powers distributed systems. After several decades of formal middleware creation and consolidation, the methodologies used to create middleware can be classified. Emmerich’s [6] widely accepted [46] [8] classification scheme first explores five middleware requirements, because the way different middleware schemes handle these requirements allow them to be classified: network communication, coordination, reliability, scalability, and heterogeneity. Using the methods that middleware use to address these requirements, distributed middleware can be classified into four different groups: transactional, message-oriented, procedural, and object/component.

#### 2.4.2.1 Transactional

Transactional middleware connects distributed hosts using a two-phase commit protocol. In this middleware architecture, hosts are coordinators, participants, or non-participants. The coordinator is typically the host process that needs to communicate with a distributed resource manager and coordinates the transaction. This transaction is shown in Figure 3. Participants are the resource managers that host resources desired by the coordinator. Non-participants are resource managers not participating in the transaction. The coordinator queries all desired participants through a prepare message. The participants reply to the prepare message, voting either to abort the transaction or commit their requested resources. If the participants all vote to commit, the coordinator issues a commit message and the transaction proceeds [50] [51].

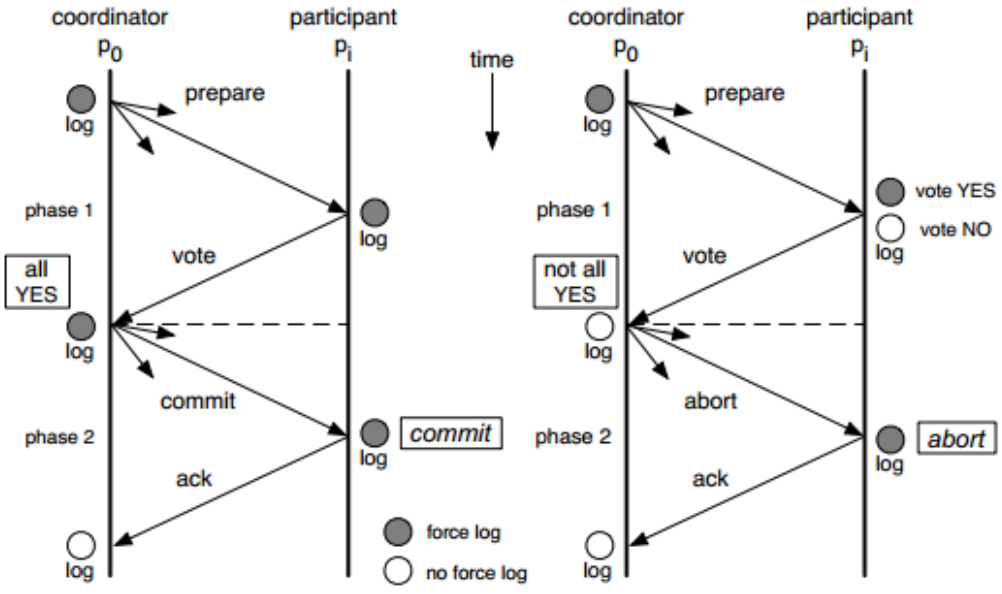


**Figure 3: Basic Two-phase Commit Protocol [51]**

This process is based on hundreds-year-old contract law, where transactions between parties obey three properties: consistency, where the transaction follows an established protocol; atomicity, where the transaction binds either all participants or none; and durability, where commitment to the transaction is final and cannot be violated [52]. The Open Group has adopted the Distributed Transaction Processing (DTP) model and XA specification to incorporate an implementation of this two-phase commit protocol. An example implementation of this middleware architecture is International Business Machine Corporation's (IBM) Customer Information Control System (CICS) family of online transaction management servers and clients.

#### 2.4.2.2 Message-Oriented

Message-oriented middleware (MOM) uses message exchange to connect distributed hosts. A client sends a message to a message queue, which is a temporary, persistent storage location for messages. Server components check the message queue and retrieve any pertinent messages, execute the request, then place messages back in the queue for the client. This messaging is asynchronous, meaning the client continues execution of other tasks after the message is given to the middleware. Additionally, if the server component or client experience a failure, the messages remains in the queue and can be retrieve once the component has restarted. The advantage of this architecture is that the distributed hosts can be running at different times and speeds; messages are placed in the queues and retrieved from the queues, requiring no logical link or synchronization between hosts, further separating SMs from the distributed network. An example implementation of this architecture is IBM's MQSeries product line, which implements a MOM architecture and provides an API for SMs to utilize it in distributed systems [53].

#### 2.4.2.3 Procedural

Procedural middleware relies on remote procedure calls (RPCs) to connect distributed hosts. Each host has a set of available programs that are visible as server components to clients. A client "calls" the procedure by passing the call to the middleware, which marshals the call into a network message using an automatically-generated client stub, and transmits the message. The server unmarshals the message, using an automatically-generated server stub, and executes the program. Acknowledgement and any other data transfer is accomplished via similar marshalling/unmarshalling of messages.

The key advantage of this style of middleware is in the interface definitions, which are by necessity explicitly defined for every available RPC. However, these systems are not scalable, as newly available RPCs still need to be programmed or handled by SM developers, as they are not

handled by the middleware. RPCs are available on Microsoft Windows and Unix operating systems, and were first developed by Sun Microsystems for the Open Network Computing platform.

#### 2.4.2.4 Object/Component

Object middleware is an extension of procedural middleware, but with adoption of many of the object-oriented principles from C++. All available resources and hosts are objects, which can call other objects through references. These references are automatically marshalled and unmarshalled on the client and server, maintaining access and location transparency. Furthermore, object middleware implementations allow for both synchronous and asynchronous communication and transaction-based communication. While the architecture of invoking objects on other hosts is very similar to procedural middleware, object middleware integrates transactional and message-oriented principles. Middleware in this category are the subject of this thesis. An example of object middleware is the Common Object Request Broker Architecture (CORBA), which is further detailed in Chapter 4.

## 2.5 Space Systems Laboratory

The Space Systems Laboratory (SSL) at the University of Kentucky (UK) began as an embedded systems design lab specializing in autopilot instrumentation for autonomous UAVs. In 2006, the SSL joined the newly-created Kentucky Space consortium to develop an aerospace infrastructure in the state of Kentucky. The goal of Kentucky Space was to design, build, launch, and operate spacecraft every 12-18 months, creating a technical infrastructure and talent pool to facilitate Kentucky's permanent presence in space [54]. The flagship project for this endeavor was the cubesat KySat-1, a small satellite measuring 10x10x10 cm^3. KySat-1 launched in March 2011, and was lost along with the NASA Glory mission [55].

While developing KySat-1, other missions were completed to test subsystems and train new students. These missions included development and flight of instrumentation payloads on three different suborbital missions, as well as one high-altitude balloon flight. In late 2009, the SSL designed and constructed the NanoRack platform in partnership with NanoRacks, LLC for use on the International Space Station (ISS). Conceived as a standardized experiment locker, the NanoRack provides CubeSat-sized payload volumes to ground researchers, lowering the barrier to entry for microgravity research. The SSL developed the CubeLab Standard and operated the first two NanoRack platforms, as well as the first ten CubeLabs, on the ISS from a remote console located at UK. In 2012, the SSL partnered with NASA Ames Research Center's Small

Spacecraft Payloads Technology (SSPT) Office to create the AmesLab Bus, a NanoRack-extension to supply additional power and commanding capabilities to NASA small satellite science payloads on the ISS. In 2013, the SSL extended the capabilities of the AmesLab Bus through a collaboration with COSMIAC at the University of New Mexico, bringing Space Plug-and-play Avionics (SPA) compatibility to the ISS. Finally, in 2012 Kentucky Space was granted a launch opportunity on NASA's ELaNa IV mission to re-fly KySat-1. Both the SSL and Morehead State University's Space Science Center (SSC) worked to design and build KySat-2, which successfully launched in November 2013.

# 3 Host-Infrastructure Middleware

This chapter details host-infrastructure middleware. Host-infrastructure middleware is the lowest layer of object/component middleware, and is closest to the PE. While Schantz and Schmidt define this layer as an encapsulation of the native operating system's network mechanisms, this thesis extends host-infrastructure middleware to include middleware present without an operating system. This layer is divided into three sub-layers to account for middleware closer to the hardware level of the PE: hardware, network, and operating system. Middleware in each of these categories only encapsulate the functionality provided by that category, and do not significantly enhance that functionality. Middleware in these categories also do not account for the meaning of the bytes being transferred.

## 3.1 HI-Hardware

The HI-hardware sub-layer is the lowest layer of middleware. Middleware in this layer provides encapsulations of hardware registers to achieve communication between PEs. PEs found in CubeSats and UAVs typically provide hardware communications modules to implement serial communications protocols. Middleware in this layer can provide encapsulations for use of these modules. Only communications protocols that offer physical and data link layer services are considered. These include CAN, $I^2C$, USB, and Ethernet.

### 3.1.1 CAN

Controller Area Network (CAN) is a network communication protocol originally developed by Bosch in 1985 to answer the need for less complicated wiring harnesses in the automobile industry. It has since been adopted as ISO standard 11898 (1993), and is extensively used in the automobile, medical, manufacturing, and aerospace industries due to its low-cost, lightweight networking solution and fault-tolerance [56].

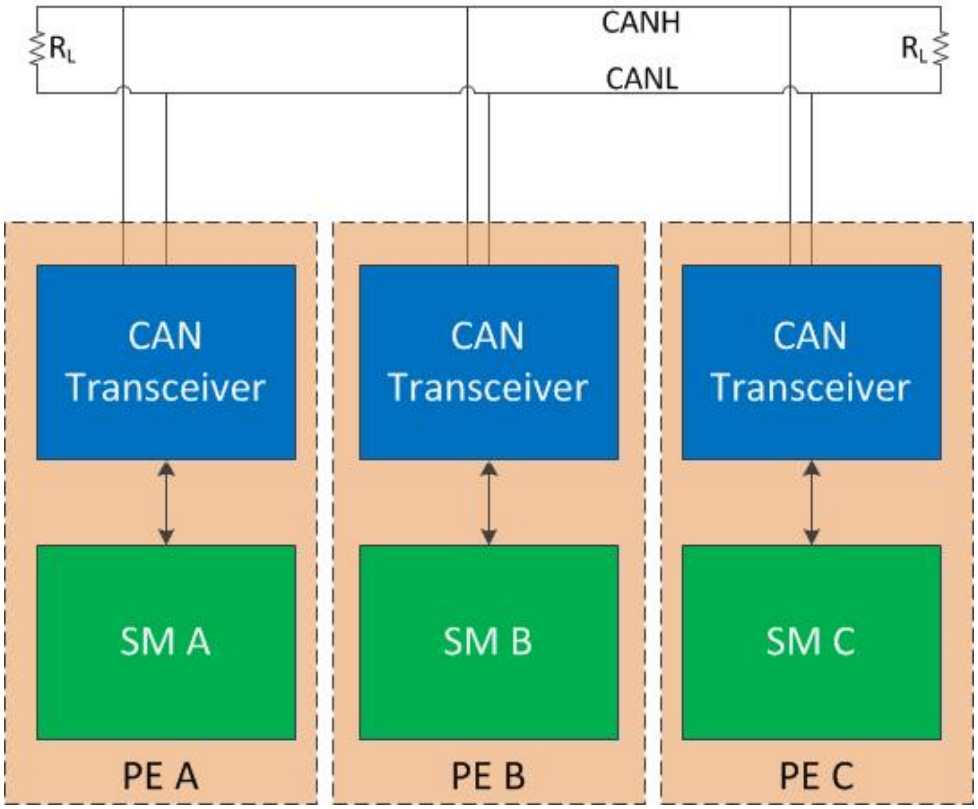


**Figure 4: CAN Bus**

CAN specifies a two-wire interface, CANH and CANL, that is terminated with load resistors; this allows a CAN bus can be up to 40 meters in length. CAN uses a broadcast messaging style, where messages are delivered to all PEs to ensure consistent messaging. CAN is message-oriented as well, meaning that instead of addressing PE recipients, the PEs themselves decide whether to interpret the data based on message identifiers. CAN uses carrier-sense, multiple-access (CSMA) where each PE must wait until a certain period of inactivity has elapsed before attempting to transmit on the bus. CAN additionally uses collision detection and arbitration on message priority (CD+AMP), where message priority flags arbitrate multiple PEs trying to transmit at the same time. Finally, CAN includes a cyclic redundancy check (CRC) within all message frames, providing for fault detection; errors during transmission prompt retransmission, providing fault correction [57].

### 3.1.2 $I^2C$

Inter-integrated Circuit ($I^2C$) is a network communication protocol developed by NXP Semiconductor (formerly Philips Semiconductors). $I^2C$ was originally developed to link discrete digital devices on small surface areas, such as PC cards. However, as data rates (increasing from

100 kilobytes per second to 3.4 megabits per second) and capacitive limits (extending beyond 400pF through isolation devices and improved printed circuit board (PCB) design) increase, $I^2C$ has found use in servers, home electronics, and aerospace devices [58].

$I^2C$ requires two wires, serial data (SDA) and serial clock (SCL); each is pulled up to the operating digital voltage via pull-up resistors. The network is composed of masters and slaves, and masters may only initiate transfers on the $I^2C$ bus. $I^2C$ messages contain a slave address, unique to each slave, and can either read from or write to an $I^2C$ slave.

### 3.1.3 USB

Universal Serial Bus (USB) is a plug-and-play protocol widely used in the PC industry. First released in 1996, the USB standard has gained worldwide acceptance and is incorporated on millions of devices for its simplicity of use and device support [59]. Its plug-and-play architecture makes it an ideal candidate for plug-and-play spacecraft, and SPA-U is based on it.

There are four types of transfers in USB: control, bulk, interrupt, and isochronous. The purposes and features of each are summarized in Table 6.

**Table 6: Summary of USB transfer modes**

| | Control | Bulk | Interrupt | Isochronous |
|---|---|---|---|---|
| **Purpose** | Configuration | Non-time-critical data transfers | Time-critical data transfers | Streaming, real-time transfers |
| **Error Detection** | Yes | Yes | Yes | Yes |
| **Error Correction** | Yes | Yes | Yes | No |

Control transfers are the only required transfer mode for all USB devices, and are used to enumerate a network, assign device addresses, and determine the properties and capabilities of newly connected devices. This is the feature most directly emulated on higher-level middleware such as SPA, discussed in Chapter 4. Other transfer types seek to address other data transfer situations, such as bulk transfers where data integrity is important but time to transfer is unimportant (such as print jobs); interrupt transfers where time is important (such as keyboards); and isochronous transfers where the rate of transfer is important but data integrity is unimportant (such as video or audio streaming) [59].

### 3.1.4 Ethernet 10/100 Base-T

IEEE 802.3 (Ethernet) is a network communication protocol developed by Robert Metcalfe and David Boggs at Xerox PARC in 1974. It became an IEEE standard in June 1983, and has continued to evolve since [60]. Ethernet has grown into a very popular standard for connecting personal computers and servers, typically used to support Transmission Control Protocol/Internet Protocol (TCP/IP).

Ethernet implements the bottom two layers in the OSI network model: the physical layer and the data link layer. On the physical layer, Ethernet encodes frames for transmission and decodes received frames over a variety of supported physical media, including twisted pair copper wire and fiber optic cable. On the data link layer, Ethernet implements Media Access Control (MAC) and Management Information Base (MIB). The MAC protocol is carrier-sense, multiple-access with collision detection (CSMA/CD), meaning a transmitting PE must detect an idle connection for certain period of time before transmitting; if a collision occurs, all transmitters that collided complete transmission to allow the collision to propagate, and then remain silent for a random period of time and attempt transmission again. The MAC protocol is node-oriented, meaning an addressing scheme is used to only address a selected recipient [60].

Ethernet frames are organized into octets, which are eight bits in the historical absence of a standard “byte”. The first seven octets are preamble octets of 0x55, which allows a receiver to prepare for the arrival of a frame. The next octet is a start frame delimiter (SFD), which denotes the beginning of the frame. The next six octets form the destination MAC address, and the next six after that the source MAC address. There are two octets for the length and type of data, and finally between 46 and 1500 octets of data. The frame concludes with four octets comprising the frame check sequence, consisting of a 32-bit CRC over the all frame fields excluding the preamble and SFD [61].

### 3.1.5 UART

Universal Asynchronous Receiver/Transmitter (UART) is a serial communications protocol dating back to the 1960’s. Its most basic function is translating bytes into bits for transmission between, historically, Data Terminal Equipment (DTE) and Data Communications Equipment (DCE), though these terms have been replaced by transmitters and receivers for communications between embedded systems. UART subsystems are offered on most modern microcontrollers, particularly the target 8051 and ARM Cortex-M processors; many specific chips from both architectures contain multiple UART subsystems.

UART consists of, minimally, a two-wire interface and shared ground: a transmit line and a receive line. Communications between nodes using UART are asynchronous and there is no native addressing scheme; both nodes must, however, use the same UART clock rate in order for the bits to be successfully recomposed into the correct bytes after transmission. Transmission of a byte includes a start bit, eight data bits, an optional parity bit, and a stop bit [62].

While there is no native support in UART for a multidrop bus, there are additional standalone chips that implement a nine-bit mode for supporting such a bus for network communications. This nine-bit mode is a master/slave network, where the master will use a ninth bit, usually the optional parity bit, to differentiate whether the byte being transmitted is an address byte or a data byte. If it is an address byte, all devices connected via UART to the master device must check their own address, with the correctly-addressed slave responding with a data byte to the master. The master then transmits normal data bytes to that slave [63].

### 3.1.6 Middleware Aspects

The degrees to which each HI-Hardware layer middleware address the five key aspects of middleware will now be compared and contrasted. The results are summarized in Table 7.

**Table 7: Comparison of HI-Hardware Middleware**

| | Network Communication | Coordination | Reliability | Scalability | Heterogeneity |
|---|---|---|---|---|---|
| **CAN** | Message-oriented | Asynchronous | At-least-once | Location, Replication | Hardware, Software |
| **$I^2C$** | Node-oriented | Synchronous | At-most-once | None | Hardware, Software |
| **USB** | Node-oriented | Mixture | Mixture | None | Hardware, Software |
| **Ethernet** | Node-oriented | Asynchronous | At-most-once | None | Hardware, Software |
| **UART** | Node-oriented | Asynchronous | At-most-once | None | Hardware, Software |

#### 3.1.6.1 Network Communication

CAN handles network communication by specifying a message-oriented networking style. This means that all PEs on the network receive messages placed on the bus, but must each individually interpret the message to determine whether or not to handle the message. This implies that no addressing scheme is used to uniquely address each PE, but also implies no unicast messaging. CAN frames contain 11-bit message identifiers at the beginning of the frame; every PE on the network receives the message, but every PE must read the 11-bit identifier to determine if it needs to continue reading and interpreting the frame. This identifier yields 2048 possible messages; an "extended" CAN with a 29-bit identifier yields 537 million possible messages.

$I^2C$ handles network communication by specifying a node-oriented networking style. This means that messages contain physical PE addresses, and only the designated PE listens to and handles the message. An $I^2C$ network consists of masters and slaves, where the master (or sending) PE must know the address of the slaves on its network, and must specify which slave to send the message to. The number of slaves are inherently limited by the single byte used to address them, for 256 possible slaves; this can however be extended through $I^2C$ bus expanders.

USB handles network communication by specifying a node-oriented networking style. Like $I^2C$, each PE has an address; however, unlike $I^2C$, each PE is assigned the address during enumeration from the USB host controller. The USB bus is defined as a four-wire interface, with a 5V and ground line and two data lines, D+ and D-. These data lines use differential signaling, and can be in one of four possible states: single-ended 0, single-ended 1, data J, and data K. These states allow for low- and full-speed communication on the same bus.

Ethernet handles network communication by specifying a node-oriented networking style. Like $I^2C$, this means that every message contains a physical PE address, called a MAC address, and the message is routed to that PE. Unlike $I^2C$, any PE on the network can send a message. Ethernet MAC addresses use a 48-bit address, yielding 474,976,656 possible PEs [60].

UART handles network communication through its nine-bit mode as a node-oriented networking style. In nine-bit mode, the parity bit is used as an indicator of whether the byte transmitted from a master is an address or a byte of data. If it is an address, all connected slaves must check their own addresses, with the correct slave responding and allowing data transmission.

#### 3.1.6.2 Coordination

CAN handles coordination through asynchronous communication. There is no shared clock line on the CAN bus, and transmitting PEs do not block waiting for an acknowledgement or reply from receiving PEs.

$I^2C$ handles coordination through synchronous communication. One of the two $I^2C$ bus lines is a clock line, allowing the receiver to only record or supply bits during specified clock pulses. At the end of each byte, the receiver must acknowledge reception of the previous byte, forcing the transmitting PE to block while waiting for this acknowledgement.

USB handles coordination for control, bulk, and interrupt transfer modes with synchronous communication. While there is no shared clock line to synchronize USB hosts and receivers, a handshake sequence follows data transmission where the receiver must provide an acknowledgement of error-free data, causing the transmitting host to wait until that acknowledgement has been received. For the isochronous transfer mode, coordination is asynchronous; the host transmits data at a guaranteed rate but does not correct for errors or wait for acknowledgements from the receiver.

Ethernet handles coordination through asynchronous communication. Messages are created and transmitted, and the transmitting PE continues execution while the message is routed to the intended receiver.

UART handles coordination through asynchronous communication. Messages are decomposed into individual bits and transmitted using a previous agreed-upon clock rate; however, no clock signal is shared between transmitter and receiver. Furthermore, mismatched clock rates between the transmitter and receiver will not stop communications; rather, the receiver will recompose the received bits incorrectly.

#### 3.1.6.3 Reliability

CAN's reliability is at-least-once. Its specification of physical network characteristics specifies low-noise differential signaling for communications between PEs, meaning that the voltage difference between the wires yields the signal instead of an absolute voltage threshold, as in $I^2C$. CAN also specifies 120 ohm resistors at either end of the network to maintain the differential impedance of the bus and further reject noise [57]. Beyond the physical medium, CAN frames include CRC checksums. The transmitting PE computes and appends the CRC checksum onto the frame; the receiving PE(s) computes its own CRC checksum on the received

frame and compares it to the transmitting CRC checksum, requesting retransmission if they do not match. This guarantees successful message delivery, but the message could be delivered multiple times.

$I^2C$'s reliability is at-most-once. Its specification of physical network characteristics specifies the maximum allowed bus capacitance and the presence of pull-up resistors to guarantee compliant voltage rise times on both clock and data lines [58]. Beyond the physical medium, $I^2C$ slaves must acknowledge each byte that is transmitted by the master, guaranteeing that the master will know whether the byte was received. There is no built-in checksum to verify the validity of the actual bits, meaning the message may not be correct but will not be retransmitted natively.

USB's reliability is exactly-once for control, bulk, and interrupt transfer modes. The handshake sequence of transfers allows for receivers to notify the host of errors in data, prompting the host to retransmit. A sequence number increments with every successfully transmitted packet, enabling receivers to tell the host which packets failed and to know which, if any, packets are duplicates. For the isochronous transfer mode, the reliability is at-most-once, where the host transmits data to the receiver at a guaranteed rate but with no error correction, with no guarantee of delivery but ensuring no duplicate packets as well.

Ethernet's reliability is at-most-once. It specifies different data transfer speeds and the physical interconnects required to achieve them, such as 1Mb/s: 1Base5 with two twisted telephone lines; 10Mb/s: 10Broad36 with one broadband cable or 10Base-F with one optical fiber; 100Mb/s: 100Base-TX with two twisted pairs of Category-5 (CAT5) cable; and even 1Gb/s: 1000Base-T with four CAT5 cable pairs [61]. Ethernet frames contain a frame check sequence, which contains a 32-bit CRC checksum over all variable frame fields. The transmitting PE computes this value and includes it in the frame, and the receiving PE computes it and compares to the sent CRC. If they do not match, Ethernet does not natively trigger a re-request, and instead just discards the message; thus the message is not necessarily sent correctly, the receiver is aware of the fault, and no duplicates will occur.

UART's reliability is at-most-once. UART does offer a parity bit that can help detect the presence of errors, but there is no native mechanism for correcting these errors or retransmitting incorrect bits. It is left to the application to either check this parity bit and attempt retransmission, or to implement some other form of error detection and correction.

#### 3.1.6.4 Scalability

CAN provides location and replication transparency. CAN's message-oriented network style is agnostic to the number and location of other PEs on the bus, and SMs using CAN do not need to know any endpoint addresses. Furthermore, replicated functionality need only be added to the bus and know the CAN identifiers of the messages they need to handle.

$I^2C$ does not natively provide any transparency. Many $I^2C$ devices have built-in hardware addresses that are not configurable, whereas "smarter" $I^2C$-capable devices, such as microcontrollers, have software-settable addresses.

USB does not natively provide any transparency. When new devices are connected to the bus, the USB host controller follows an enumeration process where the new device is assigned a unique device address for use in future communication.

Ethernet does not natively provide any transparency.

UART does not natively provide any transparency.

#### 3.1.6.5 Heterogeneity

CAN provides hardware and software heterogeneity. It exhibits hardware heterogeneity because different computer architectures can be used by PEs on the bus, as long as they can correctly format CAN packets. Modern microcontrollers include CAN modules in the hardware. Furthermore, companies provide commercial CAN transceivers that can be connected to nearly any systems, ranging from 8-bit to 32-bit microcontrollers and more powerful embedded computers. CAN exhibits software heterogeneity for the same reason; CAN transceiver solutions exist for a range of embedded real-time operating systems up to full personal computers, and can be interfaced to a number of programming languages. However, CAN does not exhibit network heterogeneity because the network signaling and protocols are tightly defined.

$I^2C$ exhibits hardware and software heterogeneity. Like CAN, $I^2C$ is available as a standalone module on nearly every modern microcontroller architecture, and range in availability from low power 8-bit to higher-power 32-bit microcontrollers. Furthermore, companies such as Texas Instruments, Silicon Laboratories, and NXP sell a wide variety of $I^2C$ peripherals for a range of computing equipment. Again like CAN, $I^2C$ can interface to different operating systems and programming languages as well, and is not dependent on any SMs. However, $I^2C$ does not exhibit network heterogeneity because the network signaling and protocols are tightly defined.

USB exhibits hardware and software heterogeneity. Compared to CAN and $I^2C$, USB is a much more complex serial communications protocol and hasn't historically been included on low-power microcontrollers. However, as the computational capabilities increase and power consumption decreases on modern microcontrollers, USB is increasingly being offered in low-power 8-bit microcontrollers, yielding a wide range of available hardware architectures. Furthermore, USB implementations are available on nearly every popular operating system.

Ethernet exhibits hardware and software heterogeneity. Like CAN and $I^2C$, Ethernet modules are offered on many modern microcontroller architectures, and standalone modules can be purchased for integration on many systems. Ethernet exhibits more complexity and overhead than CAN or $I^2C$, however, and is a bit more restricted in what systems can support it (need references here). Like CAN and $I^2C$, Ethernet is not restricted to a particular operating system and can interface with a variety of programming languages and any SM. However, unlike CAN and $I^2C$, Ethernet does exhibit partial network heterogeneity. While the signaling and protocols are tightly defined in CAN and $I^2C$, the Ethernet physical layer allows for telephone wire pairs, broadband cable, coaxial cable, optical fibers, and wireless transmission. Full network heterogeneity would imply that Ethernet is unconcerned with the physical media and signaling; hence Ethernet's restricted provision of supported media and signaling give it partial network heterogeneity.

UART exhibits hardware and software heterogeneity. Like the other HI-Hardware, most modern microcontrollers, particularly those of the target processors, offer multiple UART modules onboard, and many include sample software to use the modules.

### 3.2 HI-Network Middleware

The HI-network sub-layer is above the HI-hardware sub-layer, and actively routes messages. It uses the HI-hardware layer's encapsulations to transmit streams of bytes to desired recipients. These desired recipients can either have a network address in node-to-PE messaging, such as in SPA-1L, or can be programmed to respond only to certain messages in broadcast networks, such as in MAVLink.

The HI-network middleware implementations to be reviewed in this chapter are MAVLink, which is a header-only message marshalling library originally created for micro-UAV communications; SPA-1L, which is a "lite" implementation of SPA that provides communications between the PEs on a distributed network in the CubeSat KySat-2; Avionics Full-Duplex Ethernet (AFDX); and Time-Triggered CAN (TTCAN).

### 3.2.1 SpaceWire

SpaceWire is a middleware developed by Steve Parkes at the University of Dundee in 2008 (check this date) specifically for spacecraft communications. It has been used on a variety of space missions, including the European Space Agency's (ESA) ExoMars surface rover and NASA's Swift gamma-ray burst observation satellite, Lunar Reconnaissance Orbiter (LRO), and the James Webb Space Telescope [64].

Instead of specifying a particular layer's worth of middleware, the SpaceWire standard specifies the physical layer through the transport layer in an attempt to control the end-to-end process of transporting packets. It is an ETP, point-to-point network similar to Ethernet, but with more functionality above the physical and data link layers implemented by Ethernet. Adopting its own naming convention, SpaceWire is divided into the following "levels": physical level, signal level, character level, exchange level, packet level, and network level [65].

#### 3.2.1.1 Physical Level

The physical level is the bottom level of SpaceWire, and defines the PCB tracks, cables, and connectors used for SpaceWire. The physical level is designed to allow for up to 10 meter-long cables and to meet typical spacecraft electromagnetic compatibility specifications. SpaceWire consists of four twisted pair wires with separate shielding. These pins are: Data_In+, Data_In-, Strobe_In+, Strobe_In-, Data_Out+, Data_Out-, Strobe_Out+, and Strobe_Out-. The connector specified for SpaceWire is the 9-pin micro-miniature D-type connector.

#### 3.2.1.2 Signal Level

The signal level is above the physical level, and defines the data rates, acceptable noise levels, and encoding used for transmitting bits with SpaceWire. Low voltage differential signaling (LVDS) is used to transmit bits, relying on a small voltage swing between differential wires to denote bits. This provides low noise and low power consumption, as well as constant drive current and independence from endpoint voltage levels. For encoding, Data-Strobe (DS) encoding is used. The data is sent on the data line, and the clock signal is encoded as the exclusive-OR (XOR) of the data and strobe lines. This prevents clock skew, which is the RC delay of wire causing variations in clock signal arrival times. Additionally, this clock signal implies synchronous coordination, requiring both the transmitting and receiving PEs to suspend activity for the duration of the message.

#### 3.2.1.3 Character Level

The character level is above the signal level, and defines the data and control characters that manage data flow. There are two types of characters in SpaceWire: data characters and control characters. Data characters are 10 bits, with one parity bit, one data-control flag to indicate that the character is a data character, and eight bits of data, transmitted least-significant bit first. Control characters are four bits, with one parity bit, one data-control flag to indicate that the character is a control character, and two control bits: a flow control token (FCT), a normal end of packet (EOP), an error end of packet (EEP), and an escape (ESC). These are then combined to form control codes. Two of these control codes are NULL, which is formed from ESC and FCT and indicates an idle connection, and Time-Code, which is formed from ESC and a data-character containing system time.

#### 3.2.1.4 Exchange Level

The exchange level is above the character level, and defines the initialization and error detection processes across a link. The exchange level offers a series of services to the next level, the packet level, to packetize and deliver data to a recipient. These services manipulate the control and data characters defined in the character level, and are: initialization, flow control, disconnect error detection, parity error detection, and link error recovery.

#### 3.2.1.5 Packet Level

The packet level is above the exchange level, and defines how data is split into packets for transmission across a link. A packet is composed of a destination address, the cargo or payload, and an end of packet marker. Destinations are address in one of two ways: path addressing or logical addressing.

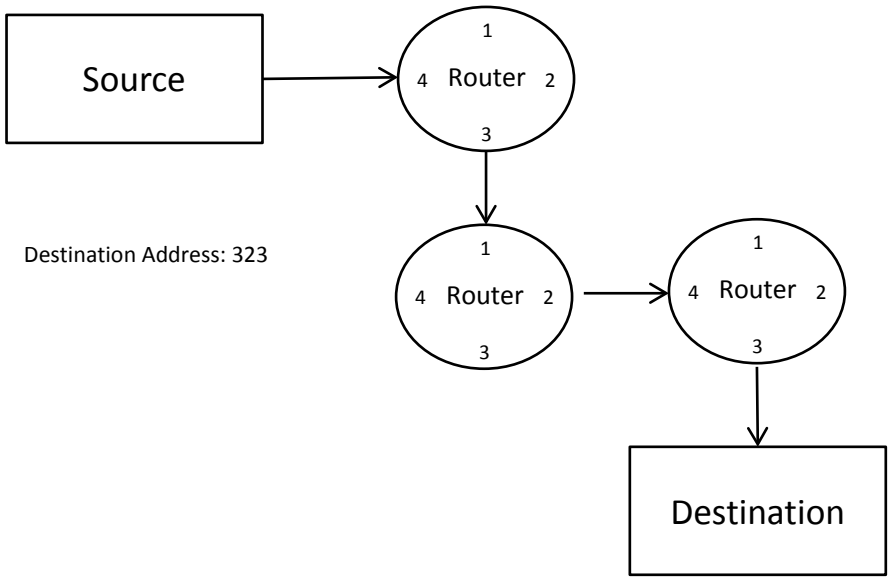


**Figure 5: Path addressing in SpaceWire**

Path addressing means that the destination address field is encoded as the list of output ports that each router must forward the packet out of to reach the destination. Each router forwards the packet out of the port corresponding to the first byte of the destination address field, then discards that byte for the next router. An example of this path addressing scheme is shown in Figure 5: a destination address of 323 in a three-router network would take the following path: router 1 would forward the packet out of port 3, router 2 would forward the packet out of port 2, and router 3 would forward the packet out of port 3 to the destination. This path addressing scheme implies explicit knowledge of the network to every PE.

Logical addressing moves network knowledge from the PEs to the routers. With logical addressing, each PE in the network is assigned a logical address in the range 32 to 255. PEs address each other using this address, and routers maintain routing tables defining how to reach each PE. The cargo is the actual data payload to transfer. The end of packet marker is either the EOP or EEP control characters. This level is the upper level available to SMs for transmission of data.

#### 3.2.1.6 Network Level

The network level is above the packet level and is the top-most level of SpaceWire. This level defines how packets are transferred and routed between PEs, and is composed of PEs, routing switches, and the links between them. Routing switches in a SpaceWire network maintain routing tables, and parse the destination address emitting from PEs in order to replace that address

with the appropriate next-hop address. This is similar in structure to the routing of TCP/IP packets.

### 3.2.2 MAVLink

The Micro Air Vehicle Communication Protocol (MAVLink) was developed by Lorenz Meier in 2009 for ground station-to-MAV communications for the PIXHAWK autopilot. As a message marshalling library, it also serves for data-passing between onboard components as well [66]. It has since been adopted as the ground station-to-MAV and internal communications protocol for many other commercial and open source MAV products, including ArduPilotMega, SmartAP, and AutoQuad 6 [67].

MAVLink is a header-only message marshalling library that packs C-structures over a serial channel. C-structures are structured objects in the C programming language, which is the implementation language of MAVLink. They are similar to arrays in that they are a container for named objects; however, the objects in arrays must be of identical type. The data types of objects in a structure can all be of different types, allowing more flexibility in the custom objects that structures can describe [68]. With only eight bytes of overhead per message and automatic dropped packet detection, MAVLink can be deployed on microcontrollers and over low-bandwidth radio connections with ease due to such low overhead. Examples of serial channels include UART, $I^2C$, SPI, CAN, and User Datagram Protocol (UDP). MAVLink is hardware-independent, and the choice of serial channel does not affect the protocol or message passing.

Messages themselves are specified in Extensible Markup Language (XML), which specifies a format and syntax for electronic information publishing. XML documents contain a set of elements, delimited by tags [69]. The XML messaging libraries are then auto-generated into their corresponding, MAVLink-ready C-structures using a Python graphical user interface (GUI). This guarantees compliant C-structures, and allows for MAVLink version upgrades and expansions of functionality with no burden on end users. There are a set of preset MAVLink messages specifically intended for popular autopilots and their ground control systems; these messages do everything from basic heartbeat messages to telemetry requests, video streaming, and motor control. An example such heartbeat message encoded in XML is shown in Figure 6. This message is required for use with the popular UAV ground control software QGroundControl, and since MAVLink is stateless, is used to periodically poll the UAV to make sure it’s alive and operating. This heartbeat-style message is recommended for any SM, however, as it maintains knowledge of the network.

A MAVLink message consists of four primary fields, as seen in Figure 6. These fields are: ID, name, description, and field, which is further composed of type and name. The ‘ID’ field gives a unique numerical identifier to the message, and is how sending and receiving MAVLink implementations address the message. This ID is a single byte, and ranges from 0 to 255, yielding 256 possible messages. If MAVLink is running on a UAV using an autopilot and QGroundControl, IDs between 150 and 240 can be used for custom messages. The ‘name’ field gives a human-readable name to the message, and is not actually transmitted by MAVLink. The ‘description’ field is similarly a human-readable description of what the message is, and is not transmitted by MAVLink. The ‘field’ field is composed of two fields, and encodes the value in the MAVLink message. It is composed of a type, which is a variable size/type that is unique to the system (for example, uint8_t or unsigned char), and a name, which is the name of the variable as it will be addressed when reading the generated C-structure. All messages must follow this format, and can either be placed in an existing MAVLink message definitions file, which is included automatically with popular autopilots and QGroundControl, or in a custom file. If placed in a custom file, the <mavlink> tags must be used, as well as the MAVLink <version>. This format is specified in the example custom message file in Figure 7.

Once the MAVLink messages have been translated into C-structures, MAVLink handles transmission of these structures by composing them into frames. A MAVLink frame consists of a six byte header, a maximum 255-byte payload, and two checksum bytes. Figure 8 shows the MAVLink frame composition.

```
<message id="0" name="HEARTBEAT">
    <description>The heartbeat message shows that a system is present and responding. The type of the MAV and Autopilot hardware allow the receiving system
        to treat further messages from this system appropriate (e.g. by laying out the user interface based on the autopilot).</description>
    <field type="uint8_t" name="type">Type of the MAV (quadrotor, helicopter, etc., up to 15 types, defined in MAV_TYPE ENUM)</field>
    <field type="uint8_t" name="autopilot">Autopilot type / class. defined in MAV_CLASS ENUM</field>
    <field type="uint8_t" name="base_mode">System mode bitfield, see MAV_MODE_FLAGS ENUM in mavlink/include/mavlink_types.h</field>
    <field type="uint32_t" name="custom_mode">Navigation mode bitfield, see MAV_AUTOPILOT_CUSTOM_MODE ENUM for some examples. This field is
        autopilot-specific.</field>
    <field type="uint8_t" name="system_status">System status flag, see MAV_STATUS ENUM</field>
    <field type="uint8_t_mavlink_version" name="mavlink_version">MAVLink version</field>
</message>
```

**Figure 6: MAVLink Heartbeat Message [67]**

```xml
<?xml version="1.0"?>
  <mavlink>
      <version>3</version>
       <include>common.xml</include>
    <enums>
    </enums>
    <messages>
        <message id="150" name="RUDDER_RAW">
            <description>This message encodes all of the raw rudder sensor data from the USV.</description>
            <field type="uint16_t" name="position">The raw data from the position sensor, generally a potentiometer.</field>
            <field type="uint8_t" name="port_limit">Status of the rudder limit sensor, port side. 0 indicates off and 1 indicates that the limit is hit. If this
                                                    sensor is inactive set to 0xFF.</field>
            <field type="uint8_t" name="center_limit">Status of the rudder limit sensor, port side. 0 indicates off and 1 indicates that the limit is hit. If this
                                                     sensor is inactive set to 0xFF.</field>
            <field type="uint8_t" name="starboard_limit">Status of the rudder limit sensor, starboard side. 0 indicates off and 1 indicates that the limit is hit.
                                                        If this sensor is inactive set to 0xFF.</field>
        </message>
    </messages>
  </mavlink>
```

**Figure 7: MAVLink Custom MAVLink File Specification [67]**


**Figure 8: MAVLink Frame [67]**

The header contains the following bytes: STX, which notes the beginning of a packet and is always 0xFE; LEN, which the length of the payload field, ranging from 0 to 255; SEQ, which is the sequence number and increments every message and rolls over at 255; SYS, which is the source ID and identifies the system sending the message; COMP, which is the component ID and identifies the component of the system sending the message; and MSG, which is the message ID from the MAVLink XML message definition. Following the header is the payload field, which contains the data to be transmitted. Finally, the two checksum bytes contain a 16-bit CRC checksum that validates the integrity of the message and checks for reordering implementation. With the latest MAVLink version, MAVLink reorders fields in messages according to their data type size to prevent word/half word memory alignment issues; this is validated in the CRC_EXTRA byte at the end of a MAVLink frame.

In order to transmit a message using MAVLink, only the send/receive functions within the MAVLink header need to be linked to the chosen protocol. The user must create custom functions containing protocol/hardware-specific communications functions, such as UART_send/receive, $I^2C$_send/receive, etc. MAVLink also provides convenience functions in the form of adapter headers, where the functions “comm_send_ch” and “comm_receive_ch” are already implemented, and only need the user to place the protocol/hardware-specific function calls in these functions [67].

### 3.2.3 SDM-Lite

The Satellite Data Model-Lite (SDM-Lite) strips down the functionality and thus computing requirements of Space Plug-and-play Avionics (SPA), in order to be better supported by low-power, 8-bit microcontrollers. The SDM-Lite resulted from a partnership between the UK SSL and the University of New Mexico’s COSMIAC. The SDM-Lite has seen flight heritage on the SSL’s KySat-2 CubeSat and COSMIAC’s Trailblazer CubeSat, as well as the CubeLab Bus International Space Station payload (ref).

The goal of SDM-Lite is to control a plug-and-play distributed embedded network on low power 8-bit processors [70]. The governing architecture is SPA, and the goal of this architecture

is to reduce the complexity and time of completion for spacecraft avionics and integration. Since spacecraft avionics significantly vary in processing power and speed, four different varieties of SPA exist, each using a different network communication medium that reflects the relative capabilities and speed of the host MCUs: SPA-O (optical), SPA-S (spacewire), SPA-U (USB), and SPA-1 ($I^2C$). Since all but SPA-1 require Linux or VxWorks operating systems and thus more power-hungry processors, the SDM-Lite focuses on creating a lighter version of SPA-1 that can be run in low-power 8-bit environments. Creating this lighter version involved creating a lighter version of the Satellite Data Model (SDM), which is the "traffic cop" that manages the SPA network.

The purpose of the SDM-Lite is to provide a discovery and join mechanism for SPA-1 devices, while maintaining compliance with larger full-SPA networks. This allows for new devices to be detected and their capabilities and needs discovered by the SDM-Lite. These devices are either SPA-compliant by design or legacy devices that must be adapted for compliance with SPA. Applique Sensor Interface Module (ASIM). The ASIM acts as a bridge between non-SPA devices and the SPA network. The SDM-Lite discovers the capabilities of ASIMs through their Extended Transducer Electronic Datasheets (xTEDS), which are XML-style sheets that describe the capabilities of the ASIM.

The SDM-Lite is broken into four categories of tasks: network enumeration, round robin, data handling, and process information. There are three primary structures for queuing and handling information-passing between the four tasks: the processing structure, the data structure, and the output queue. The processing structure is a buffer for the data and state read from each ASIM during the round robin task. The processing structure contains the device ID, the address, and the data from the ASIM, as well as a pointer to the next ASIM structure. The processing structure only holds one round robin's cycle worth of data at a time. The data structure is the long-term storage of ASIM data from the processing structure. There is one data structure per ASIM, and the data handling task moves data read from ASIMs in the round robin cycle into this structure. Finally, the output queue is an outgoing commanding queue issued by the data handling task; commands to issue are placed into this queue during the process information task. These data structures and their contents are pictured in Figure 9.

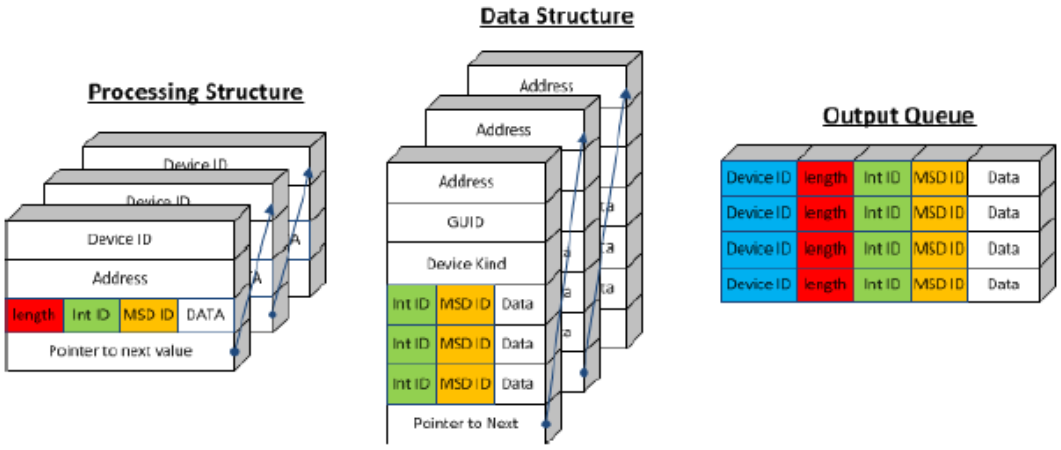


**Figure 9: SDM-Lite Structures [70]**

#### 3.2.3.1 Network Enumeration

The network enumeration process registers SPA devices on the network, allowing for the SDM-Lite to keep track of each device's properties, addressing information, etc. This process brings ASIMs into the SPA network by interrogating new ASIMs for their Global Unique Identifier (GUID), the version of their software, their status, and their xTEDS. This enumeration process is repeated every 100 round robins to check for new devices on the network. This process implements the primary plug-and-play aspect of the SDM-Lite.

#### 3.2.3.2 Round Robin

The round robin periodic process polls each possible ASIM address on the network, determining if any ASIMs have joined, dropped off, or changed configuration. This process writes to each ASIM address, then successively reads from each ASIM address to allow the ASIMs time to respond while avoiding system downtime or blocking loops. This process is shown in Figure 10. If an ASIM goes offline and comes back online, address resolution takes place where the ASIM temporarily becomes master of the network, sending messages to each available address until an empty address is found. The processing structure is used during this task for storing the data read from each ASIM.

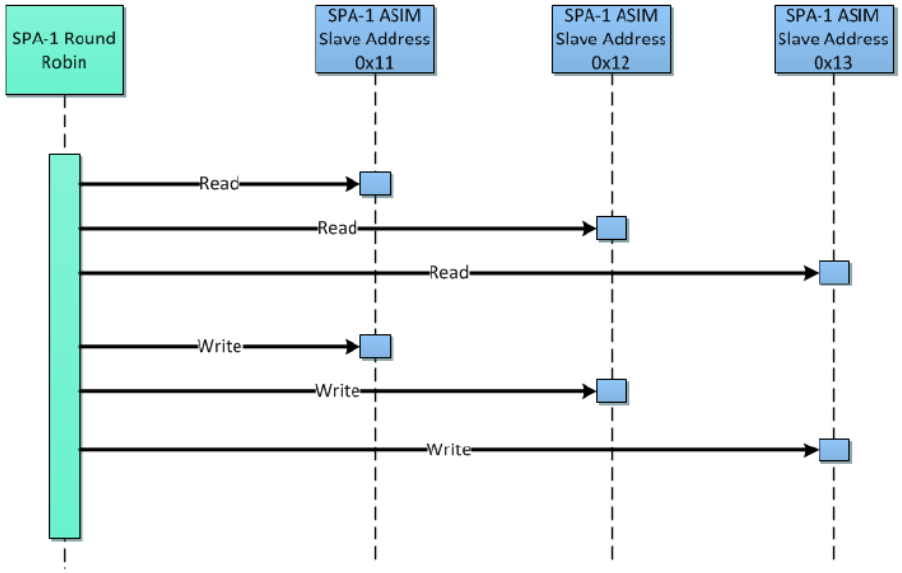


**Figure 10: Round Robin Task [71]**

#### 3.2.3.3 Data Handling

The data handling process involves servicing the output queue and populating each ASIM's data structure. The output queue contains commands to be issued to ASIMs, and is populated during the process information task. The queue is organized by the device ID for each ASIM. This utilizes the plug-and-play aspect of the SDM-Lite, in that the device ID does not directly correspond with that ASIM's network address. This allows for changes in ASIM addresses due to re-enumeration if any devices drop off or are added to the network, with no knowledge required of the process information task. Additionally, the data handling task stores any data read from each ASIM during the round robin task into that ASIM's specific data structure.

#### 3.2.3.4 Process Information

The process information task is the "SM layer" of SDM-Lite. This assumes that ASIMs all have SMs that perform missions on top of their SPA implementations. During that time, the SDM-Lite can also perform SM-level actions. This task operates on each ASIM's data in that ASIM's data structure, and places any commands that need to be issued in the output queue. The process information task addresses ASIMs to be commanded by their device ID, assigned during the network enumeration process. This allows for physical address changes without compromising the validity of the device ID between the process information and data handling tasks.

In order to be SDM-Lite-compliant, all ASIMs must respond to three different phases when connected to the SDM-Lite: address resolution, network enumeration, and round robin. SM-level actions are implemented on an ASIM-by-ASIM basis.

### 3.2.4 SPA-1 Lite

SPA-1 Lite (SPA-1L) is a hardware and software implementation of a SPA-1-style distributed processing network for 8-bit microcontrollers. SPA-1L was demonstrated on-orbit in the C&DH of the CubeSat KySat-2 [5]. SPA-1L is composed of a modified SDM-Lite and Lite Applique Sensor Interface Modules (LASIMs), and is described as both a hardware architecture and a software architecture.

#### 3.2.4.1 Hardware Architecture

The KySat-2 C&DH utilizes a distributed processing design philosophy, and includes hardware time-keeping, mass storage, and all processors on the same physical board, pictured in Figure 3. There are five total processors that make up the KySat-2 C&DH: one central processor and four subsystem interface processors. The C&DH is not strictly limited to four subsystem interface processors, and can theoretically scale to support any number of subsystems below the address limit of $I^2C$. Practically, as the number of interface processors increases, the increase in bus capacitance will require larger pull-up resistors, resulting in more power consumed by the network.

The naming convention for the architecture is taken from the SPA standard [4]. The central processor is called the Satellite Data Manager-Lite (SDM-L). This processor handles the mission-specific implementation software, including ground command handling. The SDM-L also implements routine health and status monitoring, file system management, and data exchange. The processor chosen for KySat-2 was the Silicon Labs 8051F120, with 8kB of RAM and 128kB of flash memory. This processor was chosen for its low power consumption and peripheral options, including communications modules ($I^2C$, SPI, UART). The C8051 family from Silicon Labs also has flight heritage with prior SSL missions, including the Sub-Orbital CubeSat Experimental Mission (SOCEM) [72]. The subsystem interface processors are called LASIMs and utilize the Silicon Labs 8051F930, a smaller form-factor, lower power processor that includes 8kB of RAM and 64kB of flash memory. This processor was also chosen for its low power consumption and similar availability of serial communications peripherals, including $I^2C$, SPI, and UART. Figure 11 shows the KySat-2 C&DH SDM-L and LASIMs, and their physical locations.

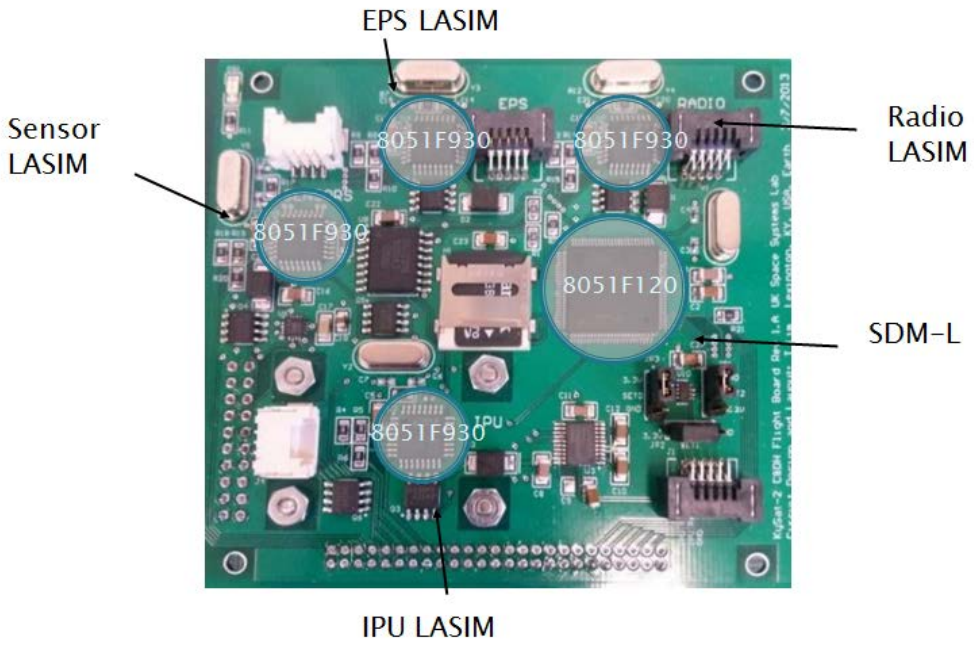


**Figure 11: KySat-2 C&DH processing elements [5]**

The KySat-2 C&DH includes internal subsystems. The SDM-L implements a FAT file system using a micro-SD card and the SPI communication protocol for reading and writing data. Timekeeping onboard the satellite is managed using a Real-Time Clock (RTC). The RTC serves as the time-base for satellite's custom operating system, and has a resolution of one second, limiting the periodicity of command execution and time-keeping to one second. Fault-tolerant hardware features were designed to complement the software's fault tolerant features, and consist of individual MOSFET power switches on the LASIMs, controllable from the SDM-L in the event of LASIMs becoming unresponsive. Finally, an external watchdog timer (WDT) maintains reset control over the SDM-L. The WDT has jumper-selectable timeout intervals measuring from one millisecond to 60 seconds, allowing for development flexibility in timeout selection. The WDT is kicked with a frequency of approximately six hertz during routine command servicing by the SDM-L.

One of the goals of the CubeSat Standard is to allow for rapid construction of spacecraft, and the KySat-2 C&DH design supports this goal by addressing post-integration reconfiguration. The ability to reprogram a mission-critical processor typically becomes difficult after the satellite has been integrated, a problem magnified four-fold by KySat-2's four extra processors. As a result, reprogrammability was added by breaking all processor programming pins out to a 50-pin ribbon cable with an external interface on the –Z side of the spacecraft. This cable connects to a

custom-designed programming board, allowing for each of the C&DH processors to be reprogrammed post-integration and post-environmental testing. The ribbon cable also breaks out UART debugging lines, reset switches, and provides individual power to each processor for further debugging.

#### 3.2.4.2 Software Architecture

The goal of the software architecture design for KySat-2 was to mimic the distributed nature of SPA. The SPA software architecture is managed by a middleware component called the SDM. The SDM provides network services that allow data producers and consumers to dynamically join or leave the network and be paired with appropriate resources. When a new device joins the SPA network, an enumeration process begins that includes giving the device an address and registering its XTEDs, an XML format that describes the needs and capabilities of the subsystem in the form of interfaces. The result is a plug-and-play network of devices.

While SPA greatly reduces the amount of time required to integrate a complex system, it requires a system with more power and computational capabilities seen in previous CubeSat designs. With an emphasis on a pure PnP methodology, a typical SPA network is primarily composed of a heterogeneous mixture of 32 bit processors, all running a complex RTOS such as VxWorks or Linux [73]. The SPA standard would dictate one of these processors act as a gateway for every subsystem to the SPA network greatly, increasing power consumption. Due to wildly varying spacecraft requirements of data transfers and power, SPA exists in four different interfaces: optical (SPA-O), SpaceWire (SPA-S), USB (SPA-U), and $I^2C$ (SPA-1). While the SPA-1 variant allows for devices with lower processing capabilities and power requirements, we worked with COSMIAC at the University of New Mexico to develop a lighter $I^2C$ derivative of SPA intended to be used with extremely low power eight-bit processors, called SPA-1L.

SPA-1L differs from a full SPA design in several key ways to facilitate its use in lower power SMs. Among the removed features are self-describing network entrance and discovery and enumeration through the transfer of XTEDS. With this change, mission specific software must have a known network configuration and addressing scheme, therefore losing pure PnP operation. This was considered to be an acceptable trade off as hardware components and network configuration are typically established before SM specific software development begins, making it possible to address the issue with configurable software.

SPA-1L has currently been implemented in two different SMs. The Trailblazer CubeSat, built by COSMIAC and manifested on ELaNa IV, has a SPA-centered C&DH. In addition, the SSL has worked to extend the CubeLab Bus adaptor for the NanoRacks platform aboard the ISS [74]. This technology provides a SPA-1 bus and experiment scripting capabilities for microgravity testing of SPA-1 devices on orbit [71]. Finally, the KySat-2 C&DH utilizes the SPA-1L communication layer as its communications bus protocol between processors.

The modular architecture created for KySat-2 functions as a distributed kernel, executing system and SM tasks across the command network at programmable priority and frequency. These tasks range from mission-specific ground commands executed by a LASIM in the form of an RPC to network maintenance operations performed by the SDM-L. To facilitate reuse, the layered design approach shown in Figure 12 was used; this will now be discussed in detail.

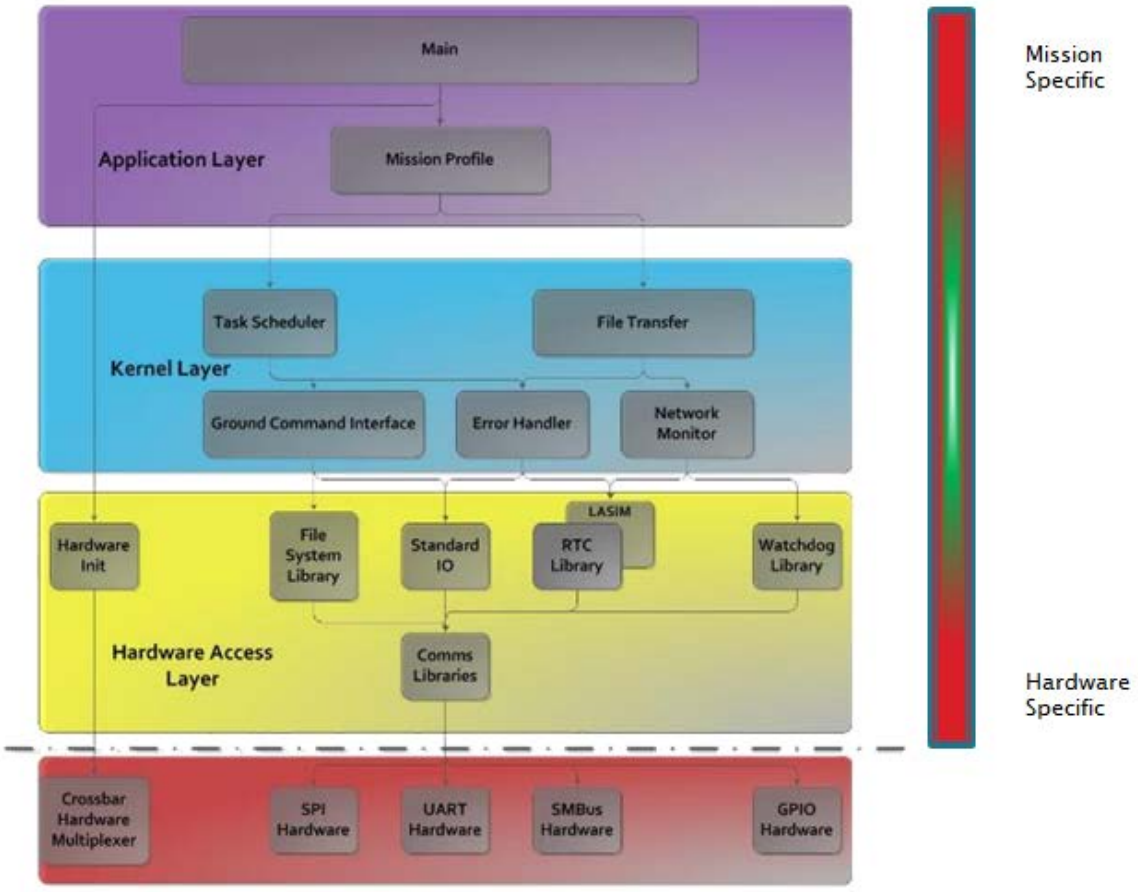


**Figure 12: KySat-2 Software Architecture [5]**

The abstracted software model of KySat-2 allows for rapid porting to multiple hardware platforms or missions due to its layered approach. At the bottom of this hierarchy resides the Hardware Access Layer (HAL), which is the only layer to directly access the micro-controller-specific subsystems. For KySat-2, this includes the SPI, UART, and $I^2C$ for serial drivers, general purpose input/output (GPIO) drivers, and hardware registers for initialization configuration; however in general this could include any number of hardware-specific peripheral drivers. The HAL also provides an API to provide software access to external peripheral support devices through use of on-chip communications drivers. The peripheral hardware libraries include the WDT library, providing a watchdog-kick function and timeout configuration; and the RTC library, providing both time and alarm setting and reading. This layer is the only purely device-specific layer, and is the only layer that would need to be re-configured for modified hardware architectures.

The next layer is the distributed kernel layer. This layer provides many reusable protocols such as network status and health management, data transfer and storage, task scheduling, and system debugging. The kernel acts as a distributed operating system by carrying out tasks both locally on the SDM-L and also through the use of specific remote procedure calls to the LASIMs, adhering to the messaging standard in Figure 6. Within the kernel, the mission task handler acts as a non-preemptive priority scheduler. Task execution timing precision is a function of both the RTC resolution and also the resulting latency from communication rates across the distributed network. This allows the kernel to be configured for a wide range of requirements including sub-second task execution resolution at approximately 30Hz and facilitates ground command scheduling periods up to 30 days in the future in its current revision. Due to the non-preemptive nature of the kernel, the task scheduler maintains a requirement that no individual thread or task contain a critical section longer than the WDT period. This allows the WDT kick subroutine to only be called by the scheduler, thus increasing the redundancy of the software by eliminating the ability for a potentially hung task to kick the WDT.

One of the primary features of the kernel layer is the network monitor in Figure 13. The network monitor is responsible for maintaining the status of all the LASIMs on the network and take corrective action if an error is encountered. The network monitor functions by successively sending each LASIM a report status message. After each network member has been sent the message, the SDM-L then reads the status of each LASIM and takes an appropriate action. LASIMs report to the SDM-L their current task status or completion, general health, or in the case of the Radio LASIM, a pending up-link packet to be processed. The general API of these

status messages are enumerated in Figure 14. The purpose of this round-robin style polling of each peripheral processor is to uniformly limit the latency during task execution. Each communication transaction, each remote RPC, etc. to any LASIM rides on the network monitor process, ensuring each LASIM is visited periodically no matter what task is executing and no matter how long that task takes to execute.

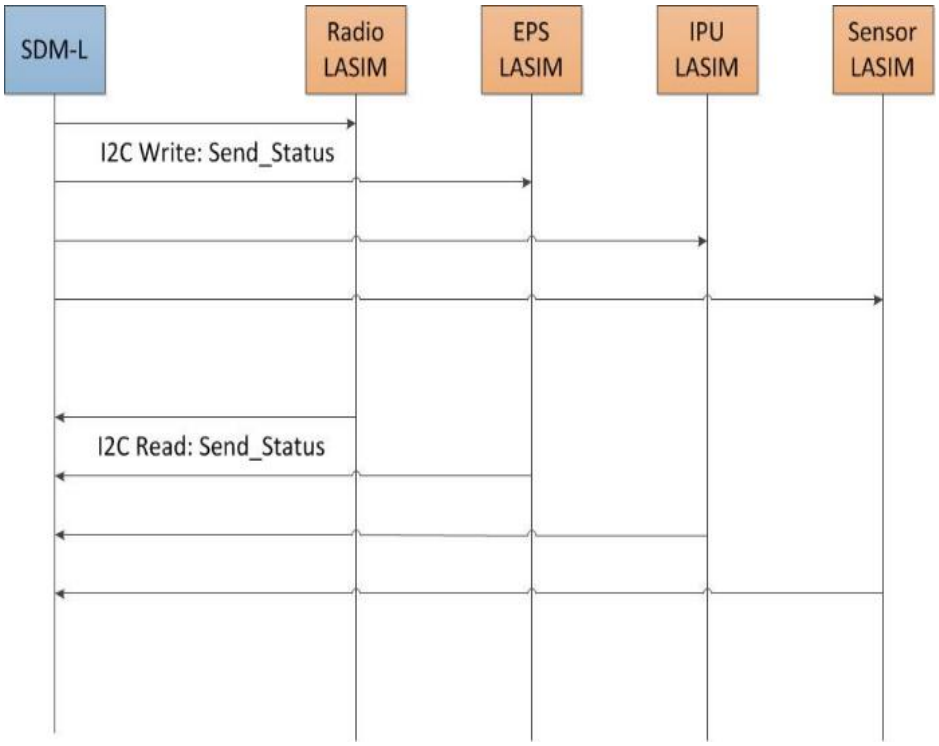


**Figure 13: Network Monitor Process [5]**

| Send_Status() | Data | CKA | CKB |
|---|---|---|---|
| 'S' | 1 byte | 1 byte | 1 byte |
| Notes: | Don't Care | | |

| Send_Status() | Data (3 bytes) | | | CKA | CKB |
|---|---|---|---|---|---|
| 's' | 1 byte | 1 byte | 1 byte | 1 byte | 1 byte |
| | RDY? | CMD | SIZE | | |
| Notes: | Data Ready = 0xFF<br>No Data = 0x00 | Response CMD | Size of data | | |

**Figure 14: (Blue) SDM-L Network Monitor message and (Red) LASIM response message [5]**

To keep track of these LASIM status messages, the kernel layer also provides a general LASIM structure that contains parameters for each individual LASIM. This equips network monitor with a configurable allowable LASIM latency before corrective action is taken in the event of an unresponsive subsystem. When the SDM-L receives a corrupt packet from a LASIM, there are three possible conditions:

1. The physical characteristics of the $I^2C$ network has dropped or corrupted the packet.
2. The LASIM is executing a critical section of code, and has turned off communication interrupts in order to avoid data corruption.
3. The LASIM is in a fault condition.

The first condition stems from the physical connection over the $I^2C$ bus, which does not guarantee successful delivery of data. If $I^2C$ data is corrupt from a LASIM, the SDM-L first assumes the lossy connection is to blame, and immediately re-requests the data. If the LASIM continues to either provide corrupt data or is unresponsive, the SDM-L proceeds to assuming the second condition. This condition represents expected corruption indicating a temporary suspension of communications interrupts. This allows each LASIM to execute critical sections of remote procedure calls without disrupting network performance or management. If a LASIM is unresponsive during successive network monitoring sessions, the SDM-L allows that LASIM to carry on until it reaches its configurable maximum latency. If this limit is reached, the third condition is assumed to be the case. The LASIM is flagged as a runaway device and is hardware reset. The LASIMs themselves are required to start up in an environment-independent state, meaning they are able to recover from any number of hardware resets. The ability to recover in known states increases the robustness of the network, but incurs additional overhead for the SDM-L, since the hardware reset aborts any commands being executed and requires resending the command.

One limiting factor of any communication system is its protocol-limited max transfer size and speed. The kernel manages this limitation by including a data management protocol, depicted in Figure 15, to facilitate transporting and reconstructing large files both around the network and down-linked over the air. This protocol decomposes a large file into individual meta data-encoded packets for transfer, each transfer corresponding to the data size of a radio packet. This allows for seamless packetization of data, and allows for straightforward data request both

from the SDM-L and from the ground. Transfer of files uses the network monitor process from the kernel layer, transferring only one packet at a time before polling the other LASIMs and checking the scheduler. As a result, the transfer of small files is no different than the transfer of large files in terms of latency or impact on the system's resources, preventing starvation of other LASIMs and allowing for multiple tasks to be executing. On KySat-2, this protocol is used to transmit both inertial measurement data and captured images to the ground for processing.

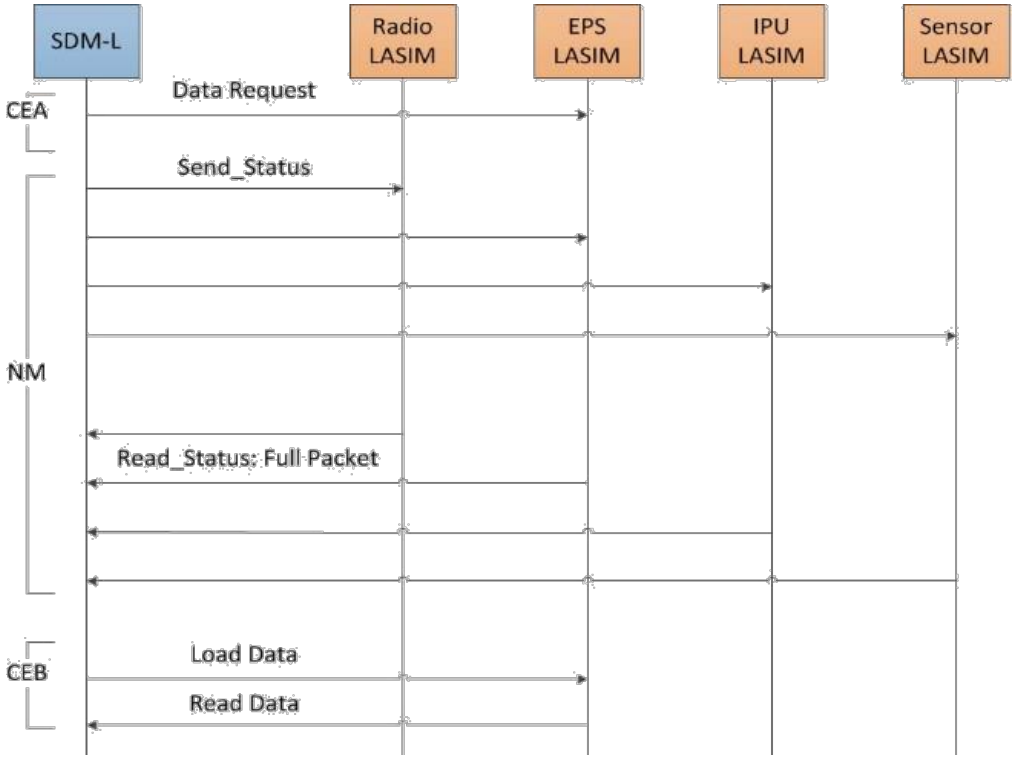


**Figure 15: Data Transfer Exchanges [5]**

This data exchange process is shown in Figure 15, and is split into three phases. The first is Command Execution A (CEA), which is executed when a mission profile command in the SM layer wishes to retrieve data from a LASIM. This phase consists of sending the RPC command to gather the required data from the LASIM's subsystem. The SDM-L then drops into the Network Monitor (NM) phase, which is just the normal network monitor routine, checking the status of the LASIMs and executing normal health and status routines. Once the LASIM completes its task and is ready for the data to be read, it notifies the SDM-L in its network monitor status response by giving the full packet flag, the API command ID it is responding to, and the size of the data to be read. This prompts the command to be rescheduled and the Command Execution B (CEB) phase to execute, which sends the RPC commands to load and read the data. Data transfer

proceeds in this manner until the desired file has been transferred. This phased approach makes the size of the total file irrelevant to the process, which transfers one packet a time before tending to the health and status of the satellite. The transfer protocol takes advantage of the introduced polling latency, allowing the LASIM time to gather and load the next packet of data for transfer, requiring no computational downtime at the master PE. By transferring data in stages, the normal health and status and scheduler maintenance operations continue to function regardless of the file size, yielding a more responsive and fault tolerant system.

The highest software layer is the SM layer. The SM layer houses the variable mission specific functionality such as ground command processing and LASIM RPC definitions. Each LASIM contains two sets of command APIs: internal network operations and subsystem-specific operations. The internal network operations are handled by the network monitor functionality in the kernel layer; however, the subsystem-specific operations are handled here in the SM layer. This includes the functionalities for operating each subsystem in order to implement the mission. For KySat-2, these include: powering on and off different voltage rails through the EPS LASIM, taking pictures with the IPU LASIM, gathering sensor data with the Sensor LASIM, and transmitting files through the Radio LASIM. As mentioned previously, one of the purposes of the distributed C&DH approach was to allow an SM-level programmer to utilize the subsystem-specific operations API to abstract away hardware knowledge and complexities, yielding more time for other SM-level programming. Furthermore, it should be noted that the subsystem-specific operations API is specific to the general subsystem, not the implementation of the subsystem. For example, the EPS LASIM does not imply the specific design or model number of the EPS to the SM-level programmer. Rather, it implements the functionality of that subsystem and could be changed to accommodate a different kind of EPS that makes no difference to the mission.

### 3.2.5 Avionics Full-Duplex Switched Ethernet (AFDX Ethernet)

Avionics Full-Duplex Switched Ethernet (AFDX) was developed by Airbus for the Airbus A380 passenger airliner, and is in use as well on the Boeing 787 Dreamliner [75]. It is used to link the processing elements and route messages and data in highly safety-critical systems; for example, it is the data bus that links the aircraft cockpit, cabin, utility measurement and management systems, and energy systems on the Airbus A380.

The goal of AFDX is to answer the need for a more robust and faster network to support next-generation Aircraft Data Networks (ADN). These ADNs must exhibit improved quality of

service, speed, and cost over the previous generation. In order to support these AFDX is based on 10/100 Base-T Ethernet, giving it 10 or 100 Mbps speeds, copious commercial support for development and testing, and a proven data delivery infrastructure and existing suite of SMs. Specifically, AFDX use twisted pair copper wires and fiber optics for the physical layer, Ethernet framing and MAC protocols for the data link layer, Internet Protocol (IP) for the network layer, and UDP for the transport layer. AFDX also adds two additional features to conventional Ethernet: deterministic timing and redundancy management.

Deterministic timing is provided by defining virtual links between PEs. These virtual links specify the maximum bandwidth, bounded latency, and frame size of those links, allowing configuration tables to be made routing information along the links that meet the required message delivery parameters for different kinds of data. Redundancy management is provided through a required duplicate network. Transmitting PEs send the same data onto both networks, and receiving PEs discard duplicates only when successful delivery occurs. This management is handled by separate integrity checkers in the data link layer as the data arrives, with the redundancy management routine eliminating the redundant frames before passing them to the network and transport layers above [76].

### 3.2.6 Time-Triggered Controller Area Network (TTCAN)

Time-Triggered Controller Area Network (TTCAN) extends CAN to be time-triggered instead of event-triggered. This extension was completed by Bosch, the inventor of CAN, and has been standardized as ISO 11898-4 as an additional layer on top of CAN.

TTCAN still allows for event-triggered transmission because it is just an additional layer on top of CAN's functionality. For this aspect, TTCAN still uses carrier sense multiple access with collision detection and arbitration on message priority (CSMA/CD+AMP), which means that messages with the lowest ID are transmitted first when multiple PEs attempt to transmit simultaneously. For the time-triggered aspect however, a system matrix defines a messaging schedule, and a single PE is designated as the time master. This time master PE sends a reference frame periodically, kicking off the messaging schedule cycle [48] [77]. As such, TTCAN is not masterless, and has slower transmission speed than other architectures, at 1MB/s. TTCAN, while specified for the automotive industry, is used in aerospace SMs as well.

### 3.2.7 CAN-Aerospace

Controller Area Network-Aerospace (CAN-Aerospace) was established in 1997 by Stock Flight Systems, and was standardized in 2001 by NASA as the AGATE Data Bus and in 2007 by Aeronautical Radio, Incorporated (ARINC) as ARINC 825 [78].

The goal of CAN-Aerospace is to enhance the CAN protocol for use on safety-critical avionics. CAN-Aerospace provides further definition and handling of CAN frames, and specifies timing requirements and connectors/cables. A network using CAN relies on broadcast messaging, meaning all PEs on the network see the message and any PE on the network can initiate a message. However, this can lead to extra processing time being wasted by PEs who shouldn't be parsing the message to determine whether or not to respond. CAN-Aerospace provides a peer-to-peer (PTP) mechanism to allow for individual PEs to act as clients and servers. To accomplish this, CAN-Aerospace implements a Logical Communication Channel (LCC) layer that groups messaging types and priorities. The LCCs distinguish between broadcast messages (anyone-to-many: ATM) and PTP messages. These LCCs decompose into seven different channels with descending priority, enumerated in Table 8. This allows for PEs with lower computational capabilities and power requirements, such as those featured on avionics platforms, to avoid the typical CAN communication layer.

**Table 8: CAN-Aerospace LCCs [72]**

| LCC | CAN ID Range | Communication Type | Coordination |
|---|---|---|---|
| **Emergency Event Data** | 0-127 | ATM | Asynchronous |
| **High Priority PE Service Data** | 128-199 | PTP | Either |
| **High Priority User-Defined Data** | 200-299 | ATM | Synchronous |
| **Normal Operation Data** | 300-1799 | ATM | Either |
| **Low Priority User-Defined Data** | 1800-1899 | ATM | Synchronous |
| **Debug Service Data** | 1900-1999 | Either | Either |
| **Low Priority PE Service Data** | 2000-2031 | PTP | Either |

In addition to the LCCs that give CAN-Aerospace a P2P mechanism, CAN-Aerospace also extends the CAN frame structure to be self-identifying. The standard CAN frame has an 11-bit CAN identifier identifying the message, followed by bit flags and up to eight bytes of data [57]. The CAN-Aerospace frame still has the 11-bit CAN identifier and bit flags, but specifies the

first four bytes of the data field as the node-ID, the data type, a service code, and a message code, respectively. The node-ID is the software-defined address of the PE, with a "0" being broadcast and extending up to 255 possible individual PEs. The data type informs the receiving PE on how to interpret the data. The service code contains eight single-bit flags that can be used to determine the state of the transmitting PE and the data itself. The message code is a counter that increases monotonically for each message, allowing for the sequence of messages to be monitored and arranged if necessary.

Finally, CAN-Aerospace addresses timing considerations by offering deterministic timing through a time-triggered architecture. CAN-Aerospace allocates a finite period of time during which each PE may transmit messages; this time may vary from PE to PE. This is similar to TTCAN's provision of a time-triggered architecture, but does not explicitly use TTCAN [78].

### 3.2.8 Middleware Aspects

The degrees to which each HI-Network layer middleware address the five key aspects of middleware will now be compared and contrasted. The results are summarized in Table 9.

**Table 9: Comparison of HI-Network Middleware**

| | Network Communication | Coordination | Reliability | Scalability | Heterogeneity |
|---|---|---|---|---|---|
| **SpaceWire** | Node-oriented | Synchronous | At-most-once | Location | Hardware, Software |
| **MAVLink** | Message-oriented | Asynchronous | At-most-once | Location, Replication | Hardware, Network, Software |
| **SDM-Lite** | Node-oriented | Asynchronous | At-most-once | None | Hardware |
| **SPA-1L** | Node-oriented | Asynchronous | At-least-once | Location | Hardware |
| **AFDX** | Node-oriented | Asynchronous | Exactly-once | None | Hardware, Software |
| **TTCAN** | Message-oriented | Asynchronous | At-least-once | Location, replication | Hardware, Software |
| **CAN-Aero** | Both | Both | At-least-once | Location, partial replication | Hardware, Software |

#### 3.2.8.1 Network Communication

SpaceWire handles network communication through a node-oriented architecture by specifying the exact network communication medium, link, encoding, packetization, and routing. PEs transmit packets containing only a destination address, leaving SpaceWire routers to determine the path and intermediate addresses required for the packet to reach its destination.

MAVLink handles network communication by providing a user-implemented function for basic transmit/receive functionality. This function is abstracted from the MAVLink message marshalling implementation, and can use any serial communications protocol that allows for broadcast messages desired by the user. MAVLink messages are message-oriented, containing a message ID that is interpreted by all receivers to determine whether or not they need to process the message.

The SDM-Lite handles network communication by necessitating a node-oriented architecture, implying an addressing scheme. The SDM-Lite targets SPA-1 simplification, requiring $I^2C$ from the HI-Hardware layer as the network protocol. This protocol supports a master-slave, addressed network. The SDM-Lite abstracts each ASIM's physical address by assigning each a logical address, allowing for network reconfigurations while maintaining a static address to higher-level tasks.

SPA-1L handles network communication very similarly to the SDM-Lite: an $I^2C$ master-slave, node-oriented network with each LASIM's physical address and location abstracted to a logical address. Additionally, SPA-1L provides checksums and failed response counts for each message and LASIM for error handling and reliability.

AFDX handles network communication by specifying IEEE 802.3 Ethernet for its physical and data link layers. Ethernet relies on twisted pair copper wire and fiber optic cables to transmit bits, and implements the MAC protocol for hardware addressing. AFDX frames are transmitted by end systems and are routed through AFDX switches to other end systems.

TTCAN handles network communication by adding a time master to implement a time-triggered variant of CAN. The time master provides a periodic reference frame, kickstarting a predetermined allotted time for each PE to perform messaging in turn. The frames themselves are still purely message-oriented, with messages broadcast during the PE's allotted time to all PEs, which then process the message for the CAN identifier to determine whether or not to respond during their upcoming time slot.

CAN-Aerospace handles network communication by relying directly on CAN for ATM messaging and by providing a node-addressing scheme for PTP communications. This node-addressing scheme redefines the first four bytes of CAN frames to incorporate self-identification, allowing for less processing power and time to be wasted by PEs not requiring the message. This provides a scaled-down solution for resource-constrained, safety-critical components often seen in avionics platforms.

#### 3.2.8.2 Coordination

SpaceWire handles coordination through synchronous-only messaging. SpaceWire itself is not concerned with the meaning of any data characters, and thus does not contain any implicit mechanisms for ensuring coordination between tasks on different PEs. The SpaceWire physical and signal levels feature a clock signal recovered by XORing the strobe and data signals, ensuring that the receiving PE is ready to receive the message from the transmitting PE. As a result, both the transmitting and receiving PEs must block and maintain this clock signal for the duration of the message.

MAVLink handles coordination by abstracting the communications protocol and leaving its implementation to the user. Instead of specifying this protocol, as in SpaceWire, SPA-1-, and CAN-variants, MAVLink performs message marshalling and packetization while assuming that the user will supply the HI-Hardware level communications. MAVLink's coordination is thus asynchronous.

The SDM-Lite presents an interesting situation: since it relies on an HI-Hardware layer middleware, I2C, it handles coordination through synchronous-only messaging. However, the SDM-Lite implements a pseudo-time-triggered architecture where each ASIM is allotted its period of time to respond to SDM-Lite requests, and the SDM-Lite doesn't block or wait for that ASIM to respond. This period of time is equal to the period of time required for the SDM-Lite to service the other ASIMs, continuing the round robin and allows the data handling and process information tasks to complete. So while the SDM-Lite uses a synchronous middleware for its HI-Hardware layer interactions to send individual messages, it is classified as an asynchronous middleware for how it handles HI-Network layer interactions.

SPA-1L handles coordination in the same way as the SDM-Lite: as a pseudo-time-triggered architecture, where each LASIM is commanded and given the length of a round robin cycle to execute the response. Despite its use of I2C from the HI-Hardware layer, SPA-1L as a

HI-Network layer middleware extends the provisions of I2C making it asynchronous for node-to-node communications.

AFDX handles coordination through asynchronous communication. It is based Ethernet and uses Ethernet's default coordination, with no shared clock signal between PEs and messages requiring no acknowledgement or pause in execution from transmitting PEs.

TTCAN handles coordination through the asynchronous messaging of CAN. Each PE broadcasts its message onto the CAN bus during its allotted time, and does not wait for an acknowledgement or response from any receivers and can continue execution.

CAN-Aerospace handles coordination through the provision of the LCCs. These LCCs each offer different levels of coordination; the Emergency Event Data channel is the only asynchronous-only channel, and the two User-Defined Data channels are the only two synchronous-only channels; the remaining channels offer both asynchronous and synchronous coordination according to user selection.

#### 3.2.8.3 Reliability

SpaceWire's reliability is at-most-once. It handles reliability both through its end-to-end layer specification of packet transport and its exchange level. By specifying the end-to-end mechanisms for packet transport, the designers of SpaceWire can assume compliant and EMC-tested connectors, cables, circuit board routing, and signal noises and levels. Additionally, the exchange level makes use of the character level control codes to offer fault detection services; however, SpaceWire only detects and reports these faults, leaving the decision to the user of whether or not to attempt to correct the error. Once detected, SpaceWire reports the error, reestablishes the link, and transmits the next packet to avoid duplication. Thus SpaceWire does not guarantee the validity of the transmission, but does guarantee that no duplicate messages will be sent.

MAVLink's reliability is at-most-once. It relies on user-provided transmit/receive functions, and natively only provides for fault detection through the ITU X.25 checksums and the sequence number natively included in MAVLink frames. It is up to the user to verify these checksums and re-request faulty frames.

The SDM-Lite's reliability is at-most-once. Since it relies on $I^2C$, the SDM-Lite is limited by $I^2C$'s inherent lack of guaranteeing reliable communication. Messages to each ASIM are transmitted once, but not guaranteed to reach the destination correctly. While ASIMs could

potentially fail to acknowledge individual bytes per the $I^2C$ specification, indicating network or coordination faults, there is no knowledge that the message contents arrived correctly.

SPA-1L's reliability is at-least-once. Similar to the SDM-Lite, it relies on $I^2C$ with its inherent lack of guaranteeing reliable communication. However, SPA-1L adds a software acknowledgement features that includes Fletcher's checksums in packets, with mismatched checksums prompting notification from the slave device and triggered retransmission from the master device. This provides for at-least-once reliability, since the message is delivered correctly but may take many transmissions to do so. Additionally, SPA-1L maintains counters for the number of times each LASIM fails to acknowledge during the $I^2C$ acknowledgement byte, and resets the LASIMs that exceed their maximum allowable number of failed acknowledgements. This provides for fault handling on a per-round robin basis; while this sensitivity minimizes the amount of time a LASIM experiences the fault condition, it necessitates design-time cognizance of reset handling.

AFDX's reliability is exactly-once. It handles reliability by extending Ethernet to include deterministic timing and redundancy management. The deterministic timing stems from the definition of virtual links that define the bandwidth capabilities and latency for each link, allowing for traffic policing and fault containment if a switch ever fails. The redundancy management stems from requiring an identical redundant network, and an integrity checker that compares data sent over both channels to ensure reliable transport. For exactly-once reliability, AFDX guarantees both successful message delivery and no duplicate packets, with the integrity checker/redundancy management ensuring message validity and a sequence number allowing receivers to guard against duplicate messages.

TTCAN's reliability is at-least-once, mirroring CAN's reliability from the HI-Hardware layer. TTCAN provides an extension onto CAN to make it a time-triggered architecture, providing a master time base and allocated a unit of time for every PE to control the bus and send messages. This does not change the reliability guarantee of CAN, but does aid to ensuring that every PE will gain priority access to the bus and will be able to send messages with no collisions, improving latency.

CAN-Aerospace's reliability is at-least-once, again mirroring CAN's reliability from the HI-Hardware layer. While is does not increase the reliability guarantee, CAN-Aerospace provid ea further extension to CAN including the Message Code field of the CAN-Aerospace message header. This Message Code increases monotonically with each CAN-Aerospace frame, similarly

to a MAVLink frame's sequence number; this message code allows for detection of missing frames, and determines the age of a frame if an identical frame is delievered multiple times. While this allows for exactly-once reliability detection by the user, CAN-Aerospace does not natively handle or interpret this Message Code.

#### 3.2.8.4 Scalability

SpaceWire optionally implements location transparency. PEs are addressed using either path addressing or logical addressing. Path addressing means that the destination address of the intended receiving PE is encoded as the series of output ports on each router along the path that the message must be forwarded from; this implies strict knowledge of the location of the receiving PE. However, logical addressing may optionally be used, where each PE in the network is assigned an address 32-255, and each router maintains a routing table on how to route messages to each PE. This logical addressing represents the ability for SpaceWire to be location transparent.

MAVLink implements location and replication transparency. As a broadcast-only network with no designated endpoints in messaging, the locations of receiving PEs are unknown and irrelevant to the SMs. Furthermore, redundant PEs need only identify the proper messages to respond to and be fully MAVLink-compliant, again with no knowledge from SMs.

The SDM-Lite does not implement any form of transparency. ASIM addresses are hardcoded and must be directly known by SMs. By using $I^2C$, the network itself is scalable up to the theoretical address limit of $I^2C$, meaning 255 possible devices, and the practical bus capacitance limit from traces and the number of devices.

SPA-1L implements location transparency. Each LASIM on the network is assigned a logical address, meaning that SMs do not know the hardware addresses. This is the only form of transparency implemented. Like the SDM-Lite, SPA-1L relies on $I^2C$ and can theoretically support up to 255 PEs, though practically far fewer due to bus capacitance.

AFDX does not implement any form of transparency. AFDX adopts and extends Ethernet, meaning that end systems need only be Ethernet-compliant. However, the AFDX switches, while based on Ethernet switches, extend Ethernet and include traffic policing and bandwidth monitoring. Thus, the switches are restricted to AFDX switches, and the traffic will be subject to bandwidth restrictions that guarantee delivery and AFDX-compliant latency.

TTCAN implements location transparency and replication transparency due to its reliance on CAN. TTCAN extends CAN into a time-triggered architecture, but does not reduce or enhance the scalability because of CAN's message-oriented network. New or redundant PEs can be added to the network with no knowledge from other PEs, since all PEs receive all messages on the bus.

CAN-Aerospace implements location transparency and partial replication transparency due to its reliance on and extension of CAN features. CAN-Aerospace extends CAN into a time-triggered architecture, similarly to TTCAN, but introduces the LCCs that offer both standard message-oriented networking and a new node-oriented networking. Location transparency exists in both the generic message-oriented (called ATM) channels and the new node-oriented (called PTP) channels, since all PEs receive the message in the ATM channels and all PEs are addressed by logical PE identifiers in the PTP channels. However, replication transparency only exists in the ATM channels, and not in the PTP channels, because those replicates would require their own unique identifier to be addressed in the PTP channels.

#### 3.2.8.5 Heterogeneity

SpaceWire exhibits hardware and software heterogeneity. By completely defining the character encodings and exchange parameters above the physical level, SpaceWire can be run on multiple architectures and has hardware heterogeneity. Furthermore, SpaceWire exhibits software heterogeneity because it can be called from any SM to transfer data to other PEs on the network, supports multiple operating systems including VxWorks, Linux and Windows, and is contains C, C++, and Java APIs [64].

MAVLink exhibits hardware, network, and software heterogeneity. As a header-only library included at compile-time, hardware is only restricted to that hardware that can execute the code and has some network communication method for serial communication. This allows MAVLink to run on both microcontrollers, including most popular 8-bit and 32-bit architectures, and desktop computers for ground stations. MAVLink exhibits network heterogeneity because it abstracts serial communication, relying on the user to provide the implementation-specific code transfer MAVLink frames off-chip. Finally, MAVLink exhibits software heterogeneity because it is not restricted to operating systems or SMs, and its code generator allows output into a variety of programming languages including C, Python, and JavaScript.

The SDM-Lite exhibits hardware heterogeneity. The goal of the SDM-Lite was to strip down full SPA SDM functionality so that it could run on a lower-power network, while retaining compatibility with normal SPA networks. Accomplishing this goal, the SDM-Lite is able to run

on 8-bit and 32-bit microcontrollers as long as they allow $I^2C$ communications; additionally, the SDM-Lite is compatible with SPA networks running on more powerful hardware, provided xTEDS registration and network enumeration.

SPA-1L exhibits partial hardware heterogeneity. SPA-1L builds on the SDM-Lite, inheriting the SDM-Lite's requirement of $I^2C$ communications as the only requirement between processors. SPA-1L is not compatible with other SPA networks on more powerful hardware, however, and has only been demonstrated on a network of 8-bit 8051-archtiecture microcontrollers.

AFDX exhibits hardware and software heterogeneity. Since AFDX is Ethernet with additional reliability and fault-tolerance, it exhibits Ethernet's heterogeneity traits.

TTCAN exhibits hardware and software heterogeneity. Since TTCAN is CAN with a time-triggered architecture, it exhibits CAN's heterogeneity traits.

CAN-Aerospace exhibits hardware and software heterogeneity. Since CAN-Aerospace is CAN with a node-oriented, unicast capability in addition to normal CAN, it exhibits CAN's heterogeneity traits.

# 4 Distribution Layer Middleware

This chapter details distribution layer middleware. Distribution layer middleware is above host-infrastructure middleware, and extends the encapsulations provided by the host-infrastructure layer. These extensions enable end-to-end transport of data from SMs, and provide additional fault-tolerance because unlike host-infrastructure middleware, distribution middleware is concerned with the meaning of the bytes being transmitted, and can detect errors or anomalies and take corrective action. Middleware in this layer is divided into two classifications: transport and object request broker.

## 4.1 D-Transport Layer

The D-transport layer describes middleware that account for transportation of data between SMs on a network of PEs without using object request brokers (ORBs). Whereas the HI-network layer was concerned with transmitted bytes between PEs, the D-transport layer is concerned with transmitting meaningful messages between the SMs running on PEs. Since middleware in this layer actively knows the meaning of messages instead of simply transmitting the bytes, the middleware surveyed here targets fault-tolerance and distributed network knowledge and management. The middleware to be reviewed in this layer include Space Plug-and-play Avionics (SPA), Message Routing Layer (MeRL), Automatically Reconfigurable Dependable Embedded Architecture (Ardea), MIL-STD-1553, and LonTalk.

### 4.1.1 Ardea

The Automatically Reconfigurable Distributed Embedded Architecture (Ardea) was developed by the SSL at UK in 2005. It is targeted toward low-power distributed embedded systems, particularly those found linking the instrumentation and control surfaces of UAVs [79]. Its use can also be extended to small satellites.

The primary fault-tolerant concept under study, and the driving factor in the creation of Ardea, is graceful degradation. This concept specifies a system that reconfigures in response to faults in hardware and/or software such that the system exhibits reduced quality and/or capability instead of total failure. Ardea addresses this concept with its central feature: software module dependency graphs (DGs). These graphs provide a graphical representation of the software and hardware dependencies, making recomputation of dependencies possible when any of the hardware or software components fail. Figure 16 depicts the hardware architecture of an Ardea system.

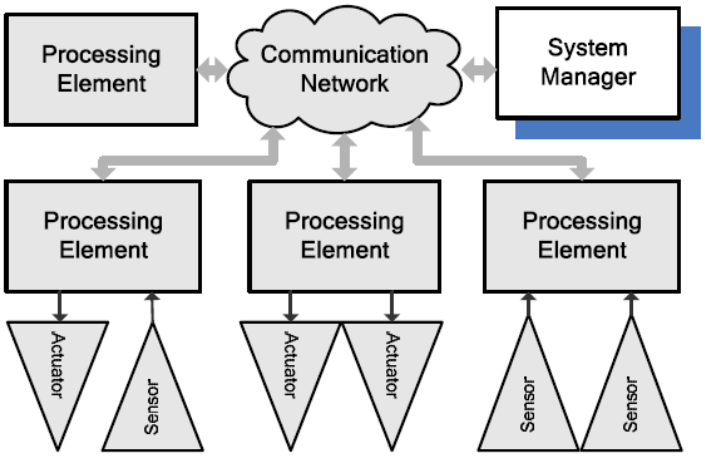


**Figure 16: Ardea Architecture [79]**

The Ardea system consists of four primary elements: processing elements, the communication network, the system manager, and input/output (I/O) devices.

#### 4.1.1.1 Processing Elements

Processing elements are computational units/resources, assumed to be homogeneous, and hold the software modules which produce and consume data for the system. The processing elements consist of local management tasks and an RTOS, as well as any unique SM software required to produce or consume requisite data.

#### 4.1.1.2 Communication Network

The communication network allows for bidirectional data flow between processing elements, and with the system manager. There are two sets of messages on this network: SM data, which consists of data variables that are produced and consumed by processing elements, and management messages, which consist of messages between processing elements and the system manager. These management messages can be further split into two groups, depending on the direction of flow: messages from a processing element to the system manager consist of status messages and fault reporting, as well as any DG modification commands; and messages from the system manager to processing elements consist of state data, scheduling commands, and module object code if that module needs to be reconfigured in the event of faults. SM data is periodic in nature, being produced and consumed in orderly, routine fashions by processing elements. Management messages, however, have priority access to the network since they result aperiodic events, with the exception of heartbeat messages.

#### 4.1.1.3 System Manager

The system manager tracks the status of hardware and the availability of software resources to ensure that no reconfiguration is required. If faults occur, new configurations are computed and deployed. The system manager also handles state data and checkpointing from processing elements.

#### 4.1.1.4 I/O Devices

I/O devices are the system's interface with the outside world, reading inputs to measure aspects of the system's interaction with the physical world and driving outputs to change the system's interaction with the physical world. These can be monitored for correct operation, and replaced by redundant units if available.

The above four primary elements represent the hardware in the Ardea system. These host the software units of the Ardea system, which are the components of the DG. These include software modules, data variables, dependency gates, and I/O devices.

#### 4.1.1.5 Software Modules

A software module is a "quantum of executable machine code that is schedulable on a processing element." The attributes associated with software modules are: unique ID, execution time, and output rate factor. Software modules produce and consume data variables.

#### 4.1.1.6 Data Variables

Data variables are produced and consumed by software modules, and come in two forms: state data variables, which specify state information about the software module, are required to start or restart a software module in the correct state and are stored both locally by the processing element and globally by the system manager; and management data variables, which are the fault-reporting and DG-modification mechanism, and are only consumed by the system manager. The attributes associated with both types of data variable are: ID, size, quality, and fail-safe value.

#### 4.1.1.7 Dependency Gates

Dependency gates resemble digital logic gates, and specify the dependence of software modules on data variables. There are a total of four gates that comprise the Ardea framework: K-out-of-n OR gates, AND gates, XOR gates, and DEMUX gates. K-out-of-n OR gates accept any of n inputs, as long as there are enough for k outputs. AND gates require all of the data variable inputs. XOR gates require exactly one of the inputs. DEMUX gates have one input and two or more outputs, and allows for redundant outputs of other gates.

#### 4.1.1.8 I/O Devices

I/O devices are the endpoint sources and syncs for software modules. The attributes associated with I/O devices are: criticality, priority, status, and rate. These attributes allow the system manager to ensure that required input/output rates are met, as well as prevent overloading processing elements or network bandwidth.

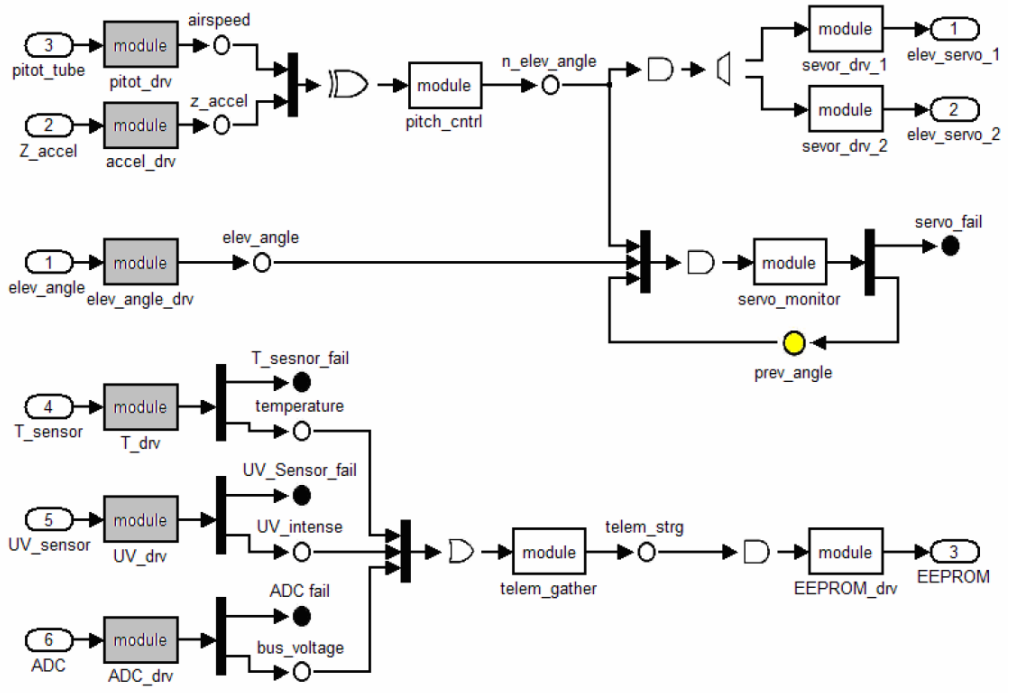


**Figure 17: Example Ardea DG [79]**

Figure 17 depicts an example dependency graph for subsystems in a UAV. The top graph represents an airspeed measurement system. Two different sensors are used to measure the airspeed: pilot_tube and Z_accel. Exactly one of these measurements are used in the pitch_cntrl software module, producing an elevation angle. This elevation angle is fed into both the servo_monitor and servo drivers. The servo_monitor requires three data variable inputs: n_elev_angle, elev_angle, and the prev_angle state variable to determine the servo fail state.

The bottom graph is a telemetry recording system, accepting any (or all) of three data variables: temperature, UV_intense, and bus_voltage. These are fed to the telem_gather software module, which produces telem_strg. This telem_strng is required for the EEPROM_drv software

block, which stores the variable to electrically erasable programmable read only memory (EEPROM).

The above example highlights several potential fault detection methods, and others are available. One such option is N-redundancy and voting, where several versions of the same I/O device drive identical software modules, with a “voter” software module choosing the reading that is most common between the devices. Another method is basing the selection of data based on quality.

At run-time, each processing element is running three common tasks: a network interface task, a scheduler task, and a memory loader task, with mailboxes controlling data variable flow between these and software modules. The network interface task manages access to the communication network between processing elements. The scheduler task schedules and unschedules SMs, including those from local memory or those mandated and sent by the system manager in the event of system reconfiguration. Finally, the memory loader task handles reconfiguration of the processing element, as directed by the system manager.

### 4.1.2 Message Routing Layer (MeRL)

The Message Routing Layer (MeRL) was created by the SSL at UK in 2009. MeRL is an implementation of the communication network component of Ardea. The goal of MeRL is to provide a generic interface for message-passing that abstracts the endpoint location from the sending SM. The approach to designing MeRL first defines the embedded distributed network envisioned by specifying the network style and protocol. The network styles available are broadcast/multicast, where messages sent over network are visible and read by all PEs, and are the same to all PEs; and point-to-point, where messages are only sent to specific targets. The broadcast/multicast scheme was chosen for this work, so that faults cannot be blamed on messages being delivered inconsistently to different PEs. This style also cuts down on the number of global messages that need to be sent. Next, the protocol format was chosen from between two general options: node-oriented, where messages are addressed and are only visible to PEs identified as receivers; and message-oriented, where receivers choose whether or not to process message and are location independent. The CAN protocol is chosen as the best candidate wired protocol, and the Zigbee protocol is chosen as the best candidate wireless protocol.

With a specific network format and protocol envisioned, the requirements for the MeRL design and operation are:

1. MeRL would be a single layer for all tasks to communicate.
2. MeRL would not allow direct communication between tasks, forcing independence.
3. MeRL would pass data with no knowledge of location of receiving or transmitting task.

MeRL's operation entails all desired task communications, whether they are between different processes on the same processor or between different processors on the same network. The message is sent to the MeRL layer, which determines if any local tasks want access to the data. No matter whether a local task uses the data or not, the data is then placed on the network for other tasks on other processors. When a task receives the data, it goes into a queue of received data that that task is subscribed to. This architecture is depicted in Figure 18.

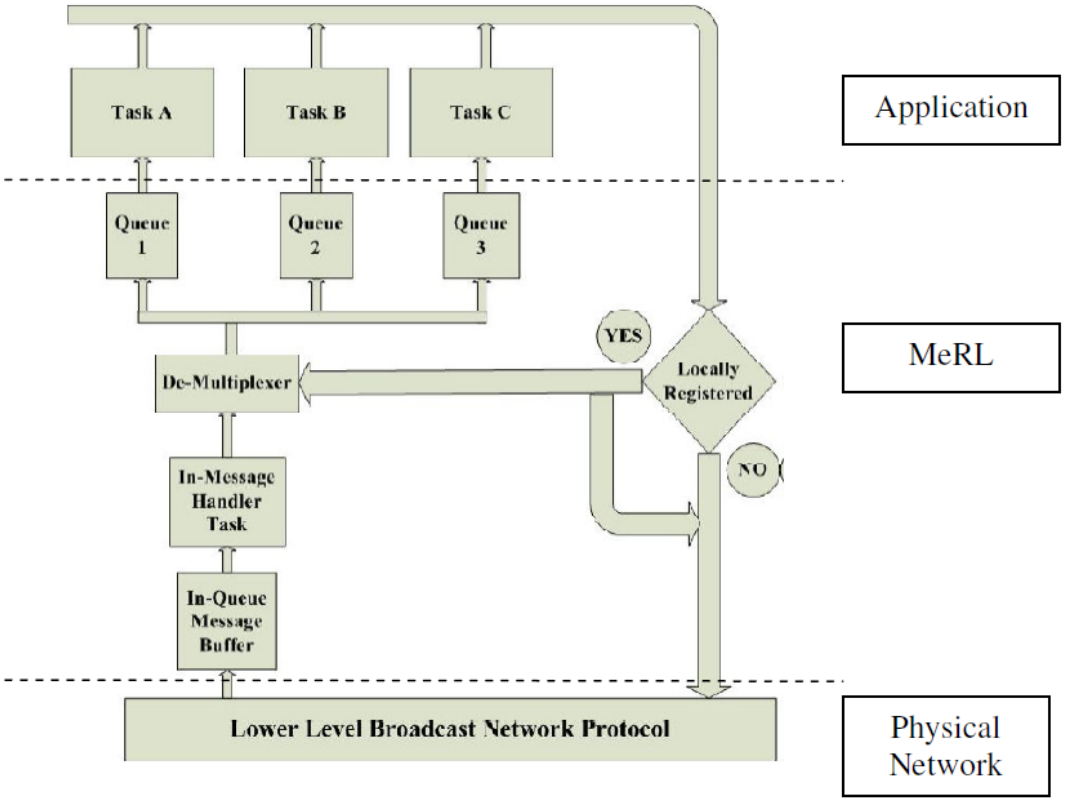


**Figure 18: MeRL message-passing architecture [44]**

For a task to have access to data, it needs to register for that data's ID. This means that all data needs to be enumerated and assigned an ID before compile-time. MeRL provides two APIs:

an SM layer API and a Network Interface API. The SM layer API is present no matter the implementation, and is used to initialize the system, register tasks to receive messages, send messages, and receive messages. This layer is called from SM code, no matter if the recipients are local tasks or other processors. The network interface API is present if the system consists of multiple processors on a network, and provides a similar set of commands to the SM layer API.

An implementation of MeRL exists and runs on a Silicon Labs 8-bit 8051F040 microcontroller running IDEAnix, a microC/OS-II port for the 8051. This RTOS is a proposed setup for MeRL integration, and thus contains #define statements that enable/disable features of MeRL, and provide for user management of memory allocation.

### 4.1.3 Space Plug-and-play Avionics (SPA)

The Space Plug-and-play Avionics (SPA) architecture is the result of years of work within the aerospace and defense communities to enhance rapid systems integration. SPA specifically aims to improve the robustness and time to integration of other popular avionics protocols, such as MIL-STD-1553, and to create a plug-and-play architecture for space SMs. SPA draws upon the design challenges and implementation of other plug-and-play approaches in other industries, such as the Highway Addressable Remote Transceiver (HART) protocol that superimposes digital telemetry on top of analog current loop measurements for precise industrial control [80]; LonTalk, used in industrial sensor networks [81] and discussed later in this chapter; IEEE 1451, a smart sensor standard that defines Transducer Electronic Datasheets (TEDS) [82]; and Universal Plug-and-play (PnP), a publish/subscribe self-organizing network standard for the PC industry [83].

There are several motivations for such a system. The cost of spacecraft construction, in time, money, and human resources, has always been exorbitantly high, requiring millions of dollars and years or even decades to reach orbit. The miniaturization of satellites, notably into micro- and nanosatellites, has alleviated this problem somewhat; however even this small satellite revolution still requires an extra reduction in development and integration timelines. Two notable terrestrial industries have embraced the PnP concept: consumer PCs with their embrace of USB and Peripheral Component Interconnect (PCI), and industrial factory adoption of PnP sensor networks through Echelon's industrial Internet of things, the LonTalk protocol. SPA forsakes simply relying on these existing PnP architectures to "adapt" existing components through interfaces to communicate on USB or PCI networks; rather, a new standard that is "built-in" to spacecraft components is needed. While SPA is only the network management aspect of this

standard, fully reconfigurable software defined radios (SDR), programmable wiring systems, malleable signal processors, and radiation-hardened components form the full PnP picture for a satellite [84].

SPA itself aims to differentiate itself from terrestrial PnP implementations by addressing special constraints more unique to space SMs. These include environmental constraints, synchronization, high power delivery, and driverless operation.

1. Environment – processing elements in space SMs must be cognizant of radiation effects, such as total ionizing dose, latchups, and single event upsets. These can temporarily wreck individual task execution, corrupt memory elements, and even destroy processing elements.
2. Synchronization – all systems in the satellite must have a "unified notion of time"
3. High Power Delivery – many terrestrial PnP implementations provide some kind of power/data bundling; however, these are not well-suited to most spacecraft power requirements (such as a 28V bus).
4. Driverless – again, terrestrial PnP implementations oftentimes require drivers to operate with new devices; this is not desirable for SPA

With these motivations and constraints in mind, an example implementation of SPA is SPA-U, the USB-based variant of SPA. The term "variant" is used because SPA-U, while borrowing the data transfer characteristics and capabilities from USB, provides additional power and synchronization facilities. Like USB, the three types of SPA-U components in a network are hosts, endpoints, and hubs. SPA-U hosts are the root of the tree-structured SPA-U network, with all communication being between hosts and endpoints. SPA-U endpoints are the PEs of the SPA-U network, and serve as the interface between the network and SPA-U devices. Finally, SPA-U hubs are similar to USB hubs, providing connections between multiple hosts and endpoints. Additionally, SPA-U hubs provide power switching to connected endpoints. Within SPA-U, two models exist. These are applique sensor interface modules (ASIM) and the satellite data model (SDM) [73].

**ASIM**

The ASIM bridges between a compliant SPA network and a user's implementation of a system, seamlessly handling the system's electronic datasheet, power requirements and management, and synchronization. ASIMs should contain the requisite circuitry and services to

adapt a device or system to the SPA network. To follow the ideal SPA design guidelines, these should include:

1. Central Processing Unit (CPU) – a processing unit of some kind is required to respond to SPA commands.
2. Non-volatile memory – the extended transducer electronic datasheet (xTEDS), which is an XML document that describes the device's capabilities and requirements, must be persistently stored on the ASIM.
3. SPA network interface – for a SPA-U network, a USB interface should be implemented by the ASIM. Similarly, for SPA-E (Ethernet) and SPA-S (Spacewire), similar interfaces should be implemented.
4. User facilities – the ASIM should provide commonly-used services in embedded SMs, such as digital and analog input/output channels and serial ports, simplifying coding and complexity on the device.
5. Power management – the ASIM should take care to power before the device, in order to enumerate on the network and provide some control over the connected device's power.
6. Clock management – in service of the synchronization goal of SPA, a 1 Hz clock pulse services to unify time-keeping on all devices in the SPA network. The ASIM should be able to manage this pulse, and keep track of pulses to provide timestamping to the device.
7. Test bypass interface – the ASIM should provide a secondary connection for in-system testing.
8. SPA software API – the ASIM should provide a simple "client-side" API for programmers setting the xTEDS values and interacting with other SPA devices.

**SDM**

The ASIM is a piece of hardware that implements both hardware and software services. The SDM is usually a similar piece of hardware, but only implements software services. These services are the primary middleware layer that implements the services of SPA. The goal of the SDM is provide an interface that allows devices to communicate with each other as processes or services, instead of physical devices. This abstraction allows for devices changing address, location, makeup, etc., all without knowledge of other devices. The purpose is to force SPA designers and users to focus on adhering to software interfaces instead of physical electrical interfaces. Specifically, adherence to a Command Data Dictionary (CDD) that describes sensors, computing resources, subsystems, etc. must be used to ensure correct, device-independent routing

of messages. To accomplish this, the SDM implements a set of five software managers: the processor manager, the data manager, the task manager, the sensor manager, and the network manager.

1. Processor Manager – This manager keeps each processor busy by checking its parent processor and executing any pending tasks. This is particularly important in view of the "reconfigurability" aspect of SPA, and allows processors to pass off executables of tasks to other processors experiencing down time.
2. Data Manager – keeps track of all data available and routes data requests and responses
3. Task Manager – indexes tasks that are both executing and pending
4. Sensor Manager – implements the PnP network interface
5. Network Manager – manages the network and creates/updates routing tables for messages, keeping track of endpoint locations as they vary

While the ultimate goal of SPA is a catalogue of SPA-compliant components, adoption of SPA has been slow after initial successes due to the restricted computing and power capabilities of small satellites. An additional SPA standard, SPA-1 based on $I^2C$, was created to answer the need for a smaller-footprint SPA implementation.

#### 4.1.3.1 SPA-1

SPA-1 is intended for use on the simplest SPA devices [4]. This variant uses $I^2C$ as its network communication protocol, since the potential majority of SPA devices are simple and lightweight enough to only require $I^2C$ data rates. However, despite the reduction in data rates, a SPA-1 network is still compliant with the other versions of SPA because SPA-1 still supports the hallmark features of SPA, including network self-enumeration and automatic discovery and self-description.

The role of the ASIM is the same as in SPA, and perhaps even more applicable as SPA-1 describes connections to inherently less capable devices, making the SPA task easier for users of devices not built to SPA specifications. The ASIM takes care of all power management for the device, as well as commanding, synchronization, and data transfer mechanics. These are envisioned as comparable to USB chips that take care of the complicated protocol-level translations, giving the end device an easier interface than that exhibited by the network.

To specify SPA-1, a design goal was formulated through which all SPA-1 decisions were passed: the new SPA variant was to minimize the size, weight, and power footprint of SPA.

Implicit in this need for minimum wires is scalability, as SPA networks are agnostic to the number of connected devices. Several different communication protocols were considered, including RS-485, SPI, $I^2C$, and the wireless protocols Zigbee, Bluetooth, and 802.11. In light of the advantages and disadvantages of each communications protocol, such as the lack of network management in RS-485, wire overhead in SPI, and power overhead and interference issues with wireless technologies, $I^2C$ was chosen since it meets the goal of having the smallest design footprint possible.

In light of the advantages and disadvantages of each protocol, as well as their utility to the SPA-1 network, $I^2C$ was chosen since it meets the design goal of having the smallest footprint possible. With the protocol chosen, the functionality was then defined. This functionality was broken into two groups of functions: common functions and device-specific functions. Common functions are commands that any SPA-1 device must respond to, and are instrumental in making the SPA-1 device a fully compliant member of the network, allowing it to be discovered and enumerate on the network. Table 10 lists these common functions.

**Table 10: SPA-1 Common Functions**

| Command | Mnemonic | Response |
|---|---|---|
| **Reset** | R | Status Message |
| **Initialization** | I | Status Message |
| **Self-test** | T | Status Message |
| **Version** | U | Version Message |
| **Time-at-tone** | O | Status Message |
| **xTEDS** | X | xTEDS Message |

Device-specific functions are those that perform the capabilities listed in the device's xTEDS. These functions consist of an interface identification byte and a message identification byte, along with any arguments for the function. In addition to these two sets of functions, SPA-1 devices must support all elements of SPA-1 network operation, which consists of three phases: address resolution, enumeration, and routine network operation.

**Address Resolution**

$I^2C$ offers no address resolution protocol (ARP), but SPA-1 implements this by assigning a global unique identifier (GUID) to each ASIM, allowing that ASIM to change $I^2C$ addresses should conflicts arise. The address resolution process involves ASIMs finding an "open" $I^2C$ address.

**Enumeration**

After the ASIM has an address and is on the SPA-1 network, the SDM initializes the ASIM, reads the version identification, performs any self-testing on the ASIM, and reads and registers the xTEDS.

**Routine**

After address resolution and enumeration, the ASIM is ready to participate on the network. The SDM performs round robin cycling on all registered ASIMs, checking for new ASIMs and performing any commanding or data requests as they arise.

The creation of SPA-1 has led to more adoption of the standard, as well as formal support for SPA services. xTEDS can now be generated using online tools, and SPA-U, SPA-S, and SPA-1-compliant devices are being offered at commercial prices by companies such as AAC-Microtec and Micro-RDC [4].

### 4.1.4 MIL-STD-1553

MIL-STD-1553 is the most widely deployed serial communications architecture. It was first published in 1978 by the U.S. Air Force for F-16s and U.S. Army Apache warfighters. Other SMs include the Space Shuttle and International Space Station (ISS), along with countless satellites currently in orbit. The frenetic pace of air systems development during the Cold War, particularly the late 1960's, saw the need for new aircraft to incorporate distributed processing to make up for the inability of period hardware to offer the speed and throughput necessary. These new distributed networks reduced the load central computers, and required the creation of a standard serial bus [85].

MIL-STD-1553 describes an asynchronous time division command/response multiplex data bus. There are three interfaces to this bus: the bus controller (BC), remote terminals (RTs), and monitoring terminals (MTs). The BC supervises time division multiple access (TDMA) to a multidrop bus of interconnected RTs. MTs listen to traffic on the bus and can record transactions

for telemetry or development/debugging. This represents an ETP architecture: RTs only respond to requests for data from the BC, which may or may not follow a periodic pattern. Typically, there are other backup BCs that can take over BC responsibilities in the event of BC failure, and there is a secondary bus in a dual-redundant configuration in case the primary bus goes down [48]. This topology is shown in Figure 19.

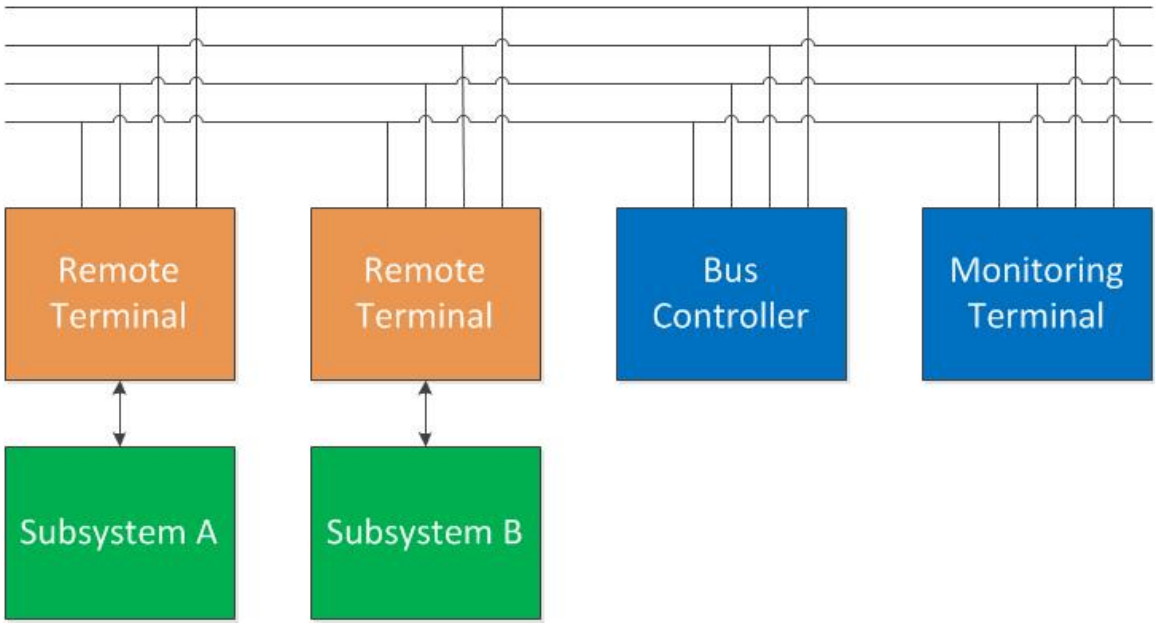


**Figure 19: MIL-STD-1553 Bus Topology [86]**

On the byte level, there are three words: command words, which are issued only from the BC and contain the RT address and command; data words, which are two bytes of data; and status words, which are issued only from RTs and contain the RT address and a status byte. These three words form three basic message transfers: BC to RT, where the BC commands the RT; RT to BC, where the RT responds to a BC command with its status word; and RT to RT, where the BC commands one RT to transmit and another RT to receive. Both RTs perform an RT-to-BC transfer with their status words [86]. While the BC initiates all communications and messages on the bus, the RTs are required to perform word validation for every received byte of data. Failure of any of these validations prompts the message to be discarded: valid sync field at the beginning of the word, valid Manchester II code, a 16-bit information field, and valid odd parity. While the word is discarded if any of these conditions fail to be met and a message error bit is set in the RT's status message to the BC, no native action is taken to retransmit the word [87].

The primary drawback of this architecture is the limited transfer speed of 1MB/s. While much work has been done to increase this speed (with different star topology configurations and standard add-ons yielding 10MB/s and 200MB/s, respectively), MIL-STD-1553 is an antiquated architecture that will continue to find use not for its performance, but for its reliability and the expense required to replace it in existing systems [48].

### 4.1.5 LonTalk

LonTalk is a communications protocol that implements the EIA-709.1 standard, designed for terrestrial control networks whose messages are very short and require low bandwidth, power, and maintenance [88]. LonTalk was originally developed by the Echelon Corporation, and is now part of a networking platform called LonWorks that includes physical interconnect specifications and a commercial chip called the Neuron.

The design goals of LonTalk are many-fold. These goals include: media independence, meaning LonTalk can be deployed in a very wide range of environments; scalability, from only a few PEs to many thousands; low cost; no central controller necessary, meaning no single point of failure; peer-to-peer, and no protocol subsets so that all PEs are interoperable [81]. There are many “internet of things” SMs, and many that already use this protocol. These include heating, ventilating, and air conditioning (HVAC) systems, industrial control, medical instrumentation, security, home automation, etc. The protocol envisions a vastly deployed sensor network, all communicating with only two or three byte payloads; for example, temperature or pressure sensors providing periodic readings to a central computer over a factory floor. To accomplish this, LonTalk implements a seven layer network stack very similar to the OSI model. These stack portions include the physical layer, link layer, network layer, transport layer, session layer, presentation layer, and SM layer.

#### 4.1.5.1 Layer 1

Layer 1 is the physical layer of LonTalk, and comprises the protocols and encodings used to transmit data over the physical media connecting LonTalk PEs. These include Manchester encoding, frequency shift-keying modulation, etc.

#### 4.1.5.2 Layer 2

Layer 2 is split into the Media Access Control (MAC) sublayer and the Link sublayer. The MAC sublayer implements a CSMA scheme, and uses the Neuron-ID of PEs as the hardware address of each endpoint. The Link sublayer frames data and checks for errors using CRC. Errors are only reported in this layer, and are not handled.

#### 4.1.5.3 Layer 3

Layer 3 is the Network layer of LonTalk. This layer implements a connection-less, unacknowledged, single-domain packet delivery service. This service can be unicast, multicast, or broadcast.

#### 4.1.5.4 Layer 4

Layer 4 is split into the Transaction Control sublayer and the Transport sublayer. The Transaction Control sublayer handles the ordering of incoming messages and checks for duplicates. The Transport sublayer implements a connection-less, reliable message delivery service over multiple domains.

#### 4.1.5.5 Layer 5

Layer 5 is the Session layer of LonTalk. This layer implements a “request/response” service to make remote procedure calls on other PEs. This layer also distinguishes between idempotent transactions, which are transactions that can be executed any number of times, and non-idempotent transactions, which are transactions whose actions depend on previous states. Idempotent transactions’ reliability is “at-least-once”, such as reading data entries from a table; non-idempotent transactions’ reliability is “at-most-once”, such as turning a valve a certain amount.

#### 4.1.5.6 Layers 6/7

Layers 6 and 7 of the OSI model are grouped into one layer, and comprise the Presentation and SM layers of LonTalk. This layer checks the application protocol data unit (APDU) header of LonTalk packets for network variable updates, and propagates these updates to other PEs. This layer also provides the highest-level generic message-passing for SMs.

Collectively, these layers provide collision detection, error checking, connection-less packets, broadcasting, acknowledged and unacknowledged uni/multicasting, and guaranteed message delivery. A typical packet using this protocol is shown in Figure 20, and notably consists of only 12 bytes, including the 2 byte data field [88].

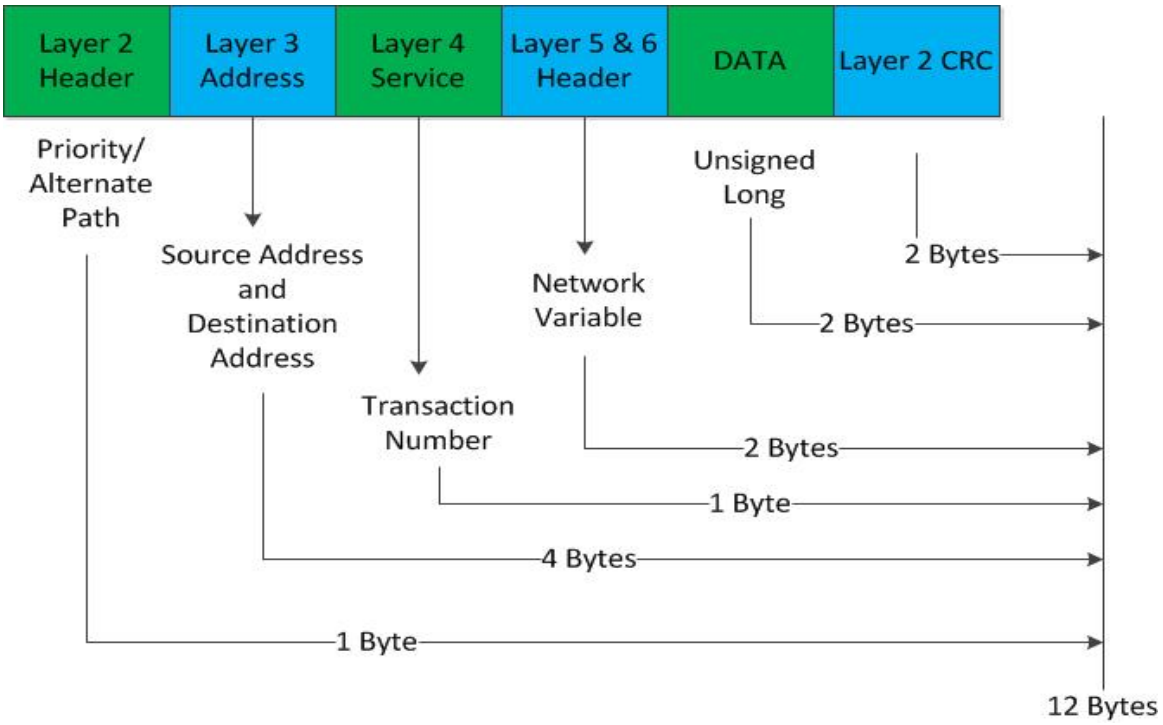


**Figure 20: Typical LonTalk packet [81]**

For the network layer (layer 3), all commercial Neuron chips are assigned a 48-bit Neuron ID for initial configuration. However, in operation, logical address specifying a PEs domain, subnet, and PE ID are used. This scheme supports many devices, allowing for 127 devices per subnet, 255 subnets per domain, and 18,446,744,073,726,329,086 domains. This yields a total of 597,397,806,827,627,450,110 devices that can communicate with each other [89]. The session layer provides authentication via the 48-bit Neuron ID burned in at manufacture, verifying the sender; the receiver must provide a 64-bit random challenge answer in order to communicate. In order for devices to communicate with each other, they use the Standard Network Variable Types (SNVT), which provides generic names and units for temperature, rotation, speed, time, etc. This is very similar to SPA's CDD. In the presentation layer, sensors "publish" information using these common terms, and actuators "subscribe" to the information using these common terms. This is again very similar to SPA and is similar to the object request brokers discussed in 4.2.

The layered approach of LonTalk can be implemented in several ways; the parent company of LonTalk, Echelon, sells Neuron chips at less than $3.00 per chip. Alternatively, since LonTalk is an open source standard, user devices can implement portions of the stack, mixing and matching commercial-off-the-shelf (COTS) components to implement other portions of the stack.

Other companies pursuing these products include Adept Systems, using an MC68360 microprocessor; Loytec Electronics implementing the Link, Transport, and Network layers on an FPGA; and Toshiba using a MIPS RISC core with Java OS. LonTalk manually defines all protocol layers in the stack for two reasons: to guarantee interoperability between devices from different manufacturers and to make the most use of memory in the smallest and lowest-cost package possible. For example, a Neuron-powered and LonTalk-connected temperature sensor requires only 344 bytes of EEPROM and 841 bytes of RAM to be fully compliant PE in the network [81].

### 4.1.6 Middleware Aspects

The degrees to which each D-Transport layer middleware address the five key aspects of middleware will now be compared and contrasted. The results are summarized in Table 11.

**Table 11: Comparison of D-Transport Middleware**

| | Network Communication | Coordination | Reliability | Scalability | Heterogeneity |
|---|---|---|---|---|---|
| **Ardea** | Message-oriented | Asynchronous | Exactly-once | Migration, replication | Network |
| **MeRL** | Message-oriented | Both | At-least-once | Access, Location | Hardware |
| **SPA** | Node-oriented | Synchronous | At-most-once | Location | Hardware, Network |
| **MIL-STD-1553** | Node-oriented | Synchronous | At-most-once | None | Hardware, Software |
| **LonTalk** | Node-oriented | Asynchronous | Mixture | Access, Location | Software |

#### 4.1.6.1 Network Communication

Ardea handles network communication through the communication network component of its architecture, and identifies a message-oriented approach. While the architecture does not specify a network communication protocol, Ardea's implementation on the BIG BLUE UAV used a subset of CAN-Aerospace, already reviewed as an HI-Network middleware in Chapter 3. Ardea only specifies the SM data and management messages of the network, using only a generic

interface to a specific network communication protocol. This interface can link to any distributed communications protocol.

MeRL handles network communication through a message-oriented API interface for SMs in the SM layer to use. The SMs are ignorant of where the destination SM is located, whether it's on the same PE or a different PE in the network. Internally, MeRL specifies CAN as the network communication protocol; however, the MeRL design analyzes other options as well, such as $I^2C$, Ethernet, SPI, and RS-485. Any of these could theoretically be swapped out for CAN. Additionally, the MeRL design specifies Zigbee as the network communication protocol in wireless network configurations.

SPA handles network communication differently through its four implementations: SPA-U (USB-based), SPA-O (optical-based), SPA-S (SpaceWire-based), and SPA-1 ($I^2C$-based). Each implementation uses that communications protocol for their physical and data link layers, but SPA extends these protocols to meet the plug-and-play challenge in the space environment, including restricted hardware, greater power delivery, self-description, and fault tolerance. All of these implementations of SPA are node-oriented.

MIL-STD-1553 handles network communication by specifying a node-oriented multidrop serial bus, with the BC controlling and initiating all traffic between RTs. Since the architecture is node-oriented, each RT is assigned a unique global address but can receive broadcast messages addressed with the reserved decimal 31. To initiate a data transfer between RTs, the BC addresses a command message to the RT that is supposed to transmit; this RT responds with a status message to the BC, and then takes control of the bus to send the data to the receiving RT. The transmitting RT then relinquishes control of the bus back to the BC [86].

LonTalk handles network communication through a node-oriented architecture and by fully specifying the seven layers of the OSI network stack. On the physical layer, LonTalk allows a variety encoding schemes, and allows for twisted pair and wireless communications. This network communication is abstracted by the MAC and Link sublayers, implementing framing, error checking, and collision avoidance algorithms.

#### 4.1.6.2 Coordination

Ardea handles coordination through asynchronous messaging, using the system manager component. This system manager is responsible for tracking the status of hardware and software resources, deploying new configurations should faults occur. The system manager also handles

checkpointing, and is able to track the completion state of software modules running on each processing element.

MeRL handles coordination by offering both synchronous and asynchronous messaging in its API. The get_msg function is blocking, and suspends the SM task until a new message is pushed into that SM task's buffer. This function is synchronous, requiring the calling SM to wait for a response. The accept_msg function, however, is non-blocking, and only checks the SM task's buffer for a new message, copying the message if one exists and returning "no new message received" if not. This function is asynchronous, and allows the SM task to continue execution of other duties after sending its message, checking for new messages as it desires.

SPA handles coordination through synchronous communication based upon its underlying architecture. SPA-U (USB-based), SPA-S (SpaceWire-based), and SPA-1 ($I^2C$-based) are all built on synchronous host-infrastructure middleware, requiring a combination of clock synchronization and acknowledgement cycles that prevent the transmitting PE from continuing execution after data has been sent. The SDM in SPA directly manages communications between PEs as well, preventing execution from continuing after a message has been sent.

MIL-STD-1553 handles coordination through asynchronous communication for BC-to-RT and RT-to-RT messages. The BC initiates all communication, telling which RTs to transmit and which RTs to receive; however, no clock signal is shared between PEs and each PE continues execution after the message has been transmitted.

LonTalk handles coordination through asynchronous communication. With no shared clock signal or acknowledgement system, transmitting PEs continue execution after sending messages.

#### 4.1.6.3 Reliability

Ardea handles reliability again through the system manager. The system manager tracks all state data from each processing element, and should a fault or reset occur, can restore the state of that processing element. If a permanent fault or temporary load balancing issue occurs, the system manager can reconfigure another processing element to adopt the responsibilities of the faulty processing element. This allows for multiple redundant processing elements in the network, and allows for other idle processing elements to adopt the roles of failed or faulty processing elements. Reliability is the key goal of Ardea, and the architecture of reconfigurable processing

elements and floating software modules was created specifically for this goal. In service of this goal, Ardea's reliability is exactly-once.

MeRL's reliability is at-least-once handles reliability both through its extensive selection of network communication protocol, CAN, and its use of message queues. CAN is a fault-tolerant network protocol for embedded systems, and was chosen as MeRL's physical layer protocol for its broadcast messaging style and its message-oriented delivery. Broadcast network messages are seen by all PEs' physical layers, ensuring that a message subscribed to by multiple PEs is consistent; this removes faults from occurring due to inconsistent message delivery. This differs from multicast messaging style, where a select set of destination PEs are chosen by the sender. Message-oriented delivery means that all PEs on the network see the message and choose whether or not to interpret the message based on a message identifier. This differs from node-oriented messaging style, where an addressing scheme limits the interpretation of the message to intended PEs.

SPA's reliability is at-most-once and is based on the underlying communications protocol. SPA-U, SPA-S, and SPA-1 are all based on host-infrastructure middleware that has at-most-once reliability.

MIL-STD-1553's reliability is at-most-once. All RT's on the bus are required to validate incoming words for a valid sync field, valid Manchester II code, 16-bit wide information fields, and odd parity. Failure of any of these tests prompts the word to be discarded and a message error bit in the RT's status message to the BC to be set. However, no native action is taken to re-transmit the word, preventing delivery of the word multiple times.

LonTalk's reliability is either at-most-once or at-least-once, depending on the interpretation of packets by the Session Layer. This layer distinguishes between idempotent messages, which can be executed any number of times but still need to be executed (at-least-once), and non-idempotent messages, which must be executed once or not at all (at-most-once).

#### 4.1.6.4 Scalability

Ardea provides migration and replication transparency. All processing elements in the network run software modules, and it is these software modules that SMs interact with. Each processing element runs the network management, task scheduler, memory loading tasks, handling all outgoing and incoming communications below the level of SMs. The system manager monitors the status of processing elements as they support their software modules, and

handles migration of software modules between processing elements with no knowledge or input from the user. Finally, the fault detection and correction algorithm of N-redundancy voting ensures that the most reliable data is delivered to software modules in the event of replicated endpoints.

MeRL provides access and location transparency. All messages are passed to MeRL, which determines whether the target task is locally registered or elsewhere on the network, providing access transparency. If the target task is determined to be elsewhere on the network, MeRL manages delivery to that task with location knowledge required from the user, providing location transparency. New data needs only be assigned an ID at compile-time, and then SMs can register for reception of that ID when available.

SPA provides location transparency. xTEDS registration and network enumeration yield logical locations for each endpoint to the user, and SMs do not have to manage or know the physical locations of PEs to communicate.

MIL-STD-1553 provides no transparency. Bus interactions are initiated and maintained by bus controller unit, and no remote terminals can initiate network communication. The locations are each remote terminal are explicitly known and referenced by the bus controller as well. Finally, there are 31 address locations for RTs, limiting the scalability of MIL-STD-1553 to those addresses.

LonTalk implements access and location transparency through its full provision of OSI layers. The location details of PEs in the LonTalk network are hidden by the MAC and Link sublayers; while SMs can use the Neuron-ID to address endpoints, logical addresses for PEs are employed. The access details in LonTalk are hidden by the Session layer's request/response service: this service allows PEs to execute remote procedure calls on the same or other PEs.

#### 4.1.6.5 Heterogeneity

Ardea exhibits only network heterogeneity. Ardea assumes homogeneous hardware because it is built around graceful degradation, migrating software modules to different hardware modules to preserve essential functionality in the face of load balancing issues and hardware failures. Additionally, Ardea assumes knowledge of and support for the C programming language, precise SMs, and operating system specification. This support is required in order for Ardea to effectively migrate tasks between hardware PEs, and has to be concise enough to

support migration. Ardea does not, however, specify the network signaling and protocols between PEs, giving it network heterogeneity.

MeRL exhibits hardware heterogeneity. As an implementation of the Ardea network component, MeRL specifies network communication component as CAN. MeRL only requires CAN support from the hardware it runs on, allowing for a range of microcontroller architectures. MeRL does not exhibit software heterogeneity because it specifies the uCOS-II real-time operating system and the C programming language.

SPA exhibits network and partial software heterogeneity. SPA does not scale to lower power processing units, requiring dynamic memory allocation and Linux or VxWorks operating systems that are not supported on 8-bit microcontrollers. This restricts SPA's hardware support to more powerful processors, many outside the target scope of this thesis. SPA does support different network protocols, such as USB, $I^2C$, Ethernet, and SpaceWire, giving it network heterogeneity. Finally, SPA requires an operating system and restricts these to Linux and VxWorks, providing partial software heterogeneity.

MIL-STD-1553 exhibits hardware and software heterogeneity. The MIL-STD-1553 specification provides a set of requirements that candidate hardware must comply to, but does not restrict the architectures or hardware used. Furthermore, MIL-STD-1553 provides a set of requirements for software to comply to and does not restrict the SM or operating systems being used to fulfill these SMs. However, the network communication protocol and signaling is tightly controlled, giving it no network heterogeneity.

LonTalk exhibits software heterogeneity. The hardware, Neuron chips, are offered by several manufacturers but rigidly defined. Similarly, the LonTalk network protocol and transfer mechanics are rigidly defined. However, there is no limitation on SMs using LonTalk. (need more here)

### 4.2 PEPt Middleware

PEPt middleware differs from D-transport middleware in that it completely obscures the programming models, data encodings, framing protocols, and frame transport from the user, presenting the user with an object-oriented-style reference to other SMs and components. These SMs and components are presented as services. Middleware in this layer follows the publish/subscribe model and represents SMs and components on the distributed network as services with object-oriented syntax. Instead of sending a message to specific SM, the message is

"published" to the middleware, which routes the message to any "subscribed" SMs. The primary reasons for PEPt middleware are two-fold: PEPt middleware allows for more scalable networks, and completely obscures the existence of other SMs. The PEPt middleware to be reviewed in this layer include Common Object Request Broker Architecture (CORBA), CORBA/embedded, and micro-ORB.

### 4.2.1 Common Object Request Broker Architecture (CORBA)

The Common Object Request Broker Architecture (CORBA) was created by the Object Management Group (OMG). The OMG was formed in 1989 to address the need for standardized, distributed, heterogeneous middleware standards in response to the growing occurrence of computer networks, such as the Internet. OMG created the Object Management Architecture (OMA) to describe an architecture for such distributed heterogeneous systems, and created CORBA as an implementation component of this architecture. CORBA 1.0 was released in 1991, with routine updates through CORBA 3.3 in 2012 [90]. CORBA is intended for large-scale computing networks, and has been used by a multitude of government agencies and large companies, such as The Weather Channel, Raytheon, Thames Water, NASA, the U.S. Navy, and over three-quarters of the world's financial institutions [91]. In response to criticism that CORBA has too large a footprint and requires too much computational power for smaller-scale embedded systems, the OMG released CORBA/e in 2008 to reduce the footprint and support distributed embedded systems. Two variants of CORBA/e exist: Compact Profile and Micro Profile [92].

CORBA is the object request broker (ORB) implementation in the OMG's OMA model. The OMA model consists of two internal models: the Object Model and the Reference Model. The Object Model describes the four types of objects in a distributed environment, and the Reference Model describes how those objects interact. The Object Model is shown in Figure 21.

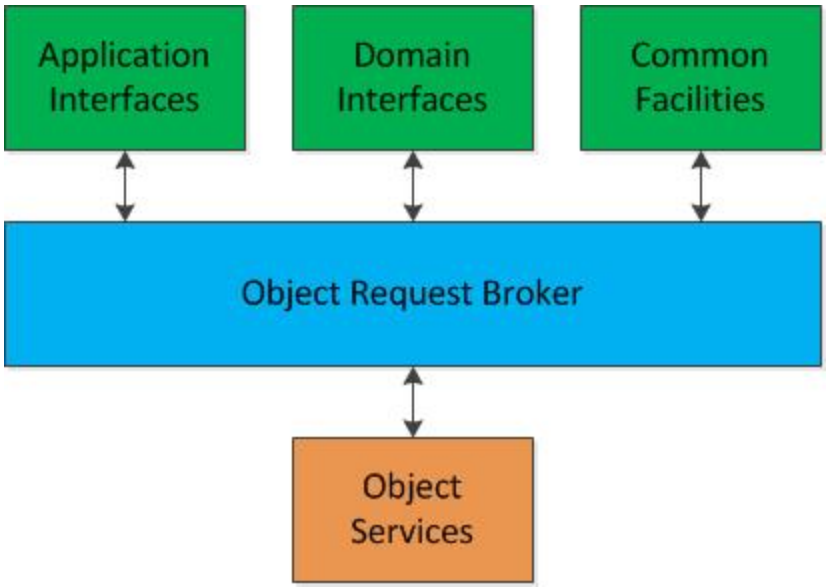


**Figure 21: OMG Reference Architecture [93]**

The above reference architecture defines the various components in a distributed computing model, particularly the component OMG fulfills with CORBA. These components exist on each PE in a distribute network, and consist of: object services, common facilities, domain interfaces, and SM interfaces, all linked by the object request broker (in this case, CORBA). Object services are domain-independent interfaces used in distributed object SMs. These include services for discovery and naming services, which allow clients to find objects based on name or properties (see Naming Service and Trading Service), security services, transaction services, lifecycle management services, etc. Common facilities provide facilities, which similar to object services, but geared more toward end users. An example facility is the Distributed Document Component Facility, which allows for linking document object components for users. Domain interfaces, again like common facilities and object services, provide services, but for specific domains. These can include telecommunications, medical, and financial SMs. Finally, SM interfaces provide services for specific SMs, and are not standardized due to their variability. The object request broker (ORB) links all of these services together, allowing for services to find each other and communicate as both clients and servers. Within this framework, different combinations of these components may exist on any PE, as shown in Figure 22.

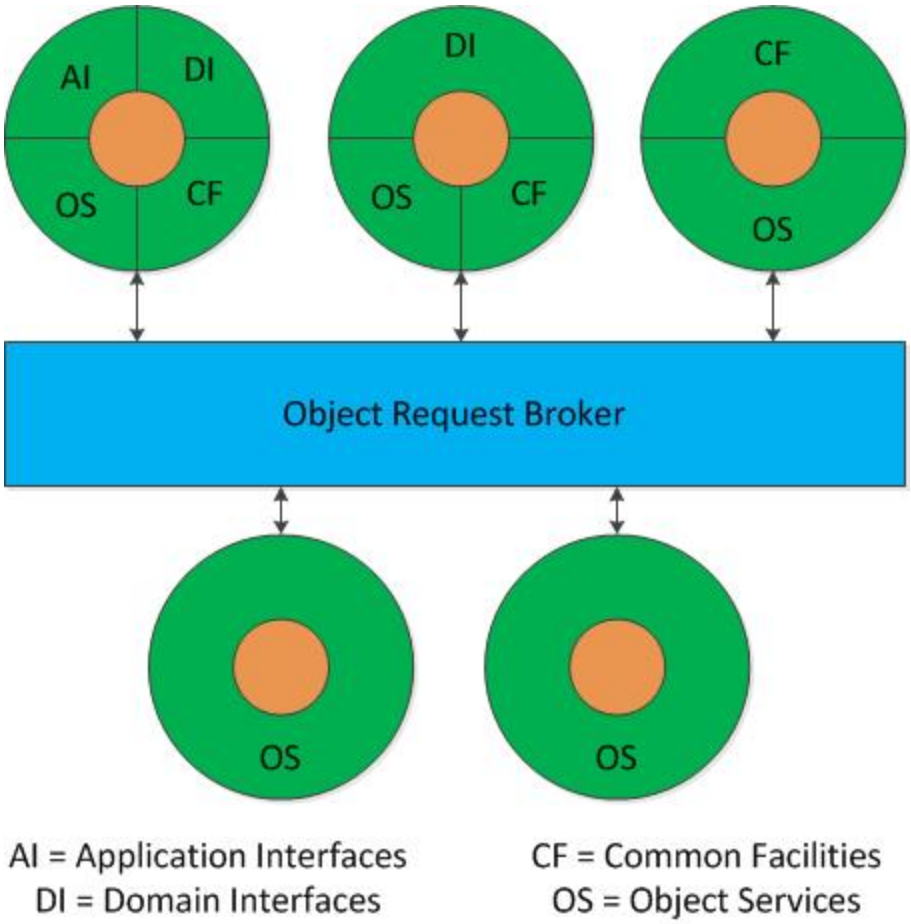


**Figure 22: Peer-to-peer network linked by ORB [94]**

CORBA is the ORB component of the OMA. The goal of CORBA is to facilitate communication between clients and objects. In facilitating communication, CORBA hides the object location, whether this location is on the same machine or a different PE; the object implementation, including the programming language, hardware, operating system, etc.; the object execution state, including whether the object is ready to accept requests; and the object communication mechanisms, which define the processes and protocols used to deliver the request and response (TCP/IP, shared memory, local call, etc.). Clients make requests to objects by invoking object references, which are available through the Naming and Trading Services; these are object services that are minimally required by each of the components in order to be implemented with the ORB, and provide object references based on name or properties to clients. These client requests and server responses are based on the client/server model, and form synchronous communications between the two. However, CORBA also offers a publish/subscribe

model through its Event Service, which allows data to be both anonymously published to the ORB core and anonymously subscribed to by SMs.

Requests for object references and operations with objects are coded according to the OMG Interface Definition Language (IDL). This IDL provides interfaces for each object, and is similar to C++ and Java in format. However, it is declarative and not compiled, and is interpreted by the ORB, allowing for full programming language independence because the object implementations are defined separately. An example interface that only creates an object is:

```
// OMG IDL
interface Factory {
    Object create();
};
```

Factory is the object interface, and allows an object of Factory to be created, returning an object reference. Additionally, interfaces can inherit from other interfaces in a format similar to C++, shown below:

```
// SpreadsheetFactory derives from Factory
interface SpreadsheetFactory : Factory {
    Spreadsheet create_spreadsheet();
};
```

The spreadsheet object interface inherits the above create() operation from Factory, and specifies its own create_spreadsheet() operation. In addition to this C++/Java-style language format, the IDL also provides variable types similar to those found in popular programming languages, including unsigned and signed long, short, char, enum, float, struct, union, and string [94].

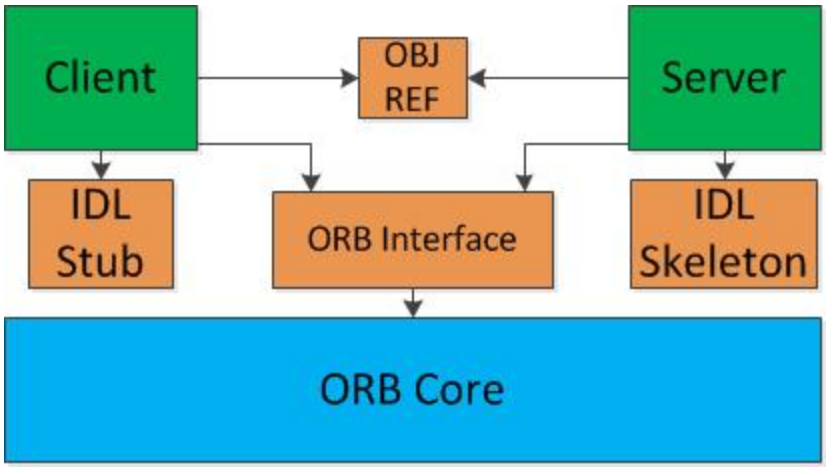


**Figure 23: CORBA ORB Implementation [93]**

Figure 23 is the implementation of CORBA. At compile time, an IDL compiler makes use of an interface repository, which contains all object interfaces, allowing SMs to traverse and discover IDL information at runtime. This compiler translates the IDL code into the target programming language, and provides stubs and skeletons. Stubs are client-side mechanisms that issue requests for the client, whereas skeletons are server-side mechanisms that deliver the requests to the specific implementation. In addition to these static requests and responses that are set at compile-time, dynamic run-time discovery of services to request are provided for as well. This is accomplished through the Dynamic Invocation Interface (DII), allowing a client to directly access the ORB to request instead of needing IDL-defined interfaces, and the Dynamic Skeleton Interface (DSI), allowing a server to respond to such requests not defined at compile-time. The final component of the ORB is the Object Adapter, which maps the ORB to object implementations [93].

For communication between distributed components on a CORBA network, two communications protocols are specified: the General Inter-ORB Protocol (GIOP) and the Internet Inter-ORB Protocol (IIOP). The GIOP specifies the syntax and composition of messages, whereas the IIOP specifies CORBA's mapping to a TCP/IP-style network transport. While both are required for CORBA 2.0 and later releases, the GIOP is not IIOP-specific, and does not contain any restriction to TCP/IP for transport; rather, the GIOP is standard for any connection-oriented transport.

CORBA/Embedded (CORBA/e) is a new version of CORBA that is targeted toward the real-time and low-footprint needs of distributed embedded environments. CORBA/e is available

in two versions, or “profiles”: Compact Profile, which is intended for 32-bit microprocessors running real-time operating systems, and Micro Profile, which is intended for low-power microprocessors and digital signal processors. The goal of the Compact Profile is to merge CORBA and Real-Time CORBA for smaller processors, creating a deterministic version of CORBA. The goal of the Micro Profile is to shrink the CORBA footprint so that it can fit on very small low-power microprocessors. Both profiles are fully compatible with the GIOP and IIOP protocols, allowing for communication with SMs running on other CORBA profiles or full CORBA. Furthermore, both profiles compile and support the entire IDL except for the dynamic aspects (the DII and DSI and the dynamic data types such as Any and Valuetype). This retains CORBA support for a heterogeneous mixture of programming languages and hardware architectures [92]. This comparison of CORBA profiles is summarized in Table 12.

**Table 12: Comparison of CORBA Profiles**

| | CORBA | CORBA/e Compact | CORBA/e Micro |
|---|---|---|---|
| **Target Processors** | Enterprise | 32-bit, RTOS | 32-bit/8-bit low-power |
| **IDL Support** | Yes | Yes | Yes |
| **GIOP/IIOP Comms** | Yes | Yes | Yes |
| **Real-Time Scheduling** | Yes | Yes | No |
| **Naming/Event Services** | Yes | Yes | No |

### 4.2.2 uORB

Micro Object Request Broker (uORB) is a custom ORB written for the PX4 micro air vehicle autopilot, and facilitates the passing of data structures between SMs. It follows the publish/subscribe model, and offers an API. It is intended for use on low-power embedded microcontrollers. uORB does not formally specify a network communication implementation, but interfaces with MAVLink for off-chip communications.

uORB works by managing a table of all subscriptions and publications that each individual SM maintains. Data structures are registered in the uORB core as “topics” using the ORB_advertise() function call, which returns a pointer handle for that topic. Updates to that topic are pushed to the uORB core by the publisher using the ORB_publish() function, which then updates that topic’s internal marker. Subscribers use the ORB_subscribe() function call to initially subscribe to a topic, which returns a pointer handler to that topic. These subscribers then must poll the uORB core using the ORB_check() function call, which returns a Boolean

indicating whether the topic has been updated since the last time the subscriber has copied new topic data. If the data is indeed new, the subscriber must use the ORB_copy() function call, which fetches data from the topic and prompts uORB to reset the topic's internal marker for the subscriber that copied the data, since there can be multiple subscribers [95] [96].

### 4.2.3 XML-RPC

XML-Remote Procedure Call (XML-RPC) is a middleware for calling procedures on distributed PEs over a network, historically using the OSI SM Layer program HTTP. However, work has been done to use CAN as the network communication component of XML-RPC [97].

XML-RPC is primarily distributed as C/C++ libraries; however, there are Ruby, Perl, Java, and Objective-C implementations as well. The goal of XML-RPC is to abstract away the programming languages and hardware on endpoints in the network, allowing for the calling of objects on remote hosts through the universal XML language. XML-RPC is a client-server architecture, where clients are created on PEs and make requests "methods" on servers on other PEs [98].

### 4.2.4 Middleware Aspects

The degrees to which each PEPt layer middleware address the five key aspects of middleware will now be compared and contrasted. The results are summarized in Table 13.

**Table 13: Comparison of PEPt Middleware**

| | Network Communication | Coordination | Reliability | Scalability | Heterogeneity |
|---|---|---|---|---|---|
| **CORBA Profiles** | Message-oriented | Synchronous | At-least-once | Access, Location | Hardware, Network, Software |
| **uORB** | Message-oriented | Asynchronous | At-most-once | Access, Location, Replication | Hardware, Network, Software |
| **XML-RPC** | Message-oriented | Asynchronous | Mixture | Location | Hardware, Network, Software |

#### 4.2.4.1 Network Communication

CORBA's network communication is message-oriented, and includes both client/server and publish/subscribe architectures. Clients can make requests to servers, and clients can publish and subscribe to object references anonymously. Both of these communications architectures are handled through CORBA's GIOP and IIOP, which specify a syntax and TCP/IP style connection-based, transport-level network. CORBA/e's network communication is also message-oriented, and is a new definition of CORBA that removes CORBA's dynamic aspects in order to offer deterministic timing and fit on low-power embedded microcontrollers. While exhibiting a smaller memory footprint, CORBA/e is still fully compliant with CORBA's IIOP, retaining interoperability with CORBA systems.

uORB, while a custom implementation of an ORB, is still message-oriented and relies solely on the publish/subscribe model for message-passing. Off-chip communications interface through MAVLink, which marshals the messages and packs them over a serial channel to all other processors in the network.

XML-RPC's network communication is message-oriented and relies on a client/server architecture. Traditionally based on HTTP over TCP/IP as the network communication mechanism, CAN-based XML-RPC has been demonstrated to prove XML-RPC's applicability to distributed embedded networks [97].

#### 4.2.4.2 Coordination

CORBA handles coordination by default through synchronous object requests. Full CORBA's DII allows for deferred synchronous and asynchronous object requests, but both CORBA/e profiles remove this dynamic interface and only support synchronous object requests. Stubs and skeletons are present on every PE in the network to facilitate these object requests. Stubs are client-side mechanisms for issuing requests, and skeletons are server-side mechanisms for responding to requests. Since each PE in a CORBA network can be both a server and a client depending on the services it offers or requires, these are present on a per interface basis.

uORB handles coordination through an asynchronous publishing/subscribing service. PEs with new data publish the data to the ORB and continue with execution; however, PEs subscribing to topics must poll the ORB core periodically checking for new data, copying over new data as it becomes available. Since this process requires any receiving PEs in the network to devote execution time dedicated to checking for new data, instead of being notified that new data is available by the ORB, this coordination is only partially asynchronous.

XML-RPC handles coordination through asynchronous communication. While historically based on TCP/IP communication, research as shown XML-RPC running on a CAN bus, which is asynchronous. This is the target host-infrastructure architecture for XML-RPC in the target small-scale, low-power embedded networks of this thesis.

#### 4.2.4.3 Reliability

CORBA's reliability is at-least-once. While the addition of the Event Service allowed for a publish/subscribe network communication model, it did not address CORBA's inherent lack of a reliability guarantee. The addition of the Notification Service, however, provided at-least-once reliability by offering a set of parameters, EventReliability and ConnectionReliability, that allow the user to set the desired messaging reliability. Setting both to "persistent" provides this at-least-once reliability, where the calls to object references do not return until the references are valid and stored to persistent memory, and the push or pull request keeps retrying.

uORB's reliability is at-most-once. ORB_publish and ORB_subscribe calls contain no inherent verification that subscribers receive the published data. While the orb_publish function prompts notification to waiting subscribers, subsequent orb_publish calls will overwrite the data, no matter if subscribers have received this data or not.

XML-RPC's reliability depends on the underlying transport mechanism; for traditional TCP/IP transport, the reliability is exactly-once. For CAN-based transport, the reliability is at-least-once. XML-RPC includes no functionality to guarantee the procedure calls occur correctly or on time.

#### 4.2.4.4 Scalability

CORBA offers access and location transparency. All services in the system are abstracted to object references, and the access point, whether local or remote, and location of these references are irrelevant to SMs.

uORB offers access, location, and replication transparency. Like CORBA, all data in the system is abstracted into object references, called "topics". The access point, whether local or remote, and location of these topics are irrelevant to SMs. Since uORB uses an exclusively a publish/subscribe architecture, replicated endpoints are also irrelevant to SMs, since no knowledge of such replication is required and replicated endpoints must simply subscribe to or publish data anonymously.

XML-RPC offers location transparency. Remote procedure calls made by SMs require no knowledge of the location of the procedures on other PEs.

#### 4.2.4.5 Heterogeneity

CORBA exhibits hardware, network, and software heterogeneity. One of the primary design goals of CORBA is to mask the hardware, location, operating system, and programming language implementations of each PE; these are all abstracted from CORBA operation. Furthermore, CORBA's IDL exists for the purpose of allowing usres to map specific programming languages onto a common declarative language, with official support for Ada, C, C++, COBOL, Java, Lisp, Python, and Ruby. This gives CORBA hardware and software heterogeneity. Furthermore, CORBA suggests a network communication similar to TCP/IP, but does not require or implement any network communication, giving it network heterogeneity as well.

uORB exhibits hardware, network, and partial software heterogeneity. As a C library, uORB is restricted to the C programming language, but provides an API to interface with any SM or operating system, giving it partial software heterogeneity. uORB does not specify any network communication, and in implementation relies on MAVLink, giving it network heterogeneity. Finally, uORB does not specify hardware, and is a low-footprint library that is not restricted to any particular architecture, giving it hardware heterogeneity.

XML-RPC exhibits hardware, network, and software heterogeneity. By using XML as the encoding, PEs may use different programming languages and hardware architectures to format the XML messages. Furthermore, the transport mechanics of XML-RPC are not rigidly defined; usually implemented using TCP/IP, CAN-based transport for XML-RPC has been used, and any message-oriented network communication style could theoretically support XML-RPC messages.

# 5 Common Services Layer Middleware

This chapter details common services layer middleware. Common services middleware is above distribution middleware, and provides common distributed embedded network services in addition to end-to-end transport mechanisms. The set of services provided vary, but their goal is to utilize the kind of encapsulations and extensions provided by host-infrastructure and distribution middleware to form SM-level services that are typical in distributed SMs. These kinds of services include the message transportation found in distribution middleware, but also other services: file manipulation, timing and synchronization, logging, and device virtualization. The middleware to be reviewed in this layer are Spacecraft Onboard Integration Services (SOIS) and Core Flight System (CFS).

## 5.1 Spacecraft Onboard Interface Services (SOIS)

The Spacecraft Onboard Interface Services (SOIS) middleware is currently being defined and standardized by the Consultative Committee for Space Data Systems (CCSDS), which is a collection of 11 member space agencies from around the world seeking to improve interoperability between international space systems. CCSDS organizes standards releases by color: the SOIS handbook is currently in Green Book form, meaning it is an Informational Report that describes the desired design process and methodologies for the adoption of SOIS. This Green Book was first released in June 2007, and was updated in December 2013. SOIS stands out from other middleware reviewed by this thesis in that it is not a standard or downloadable code base; rather it is a detailed middleware approach that is still in early stages of design and review. Individual features of SOIS, such as the Device Virtualization Service and File/Packet Store Service, are currently in the Red Book phase of review, to be completed in mid-2014.

The target system of SOIS is generically declared by CCSDS to be “all classes of civil missions, including scientific and commercial spacecraft, manned and un-manned systems” [99]. Since the SOIS Green Book defines a design process and recommended organization of a distributed computing network, there is no provision for specific hardware requirements or software restrictions; however, the Device Virtualization Service and recommended protocols have been flown on the UKube-1 CubeSat; CubeSats in general, with their goal of rapid design and integration, are ideal targets for this middleware [100].

SOIS is composed of computing services organized into three layers: the SM support layer, the transfer layer, and the subnetwork layer. The services provided in each layer attempt to disassociate users from any specifics of endpoint hardware and the network used to link them.

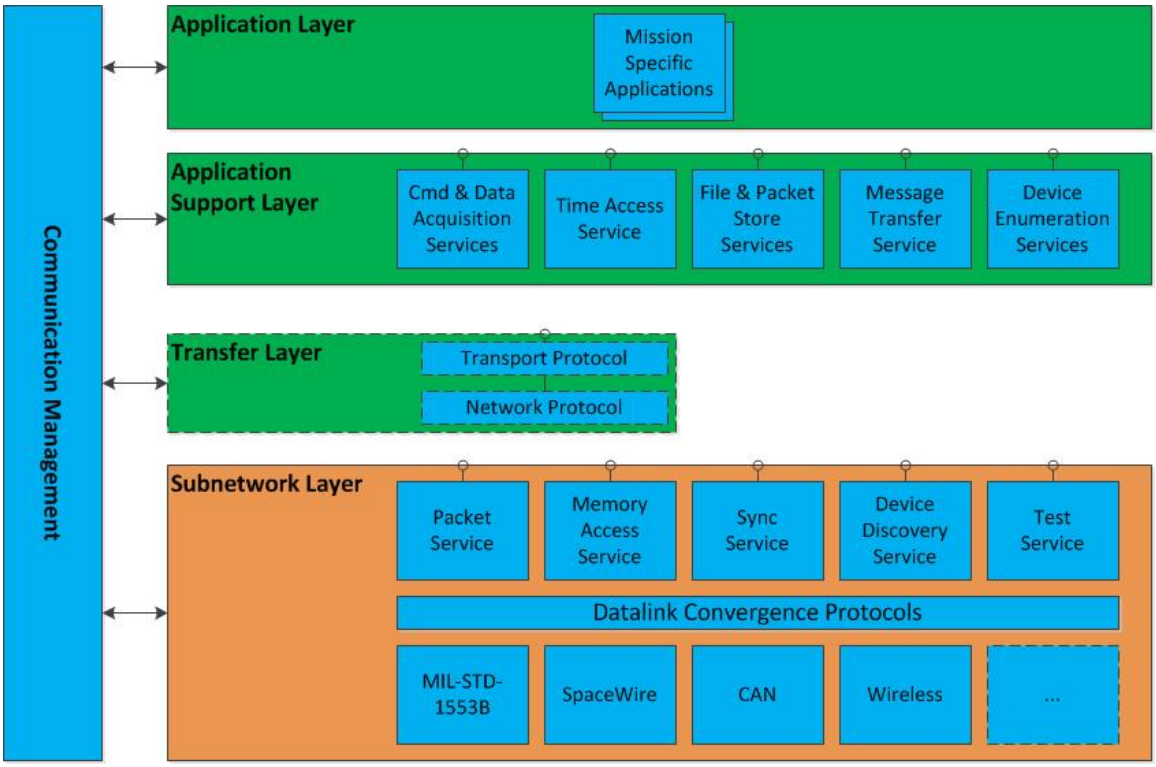


**Figure 24: SOIS Reference Architecture [101]**

Figure 24 details the layers that compose SOIS, as well as the services each layer provides. The development of this architecture results from several observations on typical spacecraft designs. Despite the ideality of a spacecraft's internal components all using the same communications medium, this is not usually the case. Rather, spacecraft usually exhibit multiple point-to-point connections, along with a single communications bus. Additionally, the choice of these communications mediums will occur on a mission-by-mission and hardware basis; essentially, no single communications protocol or method will necessarily be best for all missions, and these protocols will undoubtedly vary. Finally, the devices communicating will not typically have similar computing power or elements; microprocessors performing scheduling, file management, etc. operations may have such similar capabilities, but sensors and other actuators will have far reduced capabilities. These observation arise from the various network categories in use on satellites, many times within a single satellite: multidrop buses, where a central bus master maintains tight control over a number of slaves; point-to-point serial interface, which are mainly used for sensor and instrument connections (and sometimes bulk data transfers); and homogenous

networks, which consist of PEs with similar computing power that communicate on a peer-to-peer basis. In view of these observations, the SOIS concept attempts to completely disassociate network and hardware from users. To accomplish this, SOIS operates through three service interface layers: the SM support layer, the transfer layer, and the subnetwork layer.

### 5.1.1 SM Support Layer

The SM support layer provides common spacecraft capabilities for SMs, isolating user-space from network topology, communications architecture, and physical hardware realizations of spacecraft systems. This currently consists of five services: command and data acquisition services (CDAS), time access service (TAS), message transfer service (MTS), file and packet store services, and device enumeration service (DES).

#### 5.1.1.1 Command and Data Acquisition Services (CDAS)

The first service in the SM support layer is CDAS, which details both commanding and obtaining data from devices on the spacecraft, regardless of location; these devices are hardware devices such as sensors or actuators. The CDAS further splits into three distinct services to provide access to such devices: the device access service, device virtualization service, and the device data pooling service. Each service progressively provides more user abstraction from the endpoint hardware device. The location of this service in the SOIS framework is shown in Figure 25.

#### 5.1.1.2 Device Access Service (DAS)

The DAS is the most basic service for commanding or reading from a hardware device. This service isolates users from the physical location of the device only, providing access through a physical device identifier. This physical device identifier, as well as a value identifier for the operation and any parameters, is used to address the device for commanding or reading values, since such target hardware devices typically only asynchronously emit data. This service is best described as a Device Specific Driver.

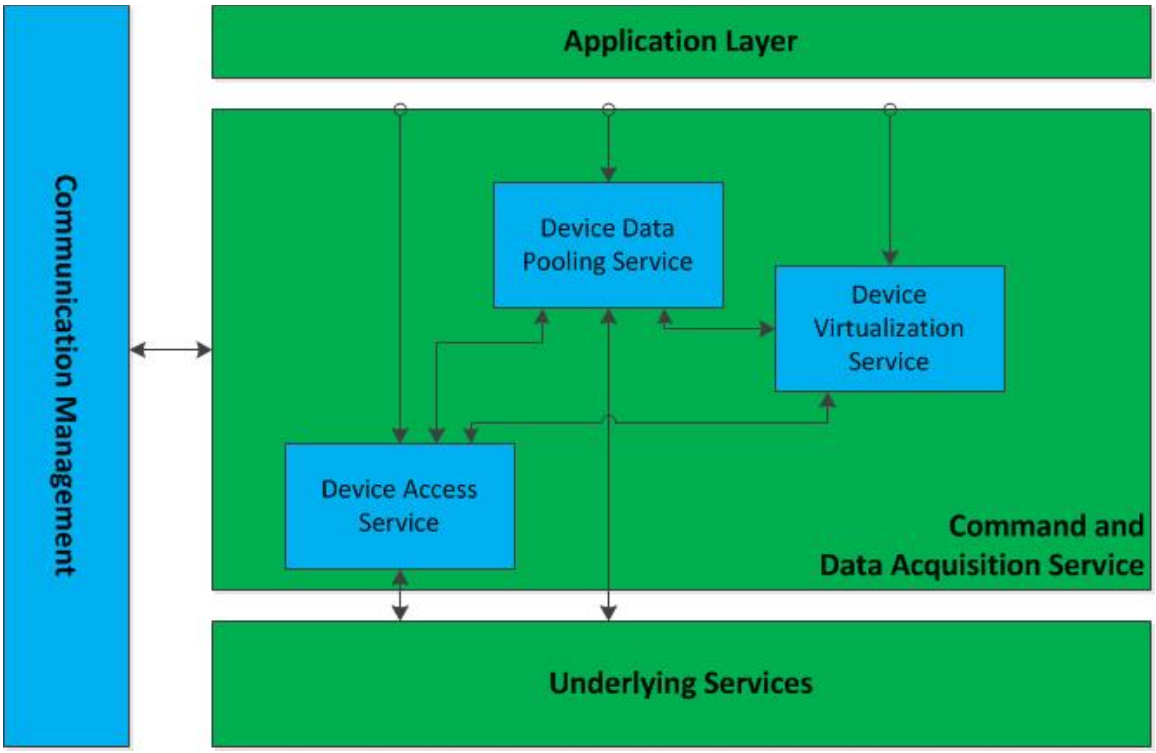


**Figure 25: CDAS Services [101]**

#### 5.1.1.3 Device Virtualization Service (DVS)

The DVS abstracts all elements of a hardware device's physical embodiment from the SM, and is instead represented by a virtual or generic image of that hardware device, such as a PC user's interaction with a disk drive. The user is entirely abstracted from the physical characteristics of the device, including its location and operation; instead, the virtual device provides an idealized interface with a set syntax and simple semantics that allow changing hardware configuration and part numbers with no knowledge to the SM, provided that these parts fulfill the simplified definition of the virtual device. These virtual devices offer commanding (parameter modification on the device) and acquisition (return of requested parameter on device) operations. These are specified using a Dictionary of Terms (DoT) to compose an engineering profile of the device, which is interpreted by the SOIS services in an Electronic Data Sheet (EDS). This is similar to SPA in its description of a device's capabilities and properties in an EDS (xTEDS in SPA); however, whereas SPA utilizes these xTEDS in run-time and dynamic discovery and registration of devices, SOIS espouses their use at design-time. SOIS claims that the real benefit of this device virtualization is with design-time adaptability, allowing hardware changes that are invisible to the SM, which sees only a generic functional interface.

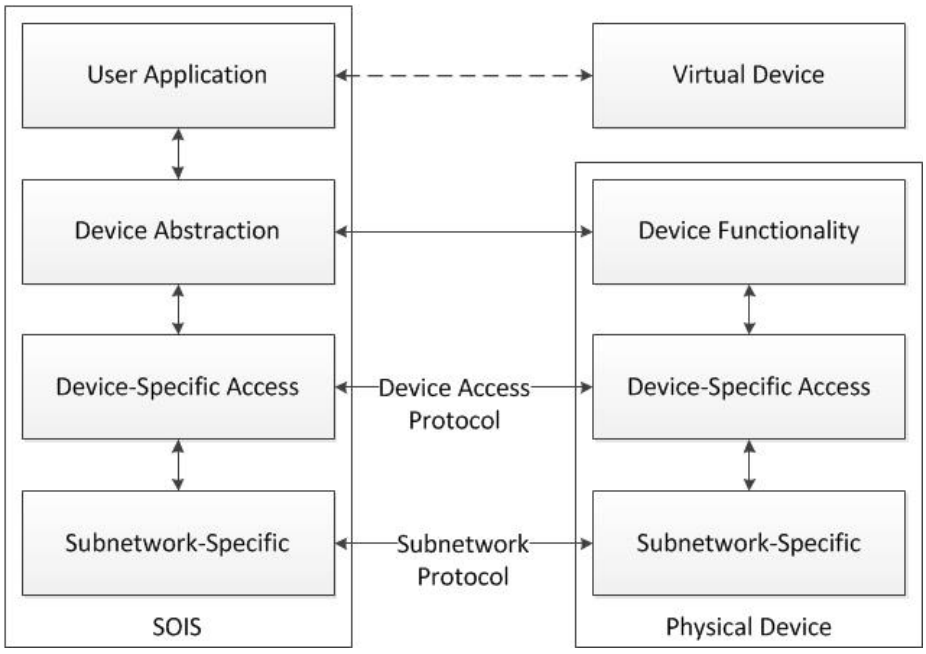


**Figure 26: SOIS Device Virtualization [101]**

Figure 26 depicts the device virtualization process, again utilizing the layering of protocols. Starting from the bottom, the subnetwork protocol transfers the actual data between SOIS and the hardware device. Above that, the device-specific access protocol (DAP) maps the generic upper-level functions to the physical device being connected to through the subnetwork protocol. While this level is device-specific, it is not subnetwork-specific. Above the DAP is the Device Abstraction Control Procedure (DCAP) block, which hides the physical device by mapping the generic functional interface to the device-specific layer. This layer may map single generic user functions onto several device-specific functions, may contain state machines to obtain several pieces of information from devices required by a generic function, and may perform any type conversions required by the DAP. This service is best described as a Standard Device Driver.

#### 5.1.1.4 Device Data Pooling Service (DDPS)

This optional service periodically requests the status and caches values from each of the hardware devices connected, in order to provide better servicing time when full data acquisition operations through the DAP or DVS are not required. This is particularly true to periodic requests with some predetermined sampling rate.

#### 5.1.1.5 Time Access Service (TAS)

The second service in the SM support layer provides an interface to a consistent local time source for all SMs. The most basic capability of this service is a “wall clock”, where any SM can request the time on demand for time-stamping and scheduling. Optionally, an “alarm clock” capability, where SMs are notified at a specific time, and a “metronome” capability, where SMs are notified at periodic intervals, can be provided as well through this service.

#### 5.1.1.6 Message Transfer Service (MTS)

The third service in the SM support layer allows for SMs to send and receive messages from each other. Each service user is identified by a unique MTS PE identifier, and messages are addressed using this identifier. The MTS provides basic sending and receiving of messages, with priority. Optionally, the MTS can provide multicast (publish) and broadcast (announce) functions.

When the transmitting and receiving SMs are on the same processor, MTS addresses the destination SM using its MTS PE identifier and places the message in a priority first in, first out (FIFO) queue. When the transmitting and receiving SMs are on different processors, MTS uses the Asynchronous Message Service (AMS) to provide prioritized delivery and bounded delivery times for messages. AMS is a CCSDS protocol that is defined in Blue Book form, meaning it is a recommended standard. Broadly, AMS is an OSI SM Layer service that relies on underlying transport layers to transfer physical information. These transport layers can include any network transportation services that give by senders and receivers access, including TCP/IP, multi-master $I^2C$, CAN, etc. AMS implements four messaging models: asynchronous send/receive, where single messages are sent to designated PEs; synchronous query, where sending SMs suspend activity until receiving SMs reply; publish/subscribe, where anonymous messages are published to set of subscribers; and announcement, where messages are simultaneously sent to SMs chosen by the sender [102].

#### 5.1.1.7 File and Packet Store Services

The fourth service in the SM support layer allows for users to manipulate and transfers files and packets, which can include science data, images, commands, telemetry, etc. This service also abstracts the file system implementation from the user. There are four categories of services used to accomplish this: file access service (FAS), file management service (FMS), packet store access service (PSAS), and packet store management service (PSMS).

#### 5.1.1.8 File Access Service (FAS)

This service gives user access to files, including basic open, close, read, and write operations. These can be provided through a Network File Access Protocol for use on the subnetwork service if the files are located on a different data service; however, if on the same data system as the user, a set of capabilities are required by the file store: directory list, create file, open file, close file, read file, write to file, delete file, move file, and copy file. Additional directory manipulation capabilities can be provided. Finally, this file store must maintain the name, creation time, last write time, lock status, and file size for each file.

#### 5.1.1.9 File Management Service (FMS)

The FMS allows users to manipulate existing files, regardless of their location on the same data service or across a network. The FMS provides the following capabilities to users: directory list, create a file, delete a file, copy a file, and move a file. Further directory manipulation can be optionally provided.

#### 5.1.1.10 Packet Store Access Service (PSAS)

This service is the packet analogue of the FAS, and allows the user to operate on packet stores, which are different than file stores in that they are the frames used to route and exchange messages. These operations include getting packet store information, clearing the store, writing packets, reading packets, moving packets, freeing packets, and reporting the status of a packet store.

#### 5.1.1.11 Packet Store Management Service (PSMS)

This service is the packet analogue of the FMS, allow packet stores to be created and removed, regardless of the packet store's location.

#### 5.1.1.12 Device Enumeration Service (DES)

The fifth and final service in the SM support layer assigns a system-wide unique virtual device identifier to each device in the system, and is used to verify that each of those devices meet the configurations required by the system. This service enumerates devices by discovering devices, and optionally allows for dynamic discovery through a Device Discovery Service (DDS). These enumeration styles generally follow the SOIS definitions of plug-and-play. These definitions fall into three levels: no plug-and-play, where are device IDs and network addresses are hardcoded; device capability verification, where the DDS validate each device's metadata to ensure that the expectations of the device meets the actually capabilities of the device; and Device

Discovery, where DES dynamically discovers devices and configures DAS and DVS to allow SMs to use and access them.

### 5.1.2 Transfer Layer

The second layer of SOIS provides packet routing on a spacecraft network, and may not be required if no packet routing service between subnetworks is required. Examples of such packet routing protocols are TCP/UDP/IP and Space Packet Protocol. If required, the Transfer Layer will usually consist of two services:

1. Packet Routing
2. System Addressing

As this layer is optional depending on the configuration, and is not essential to SOIS functionality, it is discussed in no more detail.

### 5.1.3 Subnetwork Layer

The third and bottom of SOIS consists of services that transfer and synchronize packets between SOIS users and devices. The SM support layer utilizes these services implement the services provided to the user. These services are: packet service, memory access service, synchronization service, device discovery service, and test service.

#### 5.1.3.1 Packet Service

This service transfers data packets over buses and subnetworks, and is invoked from a user transparent to the type of network. This service can also multiplex between different kinds of networks on the same system, including serial buses and TCP-style packets. This service is specific to the type of network being interfaced to, but essentially provides address translation, protocol identification, and segmentation of data.

#### 5.1.3.2 Memory Access Service

This service allows the user to directly read/write to a specific memory or register location on a device, bypassing higher-level convenience and virtualization layers.

#### 5.1.3.3 Synchronization Service

This service notifies users of subscribed events in a subnetwork, such as a time requests.

#### 5.1.3.4 Device Discovery Service

This service detects devices becoming active, whether the devices are connected directly to the system or to a subnetwork the system is connected to.

#### 5.1.3.5 Test Service

This service checks the go/no-go status and available error codes of all data systems on the network.

### 5.1.4 Future Work

In order to accomplish the plug-and-play and message-passes mechanisms of SOIS, electronic datasheets (EDS) are used. An EDS is like an Interface Control Document (ICD), but is machine readable. It describes all possible operations for a device, what protocols are required by the device, and human-readable documentation for that device. Since SOIS is still in the definition and standardization state, it is expected that SOIS will adopt the xTEDS format used by SPA, such that SOIS EDS are a superset of SPA xTEDS, allowing for some measure of interoperability and collaboration. Additionally, SOIS will define a DoT for use in engineering profiles, which will define ontology for units of measure, purpose of devices, syntactic type of data produced and required by devices, reference frames, and subjects. This will be especially important for device virtualization, where generic functional interfaces will absolutely require a common DoT to operate correctly. In order for "dumb" devices to comply with SOIS and not require direct connection to the primary SOIS processing unit, Smart Transducer Interface Modules (STIM) can be used as a connection point that handles communication for these dumb devices, including EDS provision. These are functionally equivalent to ASIMs in SPA nomenclature.

## 5.2 Core Flight System (CFS)

The Core Flight System (CFS) middleware was developed by the Flight Software Systems Branch at NASA Goddard Space Flight Center (GSFC). GSFC's Flight Software Systems Branch provides embedded software for on-orbit satellite science missions, and places an emphasis on software reusability and onboard autonomy. The Core Flight System (CFS) is a synthesis and formalization of software layers that have both been in use for nearly 15 years and have yet to be defined. The earliest satellite system to fly with CFS components was the Solar Anomalous and Magnetospheric Particle Explorer (SAMPEX) satellite in 1992; CFS was released as an open source package in 2011.

CFS is targeted toward satellite science observatory missions, and has flown in various forms on several missions to date. These include SAMPEX in 1992, Lunar Reconnaissance Orbiter (LRO) in 2009, Solar Dynamics Observatory (SDO) in 2010, the Van Allen Probes in 2012, the Lunar Atmosphere and Dust Environment Explorer in 2013, and the Global

Precipitation Measurement (GPM) satellite in 2014. Furthermore, GSFC has identified the CFS/cFE architecture as an area of interest for NASA CubeSat development [103]. The Intelligent Systems Division at NASA Ames Research Center has also listed CFS/cFE as the architecture for the proposed BioSentinel CubeSat mission, targeting launch on the Space Launch System (SLS) and will measure radiation-induced DNA damage [104]. In addition to proposed CubeSats, the GSFC creators of CFS/cFE are collaborating with James Lyke of SPA and CCSDS of SOIS to integrate EDS support to further enhance adoption and standardization of CFS.

CFS is composed of a set of layers, with each layer obscuring its implementation and technical details from other layers, as well as the SM. The benefits of this architecture are that it doesn't suggest hardware or operating system implementation; rather, CFS gives hardware and platform independence. CFS consists of five layers, including an SM layer for SMs; an SM library layer for translating these SM's communications into the Core Flight Executive (cFE) layer; a platform abstraction layer to isolate the above CFS layers from the specific OS and hardware implementation, translating generic OS calls into the specific OS calls of the chosen OS and processor through a Platform Support Package (PSP); and the RTOS/BOOT layer, which holds the boot information and RTOS. Currently supported RTOSs include VxWorks and RTEMS, as well as support for desktop Linux. These layers are depicted in Figure 27.

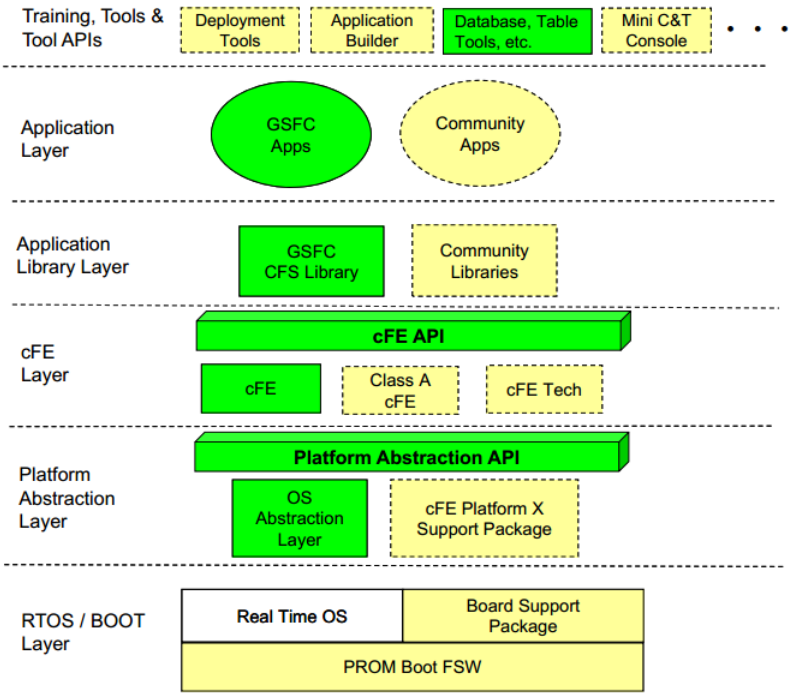


**Figure 27: CFS layers [105]**

### 5.2.1 SM Library Layer

The SM layer in CFS consists of 11 pre-built flight software SMs developed by the GSFC Flight Software Services branch. These include CCSDS File Delivery Protocol (CFDP), which is a CCSDS Blue Book recommended standard that handles file delivery from a spacecraft-based filestore to a ground-based filestore; Checksum (CS), which allows the user to schedule checksum calculations over code and data memory regions; Data Storage (DSS), which stores messages exchanged on the software bus interconnect; File Manager (FM), which provides file management for individual files and directories; Health and Safety (HS), which kicks hardware watchdog timers, monitors the execution of tasks, and takes corrective action should task errors occur; Housekeeping (HK), which can build user-specified telemetry strings; Limit Checker (LC), which monitors user-defined data watch points by checking message data against threshold values; Memory Dwell (MD), which can sample processor addresses and append values to telemetry streams for debugging; Memory Manager (MM), which can perform memory read, write, load, and dump as well as diagnostics; Scheduler (SCH), which implements a time-triggered architecture with 10 millisecond slots for each SM; and Stored Command (SC), which can follow command sequences set at relative or absolute times (ref).

### 5.2.2 cFE Layer

The Core Flight Executive (cFE) layer in CFS is the oldest layer and original middleware implementation that spawned CFS. It provides a set of five services that are used by SMs in the SM library layer, and it provides an abstraction between those higher-layer SMs and the platform abstraction layer below. These reusable core software services are: executive services, event services, software bus services, table services, and time services.

The executive services handle maintenance of spacecraft computer activities. This includes startup, task record keeping, system log, library loading, device drivers, and a Critical Data Store (CDS). After cFE code is loaded into a predetermined address in volatile memory at startup, control is transferred to cFE, which begins loading the higher-layer CFS SMs denoted in a configuration file. The event services allow SMs to send asynchronous debug/error messages, as well as local system logs. These services may also be used during debugging. The software bus service allows for inter-SM messaging. Relying on a previously developed publish/subscribe messaging middleware, SMs publish and subscribe to data completely ignorant of other SMs' requirements. Additionally, this service automatically reports errors during transfers, and can provide statistics on packet delivery and routing. The table service provides tables, which are groups of related parameters similar to a C structure. These tables are used in two ways: they are

shared between SMs, acting as a shared memory resource, and they are used to update mission parameters at runtime. This provides a configuration option after compile-time, allowing for greater flexibility after a mission has commenced by allowing for changes to be made to software without requiring a patch. Finally, the time service provides spacecraft time to SMs, both on demand and through periodic wakeup and time-at-the-tone messages.

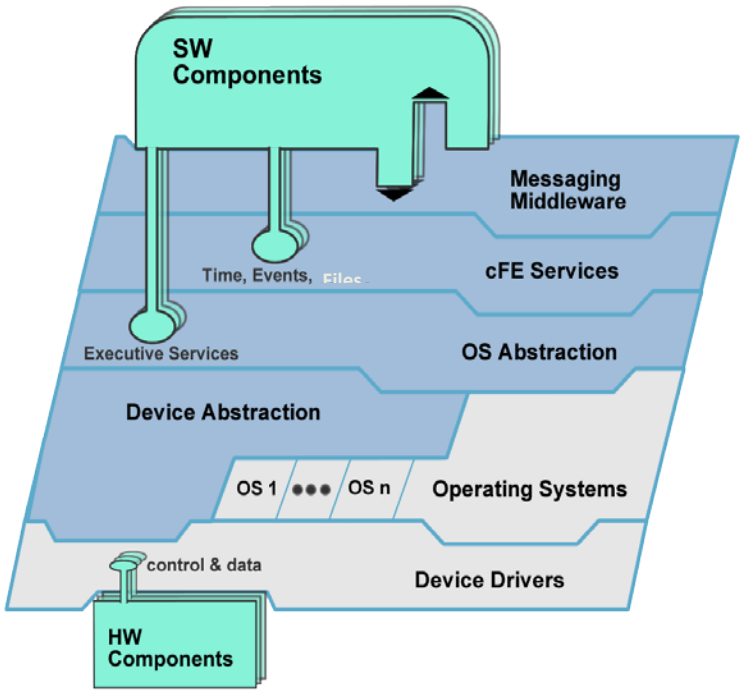


**Figure 28: cFE Layered Architecture [108]**

### 5.2.3 Platform Abstraction Layer

The platform abstraction layer consists of two components: the Operating System Abstraction Layer (OSAL) and the Platform Support Package (PSP), which is a proposed open community component that adapts other operating systems and platforms not supported by OSAL. The PSP can either be written by the user for their specific SM, or one of the existing PSPs can be used. The functions included in the PSP are the startup code, memory read, write, and copy functions, processor-specific reset and exception handler functions, and timer functions. Both the PSP and OSAL are accessed by cFE through a Platform Abstraction API.

#### 5.2.3.1 OSAL

OSAL is a software layer for embedded systems that provides an abstraction layer between specific real-time operating systems and SMs. Created and released by NASA Goddard Space Flight Center in 2010, the goal of OSAL is to allow for greater portability for embedded systems [106]. On the operating system side, OSAL combines and encapsulates the operating system-specific functions into generic functions for the SM. OSAL currently supports three operating systems: RTEMS, an open source real-time distributed operating system; VxWorks, a proprietary real-time operating system; and any other POSIX-compliant operating system, such as Contiki, Linux, and SkyOS. NASA hopes to include support for Windows XP as well, with the goal of embedded systems developers being able to port their code between various embedded systems and between embedded systems and desktop personal computers.

The APIs available to users are split into three sections: RTOS API, File System API, and Interrupt/Exception API.

**RTOS API**

The API for RTOS configuration and manipulation cover tasks and queues, as well as semaphores. Any use of OSAL must begin with an OS_API_Init function call, which sets up the internal data structures of OSAL and allows for further use. The RTOS API then splits into six APIs: miscellaneous, queue, semphamore/mutex, task control, dynamic loader and symbol, and timer.

**File System API**

The API for file system usage covers file creation and editing, directory creation and editing, and physical media actions, and is modeled after POSIX file APIs. As long as the underlying file system is POSIX-compliant, OSAL will give the user a common directory structure and a common volume organization. This means that the paths to files will not change between file systems, and the file system can be simulated on desktop computers. In order to use the file system API, the user provides OSAL a “Volume Table” consisting of a unique device name, an implementation-specific physical device name that is the mount point (“/dev” in Linux, for example), a volume type chosen from a predefined set of strings that describe the supported media types (FS_BASED, RAM_DISK, FLASH_DISK_FORMAT, FLASH_DISK_INIT, or EEPROM), a volatile flag indicating whether the volume is volatile, a free and mounted flag that should both be set to false, a volume name and mount point field that are both internal and should

be set to the empty space character, and an block size field that is left empty. After providing this volume table, the OS_mkfs and OS_mount functions can be called to create and mount the file system. The file system API splits into three APIs: file, directory, and disk.

**Interrupt/Exception API**

The API for interrupt/exception handling covers interrupt and exception setup and handling, and maps interrupt numbers to C code to handle the interrupt. The interrupt/exception API splits into three APIs: system interrupt, system exception, and system FPU exception.

### 5.2.4 RTOS/BOOT Layer

The RTOS/BOOT layer consists of the RTOS implementation and programmable read-only memory (PROM) boot software. The RTOS must either be supported by OSAL or the PSP in the platform abstraction layer. The PROM boot software handles early initialization and loads the RTOS and cFE. GSFC commonly uses RAD750- BAE SUROM, Coldfire, and LEON3 with uBoot.

Figure 29 depicts the architecture of the GPM mission. The green blocks all represent cFE layer services, including the green Software Bus for inter-task message routing. The blue blocks all represent SM Library Layer SMs. The yellow and shaded yellow blocks represent specific SMs written for the GPM mission, including command and data handling and guidance, control, and navigation SMs. This figure highlights the large percentage of reused components in the architecture [105].

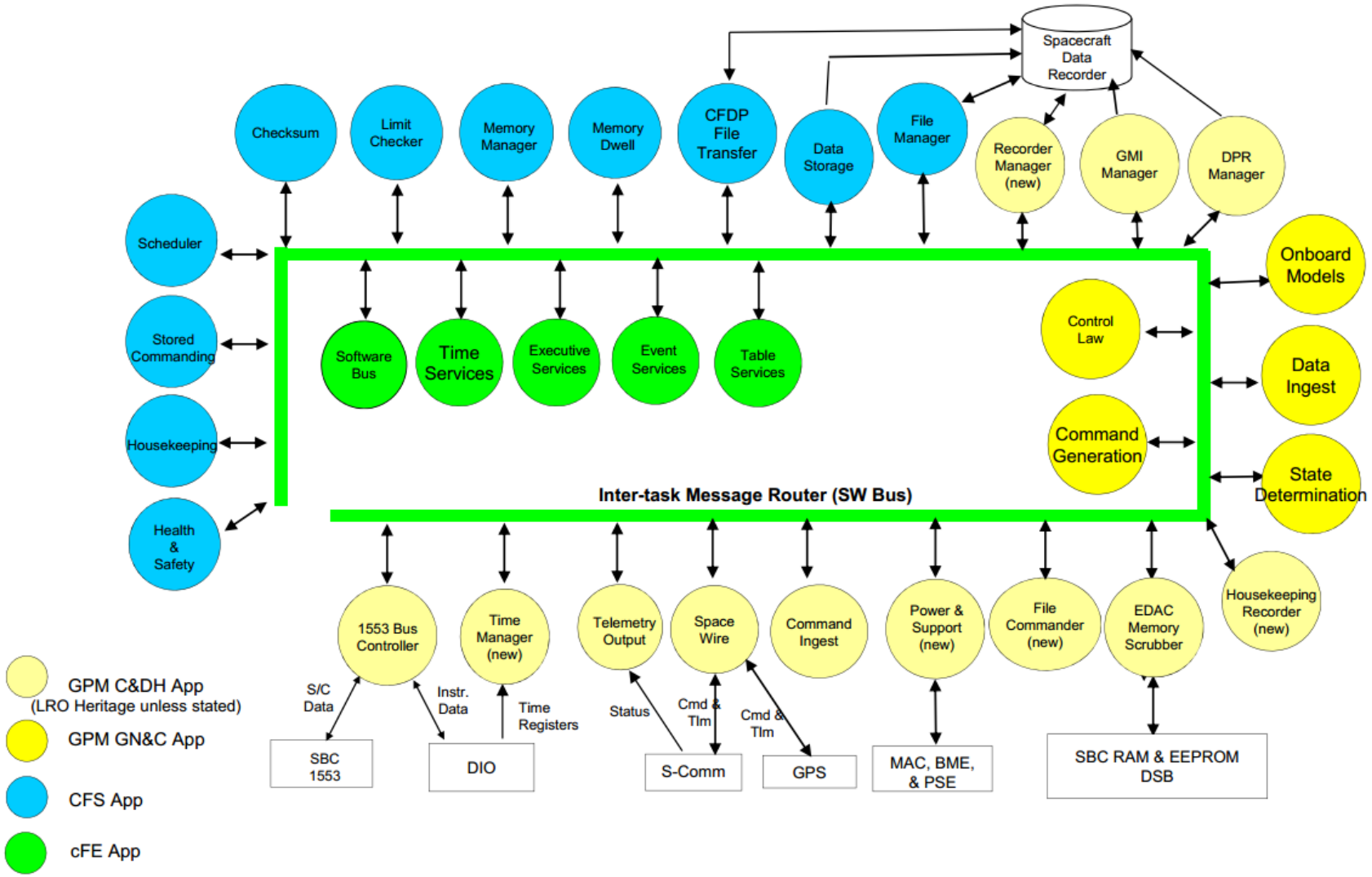


**Figure 29: Example Mission with CFS [105]**

## 5.3 Middleware Aspects

The degrees to which each Common Services layer middleware address the five key aspects of middleware will now be compared and contrasted. The results are summarized in Table 14.

**Table 14: Comparison of Common Services Middleware**

| | Network Communication | Coordination | Reliability | Scalability | Heterogeneity |
|---|---|---|---|---|---|
| **SOIS** | Node-oriented | Both | At-most-once | Access, Location, Replication | Hardware, Network, Software |
| **CFS** | Message-oriented | Asynchronous | At-most-once | Location, Migration, Replication | Hardware, Network, Software |

### 5.3.1 Network Communication

SOIS handles network communication by providing the MTS at the SM support layer. MTS provides priority FIFO queuing, no matter if the receiving SM is on the same processor or a different processor on the network, and AMS-protocol messaging services. AMS messaging uses the subnetwork layer packet services to physically transfer bytes through the network, using implementation-specific protocols such as SpaceWire and Ethernet. These are node-oriented HI-Hardware and HI-Network layer middleware.

CFS handles network communication through cFE's Software Bus service. This software bus implements a message-oriented middleware (MOM) that relies on the publish/subscribe, message-oriented model, and leaves network implementation and drivers to lower levels. It can be theoretically run over any network communication protocol.

### 5.3.2 Coordination

SOIS addresses coordination through the MTS's AMS messaging models, supporting either synchronous or asynchronous messaging. Using asynchronous messaging, sender and receiver SMs are decoupled, allowing the sender SM to continue processing and avoid blocking while waiting for the receiver SM to service the message. Additionally, group messaging is natively supported through the MTS's AMS publish/subscribe and announcement messaging models.

CFS handles coordination through cFE's Software Bus service's asynchronous messaging. SMs are allowed to continue execution after messages are dispatched, and SMs have no knowledge of other SMs through the publish/subscribe messaging model. Each SM must be independent of any other SM, and must be able to start and complete execution on its own.

### 5.3.3 Reliability

SOIS offers at-most-once reliability by requiring the MTS implementation to provide priority FIFO queuing, no matter whether the sender and receiver SMs are on the same PE or not. This means that if either the sender or receiver are currently busy, the message is still available after those SMs complete their tasks; likewise, neither sender nor receiver have to pause or abandon execution of a task in order to service a message.

CFS offers at-most-once reliability through its Software Bus service. Software Bus is a publish/subscribe inter-task router that routes data between tasks running on the same hardware, but interfaces with a serial communications architecture for distributed networking. Historically this serial communications architecture is either SpaceWire or MIL-STD-1553, and CFS's reliability is based on the reliability of those architectures. Furthermore, CFS's Software Bus service reports errors in message transactions but takes no native action to correct those errors, guaranteeing no reliable delivery but guaranteeing no duplicate delivery.

### 5.3.4 Scalability

SOIS exhibits access, location, and replication transparency. These are addressed through SOIS's DVS. By abstracting specific implementation or unique details of hardware endpoints, changes can be made to such endpoints without user interaction. This makes the system more scalable because the DVS can simply replicate and group like hardware endpoints. For example, DVS can address one or many temperature modules by providing a virtual addressing system to a user, while using the same virtualization layers and code. SOIS also addresses scalability by encapsulating common computing services into its SM support layer. SMs need not rewrite such service software, and additional SMs and PEs can be added with no knowledge of how the message transport or device interrogation mechanisms are implemented. While SOIS offers a high degree of scalability by abstracting network and hardware implementation details from the user, this scalability comes primarily at design time. Other middleware architectures, such as SPA, focus on run-time scalability by providing dynamic plug-and-play, self-description through machine-readable datasheets, and periodic device enumeration. SOIS, however, focuses on

design-time plug-and-play by virtualizing devices and automatically generating the software interfaces for Standard Device Drivers.

CFS exhibits location, replication, and migration transparency. CFS handles scalability through its publish/subscribe message model, provided by the Software Bus in cFE. As a publish/subscribe messaging model, new SMs can be added to the common software bus with no knowledge of or dependence on other SMs. These new SMs can either be local to the processor or distributed on other processors, since CFS sets no restriction on the presence of a distributed network. By pushing network communication to lower layers below the operating system, new SMs have no penalty to CFS operation.

### 5.3.5 Heterogeneity

SIOS exhibits hardware, network, and software heterogeneity. SOIS supports this heterogeneity through its layered service approach: as long as the physical hardware at endpoints can respond to MTS messaging following the specified syntax, there is no restriction on hardware or operating system. Additionally, CCSDS is currently compiling and creating the DoT, which will allow any hardware that describes itself in a compliant way to be integrated. The goal of SOIS is to enhance interoperability and encapsulate the use of common services, using no hardware-specific or operating system-specific language. MTS's AMS messaging is an OSI SM-layer protocol, and does not suggest or restrict the lower-level protocols or methods of data transportation.

CFS exhibits hardware, network, and software heterogeneity. Both are supported through CFS's platform abstraction layer. This layer exists solely to abstract the specific operating system and running hardware from the cFE and above layers. As long as each PE in the system uses a compliant operating system, the integration of OSAL allows for any combination of heterogeneous processors and operating systems to be present in the network, with no impact to the cFE and above layers. If the network consists of operating systems or hardware not supported by OSAL, custom PSPs can be written to support those platforms, still requiring no changes to the cFE layer or above layers. Furthermore, CFS's Software Bus routes messages between tasks on the same processor, and interfaces with unspecified serial communications architectures for off-chip communications; historically this is SpaceWire or MIL-STD-1553, but can be expanded to include any serial communications architecture.

# 6 Recommended Middleware Solution

## 6.1 Recommended Methods

This chapter summarizes the results of the middleware comparison from the previous chapters and recommends a solution for the target distributed embedded systems. A survey of common and widely-used middleware for distributed systems highlights how these middleware handle the five primary features of distributed middleware: network communication, coordination, reliability, scalability, and heterogeneity. Table 15 provides a summary of each of these requirements.

**Table 15: Summary of middleware aspects**

| | Network Communication | Coordination | Reliability | Scalability | Heterogeneity |
|---|---|---|---|---|---|
| **CAN** | Message-oriented | Asynchronous | At-least-once | Location, Replication | Hardware, Software |
| **$I^2C$** | Node-oriented | Synchronous | At-most-once | None | Hardware, Software |
| **USB** | Node-oriented | Mixture | Mixture | None | Hardware, Software |
| **Ethernet** | Node-oriented | Asynchronous | At-most-once | None | Hardware, Software |
| **UART** | Node-oriented | Asynchronous | At-most-once | None | Hardware, Software |
| **SpaceWire** | Node-oriented | Synchronous | At-most-once | Location | Hardware, Software |
| **MAVLink** | Message-oriented | Asynchronous | At-most-once | Location, Replication | Hardware, Network, Software |
| **SDM-Lite** | Node-oriented | Asynchronous | At-most-once | None | Hardware |
| **SPA-1L** | Node-oriented | Asynchronous | At-least-once | Location | Hardware |
| **AFDX** | Node-oriented | Asynchronous | Exactly-once | None | Hardware, Software |
| **TTCAN** | Message-oriented | Asynchronous | At-least-once | Location, Replication | Hardware, Software |

**Table 15, continued: Summary of middleware aspects**

| | Network Communication | Coordination | Reliability | Scalability | Heterogeneity |
|---|---|---|---|---|---|
| **CAN-Aero** | Both | Both | At-least-once | Location, Partial replication | Hardware, Software |
| **Ardea** | Message-oriented | Asynchronous | Exactly-once | Migration, Replication | Network |
| **MeRL** | Message-oriented | Both | At-least-once | Access, Location | Hardware |
| **SPA** | Node-oriented | Synchronous | At-most-once | Location | Hardware, Network |
| **MIL-STD-1553** | Node-oriented | Synchronous | At-most-once | None | Hardware, Software |
| **LonTalk** | Node-oriented | Asynchronous | Mixture | Access, Location | Software |
| **CORBA Profiles** | Message-oriented | Synchronous | At-least-once | Access, Location | Hardware, Network, Software |
| **uORB** | Message-oriented | Asynchronous | At-most-once | Access, Location, Replication | Hardware, Network, Software |
| **XML-RPC** | Message-oriented | Asynchronous | Mixture | Location | Hardware, Network, Software |
| **SOIS** | Node-oriented | Both | At-most-once | Access, Location, Replication | Hardware, Network, Software |
| **CFS** | Message-oriented | Asynchronous | At-most-once | Location, Migration, Replication | Hardware, Network, Software |

The survey of middleware provided by this thesis captures several methods of handling each middleware requirement. While all methods reviewed have their merits according to their deployment environment, there are preferred methods specifically for the distributed embedded systems targeted by this thesis: CubeSats and UAVs. While the ideal case is for each middleware requirement to be handled in the most fault-tolerant and robust way, this is often not possible given the resource-constrained and often harsh environments these systems operate in.

### 6.1.1 Network Communication

The recommended method for handling network communication on the target systems is through a message-oriented, mixed time-triggered and event-triggered architecture. This is also the most fault-tolerant way to handle network communication. Message-oriented, as reviewed in Chapter 2, means that the traffic between PEs on a network are not addressed to specific endpoints. Rather, the messages themselves contain a message identifier and the SMs on PEs determine whether or not to interpret the message. This means that messages normally sent to multiple PEs need only be sent once, reducing network traffic and ensuring that all required PEs see the same message.

A time-triggered architecture, as reviewed in Chapter 2, means that each PE is given a specific period of time during which to transmit data or request information; an event-triggered architecture, also reviewed in Chapter 2, means that messages and data are transferred on-demand, as needed between PEs. From a fault-tolerance standpoint, the TTP is most ideal because it prevents starvation of any one node, offers a constant and known message latency, can optimize bus loading, and is contention-free. However, pure TTP introduces latency in large data transfers, which forces data to be segmented and leads to longer transfer times. Furthermore, PEs that do not need access to the bus during their allotted time slot lead to wasted cycles, further increasing the overall power consumption and latency in responding to outside stimuli. ETP, while less fault-tolerant than TTP and yielding sometimes unpredictable latency and bus loading, can be more efficient in large data transfers and more responsive to external stimuli, both highly applicable to satellite and UAV maneuvering and science data. The recommended solution is to blend TTP and ETP: give each PE an allotted time slot for data request and transmission but a channel for ETP in high-priority situations.

### 6.1.2 Coordination

The recommended method for handling coordination on the target systems is through asynchronous communication. This is also the most fault-tolerant way to handle coordination.

Asynchronous communication means that transmitting PE does not need to wait for the receiving PE to receive the data, and there is no clock synchronization between the two. This allows the transmitting PE to service other tasks while the receiving PE checks the validity of the data, retransmitting if necessary according to the reliability needs of the system.

### 6.1.3 Reliability

The recommended method for handling reliability on the target systems is through guaranteeing at-most-once reliability. The most fault-tolerant method for handling reliability is to guarantee exactly-once reliability, where communications are guaranteed to be delivered correctly and only once. At-most-once means that the message integrity is not guaranteed, but message duplication is guaranteed not to occur. In the resource-constrained and limited communications environment of CubeSats, guarding against duplication is more important that invalid data because unpredicted or duplicate actions could lead to significant changes in the satellite system. For example, duplicate deliveries of power commands could yield an unknown subsystem power status for ground controllers, potentially altering the power budget of the satellite on orbit. Likewise, duplicate deliveries of science commands could lead to extraneous or convoluted data that ruins experiments or produces false or unknown results. Similarly, duplicate command deliveries on UAVs could significantly alter the system's flight path or flight mechanics.

### 6.1.4 Scalability

The recommended method for handling scalability on the target systems is through minimal adoption of location and replication transparency. The most scalable method of handling scalability would be to offer access, location, replication, and migration transparency. Message-oriented network communication implies both location and replication transparency because the makeup or location of PE endpoints are not needed for the transmitting PE; this means that the user SM can transmit messages agnostic to the location and redundant number of PEs on the network.

### 6.1.5 Heterogeneity

The recommended method for handling heterogeneity on the target systems is through minimal adoption of hardware heterogeneity. The most expansive method of offering heterogeneity is to adopt hardware, network, and software heterogeneity. The resource-constrained and harsh environments occupied by CubeSats and UAVs demand low-power consumption PEs; while PEs in modern CubeSats and UAVs have experience exponential

advances in processing power, they are still more limited than desktop and enterprise environments. Thus, the need for software heterogeneity to support a variety of operating systems and programming languages does not exist, as most low-power microcontrollers are programmed in C and run low-level RTOSs or light versions of Linux. Furthermore, the need to for network heterogeneity to support a variety of network protocols and signaling levels does not exist, since most PEs used in the target systems offer hardware support for several protocols and have rigidly-set signaling levels anyway. Hardware heterogeneity, however, allows for a variety of microcontroller architectures to coexist on the network as according to the computational needs of the system. The availability of special-purpose microcontrollers at different levels of power consumption allows for such a distributed embedded system to optimize the power consumption and computational capabilities of each PE in the network, and middleware support for these various architectures should be given.

### 6.2 Recommended Implementation

A comparison between the ideal and the recommended methods of handling the middleware requirements are summarized in Table 16.

**Table 16: Possible middleware configurations**

| | Network Communication | Coordination | Reliability | Scalability | Heterogeneity |
|---|---|---|---|---|---|
| **Ideal** | Message-oriented | Asynchronous | Exactly-once | Access, Location, Replication, Migration | Hardware, Network, Software |
| **Recommended** | Message-oriented | Asynchronous | At-most-once | Location, Replication | Hardware |

From the set of reviewed middleware, there are several middleware options that fulfill most of the fault-tolerant and the recommended methods for handling these middleware requirements. These options are summarized in Table 18. In this table, the middleware requirements have been abbreviated, where NC is network communication, C is coordination, R is reliability, S is scalability, and H is heterogeneity. The color codes indicate whether each stated middleware fulfills the ideal or recommended methods for handling each particular requirement. Green means that the middleware fulfills the requirement exactly, yellow means the middleware

fulfills the method partially, and red means the middleware does not fulfill the method. To numerically compare the compliance, each requirement is worth a total of 12 points. This total is to allow for ratios of points for partial compliance. For example, there are four possible scalabilities: access, location, replication, and migration. However, there are only three possible heterogeneities: hardware, network, and software. In order to compare these, the maximum compliance for each category is out of 12, which is the least common multiple between these categories. Table 17 breaks down the point categories.

**Table 17: Compliance Ratings**

| | Compliance to Model | | | | | | |
|---|---|---|---|---|---|---|---|
| | Full | Half | One-third | Two-thirds | One-fourth | Three-fourths | None |
| Network Communication | 12 | 6 | N/A | N/A | N/A | N/A | 0 |
| Coordination | 12 | 6 | N/A | N/A | N/A | N/A | 0 |
| Reliability | 12 | 6 | N/A | N/a | N/A | N/A | 0 |
| Scalability | 12 | 6 | 4 | 8 | N/A | N/A | 0 |
| Heterogeneity | 12 | 6 | N/A | N/A | 3 | 9 | 0 |

### 6.2.1 Ideal Model

The ideal model suggests that the middleware should exhibit message-oriented network communication, asynchronous communication, an exactly-once reliability guarantee, access, location, replication, and migration transparency, and hardware, network, and software heterogeneity. A summary of how each middleware compares against this ideal model is in Table 18.

**Table 18: Middleware Compliance with Ideal Model**

| Model | Middleware | Requirements | | | | | Score |
|---|---|---|---|---|---|---|---|
| | | NC | C | R | S | H | |
| Ideal | $I^2C$ | | | | | 8 | 8 |
| | CAN | 12 | 12 | | 6 | 8 | 38 |
| | USB | | 6 | 6 | | 8 | 20 |
| | Ethernet | | 12 | | | 8 | 20 |

**Table 18, continued: Middleware Compliance with Ideal Model**

| Model | Middleware | Requirements | | | | | Score |
|---|---|---|---|---|---|---|---|
| | | NC | C | R | S | H | |
| Ideal | UART | | 12 | | | 8 | 20 |
| | SpaceWire | | 12 | | 3 | 8 | 23 |
| | MAVLink | 12 | 12 | | 6 | 12 | 42 |
| | SDM-Lite | | 12 | | | 4 | 16 |
| | SPA-1L | | 12 | | 3 | 4 | 19 |
| | AFDX | | 12 | 12 | 3 | 8 | 35 |
| | TTCAN | 12 | 12 | | 6 | 8 | 38 |
| | CAN-Aero | 6 | 6 | | 6 | 8 | 26 |
| | Ardea | 12 | 12 | 12 | 6 | 4 | 46 |
| | MeRL | 12 | 6 | | 6 | 4 | 28 |
| | SPA | | | | 3 | 8 | 11 |
| | MIL-STD-1553 | | | | | 8 | 8 |
| | LonTalk | | 12 | 6 | 6 | 4 | 28 |
| | CORBA/e | 6 | | | 3 | 12 | 21 |
| | uORB | 12 | 6 | | 9 | 12 | 39 |
| | XML-RPC | 12 | 12 | 6 | 3 | 12 | 45 |
| | SOIS | | 6 | | 9 | 12 | 27 |
| | CFS | 12 | 12 | | 9 | 12 | 45 |

There are middleware implementations reviewed by this thesis that address all five requirements according to the ideal model. The criteria for partial compliance (yellow in the table) encompass two scenarios: “mixture” or “both” compliance from Table 15 or a subset of the available transparencies in scalability and heterogeneities. A highest score of 46 is held by Ardea, which exhibits full compliance with three of the five requirements and partial compliance with the remaining two. Its dependency graph ensures that the system will always supply the required data to each SM if that data is available in the system, and its ability to migrate SMs between PEs help give it the exactly-once reliability guarantee. Furthermore, its use of CAN-Aerospace in applications and recommendation of some CAN-based communications for future work give it its message-oriented and asynchronous aspects. Finally, it could be brought even closer to the ideal

model by abstracting its reliance on the uC/OS-II RTOS, and by being more tightly integrated with MeRL to adopt MeRL's access and location transparency.

### 6.2.2 Recommended Model

The recommended model suggests that middleware exhibit message-oriented network communication, asynchronous communication, an at-most-once reliability guarantee, minimally location and replication transparency, and minimally hardware heterogeneity. A summary of how each middleware compares against this recommended model is in Table 19.

**Table 19: Middleware Compliance with Recommended Model**

| Model | Middleware | Requirements | | | | | Score |
|---|---|---|---|---|---|---|---|
| | | NC | C | R | S | H | |
| Recommended | $I^2C$ | | | 12 | | 12 | 24 |
| | CAN | 12 | 12 | | 12 | 12 | 48 |
| | USB | | 6 | 6 | | 12 | 24 |
| | Ethernet | | 12 | 12 | | 12 | 36 |
| | UART | | 12 | 12 | | 12 | 36 |
| | SpaceWire | | | 12 | 6 | 12 | 30 |
| | MAVLink | 12 | 12 | 12 | 12 | 12 | ***60*** |
| | SDM-Lite | | 12 | 12 | | 12 | 36 |
| | SPA-1L | | 12 | | 6 | 12 | 30 |
| | AFDX | | 12 | 12 | 6 | 12 | 42 |
| | TTCAN | 12 | 12 | | 12 | 12 | 48 |
| | CAN-Aero | 6 | 6 | | 12 | 12 | 36 |
| | Ardea | 12 | 12 | 12 | | | 36 |
| | MeRL | 12 | 6 | | 6 | 12 | 36 |
| | SPA | | | 12 | 6 | 12 | 30 |
| | MIL-STD-1553 | | | 12 | | 12 | 24 |
| | LonTalk | | 12 | 6 | 6 | | 24 |
| | CORBA/e | 12 | | | | 12 | 24 |
| | uORB | 12 | 6 | 12 | 12 | 12 | 54 |
| | XML-RPC | 12 | 12 | 6 | 6 | 12 | 48 |
| | SOIS | | 6 | 12 | 12 | 12 | 42 |
| | CFS | 12 | 12 | 12 | 12 | 12 | ***60*** |

While there are no exact matches to an ideal model middleware in this thesis, there are two exact matches to the recommended model for the target distributed embedded systems: MAVLink and CFS. Both middleware handle the middleware requirements using methods deemed most applicable to the target distributed embedded systems of this thesis, but these are on different layers of middleware: MAVLink is HI-Network layer and CFS is Common Services layer. Further comparison between the two is based on the practical implementation limits of the distributed embedded systems themselves, instead of the theoretical handling of middleware requirements.

Since CubeSats and UAVs are the target distributed embedded systems for this thesis, it is useful to examine MAVLink and CFS's translation to those platforms. Neither MAVLink nor CFS has been implemented on CubeSats; however, NASA GSFC has identified CFS as a potential technology for CubeSats, and the NASA BioSentinel CubeSat has tentatively listed CFS for powering its C&DH [103] [104]. Despite this possibility, the class of missions that CFS has flight heritage with are large-scale satellites, such as LRO and GPM, with much larger computer systems and power budgets than those available on CubeSats. For UAVs, MAVLink was designed for and is implemented on a variety of autopilots and ground stations, including ArduPilotMega, SmartAP, PIXHAWK, QGroundControl, and APM Planner [67]; CFS has no history or heritage on UAVs.

**Table 20: Comparison of MAVLink and CFS**

| | MAVLink | CFS |
|---|---|---|
| **CubeSats** | No | No |
| **UAVs** | Yes | No |
| **Missions** | PIXHAWK, ArduPilotMega | LRO, SDO, GPM |
| **Function** | Packing C-structures over serial channels and network abstraction | Common computing services and hardware/OS abstraction |

While both have a long implementation history on missions and commercial platforms, the intrinsic functions of MAVLink and CFS are fundamentally different according to the middleware layer they are classified into by this thesis. As HI-Network layer middleware, MAVLink is chiefly concerned with moving data off one PE and transferring it through the network to another PE. It is a network layer middleware by the OSI model. CFS, however, as a Common Services layer middleware, provides common computing services required for large-

scale satellite missions. Instead of implementing any protocols or layers to transfer data off of one PE, CFS abstracts these HI-Network layer functions and instead provides a software bus interconnect for moving data between tasks. The location of such tasks is abstracted. Therefore, while CFS works well providing common computing services and an API for interfacing to middleware that manage a distributed embedded network, it is further from the hardware level typically seen on the CubeSat and UAV missions that are the motivators for this thesis. MAVLink is a portable C-header file that is lightweight enough to be used on the low power 8- and 32-bit processors favored for CubeSat and UAV missions, and can be easily expanded or restricted by editing human-readable XML files. The MAVLink generator script, freely available online, checks the validity and syntax of MAVLink messages, removing user error from encoding new messages. MAVLink is the recommended middleware for CubeSat and UAV missions.

## 6.3 Performance Analysis

The recommended MAVLink solution is currently available for download on GitHub [107], and is targeted for use in a POSIX-compliant environment and GNU toolchain. In order to provide a performance analysis of MAVLink, the communication library was downloaded and ported for direct use on an ARM Cortex-M4 STM32F4-Discovery board using the Keil toolchain. Benchmark and timing analysis were then performed.

### 6.3.1 Port to Keil Toolchain

MAVLink is a header-only communications library that is automatically generated from an XML specification. This means that the project need only include the generated “mavlink.h” and call the generated message packing and unpacking functions. Since the downloadable MAVLink is intended for a GNU toolchain, several steps were taken to include it in a project using the Keil toolchain. A new project was created in Keil uVision5 to perform the test, and under the **Target Options**, **C/C++** option for that project, **C99 mode** was selected. This allows the functions and structures in the included MAVLink files to be compiled into the project.

After C99 mode was selected, the “mavlink_types.h” file required a “#pragma anon_unions” to allow for the anonymous unions declared in that file. Anonymous unions are unions that can be declared without a class name. Furthermore, MAVLink makes use of packed structures. These packed structures contain a pointer to a running CRC checksum value for when messages are being packed and unpacked, in order to avoid memory space mismatches and thus extra overhead. The automatically generated “checksum.h” file, however, attempts to passed unpacked pointers as function parameters, generating an error within the C99 Keil toolchain. To

fix this, the packing of structures was removed from the “mavlink_types.h”, allowing the pointers being passed unpacked to the functions in “checksum.h”. This was accomplished by commenting out the “#pragma pack(push, 1)” and “#pragma pack(pop)” directives in that “mavlink_types.h” file.

### 6.3.2 Experiment Test Setup

In order to complete performance analysis of the MAVLink middleware on the ARM Cortex-M4 STM32F4-Discovery board, the RTX real-time operating system was used. Experiments were set up to test throughput, latency, and CPU usage. These were measured using software counters through Keil uVision MDK-5 and the Tektronix MSO4034B four-channel mixed signal oscilloscope with two Tektronix TPP0500 probes. This test setup is pictured in Figure 30.

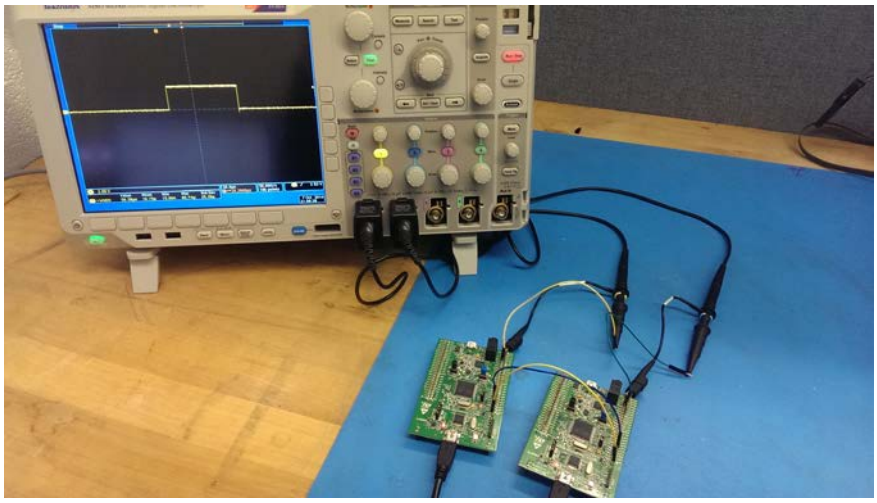

**Figure 30: MAVLink Performance Test Setup**

The software test setup involved Keil uVision MDK-5, a sample project created specifically for these tests. The code for each test is explained in the respective sections below. Finally, custom MAVLink messages were created in the MAVLink message XML specification. These created messages each varied by the number of payload bytes, from a single payload byte up to 248 payload bytes. A sample message is shown below.

```
<message id="155" name="PL1_MAV_TEST">
      <description>Test message for using MAVLink between two
      microcontrollers sending smallest (1 byte) MAVLink
      packet.</description>

      <field type = "uint8_t" name="test_variable">Test variable to
      transfer between two microcontrollers. </field>
</message>
```

This message consists of the unique identifier, a human-readable description, and the single byte data field. Other messages for the test simply added more <field> tags with more unique variables.

#### 6.3.2.1 Throughput

To test the throughput capabilities of MAVLink on the STM32F4-Discovery, two tests were conducted: the amount of time required to pack a single message with different payload sizes, and the number of messages that can be packed per unit time with different payload sizes.

To test the amount of time required to pack a single message with different payload sizes, a GPIO pin was toggled upon entering the MAVLink pack function and again upon exiting the MAVLink pack function, before message transmission. This GPIO pin was monitored by an oscilloscope and the pulse duration was measured.

To test the maximum number of messages that can be packed per unit time, two operating system tasks in RTX were created: a low-priority MAVLink transmit task that performs message marshalling with no delay, and a high-priority task that occurs every one second. Both tasks increment 32-bit counters. By setting a breakpoint in the high-priority task, the number of executions of the low-priority task can be measured by inspecting the low-priority task's counter. The tasks are reproduced below:

```
#if MAVLINK_THROUGHPUT_TEST
//------------------------------------------------------------------
// phaseC
//------------------------------------------------------------------
// Return Value: None
// Parameters:  Argument to pass task if necessary
// Description: This RTX task runs every 1 second, pre-empting a lower-
priority task counting MAVLink pack executions
//------------------------------------------------------------------
void phaseC(void const *argument)
{
    //infinite loop for task
    while(1){
        //increment 32-bit counter to show number of executions of task
        countc++;

        //RTX library call is to delay 1000ms, pre-empting the lower
        //priority task
        //and allowing that lower priority's counter to be read
        osDelay(1000);

    }//end while - will never reach
```

```
}//end phaseC

//---------------------------------------------------------------------
// phaseD
//---------------------------------------------------------------------
// Return Value: None
// Parameters:  Argument to pass task if necessary
// Description: This RTX task transmits a MAVLink message every 500ms
//              and increments a 32-bit counter
//---------------------------------------------------------------------
void phaseD(void const *argument)
{
    //infinite loop for task
    while(1){
        //increment 32-bit counter to show number of executions of task
        // -this counter is read when this task is pre-empted by the
        // higher-priority phaseC
        countd++;

        //Call MAVLink function to pack and transmit MAVLink message
        mavlink_comms_tx();

    }//end while - will never reach
}//end phaseD

//End MAVLINK_THROUGHPUT_TEST code
#endif
```

To transmit data, the mavlink_comms_tx() function was created to pack the appropriate data for the message and transmit it using a serial library function on the native processor – in this case, 115200 baud UART using the CMSIS-Driver API. The mavlink_comms_tx() function is reproduced below:

```
//---------------------------------------------------------------------
// mavlink_comms_tx
//---------------------------------------------------------------------
// Return Value: None
// Parameters:  None
// Description: Packs and transmits MAVLink packet
//---------------------------------------------------------------------
void mavlink_comms_tx(void)
{
        //mavlink_system structure definition, allowing MAVLink packet
        //header information to be set (system ID, component ID)
        mavlink_system_t mavlink_system;
        //Length variable for serial transmission
        uint16_t len = 0;
        //MAVLink message buffer (packed by pack function)
        mavlink_message_t msg;
        //Buffer for transmission
        uint8_t buf[MAVLINK_MAX_PACKET_LEN];

      //---------Information for MAVLink Heartbeat packet from example
```

```
        //Define the system type
        uint8_t system_type = MAV_TYPE_FIXED_WING;
        //Define the autopilot type
        uint8_t autopilot_type = MAV_AUTOPILOT_GENERIC;
        //Define system mode
        uint8_t system_mode = MAV_MODE_PREFLIGHT;
        //Define custom mode
        uint32_t custom_mode = 0;
        uint8_t system_state = MAV_STATE_STANDBY;
        //---------End Information for MAVLink Heartbeat packet

        //Populate system ID for MAVLink header
        mavlink_system.sysid = 20;
        //Populate component ID for MAVLink header
        mavlink_system.compid = MAV_COMP_ID_IMU;
        //Populate type of sysetm for MAVLink header
        mavlink_system.type = MAV_TYPE_FIXED_WING;

        //if this is the first execution of this function, populate
        //buffers
        if (first_run)
        {
                init_test_var();
                init_test_16var();
                init_test_32var();
                init_test_64var();
                //set state variable so that this "if" statement never
                //executes again
                first_run = 0;
        }

        //The XML->C generation of MAVLink messages creates individual
        //pack functions for each message in the specification. Call
        //these functions depending on which message needs to be sent,
        //composing the message. These are differentiated in this test
        //by size of payload

        //Uncomment the message with the desired payload size

        //1 byte payload
        mavlink_msg_pl1_mav_test_pack(mavlink_system.sysid,
            mavlink_system.compid, &msg, test_var[0]);



        //Copy the message to the send buffer
        len = mavlink_msg_to_send_buffer(buf, &msg);

        //Turn off GPIO pin to measure MAVLink latency
        LED_Off(0x00);

        //Call UART write function with the prepared buffer
        //This could be any serial library call, and is not specified
by MAVLink
        CMSIS_UART_Write(buf, USE_UART4, len);

}//end mavlink_comms_tx
```

This test was run without the physical transmission function call to isolate the overhead incurred by invoking the STM32F4-Discovery's native serial library. MAVLink does not specify a serial protocol, and instead leaves this selection to the user.

#### 6.3.2.2 Latency

To test the latency incurred by using MAVLink, two STM32F4-Discovery boards were connected via UART. Each ran the same project, but with a #define denoting the sender and receiver. A general-purpose input/output (GPIO) pin was toggled by the sender upon entering the MAVLink message-packing function call, and a GPIO pin was toggled by the receiver upon exiting its MAVLink message-unpacking function call. Using a two-channel oscilloscope, the time delay between these two GPIO pin toggles was measured to determine the amount of time spent in packing/unpacking and transmitting the message.

To create this test, separate sender and receiver tasks were created: the sender transmits a MAVLink packet every 500ms and the receiver task listens for incoming packets to parse. This receiver task runs every 5ms to minimize latency incurred by the operating system, but is required to avoid emptying the UART receiver buffer too quickly. The tasks are reproduced below:

```
#if !MAVLINK_THROUGHPUT_TEST
//only compile phaseA if programming the transmitter
#if TX
//------------------------------------------------------------------------
// phaseA
//------------------------------------------------------------------------
// Return Value: None
// Parameters:  Argument to pass task if necessary
// Description: This RTX task transmits a MAVLink message every 500ms
//              and increments a 32-bit counter
//------------------------------------------------------------------------
void phaseA(void const *argument)
{
    //infinite loop for task
    while(1){
        //increment 32-bit counter to show number of executions of task
        counta++;

        //RTX library call is to delay 500ms (prompts task switch) to
        //test latency
        //of single messages
        osDelay(500);

        //Turn on GPIO pin when entering MAVLink pack function
        LED_On(0x00);
```

```
        //Call MAVLink function to pack and transmit MAVLink message
        mavlink_comms_tx();

    }//end while - will never reach
}//end phaseA

//End tx-only code
#endif

//------------------------------------------------------------------------
// phaseB
//------------------------------------------------------------------------
// Return Value: None
// Parameters:  Argument to pass task if necessary
// Description: This RTX task checks for new MAVLink messages and
//              processes them, and increments a 32-bit counter
//------------------------------------------------------------------------
void phaseB(void const *argument)
{
    //infinite loop for task
    while(1){
        //increment 32-bit counter to show number of executions of task
        countb++;

        //check for and process new MAVLink messages
        mavlink_comms_rx();

        //RTX library call is to delay 10ms (required to avoid empyting
        //buffer too quickly)
        osDelay(5);

    }//end while - will never reach
}//end phaseB

//End !MAVLINK_THROUGHPUT_TEST code
#endif
```

To transmit MAVLink messages, the same mavlink_comms_tx() function above was used as in the throughput test in 6.3.2.1. To receive and parse data, the mavlink_comms_rx() function was created. This function checks for available data in the STM32F4-Discovery UART buffer using the CMSIS-Drvier API. Unfortunately, the STM32FX-Discovery boards do not support the latest CMSIS-Driver API, meaning that interrupt-based, lower-latency serial communications were not possible. Support for the latest CMSIS-Driver API is expected by the end of 2014. The mavlink_comms_rx() function is reproduced below:

```
//------------------------------------------------------------------------
// mavlink_comms_rx
//------------------------------------------------------------------------
// Return Value: None
// Parameters:  None
// Description: Checks for available MAVLink packets and unpacks
```

```
//---------------------------------------------------------------------
void mavlink_comms_rx()
{
        //variable to hold the number of bytes to read
        int32_t size_to_read = 0;
        //buffer to read MAVLink packed into from serial read function
        uint8_t read_buf[262];
        //counter variable
        uint8_t i = 0;
        //number of bytes actually read from serial read function
        int32_t size_read = 0;
        //flag to denote whether MAVLink unpack function returns a
        //successful message
        uint8_t correct = 0;
        //MAVLink generated message structure containing message
        //parameters
        mavlink_message_t msg;
        //MAVLink generated status structure containing the status of
        //the message
        mavlink_status_t status;

        //Check if data is available
        size_to_read = CMSIS_UART_IsDataAvailable();
        //if data is available, read the data
        if (size_to_read > 0)
        {
                //serial read function
                size_read = CMSIS_UART_Read(read_buf, USE_UART4,
                                                    size_to_read);
                if (read_buf[0] == 0xFE)
                {
        //if a MAVLink start packet is detected, turn on GPIO pin
                        LED_On(0x00);
        //reset correct message flag
                        correct = 0;
        //Loop through read buffer
                        for (i = 0; i < size_to_read; i++)
                        {
//Call MAVLink generated function state machine to parse messge,
//reading result into the msg structure
                                if(mavlink_parse_char(0, read_buf[i],
                                                        &msg, &status))
                                {
//the above function returns a 1 when the end of the message is reached
//and checksums are verified. Set the correct message flag denoting
//this
                                        correct = 1;
//break out of loop
                                        break;
                                }//end if
                        }//end for

//If MAVLink successfully parsed the message
                        if (correct)
                        {
//Handle the message according to the message ID originally coded
//during the XML specification
```

```
                                switch(msg.msgid)
                                {
                                  case MAVLINK_MSG_ID_HEARTBEAT:
                                  {
                                     LED_Off(0x00);
                                     break;
                                  }
                                  case MAVLINK_MSG_ID_PL1_MAV_TEST:
                                  {
                                    //LED_Off(0x00);
                                      break;
                                  }
                                  case MAVLINK_MSG_ID_PL16_MAV_TEST:
                                  {
                                    //LED_Off(0x00);
                                      break;
                                  }
                                  case MAVLINK_MSG_ID_PL16_2B_MAV_TEST:
                                  {
                                    //LED_Off(0x00);
                                      break;
                                  }
                                  case MAVLINK_MSG_ID_PL40_MAV_TEST:
                                  {
                                    //LED_Off(0x00);
                                      break;
                                  }
                                  case MAVLINK_MSG_ID_PL200_8B_MAV_TEST:
                                  {
                                    //LED_Off(0x00);
                                      break;
                                  }
                                  case MAVLINK_MSG_ID_PL248_8B_MAV_TEST:
                                  {
                                    //LED_Off(0x00);
                                      break;
                                  }
                                  default:
                                  {
                                     LED_On(0x01);
                                     break;
                                  }
                                }//end switch
                            }//end if
                        }   //end if
                }//end if

        //A size_to_read less than zero denotes an error within UART system
                else if (size_to_read < 0)
                {
                        size_to_read = 0;
                }

        }//end mavlink_comms_rx
```

#### 6.3.2.3 CPU Usage

To test the CPU usage incurred by using MAVLink, an STM32F4-Discovery board was used with the same RTX tasks. Using the two-channel oscilloscope and isolating the pulse created by activating the GPIO pin upon entering the MAVLink function and deactivating the GPIO pin upon exiting the MAVLink function, the number of CPU cycles required was calculated using the internal clock speed of the STM32F4-Discovery and the duration of the pulse.

The number of CPU cycles required to execute a task can be found by multiplying the number of cycles per second by the number of seconds required to execute the task, given by Equation 1.

$$CPU\ Cycles = \frac{Cycles}{second} * Number\ of\ seconds \quad (1)$$

### 6.3.3 Experiment Results

This section summarizes the results collected by following the tests outlined in 6.3.2. By collecting these results, it is shown that MAVLink is of viable code size and speed for one of the target processors, and can feasibly be integrated into the target sysetms.

#### 6.3.3.1 Throughput

MAVLink packets contain eight bytes of overhead, outlined in 3.2.2. On top of these overhead bytes, the payload field is variable between one byte and 255 bytes. This variability yields MAVLink packets between nine bytes and 263 bytes total. In order to fully characterize the throughput, the number of payload bytes were varied and the test repeated in otherwise identical conditions. The time required to pack each message was measured toggling a GPIO pin upon entering the MAVLink pack function and again upon exiting, before sending the data over UART. This test was repeated over a selection of message sizes. An example of this timing test showing the time to pack a nine-byte payload is shown in Figure 31.

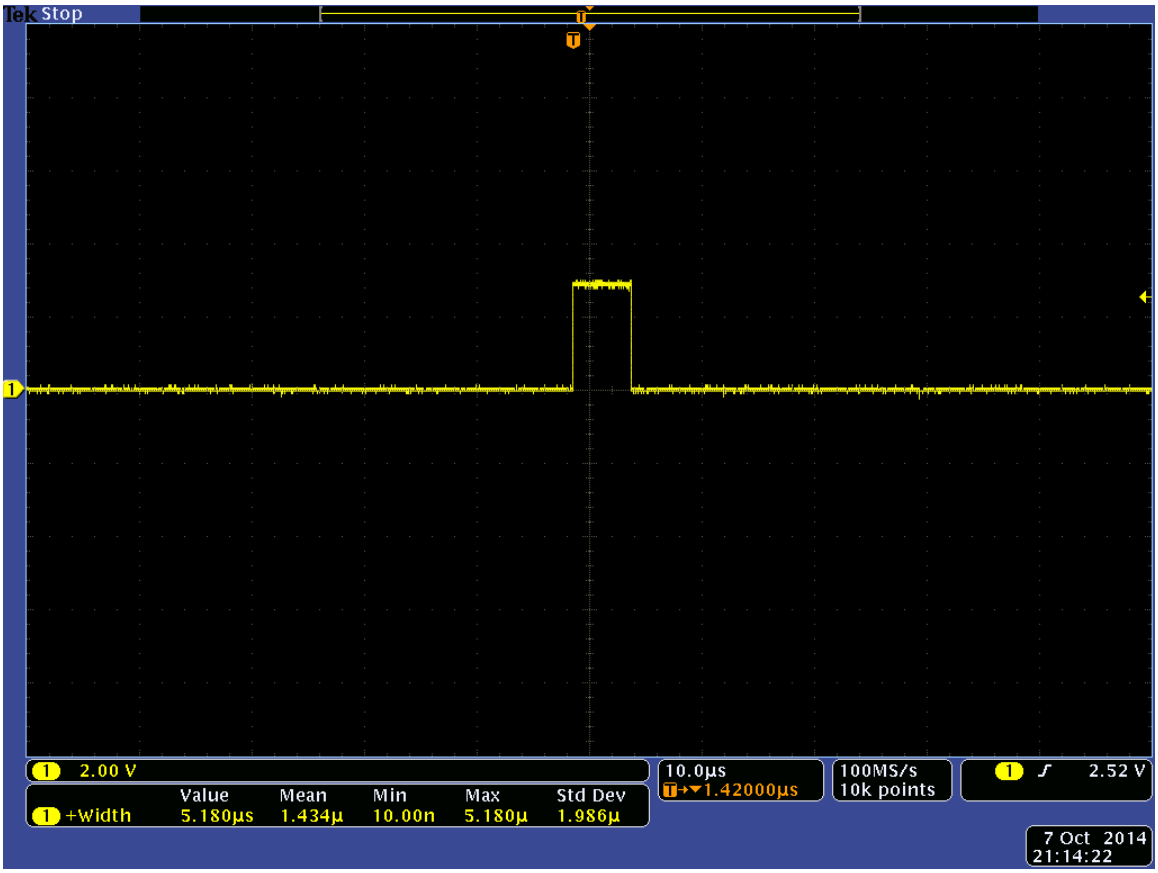


**Figure 31: Time Required for MAVLink to Pack a Nine-Byte Payload Message**

The pack times for larger packets are summarized in Table 21.

**Table 21: MAVLink Message Pack Times**

| Payload Size (bytes) | 1 | 9 | 16 | 40 | 200 | 248 |
|---|---|---|---|---|---|---|
| **Time to Pack** | 3.27µs | 5.18 µs | 6.80 µs | 10.79 µs | 39.56 µs | 48.38 µs |

Next, the number of messages that can be packed per second was measured using the RTX real-time operating system and software counters. The resulting number of MAVLink messages prepared per second for each payload size were plotted in Figure 32.

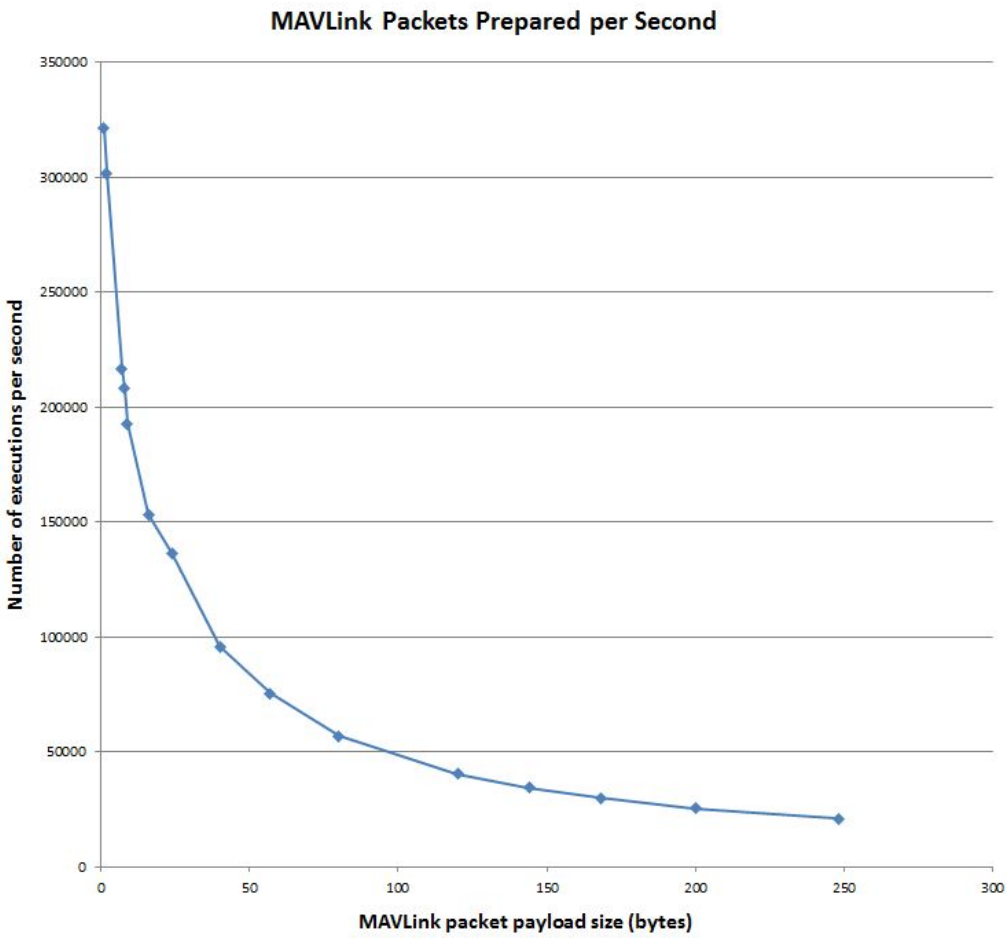


**Figure 32: MAVLink messages packed per second with varying payload sizes**

The above graph shows that MAVLink packets with a one byte payload are packed the fastest, with nearly 325,000 executions per second. As the payload size increases linearly, the number of messages packed per second experiences an exponential decay, indicating an inherent overhead with packing MAVLink packets.

#### 6.3.3.2 Latency

As presented in 6.3.3.1, there is overhead in packing MAVLink messages for transmission, and similarly overhead for unpacking the MAVLink message upon reception. The latency measured is between entering the MAVLink pack function on the transmitting node and exiting the MAVLink unpack function on the receiving node.

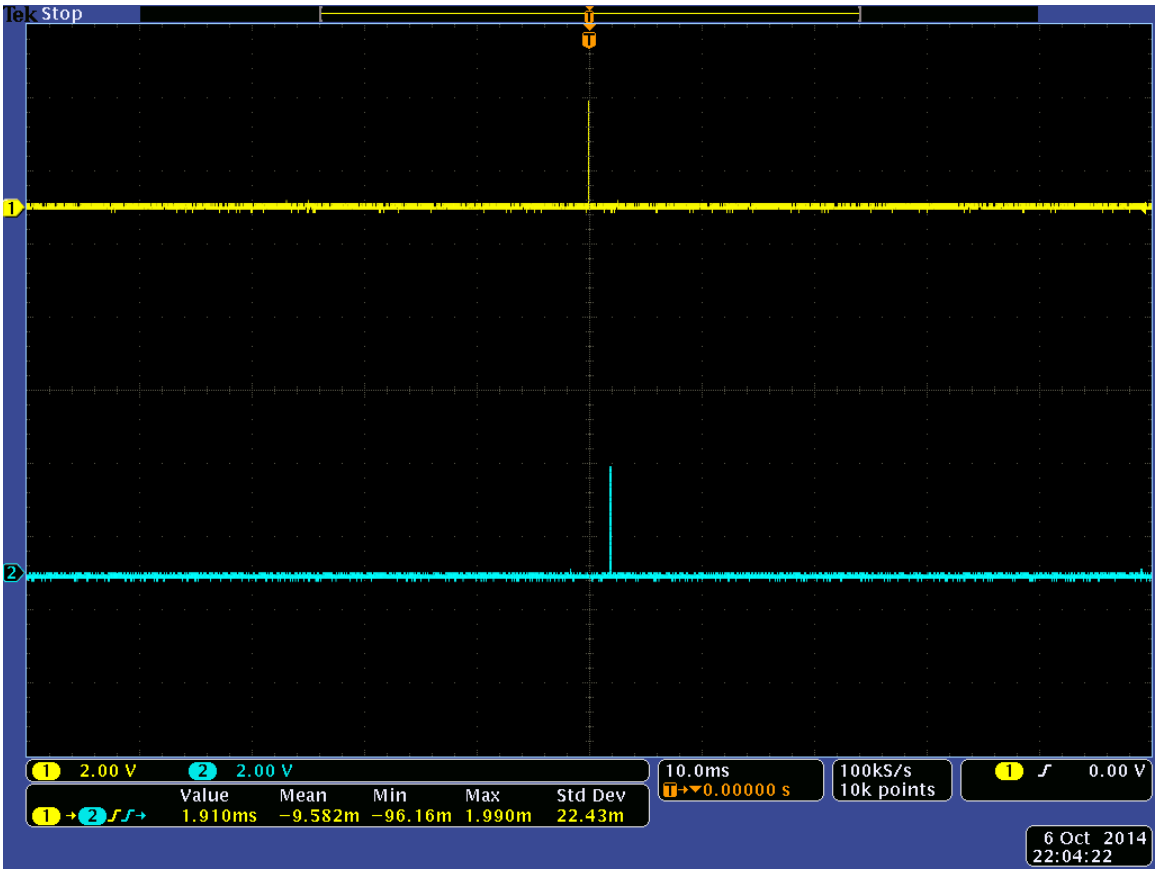


**Figure 33: MAVLink Latency for Nine Byte Payload**

Figure 33 shows an example oscilloscope reading showing the latency for a MAVLink packet with a nine byte payload. The latency between entering the MAVLink pack function on the transmitting node and exiting the MAVLink unpack function on the receiving node is 1.910 ms. Latencies for a representative grouping of MAVLink messages with different payload sizes are summarized in Table 22.

**Table 22: MAVLink Message Latencies**

| Payload Size (bytes) | 1 | 9 | 16 | 40 | 200 | 248 |
|---|---|---|---|---|---|---|
| **Latency** | 1.06ms | 1.91ms | 2.56ms | 10.28ms | 30.52ms | 64.94ms |

#### 6.3.3.3 CPU Usage

As presented in 6.3.2.3 and Equation 1, the number of CPU cycles consumed during the MAVLink packing process can be found by multiplying the system clock speed of the STM32F4-Discovery processor and the number of seconds required to execute that MAVLink pack function.

This was first completed for packing a MAVLink message with a single byte payload. This calculation is given by Equation 2.

$$CPU\ Cycles = \frac{168\ x\ 10^6 cycles}{second} * 3.27\ \mu seconds = 549\ cycles \quad (2)$$

The CPU cycles was then calculated again for packing a MAVLink message with a 248-byte payload. This calculation is given by Equation 3.

$$CPU\ Cycles = \frac{168\ x\ 10^6 cycles}{second} * 48.38\ \mu seconds = 8127\ cycles \quad (3)$$

The additional overhead in CPU cycles is incurred for two reasons: additional load and store operations for the additional bytes, and additional looping in creating the CRC checksums. These usage values are easily handled by the target platform, which in the PIXHAWK implementation of the px4 autopilot for autonomous UAVs packs and unpacks MAVLink messages while running the NuttX RTOS and managing the flight of the vehicle, as described in 2.1.

# 7 Conclusion

This chapter summarizes the motivations and research for this thesis, and proposes an area for future research and validation.

## 7.1 Summary of Work

The purpose of this thesis was to survey existing popular middleware implementations in distributed computing, form a categorization system based on an existing middleware taxonomy, recommend a solution for a targeted set of distributed embedded systems, and finally to perform preliminary experimental characterization of that solution on a target platform. These targeted embedded systems are CubeSats and UAVs, and both operate in harsh and resource-constrained environments. While neither carries human life and need be as safety-critical as an airliner, UAVs fly in proximity to humans and other structures and CubeSats often carry expensive equipment that cannot be returned or physically accessed after launch.

This thesis categorized middleware for distributed embedded applications into three broad categories, based upon an established taxonomy: Host-Infrastructure, Distribution, and Common Services. With Host-Infrastructure, two sub-categories were created: HI-Hardware where the middleware has hardware support built-in to microcontrollers, and HI-Network, where the middleware blindly transfers and routes data on a network. With Distribution layer middleware, two sub-categories were created: D-Transport, where the middleware actively interprets data and offers extension services beyond simply routing data on a network usually for additional fault-tolerance in safety-critical systems, and PEPt, which describes middleware that use object-oriented references to access objects on distributed hosts in order to mask implementation details from each host. The goal of this classification system was not to suggest a single layer as the recommended layer for distributed embedded network management; rather, it was to build upon an established taxonomy in order to better compare similar middleware approaches.

Each middleware reviewed was classified into one of these layers and evaluated on how it addressed the five fundamental requirements of middleware: network communication, coordination, reliability, scalability, and heterogeneity. Two models were offered by this thesis that prescribed specific handling of these requirements: an ideal model, where middleware adopted message-oriented network communication, asynchronous coordination, exactly-once reliability, access, location, replication, and migration transparency, and hardware, software, and network heterogeneity; and a recommended model for the target embedded systems, where

middleware adopted message-oriented network communication, asynchronous coordination, at-most-once reliability, location and replication transparency, and hardware heterogeneity. This recommended model is attainable by the target processors and uniquely suited for use within the target systems, targeting fault-tolerance in message-passing and the transparency required to easily scale the number and function of distributed hosts.

The middleware reviewed by this thesis were then scored based on how closely they followed these models. MAVLink and CFS tied, with both exactly matching the recommended model. In lieu of further quantitative comparison between these middleware on different layers, a qualitative comparison was performed to determine the best option for the target platforms. Based upon its targeted user community, scale of previously flown missions, and ease of integration into existing architectures, MAVLink was the middleware selected as the recommended solution for handling distributed embedded networking on future CubeSat and UAV missions. Finally, an introductory performance analysis was conducted. This performance analysis measured the throughput, latency, and CPU usage incurred by MAVLink. This performance analysis also resulted in a compiled, executable project for the STM32F4-Discovery board that can be used as a baseline for future projects using MAVLink.

## 7.2 Future Work

This research could be valuably extended by constructing a hardware setup consisting of a heterogeneous mixture of low power microcontrollers running a benchmarking test application, including both ARM Cortex-M processors and 8051-core processors. The use of CMSIS core and drivers for the experimental validation portion from thesis will greatly aid in adding more Cortex-M processors. This setup could be used to quantitatively verify transfer speeds, dropped packet percentages, and overall stability of MAVLink against other close competitor middleware, such as CFS, AFDX, and uORB.

This research could also be extended by adding processor-level security into its consideration. With improvements in wireless technology and the increasing use of embedded systems in safety-critical applications, security against hacking is becoming a more important issue. Many implementations of the target processors are available with built-in hardware encryption modules, and there are software techniques to encrypt internal data and verify external data. Such considerations could potentially be very important for autonomous UAVs, flying in close proximity to humans and human-built structures and thus more susceptible to security concerns with dire safety consequences.

## List of Acronyms

| | |
|---|---|
| 1U | 1 Unit |
| ADCS | Attitude Determination and Control System |
| ADN | Aircraft Data Network |
| AFDX | Avionics Full-Duplex Switched Ethernet |
| AMP | Arbitration on Message Priority |
| AMS | Asynchronous Message Service |
| APDU | Application Protocol Data Unit |
| API | Application Programming Interface |
| Ardea | Automatically Reconfigurable Distributed Embedded Architecture |
| ARINC | Aeronautical Radio, Incorporated |
| ARM | Advanced RISC Machine |
| ASCII | American Standard Code for Information Interchange |
| ASIM | Applique Sensor Interface Module |
| ATM | Anyone-to-many |
| BC | Bus Controller |
| BIOS | Basic input/output system |
| C&DH | Command and Data Handling |
| CAN | Controller Area Network |
| CAT5 | Category-5 |
| CCSDS | Consultative Committee for Space Data Systems |
| CD | Collision Detection |
| CDAS | Command and Data Acquisition |

| | |
|---|---|
| CDD | Common Data Dictionary |
| CDS | Critical Data Store |
| CEA | Command Execution A |
| CEB | Command Execution B |
| CFDP | CCSDS File Delivery Protocol |
| cFE | Core Flight Executive |
| CFS | Core Flight System |
| CICS | Customer Information Control System |
| CORBA | Common Object Request Broker Architecture |
| CORBA/e | Common Object Request Broker Architecture/Embedded |
| COTS | Commerical-off-the-shelf |
| CPU | Central processing unit |
| CRC | Cyclic redundancy check |
| CS | Checksum Service |
| CSMA | Carrier sense/multiple access |
| DAP | Device Access Service |
| DCAP | Device Abstraction Control Procedure |
| DDPS | Device Data Pooling Service |
| DDS | Device Discovery Service |
| DES | Device Enumeration Service |
| DG | Dependency Graph |
| DII | Dynamic Invocation Interface |

| | |
|---|---|
| DOC | Distributed Object Computing |
| DoT | Dictionary of Terms |
| DS | Data-Strobe |
| DSI | Dynamic Skeleton Interface |
| DSS | Data Storage |
| DTP | Distributed Transaction Processing |
| DVS | Device Virtualization Service |
| EBCDIC | Extended Binary Coded Decimal Interchange Code |
| EDS | Electronic Data Sheet |
| EEP | Error End of Packet |
| EEPROM | Electrically erasable programmable read-only memory |
| ELaNa | Educational Launch of Nanosatellites |
| EOP | Normal End of Packet |
| EPS | Electrical Power System |
| ESA | European Space Agency |
| ESC | Escape Token |
| ETP | Event-triggered Architecture |
| FAA | Federal Aviation Administration |
| FAS | File Access Service |
| FCT | Flow Control Token |
| FIFO | First in, first out |
| FM | File Manager |

| | |
|---|---|
| FMS | File Management Service |
| GIOP | General Inter-ORB Protocol |
| GNC | Guidance, navigation, and control |
| GPIO | General purpose input/output |
| GPM | Global Precipitation Measurement |
| GPS | Global positioning system |
| GSFC | Goddard Space Flight Center |
| GUI | Graphical user interface |
| GUID | Global unique identifier |
| HAL | Hardware Access Layer |
| HART | Highway Addressable Remote Transceiver |
| HI | Host-Infrastructure |
| HK | Housekeeping Service |
| HS | Health and Safety |
| HTTP | Hypertext Transfer Protocol |
| HVAC | Heating, ventilation, and air conditioning |
| IBM | International Business Machines Corporation |
| I/O | Input/output |
| $I^2C$ | Inter-integrated Circuit |
| ICD | Interface Control Document |
| IDL | Interface Definition Language |
| IEEE | Institute of Electrical and Electronics Engineers |

| | |
|---|---|
| IP | Internet Protocol |
| ISS | International Space Station |
| IT | Information Technology |
| LASIM | Lite Applique Sensor Interface Module |
| LC | Limit Checker |
| LCC | Logical communication channel |
| LRO | Lunar Reconnaissance Orbiter |
| LVDS | Low Voltage Differential Signaling |
| MAC | Media Access Control |
| MAVLink | Micro Air Vehicle Link |
| MD | Memory Dwell |
| MeRL | Message Routing Layer |
| MIB | Management Information Base |
| MIME | Multipurpose Internet Mail Extensions |
| MM | Memory Manager |
| MOM | Message-Oriented Messaging |
| MT | Monitoring Terminal |
| MTS | Message Transfer Service |
| NFS | Network File System |
| NM | Network Monitor |
| ODP | Open Distributed Processing |
| OMA | Object Management Architecture |

| | |
|---|---|
| OMG | Object Management Group |
| ORB | Object Request Broker |
| OSAL | Operating System Abstraction Layer |
| OSI | Open Systems Interconnect |
| PC | Personal computer |
| PCB | Printed circuit board |
| PCI | Peripheral Component Interconnect |
| PE | Processing Element |
| PEPt | Presentation, Encoding, Protocol and transport |
| PnP | Plug-and-play |
| P-POD | Poly Picosatellite Orbital Deployer |
| PPP | Point-to-Point Protocol |
| PROM | Programmable Read-only Memory |
| PSAS | Packet Store Access Service |
| PSMS | Packet Store Management Service |
| PSP | Platform Support Package |
| PTP | Peer-to-peer |
| RC | Radio Controlled |
| RISC | Reduced Instruction Set Computer |
| RPC | Remote procedure call |
| RT | Remote Terminal |
| RTOS | Real-time Operating System |

| | |
|---|---|
| SAMPEX | Solar Anomalous and Magnetospheric Particle Explorer |
| SC | Stored Command |
| SCH | Scheduler Service |
| SCL | Serial Clock |
| SDA | Serial Data |
| SDM | Satellite Data Model |
| SDM-L | Satellite Data Model-Lite |
| SDO | Solar Dynamics Observatory |
| SFD | Start Frame Delimiter |
| SLUGS | Santa Cruz Low-cost UAV (GNC System |
| SM | Software Module |
| SNVT | Standard Network Variable Types |
| SOA | Service-oriented Architecture |
| SOCEM | Suborbital CubeSat Experimental Mission |
| SOIS | Spacecraft Onboard Interface Services |
| SPA | Space Plug-and-play Avionics |
| SPA-1L | Space Plug-and-play Avionics-1 Lite |
| SPI | Serial Peripheral Interface |
| SSC | Space Science Center |
| SSL | Space Systems Laboratory |
| SSTP | Small Spacecraft Technology Program |
| STIM | Smart Transducer Interface Module |

| | |
|---|---|
| TAS | Time Access Service |
| TCP | Transmission Control Protocol |
| TCP/IP | Transmission Control Protocol/Internet Protocol |
| TDMA | Time Division/Multiple Access |
| TEDs | Transducer Electronic Data Sheets |
| TTCAN | Time-Triggered Controller Area Network |
| TTP | Time-Triggered Architecture |
| UART | Universal Asynchronous Receiver/Transmitter |
| UAV | Unmanned Aerial Vehicle |
| UDP | User Datagram Protocol |
| UK | University of Kentucky |
| uORB | Micro Object Request Broker |
| USB | Universal Serial Bus |
| WDT | Watchdog Timer |
| XML | Extensible Markup Language |
| XML-RPC | Extensible Markup Language- Remote Procedure Call |
| XOR | Exclusive-OR |
| xTEDS | Extensible Transducer Electrical Data Sheets |

## References


[1] The CubeSat Program, Cal Poly SLO, "CubeSat Design Specification Rev. 13," February 2014. [Online]. Available: http://www.cubesat.org/images/developers/cds_rev13_final.pdf. [Accessed 12 August 2014].

[2] G. D. Krebs, "Gunter's Space Page," 2014. [Online]. Available: http://space.skyrocket.de/directories/chronology.htm. [Accessed 23 May 2014].

[3] E. Buchen and D. DePasquale, "SpaceWorks Enterprises 2014 Nano/Microsatellite Market Assessment," 2014. [Online]. Available: http://www.sei.aero/eng/papers/uploads/archive/SpaceWorks_Nano_Microsatellite_Market_Assessment_January_2014.pdf. [Accessed 23 May 2014].

[4] J. Lyke, J. Mee, F. Bruhn, G. Chosson, R. Lindegren, H. Lofgren, J. Schulte, S. Cannon, J. Christensen, B. Hansen, R. Vick, A. Vera and J. Calixte-Rosengren, "A Plug-and-play Approach Based on the I2C Standard," in *24th Annual AIAA/USU Conference on Small Satellites*, Logan, 2010.

[5] C. Mitchell, J. Rexroat, S. Rawashdeh and J. Lumpp, "Development of a Modular Command and Data Handling Architecture for the KySat-2 CubeSat," in *2014 IEEE Aerospace Conference*, Big Sky, 2014.

[6] W. Emmerich, "Software Engineering and Middleware: A Roadmap," in *Conference on The Future of Software Engineering*, New York, 2000.

[7] D. C. Schmidt, "Middleware for Real-Time and Embedded Systems," *Communications of the ACM*, pp. 43-48, June 2002.

[8] R. Schantz and D. Schmidt, "Middleware for distributed systems," in *Encyclopedia of Software Engineering*, New York, Wiley & Sons, 2002.

[9] A. S. Tanenbaum, "Network Protocols," *Computing Surveys*, vol. 13, no. 4, pp. 453-489, 1981.

[10] H. Carr, "PEPt: A Minimal RPC Architecture," in *OTM*, Italy, 2003.

[11] B. Holman, "The first air bomb: Venice, 15 July 1849," Airminded: Airpower and Britsh society, 1908-1941 (mostly), 22 August 2009. [Online]. Available: http://airminded.org/2009/08/22/the-first-air-bomb-venice-15-july-1849/. [Accessed 19 May 2014].

[12] R. Naughton, "Remote Piloted Aerial Vehicles: An Anthology," Centre for Telecommunications and Information Engineering, Monash University, 3 August 2007. [Online]. Available: http://www.ctie.monash.edu/hargrave/rpav_home.html. [Accessed 19 August 2014].

[13] L. C. Crowell, "Improvement In Aerial Machines". Massachusetts, United States of America Patent 35,437, 3 June 1862.

[14] C. Perley, "Improvement in Discharing Explosive Shells from Balloons". New York, United States of America Patent 37,771, 24 February 1863.

[15] N. Tesla, "Method of and Apparatus for Controller Mechanism of Moving Vessels or Vehicles". New York, United States of America Patent 613,809, 8 November 1898.

[16] Public Broadcasting Service, "Tesla Life and Legacy - Race of Robots," April 2004. [Online]. Available: http://www.pbs.org/tesla/ll/ll_robots.html. [Accessed 19 May 2014].

[17] J. F. Keane and S. S. Carr, "A Brief History of Early Unmanned Aircraft," *Johns Hopkins APL Technical Digest*, pp. 559-570, 2013.

[18] L. R. Newcome, Unmanned Aviation: A Brief History of Unmanned Aerial Vehicles, Reston: American Institute of Aeronautics and Astronautics, 2004.

[19] C. Anderson, "How I Accidentally Kickstarted the Domestic Drone Boom," Wired, 22 June 2012. [Online]. Available: http://www.wired.com/2012/06/ff_drones/all/. [Accessed 19 May 2014].

[20] S. Henn, "Under the Radar: Some Pilots of Small Drones Skirt FAA Rules," National Public Radio, 13 June 2013. [Online]. Available: http://www.npr.org/blogs/alltechconsidered/2013/06/13/190369460/guidelines-for-

commercial-drones-expect-to-come-by-2015. [Accessed 19 May 2014].

[21] Federal Aviation Administration, "Unmanned Aerial Systems Roadmap," 29 April 2014. [Online]. Available: http://www.faa.gov/about/initiatives/uas/. [Accessed 19 May 2014].

[22] Amazon, LLC, "Amazon Prime Air," 2013. [Online]. Available: http://www.amazon.com/b?node=8037720011. [Accessed 19 May 2014].

[23] A. Barr and R. Albergotti, "Google to Buy Titan Aerospace as Web Giants Battle for Air Superiority," The Wall Street Journal, 14 April 2014. [Online]. Available: http://online.wsj.com/news/articles/SB10001424052702304117904579501701702936522. [Accessed 9 October 2014].

[24] I. Lapowsky, "Facebook Lays Out Its Roadmap for Creating Internet-Connected Drones," Wired, 23 September 2014. [Online]. Available: http://www.wired.com/2014/09/facebook-drones-2/. [Accessed 9 October 2014].

[25] H. Potocnik, "The Problem of Space Travel: The Rocket Motor," NASA Headquarters, September 2007. [Online]. Available: http://www.hq.nasa.gov/office/pao/History/SP-4026/contents.html. [Accessed 22 May 2014].

[26] H. Oberth, Wege zur Raumschiffahrt, Berlin: R. Oldenbourg Verlag, 1929.

[27] A. C. Clarke, "Extra-Terrestrial Relays: Can Rocket Stations Give World-wide Radio Coverage?," *Wireless World*, October 1945.

[28] J. McDowell, "Jonathan's Space Report," 17 May 2014. [Online]. Available: http://planet4589.org/space/log/launch.html. [Accessed 22 May 2014].

[29] NASA, "History of Human Spaceflight," NASA Headquarters , 24 October 2012. [Online]. Available: http://spaceflight.nasa.gov/history/. [Accessed 23 May 2014].

[30] Futron Corporation, "Space Transportation Costs: Trends in Price Per Pound to Orbit 1990-2000," 6 September 2002. [Online]. Available: http://www.futron.com/upload/wysiwyg/Resources/Whitepapers/Space_Transportation_Costs_Trends_0902.pdf. [Accessed 23 May 2014].

[31] D. Kestenbaum, "Spaceflight is Getting Cheaper. But It's Still Not Cheap," National Public Radio, 21 July 2011. [Online]. Available: http://www.npr.org/blogs/money/2011/07/21/138166072/spaceflight-is-getting-cheaper-but-its-still-not-cheap-enough. [Accessed 23 May 2014].

[32] NASA Marshall Space Flight Center, "Advanced Space Transportation Program: Paving the Highway to Space," NASA, 12 April 2008. [Online]. Available: http://www.nasa.gov/centers/marshall/news/background/facts/astp.html. [Accessed 23 May 2014].

[33] L. Guerra, "Cost Estimating Model, Space Systems Engineering," March 2008. [Online]. Available: spacese.spacegrant.org/uploads/Costs/18.%20Cost_Module_V1.0.ppt. [Accessed 23 May 2014].

[34] Futron Corporation, "Satellite Manufacturing: Production Cycles and Time to Market," May 2004. [Online]. Available: http://www.futron.com/upload/wysiwyg/Resources/Whitepapers/Satellite_Manufacturing_Production_Cycles_0504.pdf. [Accessed 23 May 2014].

[35] NASA, "CubeSat ELaNa IV Launch on ORS-3," November 2013. [Online]. Available: http://www.nasa.gov/sites/default/files/files/ELaNa-IV-Factsheet-508.pdf. [Accessed 23 May 2014].

[36] The Science Museum, "Babbage's Analytical Engine, 1834-1871 (Trial model)," 2013. [Online]. Available: http://www.sciencemuseum.org.uk/objects/computing_and_data_processing/1878-3.aspx. [Accessed 29 May 2014].

[37] IBM, "Chronological History of IBM," 2013. [Online]. Available: http://www-03.ibm.com/ibm/history/history/decade_1930.html. [Accessed 29 May 2014].

[38] G. Adams, Interviewee, *Intel 8051 Microprocessor Oral History Panel.* [Interview]. 16 September 2008.

[39] The Telegraph, "History of ARM: from Acorn to Apple," 6 January 2011. [Online]. Available: http://www.telegraph.co.uk/finance/newsbysector/epic/arm/8243162/History-of-

ARM-from-Acorn-to-Apple.html. [Accessed 29 May 2014].

[40] ARM, "Processors," 2014. [Online]. Available: http://www.arm.com/products/processors/index.php. [Accessed 29 May 2014].

[41] J. Labrosse, MicroC/OS-II: The Real-Time Kernel, San Francisco: CMPBooks, 2002.

[42] F. C. Gaertner, "Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments," *ACM Computing Surveys,* vol. 31, no. 1, pp. 1-26, 1999.

[43] A. Ebnenasir and S. Kulkarni, "Feasibility of Stepwise Design of Multitolerant Programs," *ACM Transactions on Software Engineering and Methodology,* vol. 21, no. 1, pp. 1-49, 2011.

[44] D. J. Brown, "A NOVEL MESSAGE ROUTING LAYER FOR THE COMMUNICATION MANAGMENT OF DISTRIBUTED EMBEDDED SYSTEMS," in *University of Kentucky Master's Theses,* Paper 41, 2010.

[45] H. Zimmerman, "OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection," *IEEE Transactions on Communications,* vol. 28, no. 4, pp. 425-432, 1980.

[46] S. M. Sadjadi and P. K. McKinley, "A Survey of Adaptive Middlware," Michigan State University, East Lansing, 2003.

[47] S. A. Rawashdeh, "Theses and Dissertations--Electrical and Computer Engineering," 2013. [Online]. Available: http://uknowledge.uky.edu/ece_etds/30. [Accessed 2013].

[48] D. A. Gwaltney and J. M. Briscoe, "Comparison of Communication Architectures for Spacecraft Modular Avionics Systems," NASA Marshall Space Flight Center, Huntsville, 2006.

[49] Akka, "Message Delivery Reliability," 19 July 2014. [Online]. Available: http://doc.akka.io/docs/akka/snapshot/general/message-delivery-reliability.html. [Accessed 22 July 2014].

[50] J. Gray, "Notes on Data Base Operating Systems," in *Operating Systems, An Advanced Course*, London, Springer-Verlag, 1978, pp. 393-481.

[51] S. Krakowiak, "Middleware Architecture with Patterns and Frameworks," Creative Commons License, 2009.

[52] J. Gray, "The Transaction Concept: Virtues and Limitations," in *Seventh International Conference on Very Large Databases*, Cannes, 1981.

[53] IBM, "MQSeries: An Introduction to Messaging and Queuing," International Business Machines Corporation, 1995.

[54] D. Erb, T. Clements, J. Lumpp and B. Malphrus, "Kentucky Space: A Multi-University Small Satellite Enterprise," in *23rd Annual AIAA/USU Conference on Small Satellites*, Logan, 2009.

[55] J. Buck, "NASA Releases Glory Taurus XL Launch Failure Report Summary," NASA, 19 February 2013. [Online]. Available: http://www.nasa.gov/mission_pages/Glory/news/mishap-board-report.html. [Accessed 23 May 2014].

[56] National Instruments, "Controller Area Network (CAN) Overview," 30 November 2011. [Online]. Available: http://www.ni.com/white-paper/2732/en/. [Accessed 11 July 2014].

[57] S. Corrigan, "Introduction to the Controller Area Network (CAN)," Texas Instruments, 2008.

[58] Philips Semiconductors, "AN10216-01 I2C Manual," Philips Semiconductors, 2003.

[59] J. Axelson, USB Complete: Fourth Edition, Madison: Lakeview Research LLC, 2009.

[60] Institute of Electrical and Electronics Engineers, "IEEE Standard for Ethernet," IEEE, New York City, 2012.

[61] M. Simmons, "Ethernet Theory of Operation," Microchip, 2008.

[62] Future Technology Devices International Ltd. , "Technical Note TN_111: What is a

UART?," 7 August 2009. [Online]. Available: http://www.ftdichip.com/Support/Documents/TechnicalNotes/TN_111%20What%20is%20UART.pdf. [Accessed 3 December 2014].

[63] Exar Corporation, "Multidrop/9-Bit Mode Feature," April 2009. [Online]. Available: https://www.digikey.com/Web%20Export/Supplier%20Content/Exar_1016/PDF/exar-dan-200.pdf?redirected=1. [Accessed 3 December 2014].

[64] S. Parkes, "SpaceWire User's Guide," STAR-Dundee Limited, 2012.

[65] European Cooperation for Space Standardization, "Space engeering: SpaceWire - Links, nodes, routers, and networks," ECSS Secretariat, Noordwijk, 2008.

[66] L. Meier, P. Tanskanen, F. Fraundorfer and M. Pollefeys, "PIXHAWK: A System for Autonomous Flight using Onboard Computer Vision," in *IEEE International Conference on Robotics and Automation*, Shanghai, 2011.

[67] L. Meier, "MAVLink Micro Air Vehicle Communication Protocol," 2013. [Online]. Available: http://qgroundcontrol.org/mavlink/start. [Accessed 28 May 2014].

[68] M. Banahan, "The C Book - Structures," gbdirect, March 2003. [Online]. Available: http://publications.gbdirect.co.uk/c_book/chapter6/structures.html. [Accessed 28 May 2014].

[69] T. Bray, J. Paoli, C. Sperberg-McQueen, E. Maler and F. Yergeau, "Extensible Markup Language (XML) 1.0 (Fifth Edition)," W3C, 7 February 2013. [Online]. Available: http://www.w3.org/TR/2008/REC-xml-20081126/#sec-logical-struct. [Accessed 28 May 2014].

[70] Z. A. Jacobs, "PROVIDING A PERSISTENT SPACE PLUG-AND-PLAY AVIONICS NETWORK ON THE INTERNATIONAL SPACE STATION," in *Theses and Dissertations--Electrical and Computer Engineering*, Paper 16, 2013.

[71] Z. Jacobs, "Providing a Persistent Space Plug-and-Play Avionics Network on the International Space Station," *Theses and Dissertations--Electrical and Computer Engineering*, p. 16, 2013.

[72] J. Lumpp, A. Karam, D. Erb, J. Bratcher, S. Rawashdeh, T. Clements, N. Fite, J. Kruth, B. Malphrus, I. Bland, R. Munakata, R. Coelho, J. Puig-Suari, J. Reese, C. Brodell and S. Schaire, "SOCEM: Sub-Orbital CubeSat Experimental Mission," in *31st IEEE Aerospace Conference*, Big Sky, Montana, 2010.

[73] J. Lyke, D. Fronterhouse, S. Cannon, D. Lanza and W. Byers, "Space Plug-and-Play Avionics," *3rd Responsive Space Conference*, p. 12, 2005.

[74] J. Lumpp, D. Erb, T. Clements and J. Rexroat, "The CubeLab Standard for Improved Access to the International Space Station," in *34th IEEE Aerospace Conference*, Big Sky, 2011.

[75] J. McHale, "AFDX Technology to Improve Communications on Boeing 787," Military & Aerospace Electronics, 1 April 2005. [Online]. Available: http://www.militaryaerospace.com/articles/print/volume-16/issue-4/news/afdx-technology-to-improve-communications-on-boeing-787.html. [Accessed 14 July 2014].

[76] TechSAT Gmbh, "AFDX/ARINC 664 Tutorial," 28 August 2008. [Online]. Available: http://www.techsat.com/fileadmin/media/pdf/infokiosk/TechSAT_TUT-AFDX-EN.pdf. [Accessed 14 July 2014].

[77] T. Fuehrer, B. Mueller, W. Dieterle, F. Hartwich, R. Hugel and M. Walther, "Time Triggered Communication on CAN (Time Triggered CAN- TTCAN)," in *7th International CAN Conference*, Amsterdam, 2000.

[78] M. Stock and J. Deas, "CANaerospace - the Airborne CAN Interface Standard," 2009. [Online]. Available: http://www.stockflightsystems.com/tl_files/downloads/canaerospace/CANaerospace_OSH_2009_Paper.pdf. [Accessed 4 June 2014].

[79] O. Rawashdeh and J. Lumpp, "Run-Time Behavior of Ardea: A Dynamically Reconfiguring Distributed Embedded Control Architecture," in *IEEE Aerospace Conference*, Big Sky, 2006.

[80] HART Communication Foundation, "How HART Works," HARD Communication Foundation, 2014. [Online]. Available:

http://en.hartcomm.org/hcp/tech/aboutprotocol/aboutprotocol_how.html. [Accessed 14 July 2014].

[81] LonMark International, "LonMark News Events," 2014. [Online]. Available: http://www.lonmark.org/news_events/docs/LonWorksTechnology101-AHR07.pdf. [Accessed 15 May 2014].

[82] K. Lee, "Introduction to IEEE 1451," National Institute of Standards and Technology, 31 January 2011. [Online]. Available: http://www.nist.gov/el/isd/ieee/1451intro.cfm. [Accessed 14 July 2014].

[83] W. Kastner and M. Leupold, "How Dynamic Networks Work: A Short Tutorial on Spontaneous Networks," in *8th IEEE International Conference on Emerging Technologies and Factory Automation*, Antibes-Juan les Pins, 2001.

[84] T. Morphopoulos, L. J. Hansen, J. Pollack, J. Lyke and S. Cannon, "Plug-and-Play - An Enabling Capability for Responsive Space Missions," in *2nd Responsive Space Conference*, Los Angeles, 2004.

[85] D. R. Bracknell, "Introduction to the MIL-STD-1553B Serial Mulitplex Data Bus," *Microprocessors and Microsystems*, vol. 12, no. 1, 1988.

[86] Ballard Technology, "Department of Defense Interface Standard for Digital Time Division Command/Response Multiplex Data Bus," 21 September 1978. [Online]. Available: http://www.ballardtech.com/Tutorials/Ballard%20Technology%20-%20MIL-STD-1553B-Notice2.pdf. [Accessed 14 July 2014].

[87] Data Device Corporation, "MIL-STD-1553 Designer's Guide," 2014. [Online]. Available: http://www.ddc-web.com/Documents/dguidehg.pdf. [Accessed 31 July 2014].

[88] Echelon Corporation, "Enerlon," 1994. [Online]. Available: http://www.enerlon.com/JobAids/Lontalk%20Protocol%20Spec.pdf. [Accessed 15 May 2014].

[89] A. San-Salvador and A. Herrero, "Contacting the Devices: A Review of Communication Protocols," in *Ambient Intelligence - Software and Applications*, Berlin, Springer-Verlag,

2012, pp. 3-10.

[90] Object Management Group, "CORBA Basics," 2014. [Online]. Available: http://www.omg.org/gettingstarted/corbafaq.htm. [Accessed 10 July 2014].

[91] Object Management Group, "CORBA Success Stories," 2014. [Online]. Available: http://www.corba.org/success.htm. [Accessed 10 July 2014].

[92] Object Management Group, "CORBA for embedded Specification," November 2008. [Online]. Available: http://www.omg.org/spec/CORBAe/. [Accessed 6 June 2014].

[93] D. C. Schmidt, "Overview of CORBA," Washing University in St. Louis, 28 September 2006. [Online]. Available: http://www.cs.wustl.edu/~schmidt/corba-overview.html. [Accessed 19 May 2014].

[94] S. Vinoski, "CORBA: Integrating Diverse Applications within Distributed Heterogeneous Environments," *IEEE Communications Magazine*, pp. 1-12, February 1997.

[95] "uORB," Pixhawk, 2014. [Online]. Available: http://pixhawk.org/firmware/apps/uorb. [Accessed 7 July 2014].

[96] "System for Tactical Aerial Reconnaissance - PX4 Firmware," Buskerud and Vestfold University College, 30 May 2013. [Online]. Available: https://home.hibu.no/AtekStudenter1212/doxygen/bb_handler/index.html. [Accessed 7 July 2014].

[97] S. Dissanaike, P. Wijkman and M. Wijkman, "Utilizing XML-RPC or SOAP on an Embedded System," in *24th International Conference on Distributed Computing Systems Workshop*, Tokyo, 2004.

[98] B. Henderson, "User Manual for XML-RPC," Sourceforge, November 2004. [Online]. Available: http://xmlrpc-c.sourceforge.net/doc/. [Accessed 7 August 2014].

[99] Consultative Committe for Space Data Systems, "About CCSDS," 2014. [Online]. Available: http://public.ccsds.org/default.aspx. [Accessed 11 July 2014].

[100] Bright Ascension, "Generation 1 Onboard Software," 2014. [Online]. Available: http://www.brightascension.com/products/generation1/. [Accessed 9 July 2014].

[101] Consultative Committee for Space Data Systems, "Spacecraft Onboard Interface Services Informational Report Green Book," December 2013. [Online]. Available: http://public.ccsds.org/publications/archive/850x0g2.pdf. [Accessed 20 May 2014].

[102] Consultative Committee for Space Data Systems, "Asynchronous Message Service Recommended Standard Blue Book CCSDS 735.1-B-1," September 2011. [Online]. Available: http://public.ccsds.org/publications/archive/735x1b1.pdf. [Accessed 7 July 2014].

[103] NASA Goddard Space Flight Center, "CubeSat," *NASA Goddard Tech Transfer News*, pp. 1-24, Spring 2013.

[104] M. Sorgenfrei, "BioSentinel: Enabling CubeSat-scale Biological Research Beyond Low Earth Orbit," in *Interplanetary Small Satellite Conference*, Pasadena, 2014.

[105] D. McComas, "NASA/GSFC's Flight Software Core Flight System," in *Flight Software Workshop*, San Antonio , 2012.

[106] "OS Abstraction Layer," Sourceforge, January 2014. [Online]. Available: http://osal.sourceforge.net/. [Accessed 2 July 2014].

[107] L. Meier, "MAVLink micro air vehicle marshalling / communication library," GitHub, 24 September 2014. [Online]. Available: https://github.com/mavlink/mavlink. [Accessed 25 September 2014].

[108] G. P. Rakow and J. J. Wilmot, "CCSDS Collaborative Work Environment," October 2011. [Online]. Available: http://cwe.ccsds.org/sois/docs/SOIS-APP/Meeting%20Materials/2011/Fall/Software%20Plug-and-Play%20Architectures/Distributed%20Integrated%20Modular%20Architecture_update.pptx. [Accessed 21 May 2014].

## Vita

Jason Timothy Rexroat

### Education

Bachelor of Science in Electrical Engineering, Graduated in May 2013
Bachelor of Science in Computer Engineering, Graduated in May 2013
Minors in Computer Science and Mathematics

### Awards and Activities

Graduate Research Assistantship, 2013-2014
Presidential Full Tuition Scholarship, 2008-2012
University of Kentucky Student Ambassador, 2010 to 2013
College of Engineering High School Project Mentor, 2013-2014

### Experience

Space Systems Laboratory, Lexington, Kentucky
Graduate Research Assistant August 2013 – December 2014
Undergraduate Researcher May 2009 – August 2013

NASA Ames Research Center, Mountain View, California
Research Intern July 2011 – August 2011

### Publications

- "Development of a Modular Command and Data Handling Architecture for the KySat-2 CubeSat" at 2014 IEEE Aerospace Conference, Big Sky
- "The CubeLab Standard for Improved Access to the International Space Station" at 2011 IEEE Aerospace Conference, Big Sky
- "SOCEM: Sub-Orbital CubeSat Experiment Mission" at 2010 IEEE Aerospace Conference, Big Sky

### Presentations

- "KySat-2: Status Report and Overview of C&DH and Communications Systems Design" at the 2014 CubeSat Summer Workshop, San Luis Obispo, CA
- "A Distributed Command and Data Handling Architecture for KySat-2" at the 2013 CubeSat Summer Workshop, San Luis Obispo, CA
- "The University of Kentucky Space Systems Laboratory" at the 2012 Regional NASA Space Grant Director's Meeting, Little Rock
- "The CubeLab Standard Bus for Improved Access to the Inernational Space Station", "The KySat-2 CubeSat", and "Space Plug-and-play Enabled CubeLabs for ISS Payloads" at the 2013 Kentucky EPSCoR Conference, Louisville