EXTENSIBLE SOFTWARE ARCHITECTURE FOR A DISTRIBUTED ENGINEERING SIMULATION FACILITY

A Thesis

by

JAMES FRANKLIN MAY, JR.

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2013

Major Subject: Aerospace Engineering

ABSTRACT

A need has arisen for an easy-to-use, flexible, transparent, and cross-platform communication backbone for configuration and execution of distributed simulations and experiments. Open source, open architecture, and custom student written programs have extended the capabilities of educational research facilities and opened the way for the development of the architecture presented in this thesis. The architecture is known by the recursive acronym hADES: hADES Architecture for Distributed Engineering Simulation. Included in this thesis is a discussion of the design and implementation of the novel hADES software architecture for Ethernet and wireless IEEE 802.11 network-based distributed simulation and experiment facilities. The goal of this architecture is to facilitate rapid integration of new and legacy simulations and laboratory equipment to support undergraduate and graduate research projects as well as educational classroom activities and industrial simulation and experiments.

DEDICATION

To Anne, Jim, Liz, Cali, and Scout

NOMENCLATURE

A/P	Autopilot
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
CFG	Configuration
COTS	Commercial Off-The-Shelf
EFS	Engineering Flight Simulator
GPL	GNU General Public License
GUI	Graphical User Interface
hADES	hADES Architecture for Distributed Engineering Simulation
HDD	Head-Down Display
HLA	High Level Architecture
HW	Hardware
IANA	Internet Assigned Numbers Authority
IEEE	Institute of Electrical and Electronics Engineers
I/O	Input/Output
IP	Internet Protocol
LAN	Local Area Network
LCD	Liquid Crystal Display
LQR	Linear-Quadratic Regulator
OOP	Object-Oriented Programming
PC	Personal Computer
RFC	Request for Comments
RTI	Run-Time Infrastructure

SGI	Silicon Graphics Incorporated
SW	Software
TAMU	Texas A&M University
TCP	Transmission Control Protocol
TCP/IP	Internet Protocol Suite (Transmission Control Protocol/Internet Protocol)
UDP	User Datagram Protocol
UI	User Interface
VSCL	Vehicle Systems & Control Laboratory

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

The primary use of engineering simulation is to emulate the physics behind advanced systems. It has become a major tool in engineering research, design, testing, and education. However, because advanced systems blend fields such as mechanics, electronics, chemistry, nuclear physics, and optics, the time and resources required to develop new simulations can sometimes be large and require engineers with skills in many areas outside their expertise and/or major field of study[1].

The proliferation of simulation use is fueled by increasing power and decreasing cost of modern computing technologies. Modern simulation architectures typically allow for large scale distributed computer simulations, but many usable distributions, such as some implementations of the High-level architecture[2], are highly proprietary and/or unique to the problems being studied since they are developed by private companies and engineering firms. *A problem with simulation architectures being unique to each system is the lack of a simple and common method of communication between modules. A second problem is complex interdependence between the code bases of networked modules.* Specifically and historically, the lack of a common or flexible method of communication leads to statically compiled data structures for data transmission - where a change in the data structure of one simulation module will require a change in all connected simulation modules. Without high-level, multi-platform communication codes, practical application of network distributed simulation systems, especially in an academic environment, remains near the low level of bits and bytes and requires a development team made up of students from many engineering and computer science disciplines[3].

1.1 Software

The original simulation technologies were developed using hardware and software that are primitive by today's standards. Code was restricted to the monolithic structures of single thread procedural architectures and primitive high-level programming languages, such as the original versions of BASIC[4], FORTRAN[5], and Pascal[6]. *Although procedural code is laid out linearly, functional interdependencies are difficult to comprehend and must be manually mapped out, line by line, throughout the code*[7] *including the implications and paths of branching and conditional structures.* Variables in procedural architectures have limited protection and encapsulation; much care and bookkeeping is needed to ensure the integrity of variables because they can be intentionally or unintentionally altered anywhere in the codebase.

A step forward in architecture design was the progression to modular architectures which allowed common code functionality to be broken out of the procedural paradigm into reusable subroutines and functions in programming languages such as C[8, 9] and some initially procedural programming languages, such as FORTRAN, which had function and subroutine capabilities upgraded in later versions[10]. Scalability is also an issue with modular architectures when moving from simulations with one body or modeled system to simulations hosting many bodies or modeled systems because the data for each is not encapsulated[7]. Data variables or data structures for each body of a multi-body system must be separately created and maintained but still remain available and viewable from anywhere in the codebase. Data for each body of a multi-body system must also have unique variable names or structure names if not already in an array. Data in an array will usually lack a unique name and have only array indices. Thus, an extensive simulation architecture redesign is usually required when moving from a single-body simulation to a multi-body system.

A second paradigm shift in simulations was the introduction of object-oriented architectures where similar functions and data can be encapsulated into objects using object-oriented supported programming languages such as C++[11], Java[12], and Python[13]. The use of objects allows for massive code reuse and the ability to define simulation bodies or systems into objects of which many instances can be created inside a program. Common variables and functions can be implemented in a base class which can be inherited to create more complex objects. An example of this, shown in Figure 1.1, is a base aircraft class which has position, velocity, acceleration, mass, inertia, force, and moment variables. A base aircraft class can be inherited into a fighter aircraft class which adds the fighter thrust module, number of engines, and weapons and payload modules. The same base class can be inherited into a commercial transport aircraft class which adds its own thrust module, number of engines, payload variables, passenger data, and autopilot functions. Both are aircraft and use the common aircraft data from the aircraft base class. Collections of objects can also be placed in an array, vector, list, or some other container for a higher level of organization of simulation bodies in a simulation program. These advances have allowed for simultaneous simulation of multiple vehicles' systems and environments to be easily, efficiently, and clearly coded into one computer program.

1.2 Hardware

The advent of personal computers (PCs) has expanded engineering simulation use as well. Early high-fidelity simulations were only implemented using high-end and proprietary hardware[14]. While powerful, high-end hardware traditionally has very high purchase, support, and maintenance costs[14]. Simulations that once required proprietary, specialized, and expensive computer mainframe hardware is now executed using common and cheaply procured personal computers with a much wider

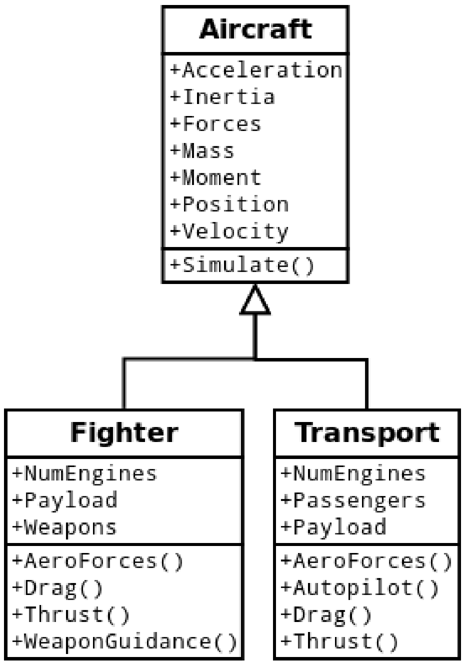


Figure 1.1: Aircraft Class Inheritance Example

support base. The ability for simulations to be executed on PCs has derived from major increases in computing power, performance, and availability of PCs. This can be visibly shown using Moore's Law[15] and derivatives of Moore's trend in Figure 1.2[16], showing microprocessor transistor count, and Figure 1.3[17], showing calculations per second per monetary cost. The key point in Moore's Law is that at a fixed cost, the number of calculations per second has been exponentially increasing over time allowing previously computationally expensive and prohibitive simulations to be executed quickly on modern hardware.

When simulations began to exhaust the hardware and software resources of a single PC, multiple network-distributed computers began to be used for executing a simulation. The expansion to multiple PCs was due to simulation of large numbers of complex simulated agents and vehicles[14, 18, 19], considerably complex and high fidelity environments and vehicles[18, 20, 21, 22], standalone commercial-off-the-shelf

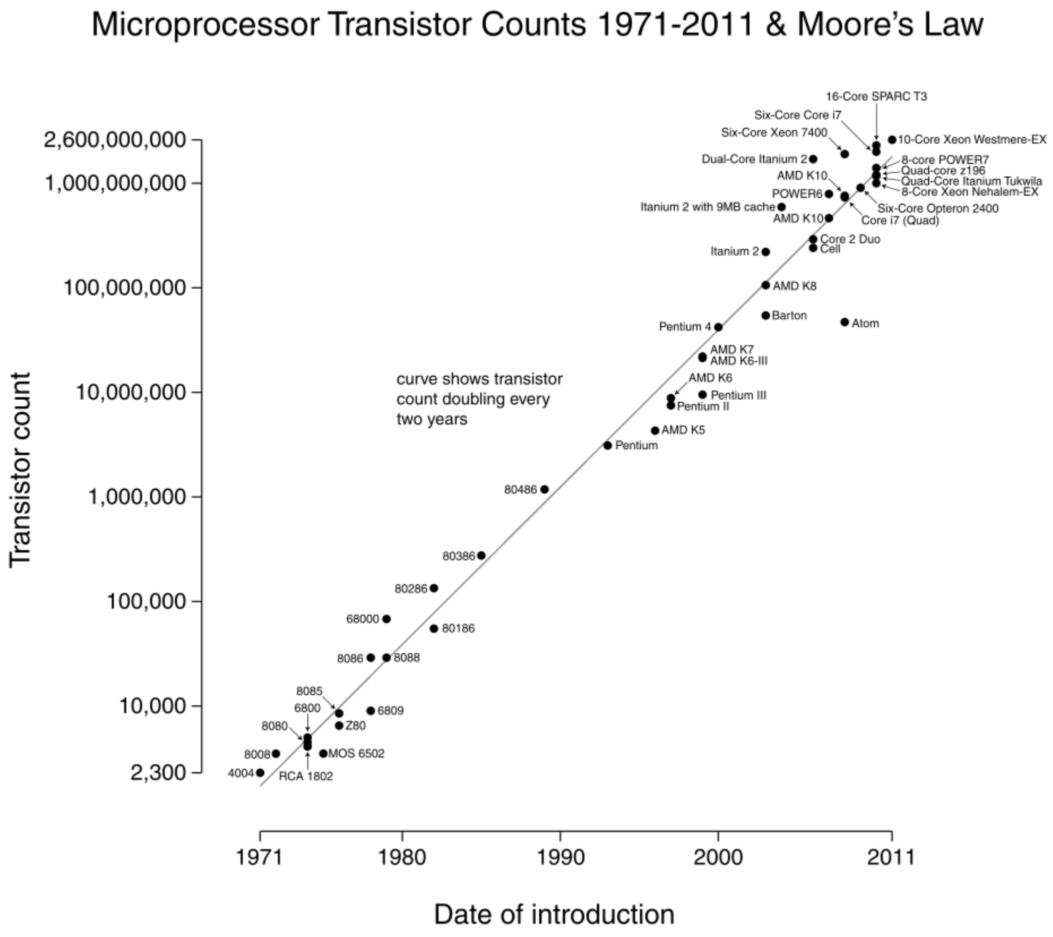


Figure 1.2: Transistor Count and Moore's Law - 2011[16]

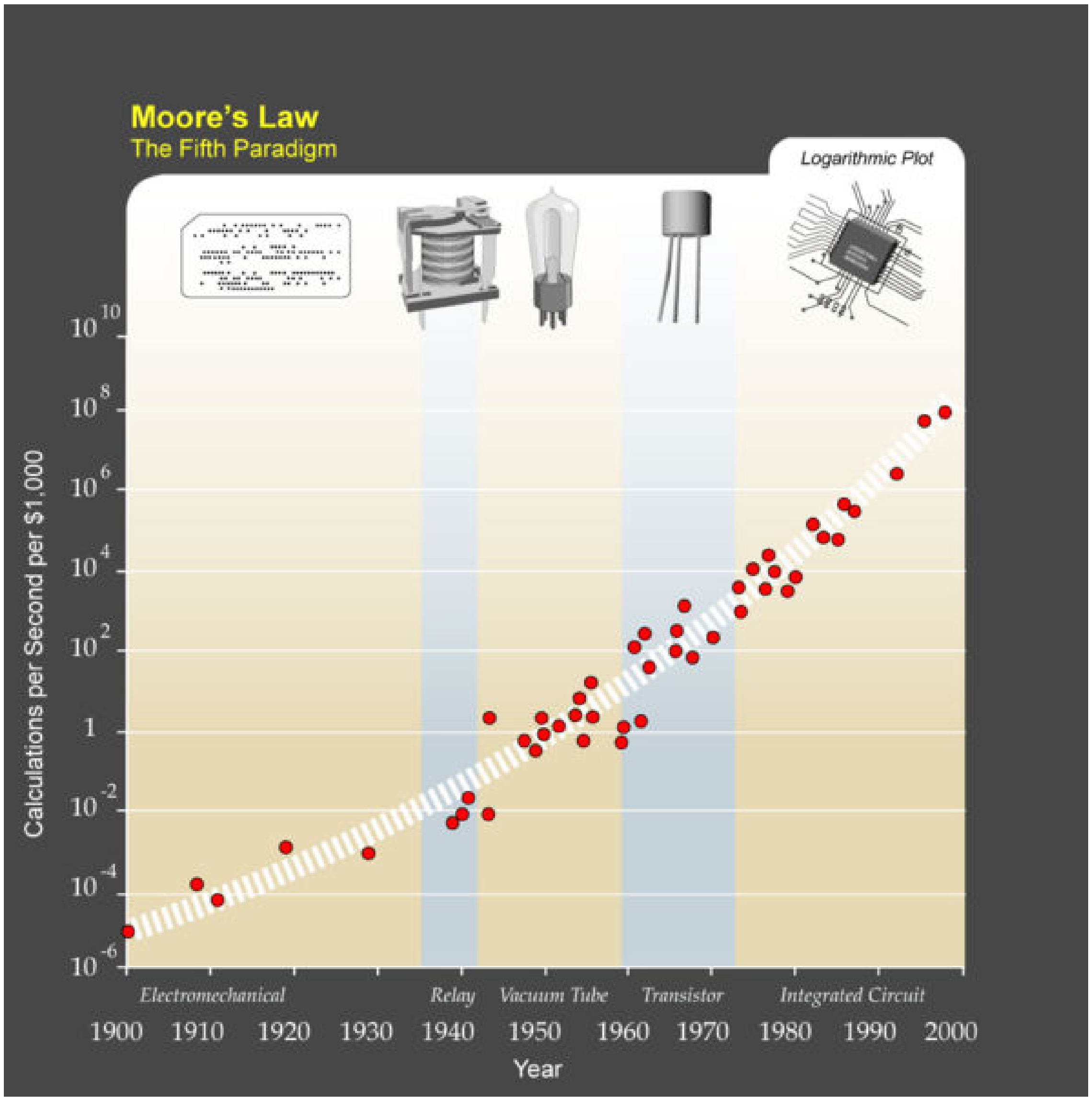


Figure 1.3: Calculations per Second and Moore's Law[17]

(COTS) software integration to simulation networks[14, 23], and other features and capabilities expanded upon from and by previous generations of architectures[14, 24]. Distributed simulations increase scalability, flexibility, and reconfigurability of a simulation environment promoting exchange, reuse, and inter-operation between multiple simulation components[25] and do not require all functions of a simulation environment to be compiled or included in a single program. Distributed simulations in this thesis are simulations spread across commonly used Linux and Windows PCs connected using Ethernet and wireless IEEE 802[26] networks.

1.3 Distributed Simulation Communication

Multiple methods exist for software programs to communicate with one another when located on the same computer, but options are limited when programs are located on separate computers. PCs are most commonly connected to one another using Ethernet and wireless IEEE 802 networks. The Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) of the Transport Layer of the Internet Protocol Suite[27] are designed to establish host-to-host (i.e. program-to-program) connectivity and handle data transmission between programs across various types of networks including IEEE 802 standard networks and even RFC 1149[28] networks. Provided drivers are available for the PC networking hardware in use, virtually all modern desktop and server operating systems support communication using these protocols. TCP is a bidirectional error checking transmission protocol that ensures exact bit-by-bit delivery of the data packets sent as well as the order of such packets. Bidirectional in this case means that a point-to-point connection is made and either end of the connection can send data. Error checking is handled transparently by the protocol across the connection. Additional overhead is required to ensure this type of error checking and thus it is a slower protocol than UDP. UDP is a unidirectional connectionless transmission protocol. Unidirectional in this case means that data is transmitted in a send-and-forget fashion where data packets are sent to a specific address and then no more information is transferred regarding them. Error checking is not done during transmission and thus less overhead is required for this protocol than TCP. Error checking can, however, be applied to UDP communication in the form of a checksum added to the packet which is handled in the Application Layer[27] instead of the Transport Layer[27] as in TCP. The programming languages used in this thesis have objects and/or data types for transmitting packets using TCP and UDP called "*sockets*" and is referred to as such in this thesis.

UDP sockets have unicast, multicast[27], and broadcast[27] transmission abilities where the recipient(s) of data are designated by IP address. Unicast is point-to-point communication where the sending address is an IP address for a specific computer. Multicast is point-to-multipoint communication. Sending data multicast is functionally identical to sending data unicast data except that the sending address is a multicast IP address. Multicast addresses are a range of addresses known as Class D[27] addresses in RFC 1122. If a client wants to receive information that is sent to a multicast address, it registers with network routers for that address and the networking hardware will handle the multipoint delivery. Sending data broadcast is also functionally identical to sending data unicast except that the sending address is in the broadcast IP address range[27] chosen to broadcast messages to a specific subnet or a whole network. Broadcast messages are not used in the architecture developed in this thesis. Figure 1.4 shows the different types of transmission and the path(s) the data takes from the sender to the recipient(s). In addition to an IP address, sockets need a port number for requesting a connection using TCP and for sending data packets using UDP. Where an IP address is analogous to a building address for sending physical mail, a port number is analogous to a specific mailbox

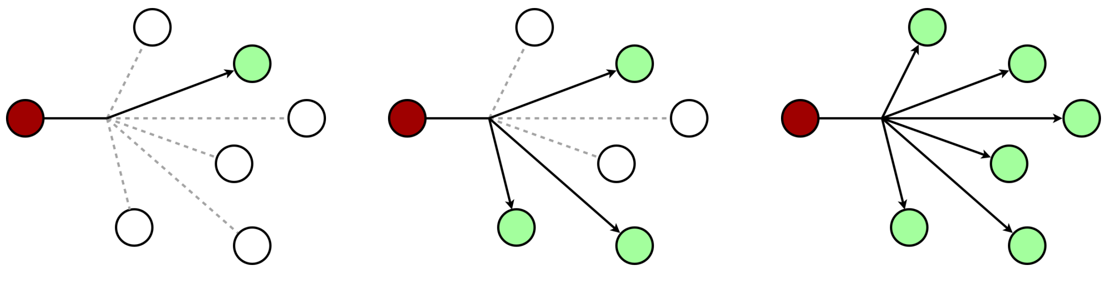

(a) Unicast (b) Multicast (c) Broadcast

Figure 1.4: UDP Transmission Types

number at that building address - both are necessary to deliver mail.

It is common for programming languages with TCP and UDP functionality to send and receive data only as a character or an array of characters data type. For other data types to be sent, they must first be converted using a direct byte-by-byte conversion to a character array. The Internet Protocol Suite does not modify the contents of the message or data, so the data sent is generally expected to be the exact same data received unless some sort of noise or transmission error occurs. With the expectation that the data sent will be the exact same data received and that any data can be converted into a character array for transmission, multiple data can be grouped together in a known order and sent to another program to be received in the same order. Both programs must agree on what order and types the data is going to be represented for transmission. An example of a data packet format for transmission is shown in Figure 1.5 where the data packet format is identical on each end of communication for successful data transmission and reception. Because it is outside the scope of many simulation programs to handle data format translation in an on-the-fly manner, this leads to hard-coded data packet formats where the data type and order is programmed before run-time for any and all programs that share data in a distributed simulation. The main cause of complex interdependence between the codebases of interconnected simulation programs has been the use of hard-coded data packet formats, where a change in one program would require all other connected programs to be adjusted and recompiled[23]. An architecture that is flexible to changes in data packet formats is needed to avoid recompilation of source code of connected simulations and manual analysis and confirmation that all data packet formats match between all connected programs upon modifications of one program in the distributed system[23].

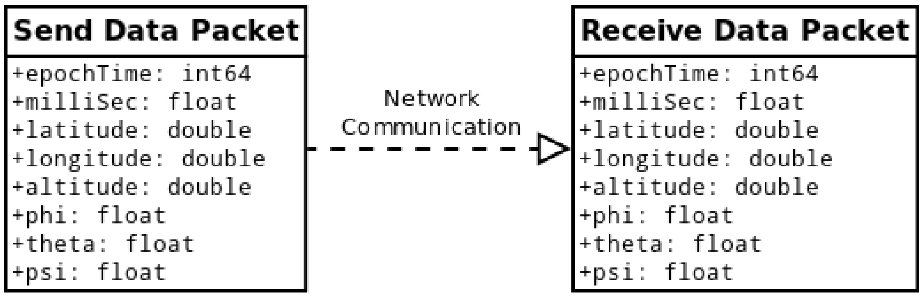


Figure 1.5: Transmission Data Packet Format

1.4 Other Distributed Architectures

Other distributed simulation architectures, such as the High Level Architecture[2] (HLA), already exist. The main component of the HLA is the Run-Time Infrastructure (RTI). The RTI is a middleware that coordinates data exchange between software modules (defined as federates in the HLA) during simulation runtime. Many RTI implementations exist; some are under commercial license or are completely proprietary and some are freely available or under GPL or US Government licensing. Creating an RTI according to the HLA standard is very time consuming due to the number of specifications required in relation to the time constraints of many student simulation designers and even industry engineers. A second consideration for not choosing the HLA is that requiring students to write a federate program to the HLA specifications is technically demanding. The limited programming experience and time constraints of students can severely prohibit other laboratory and research progress. Lastly, although a custom API to the HLA could be written to ease federate program integration, the HLA standards are excessive compared to the requirements of a typical university research laboratory and Vehicle Systems & Control Laboratory simulations[23] and the desired features discussed in Section 1.6.

Distributed Interactive Simulation[29] is another open standard for real-time distributed simulations. This standard is designed for large scale wargames. Reasons for not choosing and building an architecture according to this standard closely mirror those for the HLA[23].

1.5 Extensible Architecture

The architecture presented in this thesis is named as the recursive acronym hADES: *hADES Architecture for Distributed Engineering Simulation.* There are several novel extensible features and methods of interaction between features of hADES to be described in this section. The extensible features are implemented using three distinct functional groups shown in Figure 1.6: The simulation Daemon to handle communication and configuration between computers on the simulation network, Module API libraries for connecting a simulation Module to a network of Daemons, and the actual simulation Modules themselves. The simulation Modules are standalone computer programs that execute the actual dynamical simulations or give software access to acquired experimental data. In Figure 1.6, a computer network can be described as having N computers. One instance of the Daemon, as described in Section 1.5.1, exists on each computer. Each computer executes M_i programs locally. The Module API is paired with each simulation Module code to create a standalone program capable of connecting to, configuring with, and communicating with a simulation Daemon network.

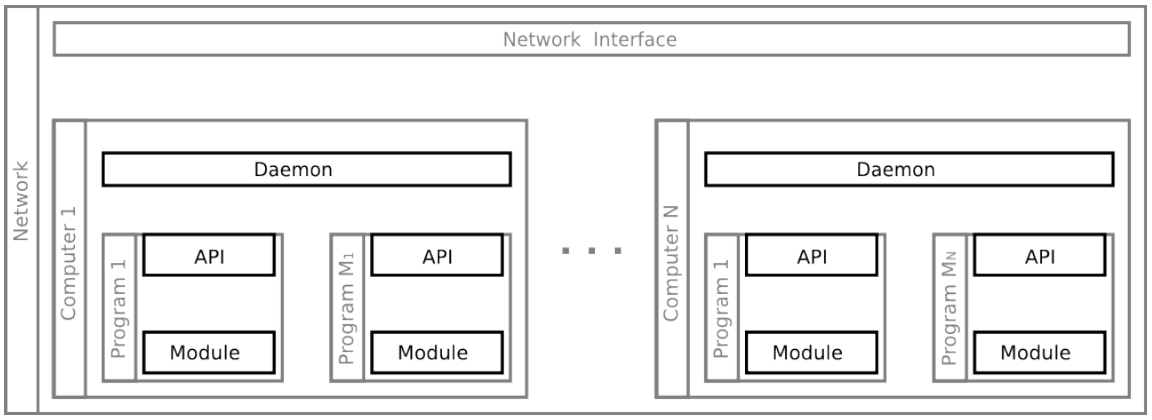


Figure 1.6: Architecture Overview: Functional Groups

1.5.1 Daemon

The simulation Daemon handles intermodule communication and configuration. It is a standalone program (usually a daemon process of the operating system) that is executed in the background of all simulation computers. The Daemon communicates the availability of and access to the actual simulation Module data between all other Daemons on the network - in other words, it tells other Daemons where to expect data that simulation Modules need. It is designed to be cross-platform (Linux and Windows) and make communication between multiple operating systems and platforms transparent to the simulation module designer.

Whereas previous distributed architectures were plagued by hard-coded data packet formats between directly connected software modules[23], the main novel feature of the hADES Daemon developed in this thesis is that it has the ability to modify data packet formats in transit between modules, combine all or parts of multiple data packets to create a new one, and create constant faux data when no input is available. An example of when faux data may be necessary is an aircraft simulation that expects a landing gear control but no control is available on the joystick being used as input. For standard flight, the landing gear control can be assumed to be constantly in the "up" state and input that way by the Daemon to the aircraft simulation Module. Modified data packet formats consist of added data to packet, data removed from packet, changing data type, and changing data units. Configuration of data paths between modules is handled by the Daemon through a user interface. Registration of a new software Module to the network of Daemons is handled using the Module API libraries.

1.5.2 Application Programming Interface

The Module API is a set of programming libraries used to integrate software Modules into the distributed architecture developed in this thesis. The purposes of the API libraries are to register input and output data packet formats with the Daemon, communicate with the Daemon, and give executive control of software modules - allowing software Modules to communicate with one another and for their execution to be controlled remotely. The API acts as a "wrapper" around the main software Module functions to start and stop modules and to send and receive data. These libraries are programming language and operating system *dependent.* Requirements on the programming language and operating system are that they have the ability to communicate via the TCP/IP protocol. *Communication methods using the Daemon API standardizes implementation of intermodule communication.*

1.5.3 Modules

Simulation Modules are the programs which execute the actual dynamical simulations or, if connected to tangible experimental hardware, give software access to experimental data. These Modules have a predefined set of inputs and outputs

(I/O) consisting of data type, order of data expected in I/O structures, data units, and expected data transmission rate. This set of information is used by the API libraries when registering with the simulation Daemon. Module execution is initiated when the Daemon has configured all I/O between connected modules. Simulation execution commands are sent through the API.

1.6 Research Objectives

High-level objectives of this thesis are to address the deficiencies in current architectures and to meet the needs of the Texas A&M University (TAMU) Vehicle Systems & Control Laboratory (VSCL) discussed in Chapter 2. The main objective and novel feature of hADES is to mitigate or remove the complex interdependence between the codebases of distributed simulation programs that communicate with one another. Previous distributed architectures were plagued by hard-coded data packet formats between directly connected software modules, but the hADES Daemon can modify data packets in transit between Modules to avoid having to statically compile and/or verify that transmission data structures match between distributed Module I/O mappings. The next objective is to standardize methods for implementing distributed simulations in a straightforward manner. Module design engineers may not be software engineers, thus the implementation methods must be minimal and *flexible* to allow the design engineers to easily create, modify, and connect simulation Modules. Lastly, an objective of the hADES system is that it must not degrade performance compared to previous architectures. Performance can be tested on a case-by-case implementation basis for any set of connected, distributed Modules.

Development of the proposed extensible software architecture for distributed simulations has four main tasks:

1.6.1 Cross-Platform Daemon Design

A minimal number of implementations of the simulation Daemon is preferred to provide similar performance across a simulation network. This can be achieved using a programming language, such as Python, where one codebase can be executed on virtually any platform that has a modern Python implementation available. In this thesis, a set of requirements for the simulation Daemon is proposed and developed according to the special needs of the VSCL that relate directly to the needs of many experimental simulation laboratories and research groups.

1.6.2 Creation of API Libraries

Development of the Daemon and the API libraries can be accomplished independently once the interface protocol between the two is defined. API libraries specific to each programming language and operating system platform can also be developed independently from one another and as needed by new or legacy software modules. Current needs of the VSCL require API libraries for the C++, Python, and MATLAB programming languages and environments. The example case in this thesis uses the Python programming language.

1.6.3 Integration of Simulation Modules with Architecture

The API libraries function as a "wrapper" around simulation modules. The API libraries feed information into modules, send information out of modules, and have executive control over execution of the main loop of a module. Methods for each function need to be developed and tested. Multiple modules will be integrated in the example case for this thesis.

1.6.4 Evaluation of System Performance

Upon completion and integration of module API libraries with software modules, performance of a distributed simulation is evaluated. Each configuration is unique and the evaluation process must be completed for every new configuration.

2. ARCHITECTURE LEGACY

The Vehicle Systems & Control Laboratory is a part of the Department of Aerospace Engineering at Texas A&M University. The purpose of the lab is to be an aerospace vehicle research, simulation, and education facility. The VSCL currently has a series of distributed networked PCs for simulation[14]. These PCs are used for manned vehicle simulations, unmanned vehicle simulations, machine learning and control, cockpit displays, and many other aerospace related uses. The expansion to multiple PCs was due to simulation of large numbers of complex simulated agents and vehicles[14, 18, 19], considerably complex and high fidelity environments and vehicles[18, 20, 21, 22], standalone commercial-off-the-shelf (COTS) software integration to simulation networks[14, 23], and other features and capabilities expanded upon from and by previous generations of architectures[14, 24].

As an educational facility, the VSCL hosts many classroom activities. One activity is for aircraft flight dynamics and design students to experience simulated flight dynamics and building and testing mathematical aircraft models. The Cockpit Systems and Displays class designs and tests displays, interfaces, and human factors in the VSCL. Students also design and test autopilot and stability augmentation systems in the laboratory. Flight test engineering students and researchers practice simulated flight test maneuvers and data acquisition in the laboratory before performing their experiments on actual test aircraft. Many of these classroom activities use multiple identical software modules but may require a few more or less sets of data input and output transmitted from the simulation system than others. Many simulation designers and maintainers are students or novice engineers with limited experience[3]. As such, it is outside the abilities for many of the simulation engineers to implement a custom distributed architecture to meet their needs; nor do they have the knowledge of the distributed computer communication programming techniques required to implement such an architecture.

The initial hardware architecture of the VSCL in 1998 (then called the Flight Simulation Laboratory) included only a single engineering flight simulator (EFS)[24] using a monolithic architecture. The EFS used a Silicon Graphics Incorporated (SGI) Onyx Reality 2 computer for scenery generation and six degree of freedom aircraft model dynamics calculations[3] in a single compiled binary executable using a mixture of procedural and modular architecture paradigms[24] in the Atlas simulation code. The SGI computer had the aforementioned high purchase, support, and maintenance costs and also created a single point of failure in the ability to run simulations for the laboratory. The EFS cockpit hardware and human interfaces were constructed from a surplus Air Force T-37 fuselage with glass displays replacing the original instrument panel and three projectors for the out-the-window view[24] as shown in Figure 2.1. The glass display interfaces were generated using Windows 98 PCs[3]. The flight controls and standard pilot input devices in the cockpit were fitted with a variety of optical rotary encoders, potentiometers, and switches which were interfaced to several data acquisition boxes from BG Systems and US Digital[14].

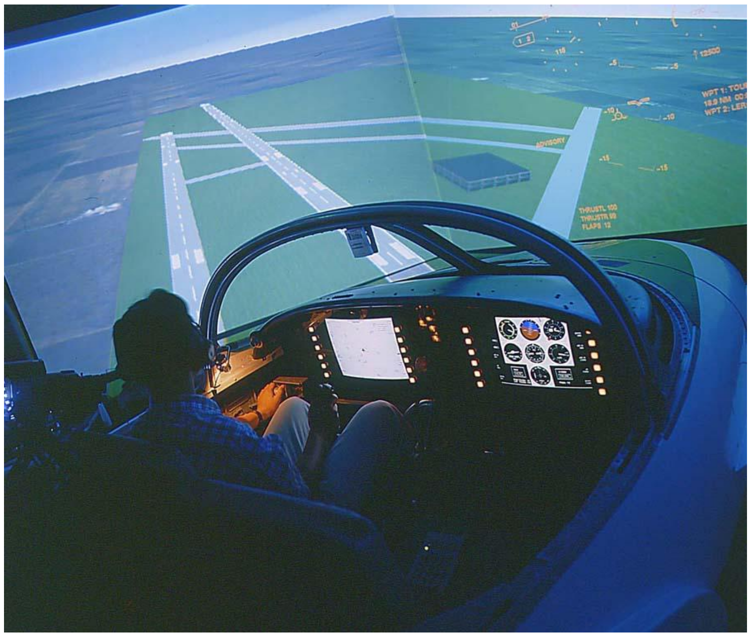


Figure 2.1: VSCL Engineering Flight Simulator c. 1998[3]

The next upgrade to the VSCL in 2005[14] distributed the main EFS computing to multiple PCs, added three PC based pilot stations, of which the hardware architecture for each is shown in Figure 2.2, upgraded the Windows operating systems to Windows XP, as well as installing touch screen liquid crystal displays (LCDs) to replace the cathode ray tube displays and input buttons of the previous HDDs as shown in Figure 2.3.

The EFS left out-the-window display and left head-down display were generated with one Windows PC, the EFS center out-the-window display were generated with one Windows PC, the EFS right out-the-window display and right head-down display were generated with one Windows PC, and the T-37 control inputs and cockpit outputs interface with one Linux PC. The pilot stations consisted of one Windows PC generating the out-the-window view and HDD. Connected to the pilot stations are COTS PC pilot yoke and pedal systems. A minor upgrade consisting of new PC

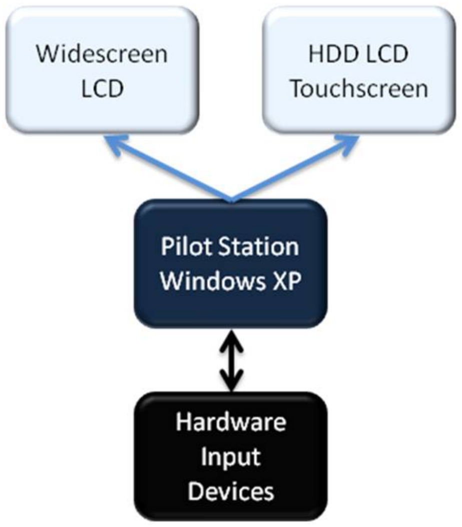


Figure 2.2: Pilot Station Hardware Architecture[30]

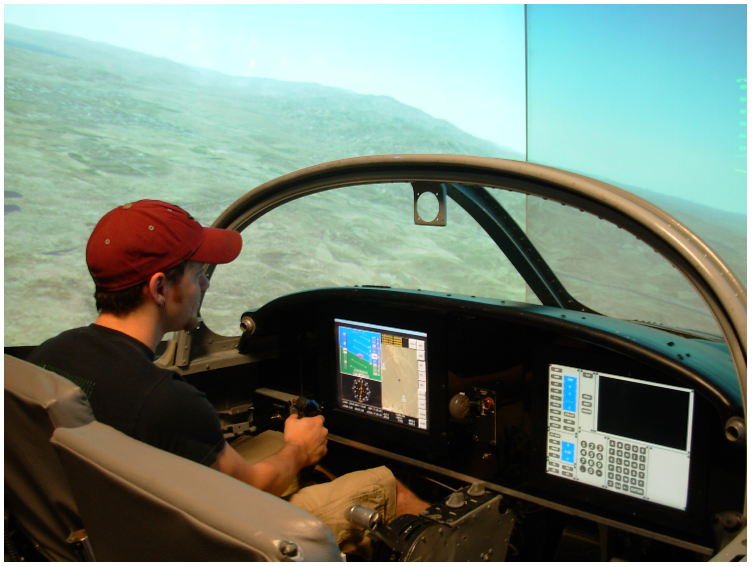


Figure 2.3: VSCL Engineering Flight Simulator c. 2009

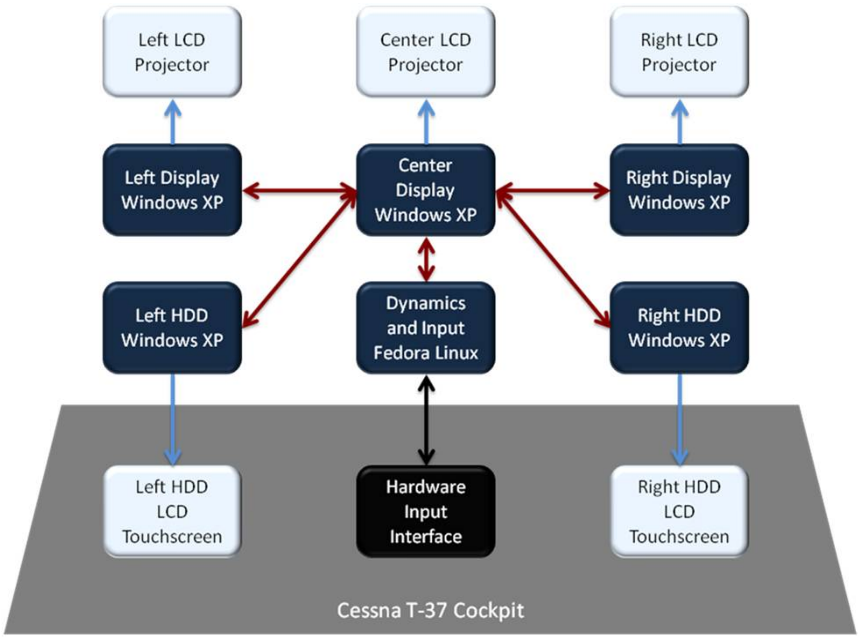


Figure 2.4: EFS Hardware Architecture[30]

hardware and COTS yoke and rudder hardware was implemented in 2009. Included in the 2009 upgrade were new Windows PCs for each of the EFS HDDs, where now each of the EFS display computers generate only one display as shown in Figure 2.4. Figure 2.5 shows the current setup of the VSCL pilot stations simulators.

Distributed simulations in the VSCL communicate data using TCP and UDP sockets of the Internet Protocol Suite. Communication can be point-to-point, multicast, or broadcast. The main cause of complex interdependence between modules has been the use of hard-coded I/O data packet formats for network-based distributed communication, where a change in one module would require all of the modules to be adjusted and recompiled[30]. A wide variety of programming and scripting languages are used in the VSCL. A large number of the educational simulations are created in MATLAB. C and C++ are used to communicate with the EFS control input hardware and for some of the dynamical simulations. Other languages used in

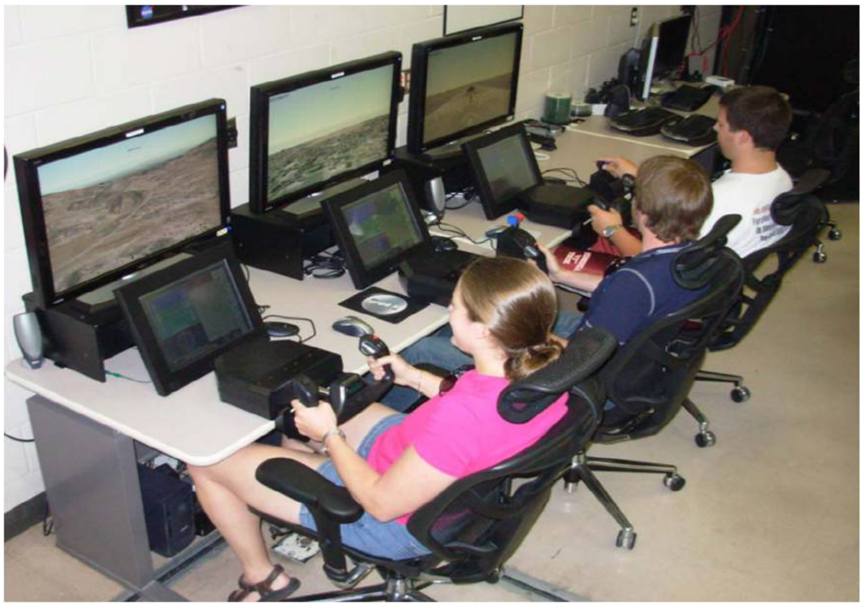


Figure 2.5: VSCL Pilot Station Simulators c. 2009

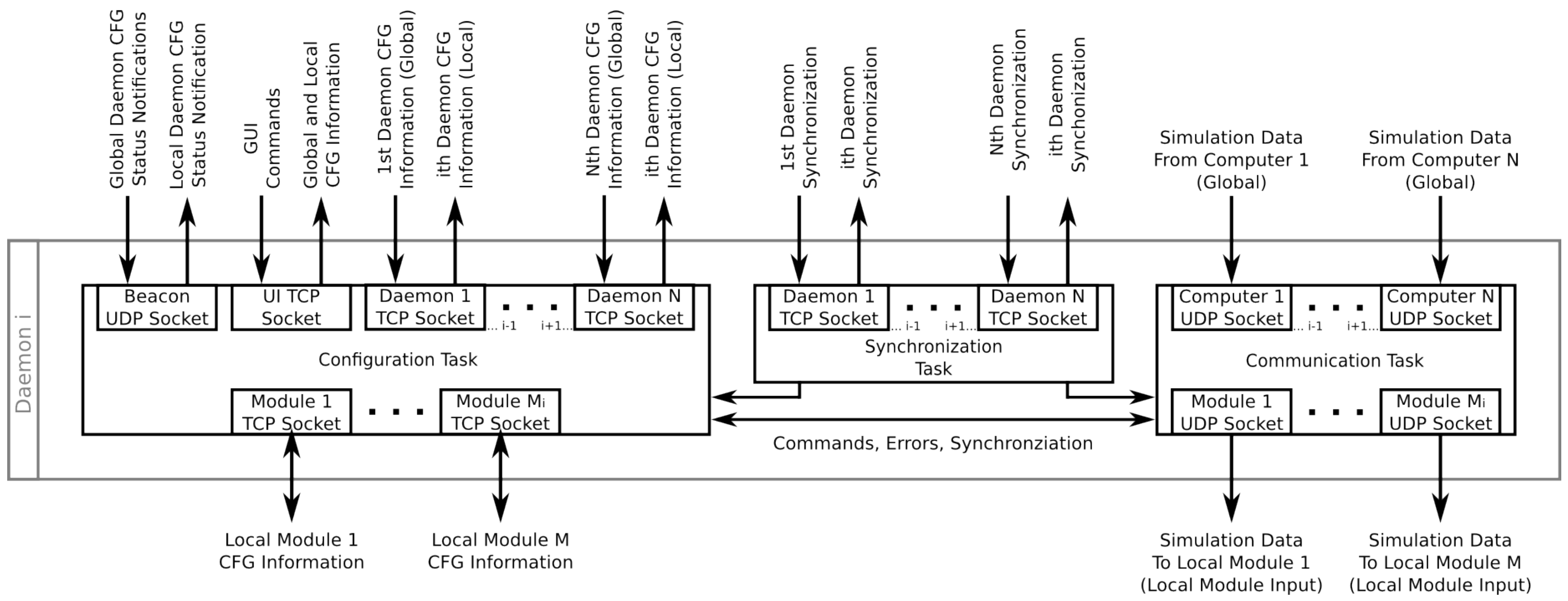
VSCL codes are Fortran, Java, and Python. An architecture implementation will be required to interface with each of these programming languages.[23]

3. DAEMON

The hADES simulation Daemon can be described as an information backbone for the simulation architecture. The Daemon is an object-oriented program that is executed in the background of all simulation computers and communicates the availability of and access to the actual Module data between all Modules on the network. The simulation Daemon is similar to the Run-Time Infrastructure of the High Level Architecture. In this thesis, the Daemon is programmed to be cross-platform (Linux and Windows) by implementation using the Python programming language. The intent is to make communication between multiple operating systems and platforms transparent to the simulation Module designer. A novel feature of the simulation Daemon is that it can convert data types, data units, and fill in missing data on-the-fly without recompiling or reprogramming connected simulation modules that may have conflicting input and output data structures.

The three main functional tasks of the simulation Daemon are synchronization, configuration, and communication. These tasks are run as parallel processes with asynchronous interprocess communication between the three. The synchronization task handles synchronization between the system clocks for all Daemon computers on the simulation network. The configuration task handles connecting to new simulation Modules and recording the input and output data information (described in Section 3.2.1) for that Module. The configuration task then relays to all other Daemons on the network the availability of data and details about that data for the simulation modules connected locally to a Daemon. This task is also used to map where the inputs to any simulation Module come from (i.e. which Module on which computer, remote or local, to read Module output data from). Another novel feature implemented in order to make the architecture easily extensible is a defined interface to the configuration task to allow data paths to be defined on-the-fly during run-time. The usual endpoint to this interface is a GUI to be used by a human operator that will display data paths for all Modules and their respective data as well as provide a way for the operator to command changes to data paths between any Modules connected to the architecture. The configuration task is designed to pass commands from a local UI to remote Daemons anywhere on the network to allow a single user from one location to be able to configure all Modules anywhere in the simulation network. The communication task receives remote Module output data, does conversions if necessary, and sends it as input data to a local Module. Because data is only converted as necessary when being received, it is not necessary to have simulation Modules send their output data through the local Daemon. Thus, output data from simulation Modules is sent directly to the simulation network avoiding extra and unnecessary transmission steps and latency of sending data from a local Module to a local Daemon and then from that Daemon to the network.

Figure 3.1 is a functional diagram showing how simulation data and configuration information are passed between the i^{th} Daemon in a network, the Daemon main tasks, and the other N-1 Daemons in a network. The Daemon uses TCP sockets for sharing configuration data and UDP sockets for all simulation data. For all further descriptions in this chapter of how a Daemon instance on a network computer communicates with other Daemons on the network and with remote and local Modules, the local Daemon is the i^{th} Daemon and is referred to as “a Daemon”, the local Modules are referred to as the 1^{st} through M_i^{th} Modules for all M_i Modules located on the i^{th} computer, and the other Daemons on the network, whose Modules are called global or remote Modules, are referred to as the 1^{st} through N^{th} Daemons noninclusive of the i^{th} which is local.



Global Daemon CFG Status Notifications
Local Daemon CFG Status Notification
GUI Commands
Global and Local CFG Information
1st Daemon CFG Information (Global)
ith Daemon CFG Information (Local)
Nth Daemon CFG Information (Global)
ith Daemon CFG Information (Local)
1st Daemon Synchronization
ith Daemon Synchronization
Nth Daemon Synchronization
ith Daemon Synchonization
Simulation Data From Computer 1 (Global)
Simulation Data From Computer N (Global)
Daemon i
Beacon UDP Socket
UI TCP Socket
Daemon 1 TCP Socket
... i-1
i+1...
Daemon N TCP Socket
Configuration Task
Module 1 TCP Socket
Module Mi TCP Socket
Daemon 1 TCP Socket
Daemon N TCP Socket
Synchronization Task
Computer 1 UDP Socket
Computer N UDP Socket
Communication Task
Module 1 UDP Socket
Module Mi UDP Socket
Commands, Errors, Synchronziation
Local Module 1 CFG Information
Local Module M CFG Information
Simulation Data To Local Module 1 (Local Module Input)
Simulation Data To Local Module M (Local Module Input)

Figure 3.1: Daemon Task Diagram

3.1 Synchronization Process

The computing systems in this thesis operate using commonly used nondeterministic or non-real-time operating system software environments. Because a broad goal of the architecture is to give as close to real-time performance as possible, a soft real-time approach, where the underlying software systems attempt to meet time deadlines as best as possible by scheduling tasks according to a system clock, is used. The system clock on each PC is not guaranteed to be accurate relative to an independent source. In order to synchronize the execution of simulation Module loops, the Berkeley algorithm[31] for clock synchronization is implemented in this thesis. The *master* Daemon, with respect to the synchronization algorithm, maintains a database of the offset for all other Daemons relative to itself. The *master* notifies the *slaves* of what offset they should maintain. The synchronization process passes offset data for its local Daemon to all other processes on the Daemon via internal process-to-process communication. The synchronization process maintains TCP connections to all the other Daemons for passing synchronization messages as shown in Figure 3.1. If the *master* Daemon goes offline, another Daemon is chosen as *master* according to the algorithm. Also, if the *master* Daemon computer is overloaded with processing tasks, another Daemon can be chosen as the *master*.

The Berkeley algorithm follows the following process inside a Daemon:

1. A Daemon is chosen as the *master* clock via an election process and all other Daemons become *slaves*.

2. The *master* polls the *slaves* with a message that contains the timestamp when the *master* sent the message. The *slaves* reply to the message by returning the original message with a timestamp from the *slave* computer appended to it.

3. The *master* calculates the round-trip time for each of the message replies for each of the *slaves*.

4. The message polling process is repeated multiple times ignoring any values far from the average for each *slave*.

5. The *master* sends an offset amount to each of the *slaves* of how much each must adjust its own clock by.

3.2 Configuration Process

The configuration process is designed to send and receive information about Modules connected to the Daemon network. This process functions using the Client-Server Model[32] for configuration information sent over TCP sockets. The configuration task on a Daemon acts as the source of all configuration properties for Daemons and the Modules located on that computer; thus acting as the "Server" in the Client-Server model. All other Daemons on the network act as "Clients" in the Client-Server model for receiving configuration information as necessary via server-push[33] style messages. Figure 3.2 shows how the TCP sockets for sharing Daemon configuration (CFG) information for a 4-computer simulation network are connected. Each circle is a Daemon on the network and the arrows show the TCP socket connections between Daemons and the direction of communication for that socket. It is important to note that even though TCP sockets are bidirectional between any two connected Daemons, this architectures uses two TCP sockets for communication of CFG information - one for sending information and one for receiving information. The purpose of having two sockets is because Daemons can join the network at any time; this method avoids a race condition where two Daemons attempt to create a connection with the other when only one connection would be wanted. This method also maintains the "Client-Server" relationship with respect to the data being transmitted across the respective sockets. Figure 3.1 shows the sockets used for this process and what type of information is sent across them. In the Client-Server model, a socket is open to accept incoming connection requests and spawn a new connection when accepted. A separate accepting socket exists for each type of data to be sent/received: UI data, global Daemon CFG data, and local Module CFG data.

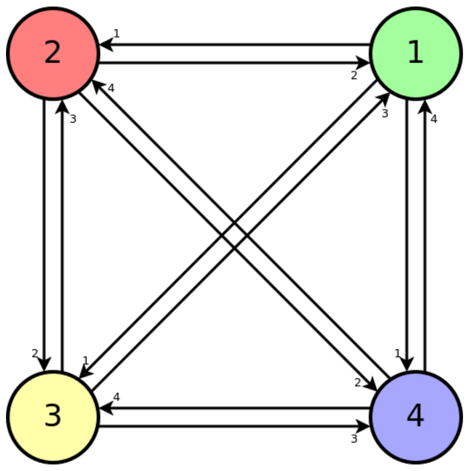


Figure 3.2: Example Network Daemon CFG TCP Socket Layout

When a Daemon joins a network, it must notify other Daemons (if any) of its existence. This is done using the Beacon UDP Socket. The Beacon UDP socket is used to send and receive CFG status information - a way to identify if the configuration has changed on a remote Daemon. The Beacon CFG status is sent at a specific rate to multicast address that all Daemons are subscribed to so only one transmission of the data is required for all other Daemons to receive it.

Figure 3.3 shows the flowchart of the Daemon configuration task process which is executed at a specific rate. When a Daemon program first starts, the CFG storage data structures and objects are initialized and sockets for accepting network communication and notifying other Daemons on the network of the i^{th} Daemon's existence are established. The main loop first checks all sockets for data and incoming connection requests. If Beacon CFG status information is received from a remote Daemon, it is processed to verify if the local copy of the remote Daemon's CFG information is correct. If a beacon signal is received from an unknown Daemon, an attempt is made to create a TCP communication link with it. If a remote Daemon is attempting to make a TCP communication link with the local Daemon, new CFG storage data structures are created for the remote Daemon and the connection is accepted. If new CFG information is received on a TCP communication link from a remote Daemon, the CFG information is updated. If commands are received on a TCP communication link from a remote Daemon, the commands are processed. If a local Module is attempting to make a TCP communication link with the local Daemon, new CFG storage data structures are created for the local Module and the connection is accepted. If CFG information is received from a local Module, the info is updated in the CFG storage data structures and forwarded to the global Daemons. If commands are received from the UI communication link, the commands are processed and sent to global Daemons if required. If any communication links are lost with local Modules or remote Daemons, the communication link sockets and data structures are cleaned up and remote Daemons are notified if necessary. The UI is then updated with the current state of the Daemon network. The loop then sleeps until the next beacon update is due or the next configuration update is due. Because CFG information and commands are not sent very often, the loop rate for the configuration process can be set to a low value to avoid slowing the computer down with many loops that do not actually receive or process data. If a shutdown is caught, the main loop ends.

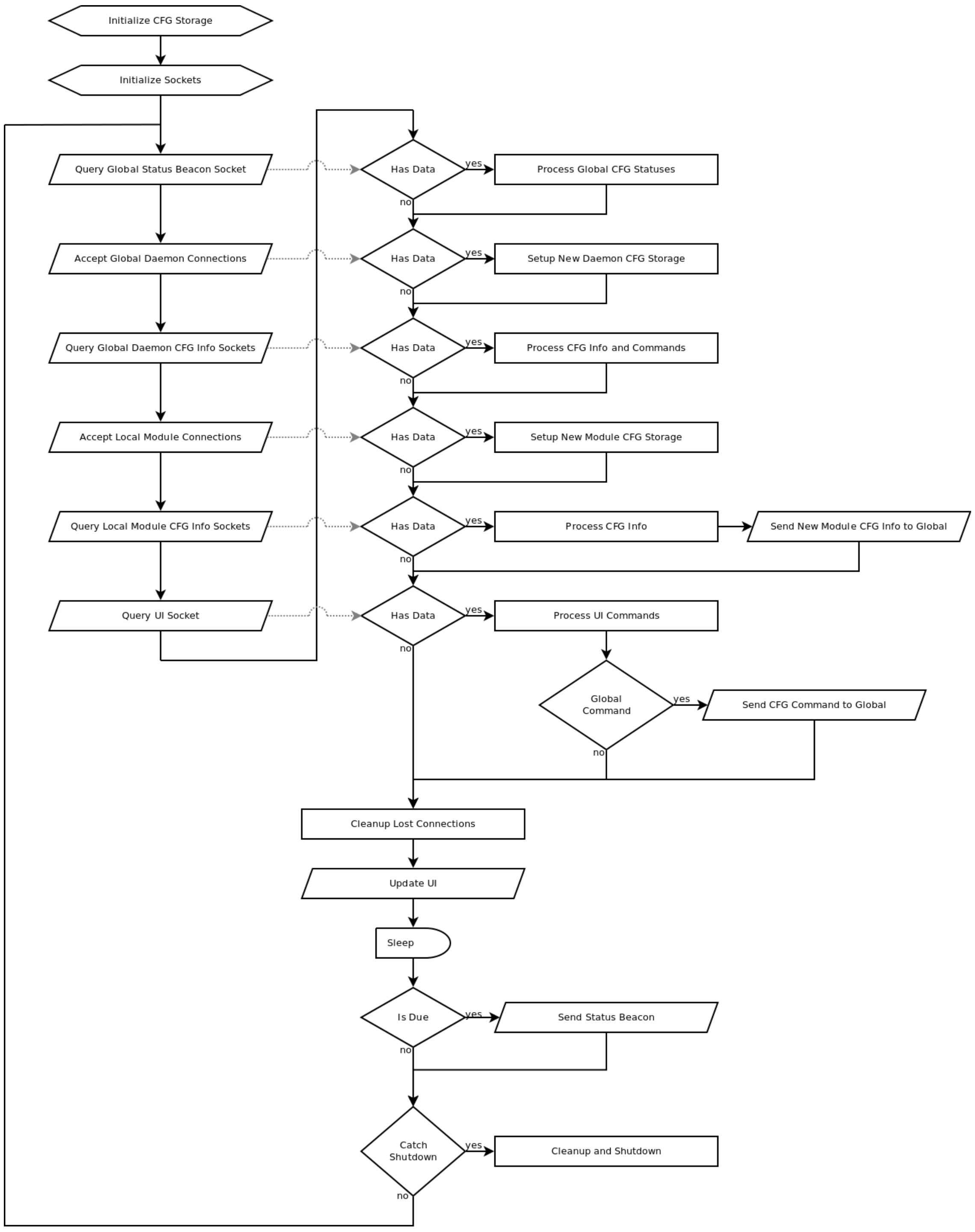

Initialize CFG Storage
Initialize Sockets
Query Global Status Beacon Socket
Accept Global Daemon Connections
Query Global Daemon CFG Info Sockets
Accept Local Module Connections
Query Local Module CFG Info Sockets
Query UI Socket
Has Data
yes
no
Process Global CFG Statuses
Setup New Daemon CFG Storage
Process CFG Info and Commands
Setup New Module CFG Storage
Process CFG Info
Send New Module CFG Info to Global
Process UI Commands
Global Command
Send CFG Command to Global
Cleanup Lost Connections
Update UI
Sleep
Is Due
Send Status Beacon
Catch Shutdown
Cleanup and Shutdown

Figure 3.3: Daemon Configuration Task Flowchart

The communication process, as shown in Figure 3.1, uses UDP sockets to send and receive data. The data output from each Module is unique. As such, a unique port number is reserved for use exclusively by each Module that has output data to send. The reason for this is to minimize any extra computation required to sort and determine which data packets came from which Module if multiple Modules were sending data to the same port. Port numbers can be reserved internationally with the IANA for use by specific programs[34]. The full range of port numbers are 0-65535. A subset of the full range of available ports are a set meant for dynamic use and cannot be registered with the IANA. Ports in the dynamic use range (49152-65535)[34] are used by the architecture presented in this thesis to avoid conflicting with international agreements and programs that may have registered and already be using specific ports in the lower range. The port designation and reservation is handled by the configuration process. Ranges of ports can be either predetermined and known by each Daemon at run-time or Daemons can choose a port number and ask if other Daemons are using it. The dynamic use range includes 16384 ports which is several orders of magnitude larger than the number of expected distributed Modules envisioned when developing the architecture in this thesis. Thus, either method of port distribution should be sufficient. The configuration process notifies the communication process of ports to use via internal process-to-process communication and notifies local Modules of ports to use via local Module CFG information TCP sockets.

3.2.1 Configuration Storage

The configuration process stores information about each Module on the simulation network regardless of whether the Module is remote or local. This information is used to route communications across the network. A mapping can be made between the outputs of one Module to the inputs of another. The configuration information for remote Modules is stored on each Daemon to allow the operator connected through the UI interface to see the state of all Modules on the simulation network as well as send commands to configure Modules anywhere on the network. All configuration information is stored as text. The storage data types for CFG text information should be consistent across all Daemons which includes character array length. The CFG information is used for defining the uniqueness of a module in the configuration process as well as for performing data type and unit conversions in the communication process.

Relevant configuration information for Modules are:

- Module Name
- Module Version
- Author
- API Language
- API Version
- Module Loop Rate
- Ordered list of input data packet
  - Data Name
  - Data Type
  - Units
  - Source
- Ordered list of output data packet
  - Data Name
  - Data Type
  - Units

3.2.2 Configuration Uniqueness

The architecture developed in this thesis provides the ability to implement a very complex and extensible simulation environment capable of connecting multiple unique heterogeneous simulation Modules and allowing them to communicate with one another. With multiple engineers authoring Modules that may serve similar purposes, it is possible that the authors will choose Module names that are not unique. However, it is still possible to differentiate Modules with the same name from one another if some of the other CFG information is not identical. Not all possible CFG information is required for a Module to be assimilated into a network and executed, however, a subset of that information is and can be used to uniquely identify it.

Several items of information required to uniquely identify a Module on a network are:

- Module Name
- Module Version
- API Language
- API Version
- Module Loop Rate
- Ordered list of input data packet
  - Data Name
  - Data Type
  - Units
- Ordered list of output data packet
  - Data Name
  - Data Type
  - Units

A concatenated string of the unique CFG information text in the order presented can be long and modules are not guaranteed to have the same concatenated string length as others. In order to shorten the length of the unique string identifying a module as well as guarantee identical length, a *hashing function* can be used to store a unique identifier for each Module implementation. A hashing function is an algorithm that maps a data set or text string of variable length to smaller data sets or text strings of fixed length. Many hashing functions can be used for this process. A hashing function that will allow only few or no collisions is desired. An example hashing function that is included in the initial implementation of this architecture is the MD5 algorithm which takes as input a message of arbitrary length and produces as output a 128-bit "fingerprint" or "message digest" of the input[35]. An advantage of this algorithm is that it is computationally infeasible or very unlikely to randomly produce two messages having the same message digest, or to produce any message having a given prespecified target message digest[35].

The CFG information for a Module is stored internally using the pair of the Module name and Module CFG hash as an identifier or key for the Module. The identifier can be used to determine how many instances of a Module are located on a local computer and/or on the entire simulation network.

3.2.3 Configuration Commands

Configuration commands are the method of communication of CFG information between a local Module and the corresponding Daemon on the computer it is located on. Commands are also passed between Daemons to share CFG information about Modules connected locally to each Daemon. Commands are also used by the UI, and passed from Daemon-to-Daemon, in order to start and stop simulations groups and to define output-input mappings. The same command syntax can be used for all types of messages. Examples of why the term "command" is used is because a local Daemon will command other global Daemons to update their CFG information when new local Modules join or leave the network or an operator will command a simulation group to start through the UI. Because the commands are used by both the Daemon and the API libraries, the command syntax must be easily implemented in any programming language. A message standard has been implemented for this architecture to be both simple to implement and easy to parse so as to not require complex or third-party parsing libraries. The message standard is also ASCII-character-based[36] to be human readable and thus easy to debug by any programmer.

Because commands are sent over TCP connections, the IP address, and thus the Daemon origin for each command is embedded in the socket/data transmission and is not included in the command text. The command syntax (Figure 3.4) has several variables. The characters available for command variables are [a-z][A-Z][0-9] of the ASCII character set to make them easily human readable. Several special characters are used as separators in the command and to define the beginning and end of a command. The first variable, `cHash`, is the command hash - a unique identifier for the command. A unique identifier is required for each command that originates on a Daemon because some commands will require responses. This will allow a Daemon to keep a cache of commands expecting a response so as to know how to properly process the response message. The second variable, `rHash`, is the response hash - it notifies which command is being responded to. The response hash is an optional variable that is only required for commands that are responding to other commands. The command hash and response hash are separated by the respond separator character: a period, '.', ASCII character. The third variable, `cName`, is the command name - a unique name used to determine how many command arguments to expect and what to do with the arguments. The command name is separated from the previous variable by the argument separator character: a comma, ‘,’, ASCII character. The last variables, `cArgs`, are the command arguments - optional variables that correspond to the arguments for each command in the order they are expected. Some commands have no arguments and some have many. The start of command character is the less-than ASCII character, ‘<’. The end of command character is the greater-than ASCII character, ‘>’.

```
<cHash[.rHash],cName[,cArgs...]>
```

Figure 3.4: Command Syntax

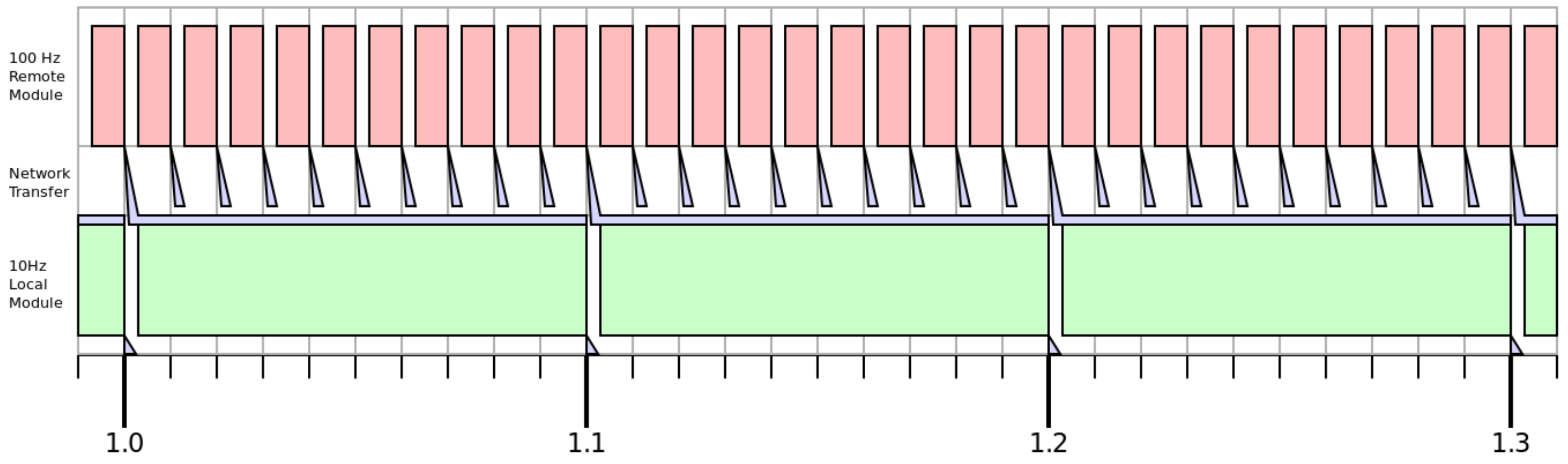
3.3 Communication Process

The purpose of the communication process is to create soft real-time performance of simulation data transmission at a desired rate. The main functions of this process are to receive output data from global Modules, convert the data as necessary, and to send input data to local Modules. Between the UI and the configuration process, a mapping is designated between the outputs of certain Modules to the inputs of others. The configuration process then notifies the communication process of which global Module outputs are required as inputs for the local Modules. The communication process will create data structures and objects for receiving global data, converting it if necessary, and then for sending it to the local modules.

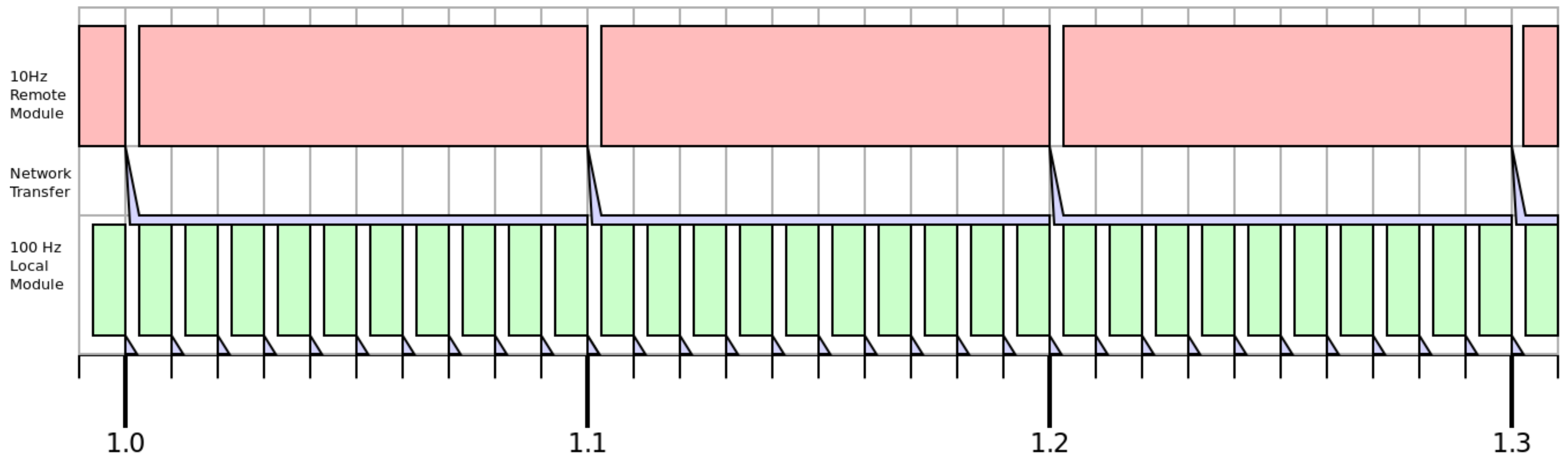
Initial implementation of the architecture communication process functions as a sampled-data system, where the most recent data from the output of a global Module is sampled, processed, and sent as input to a local Module. The purpose of this is to allow Modules with different internal loop rates to communicate with one another as shown in Figure 3.5. If a local Module main loop executes at a faster rate than the remote Module, the most recent data is used for all loops until new data is available as shown in Figure 3.5a. If a local Module is at a slower rate than the remote Module, the data available when the local Module loop starts is used and subsequent data will be thrown away until the local Module loop is finished as shown in Figure 3.5b. This is not the only option for handling extra data, but is the initial implementation for this architecture. Because a sampled-data system will require all Modules to calculate outputs in the first loop of execution without having sampled inputs, initial conditions for all states will be required to be known by each Module before run-time. Initial conditions can be prescribed internally to each Module, or can be set via commands from the UI.

Worst case transmission and processing latency from a remote Module to a local Module should be the loop rate of the remote Module if a packet transmission is missed. Empirically in the VSCL, data packet transmission rates of over 50kHz are possible across a 1Gbit/s Ethernet connection suggesting that transmission rates will be significantly faster than processing loop rates. This architecture implements a small delay (significantly smaller than the Module loop rate but larger than expected LAN transmission time) before attempting to read in data from remote Modules in order to not miss packets. The configuration process will share clock sync information with the communication process. The clock sync information paired with the delayed data reading leads to very small expected distributed transmission and processing latency. Read delay times and rate of clock syncing are adjustable parameters available to fine-tune simulation setups.

Figure 3.6 shows the flowchart of the Daemon communication task process. When a Daemon program first starts, the communication task prepares data objects and structures for storage of simulation data. The main loop checks for CFG updates to determine if the Daemon needs to prepare to receive data from remote Daemons.



(a) Faster to Slower



(b) Slower to Faster

Figure 3.5: Sampled-Data Examples

Next the communication process checks for updates on clock synchronization. After that, if any remote Modules are scheduled to have sent data to the network, that data is received. *To help avoid erroneous network data and to reduce total bandwidth usage on the network, transmission of Module data between Daemons will be sent using multicast addresses unique to each Daemon.* If data transmission deadlines are being missed, the configuration process and UI can be notified. Once data has been received, if any local Modules are scheduled to be sent data, the appropriate data is converted if necessary, packed into the proper data structure, and sent to the local Modules. The loop will then sleep until data is scheduled to be received from remote Modules or sent to local Modules or for a preset amount of time of in order to receive CFG or synchronization updates should they have been sent to the process. The sleep times are adjustable parameters available to fine-tune simulation setups. If a shutdown is caught, the main loop ends.

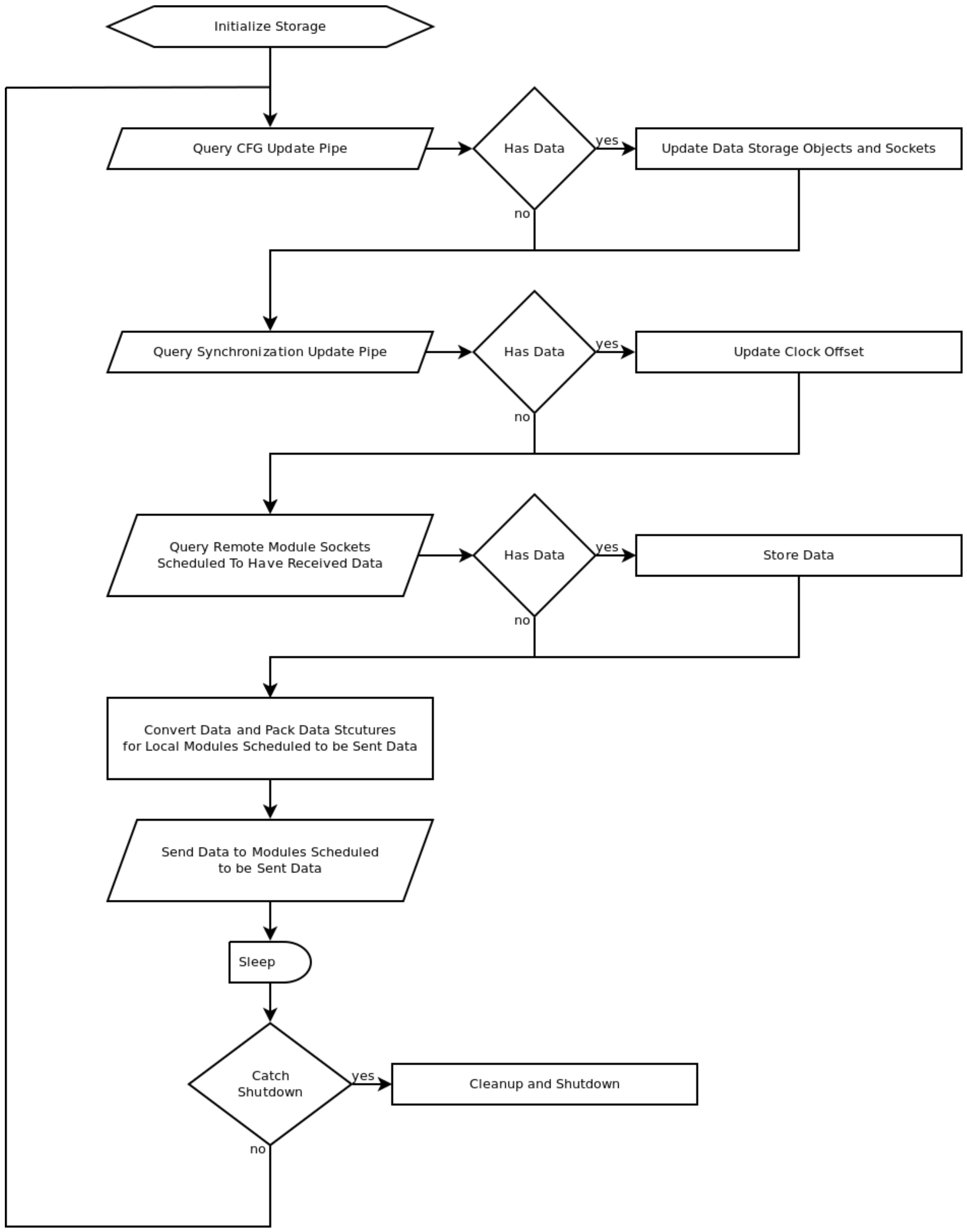

Initialize Storage
Query CFG Update Pipe
Has Data
yes
Update Data Storage Objects and Sockets
no
Query Synchronization Update Pipe
Has Data
yes
Update Clock Offset
no
Query Remote Module Sockets Scheduled To Have Received Data
Has Data
yes
Store Data
no
Convert Data and Pack Data Stcutures for Local Modules Scheduled to be Sent Data
Send Data to Modules Scheduled to be Sent Data
Sleep
Catch Shutdown
yes
Cleanup and Shutdown
no

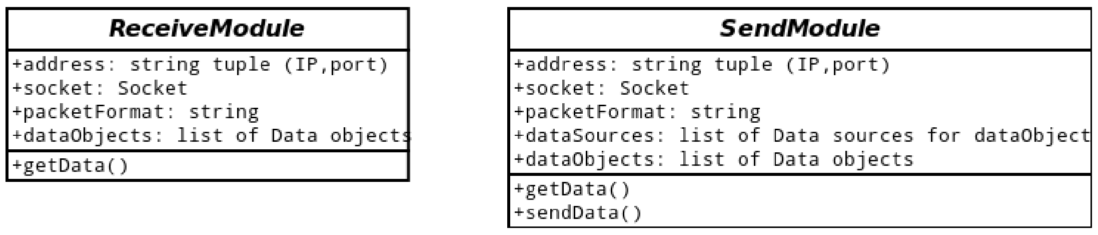
Figure 3.6: Daemon Communication Task Flowchart

3.3.1 Simulation Data Storage and Transmission

The Daemon communication process can create Object-Oriented Programming (OOP) paradigm objects to receive and store data from any simulation Module. OOP objects can also be created for any local Module in order to convert, modify, and/or create new data packets to be set as input to the local Module - a novel feature of this architecture. The classes for defining the previously mentioned types of objects can be broken into two types: `ReceiveModule` class for defining objects for receiving data from Modules and `SendModule` class for defining objects for sending data to local Modules. Module objects are designed with sending or receiving methods to be used by the communication process for packing, converting, and sending data and for receiving and storing data. The configuration process stores mappings between the outputs of Modules to the inputs of others. This mapping is shared with and also stored in the communication process to create receiving Module objects for any Module only while data is required from them as inputs to local Modules. The specific information in the mapping is which data item in the ordered list of an output data packet for a Module maps to which data item in the ordered list of an input data packet for a local Module including the corresponding data types and units for each.

The `ReceiveModule` class (Figure 3.7) is designed to receive data from any Module whose data is required to be sent to a local Module. It has an address variable for storing a string tuple of the multicast IP address and Port number that corresponds to the Module whose data is to be received. A socket for receiving data is created in the class. A packet format, which is a string representation of the data type and order for all output data to the Module, is stored as well. This class has a method for getting data which polls the socket and receives data, unpacks it according to the packet format, and stores it in the data objects. Data objects are designed to have methods for unit and type conversion. The data objects are stored in an ordered list with the order defined by the CFG information for that Module.

The `SendModule` class, as shown in Figure 3.7, is designed to take data stored in `ReceiveModule` objects, convert it if necessary, and send it to local Modules as input. It has an address variable for storing a string tuple of the `localhost` IP address and Port number that corresponds to the local Module data is to be sent to. A packet format, which is a string representation of the data type and order for all input data to the Module, is stored as well. This class has a *get method* which uses the output-input mapping to get data from the corresponding `ReceiveModule` data objects, converts the data types and units if necessary, and stores it in the data objects. As introduced in Section 1.5.1, another novel feature are faux data objects which can be created for data that doesn't exist as output on the network but can be substituted with a constant of specified data type and units. This class also has a *send method* which takes the data stored in the ordered list of data objects, packs into a packet according to the packet format string, and sends it as input to a local Module.


ReceiveModule
+address: string tuple (IP,port)
+socket: Socket
+packetFormat: string
+dataObjects: list of Data objects
+getData()
SendModule
+address: string tuple (IP,port)
+socket: Socket
+packetFormat: string
+dataSources: list of Data sources for dataObject
+dataObjects: list of Data objects
+getData()
+sendData()
Figure 3.7: Module Classes in Communication Process

Examples of data types and some of the possible type conversions are included in Table 3.1. This list is not exhaustive of all common data types available, but includes those in the initial implementation of this architecture and are based on C data types. Great care by the module programmer is needed for some conversions because converting a number from a format of longer bit length to a shorter one may result in the original number exceeding the size of the shorter format which will only be caught as an error at run-time. Initial implementation will cause an error to be sent to all Daemons when a data type conversion size violation occurs but it is also possible to truncate the number at the largest allowable size for the destination data type. Allowing conversions from longer formats to shorter ones is allowed because of historical prevalence of saving numbers in formats with maximum size significantly larger than the largest expected value of a variable in legacy code. Floating point numbers are allowed to be converted to integers by rounding because of historical prevalence of saving some integer numbers in floating point data types in legacy code. Strings of characters are handled as C-type character arrays and are converted to larger arrays by padding whitespace to the right or to smaller arrays by truncating from the right. The format code is a unique character to define the data type for the packet format string and is based on Python `struct` module[13] format codes.

Format Code	C type	# Bits	Compatible Conversions
?	Bool	8	b, B, h, H, i, I, l, L, q, Q, f, d, Ns
b	signed char	8	?, B, h, H, i, I, l, L, q, Q, f, d, Ns
B	unsigned char	8	?, b, h, H, i, I, l, L, q, Q, f, d, Ns
h	short	16	?, b, B, H, i, I, l, L, q, Q, f, d, Ns
H	unsigned short	16	?, b, B, h, i, I, l, L, q, Q, f, d, Ns
i	int	32	?, b, B, h, H, I, l, L, q, Q, f, d, Ns
I	unsigned int	32	?, b, B, h, H, i, l, L, q, Q, f, d, Ns
l	long	32	?, b, B, h, H, i, I, L, q, Q, f, d, Ns
L	unsigned long	32	?, b, B, h, H, i, I, l, q, Q, f, d, Ns
q	long long	64	?, b, B, h, H, i, I, l, L, Q, f, d, Ns
Q	unsigned long long	64	?, b, B, h, H, i, I, l, L, q, f, d, Ns
f	float	32	?, b, B, h, H, i, I, l, L, q, Q, d, Ns
d	double	64	?, b, B, h, H, i, I, l, L, q, Q, f, Ns
Ns	N len char[ ]	$Nx8$	Ms (M len char[ ])

Table 3.1: Data Type Conversion Compatiblity

4. APPLICATION PROGRAMMING INTERFACE

The Module Application Programming Interface (API) is a set of programming libraries used to integrate software modules into the distributed architecture developed in this thesis. Section 3 describes the architecture Daemon program used to synchronize configuration information across the simulation network, denote the various input-output mappings between simulation Modules, and to pass communication between the individual simulation Modules. This section describes the purposes and high-level overview of the API libraries: to register input and output data packet formats with the Daemon, communicate commands with the Daemon, and give executive control of software modules - allowing software modules to communicate with one another and for their execution to be controlled remotely.

Software modules can be implemented in many programming languages and operating systems. As such, the API libraries need to be implemented for each programming language and operating system combination for software Modules required to connect to a simulation Network. Requirements for the API implementation programming language and operating system are that they have the ability to communicate via the TCP/IP protocol.

Figure 4.1 shows the functional diagram of how the API and the Module code connect to create a single program that can communicate with a Daemon and the rest of a simulation Network. The API acts as a “wrapper” around the main software module. The API functions are used to send and received CFG and command information to and from the Daemon using the TCP port and to send and receive Module input and output using the UDP data socket. Section 3.3 explained how the architecture resembles a sampled-data system and requires initial conditions for all states in order to function. The initial conditions are loaded by the API either at load-time of the program, or on-the-fly as commands coming through the Daemon from the UI. The API functions are also used to start and stop execution of the simulation loop that calls the user implemented Module functions at the Module specified rate.

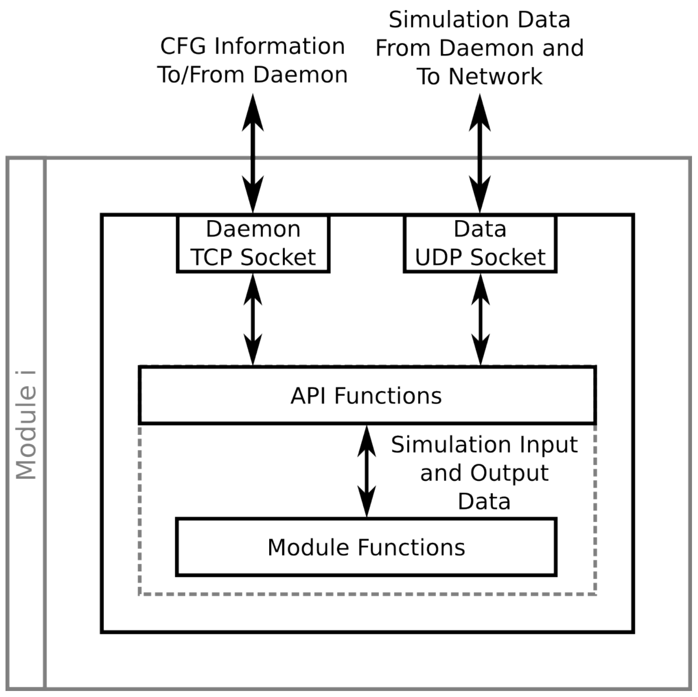


Figure 4.1: API-Module Functional Diagram

The API libraries are programming language specific and are used by engineers to design and integrate new simulation Modules and/or integrate COTS or legacy Modules to a simulation network. Communication methods using the API standardizes implementation of intermodule communication for a distributed simulation network for any organization, group, or researcher that uses this architecture. The API is meant to be minimally intrusive to Module development while still fostering extensibility.

There are several high-level functions the API fulfills:

- Connect a Module to the simulation network through the Daemon
- Transfer I/O CFG data information to the Daemon
- Start and stop module main loop execution
- Update internal loop timing with synchronization data from Daemon
- Receive other Module data as inputs from the Daemon
- Send Module outputs to the network

The main methods that need to be implemented by the API code are:

- Initializing all data structures and preparing sockets
- Reading, processing, and/or returning commands and other info through the Daemon TCP socket
- Reading and unpacking simulation data from the UDP socket
- Registering a callback function for the Module specific main loop function
- Executing the Module specific main loop function
- Packing and sending simulation data to the network through the UDP socket

Figure 4.2 shows a flowchart of Module operation using the API. A simulation Module initializes by setting up all relevant data structures, variables, and objects required for receiving commands from and communicating CFG information with a Daemon. A connection is then made with a local Daemon. CFG information is sent to the Daemon and a socket address for receiving UDP data is received from the Daemon. The socket address the Module uses for receiving data is also used for sending data to the network. If a Module has outputs, then an address to send output data to is received from the Daemon and stored. The API main loop then begins. A shutdown event inside the module is checked for first and a shutdown and data cleanup happens if caught. Otherwise the API will check for commands coming from the Daemon and process any received. If a “start Module” command has been received, the module state is changed to “running”. If the module stat is “running”, first any inputs are received and unpacked from the Daemon, the Module specific main loop functions are executed, and then any outputs for the Module are packed and sent to the network. The Module specific main loop functions are the main methods defined by the Module implementation programmer; in the context of the VSCL and related engineering simulation tasks, these functions are related to simulating dynamical systems. Lastly, the API main loop will sleep until the loop is due to process information again based on the Module rate.

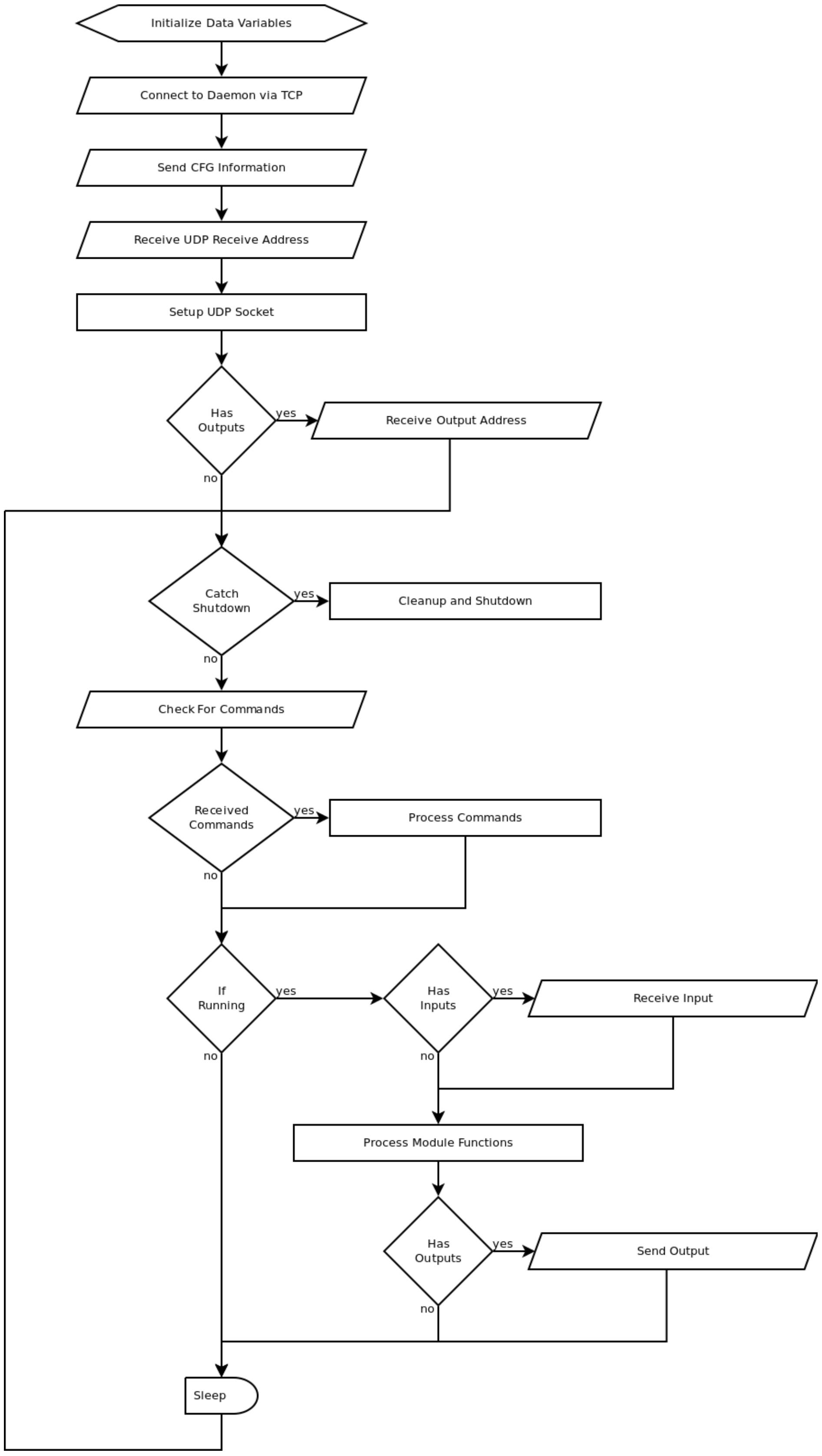


Figure 4.2: API-Module Execution Flowchart

5. EXAMPLE MODULE INTEGRATION

To test the main features of hADES, an example set of Modules is integrated to emulate a relevant use case for the VSCL. The example setup implements a piloted aircraft simulation. Modules implemented are a joystick Module for accepting pilot inputs, an aicraft simulation Module for simulating the physics of the aircraft, and a visualization Module for displaying relevant state output to the pilot.

Figure 5.1 shows the topology of Daemons and Modules and directions for I/O mapping for the example setup. Two computers are used for simulation in the network. A Daemon is run on each computer - *Daemon 1* on Computer 1 and *Daemon 2* on Computer 2. Computer 1 hosts two Modules - the *Joystick Module* and the *Aircraft Simulation Module*. Computer 2 hosts one Module - the *Visualization Module*. The *Joystick Module* has only output states. The *Aicraft Simulation Module* has input states that are converted and transmitted to it by *Daemon 1* and also has output states. The *Visualization Module* has input states that are converted and transmitted to it by *Daemon 2* and also has output states.

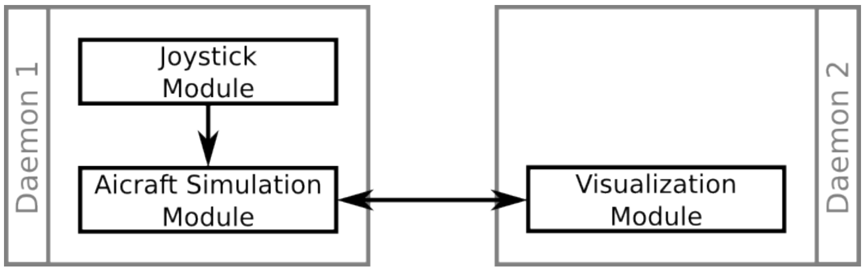
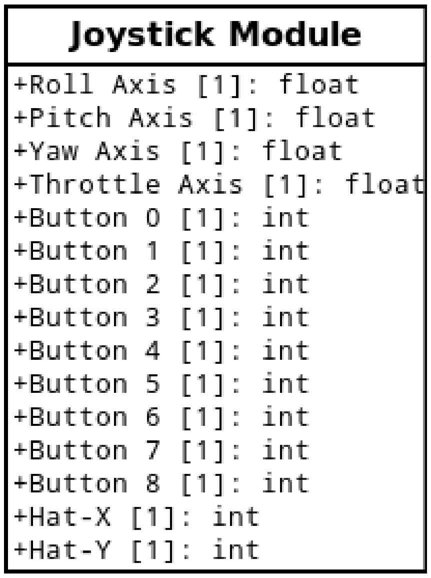


Figure 5.1: Example Module Topology

5.1 Joystick Module

The Joystick Module is a data acquisition program to record data at 50Hz from a Microsoft® Sidewinder® Precision Pro joystick. The Precision Pro joystick has four axes: a roll axis, a pitch axis, a yaw axis, and a throttle axis. It also has nine digital buttons and a four-direction (x-y) hat switch. The roll, pitch, and yaw axis positions are output as floating point numbers in the domain $[-1, 1]$ for left-to-right or down-to-up respectively. The throttle axis positions are output as floating point numbers in the domain $[-1, 1]$ for minimum-to-maximum respectively. All buttons are output as integer numbers; 0 for unpressed and 1 for pressed. Hat x and y positions are output as integer numbers in the domain [-1,0,1] or left-center-right respectively for x-position and up-center-down respectively for y-positions. The ordered output data structure for the Joystick Module is shown in Figure 5.2.



Joystick Module

```
+Roll Axis [1]: float
+Pitch Axis [1]: float
+Yaw Axis [1]: float
+Throttle Axis [1]: float
+Button 0 [1]: int
+Button 1 [1]: int
+Button 2 [1]: int
+Button 3 [1]: int
+Button 4 [1]: int
+Button 5 [1]: int
+Button 6 [1]: int
+Button 7 [1]: int
+Button 8 [1]: int
+Hat-X [1]: int
+Hat-Y [1]: int
```

Figure 5.2: Joystick Module Output Specifications

5.2 Aircraft Simulation Module

The Aircraft Module is the lateral-directional dynamical physics simulation for a Commander 700 aircraft using a discrete linear state-space model set to update at 100Hz. The Aircraft Module has inputs to turn a built-in autopilot on or off. The continuous state-space equation matrices[37] are linearized about a cruise speed, U_1, of 206.21ft/s; altitude, H_1, of 8500ft; angle-of-attack, α_1, of 5.25°; dynamic pressure, $\bar{q}$, of 37.7psf, and an elevator deflection, δ_e, of 0.1°.

The continuous state-space equation in vector form for states, $\underline{x}$, controls, $\underline{u}$, state matrix, A, and control distribution matrix, B, is given as:

$$\dot{\underline{x}} = A\underline{x} + B\underline{u} \tag{5.1}$$

For the Commander 700, the elements of the continuous lateral-directional state-space equations in vector form [37] for states: sideslip angle, β, body x-axis angular rate, p, body z-axis angular rate, r, ϕ roll angle, and ψ heading angle and controls: aileron deflection, δ_A, and rudder deflection δ_R, is given as:

$$\underline{x} = \begin{Bmatrix} \beta \\ p \\ r \\ \phi \\ \psi \end{Bmatrix}; \quad \dot{\underline{x}} = \begin{Bmatrix} \dot{\beta} \\ \dot{p} \\ \dot{r} \\ \dot{\phi} \\ \dot{\psi} \end{Bmatrix}; \quad \underline{u} = \begin{Bmatrix} \delta_A \\ \delta_R \end{Bmatrix} \tag{5.2}$$

$$A = \begin{bmatrix} -0.119 & -0.0013 & -0.993 & 0.159 & 0 \\ -1.22 & -2.00 & 0.0040 & 0 & 0 \\ 2.80 & -0.964 & -0.374 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} ; \quad B = \begin{bmatrix} 0 & 0.0038 \\ 1.92 & 0.191 \\ 0.137 & -1.59 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \tag{5.3}$$

The discrete form of the linear state-space equations for current timestep, k, and next timestep, $k+1$, is:

$$\underline{x}_{k+1} = \Phi \underline{x}_k + \Gamma \underline{u}_k \tag{5.4}$$

The continuous state-space linear equations can be converted to discrete equations for a specific update rate, T using:

$$\Phi(T) = e^{AT}; \quad \Gamma(T) = \left[\int_0^T e^{A\tau} d\tau \right] B \tag{5.5}$$

Which leads to the discrete state, Φ, and control, Γ, matrices for the Commander 700:

$$\Phi = \begin{bmatrix} 0.9987 & 0.0000 & -0.0099 & 0.0016 & 0 \\ -0.0121 & 0.9802 & 0.0001 & -0.0000 & 0 \\ 0.0280 & -0.0095 & 0.9961 & 0.0000 & 0 \\ -0.0001 & 0.0099 & 0.0000 & 1.0000 & 0 \\ 0.0001 & -0.0000 & 0.0100 & 0.0000 & 1.0000 \end{bmatrix} \tag{5.6}$$

$$\Gamma = \begin{bmatrix} -0.0000 & 0.0001 \\ 0.0190 & 0.0019 \\ 0.0013 & -0.0159 \\ 0.0001 & 0.0000 \\ 0.0000 & -0.0001 \end{bmatrix} \tag{5.7}$$

The autopilot is a full state feedback linear-quadratic regulator (LQR) controller implemented as a yaw-damper and heading regulator. The LQR controller is of the form $\underline{u}_k = -K\underline{x}_k$ to minimize the discrete cost function:

$$J = \frac{1}{2}\sum_{k-0}^{N} \left[\underline{x}_k^T Q \underline{x}_k + \underline{u}_k^T R \underline{u}_k\right] \tag{5.8}$$

To function as a yaw-damper (for r) and heading regulator (for ψ), the matrices Q and R are chosen as:

$$Q = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 10 \end{bmatrix}; \quad R = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{5.9}$$

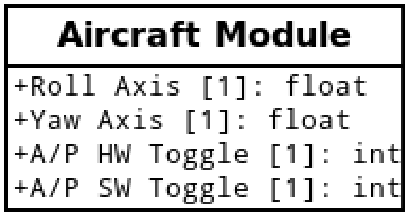
This leads to the optimal gain, K, that minimizes the cost function, J, from Equation 5.8:

$$K = \begin{bmatrix} 0.7176 & 0.4132 & -0.1370 & 0.8236 & 0.9970 \\ -1.1943 & 0.3387 & -2.4325 & -0.0498 & -2.9437 \end{bmatrix} \tag{5.10}$$

The autopilot can be turned on or off via one of two inputs: The Autopilot (A/P)

Hardware (HW) Toggle input or the Autopilot Software (SW) Toggle input. The first is to allow the autopilot to be toggled on or off from a button on a joystick. The second is to allow the autopilot to be toggled on or off from a GUI-based button on the pilot Visualization Module display. Either of the toggle buttons can be pressed to toggle the autopilot state to on or off, and the state change will happen on a downward edge detect when the button releases. A downward edge detect is when the button state goes from high/1/pressed to low/0/unpressed. This type of press detection does not toggle the state until the button is released to avoid toggling the state back and forth if the button is held down longer than one loop of the program. The state of the autopilot is output as the A/P Status where on is 1/high and off is 0/low. When the autopilot is engaged on, the joystick inputs to the Module are ignored and only the autopilot drives the system.

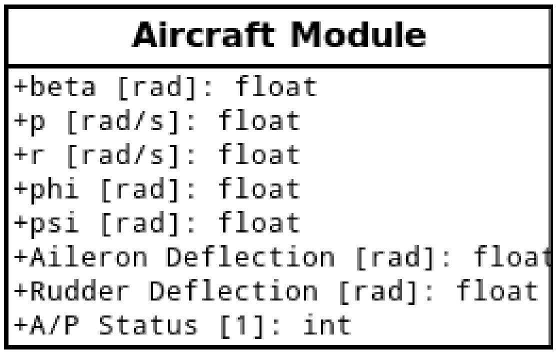
The ordered I/O data structures for the Aircraft Module are shown in Figure 5.3.



Aircraft Module

```
+Roll Axis [1]: float
+Yaw Axis [1]: float
+A/P HW Toggle [1]: int
+A/P SW Toggle [1]: int
```

(a) Inputs



Aircraft Module

```
+beta [rad]: float
+p [rad/s]: float
+r [rad/s]: float
+phi [rad]: float
+psi [rad]: float
+Aileron Deflection [rad]: float
+Rudder Deflection [rad]: float
+A/P Status [1]: int
```

(b) Outputs

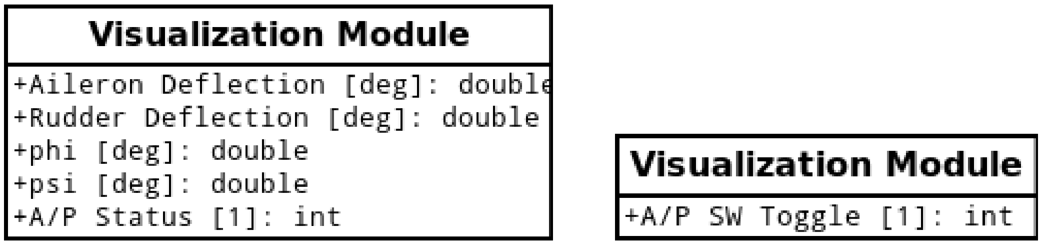
Figure 5.3: Aircraft Module I/O Specifications

5.3 Visualization Module

The Visualization Module shows the relevant aircraft navigation states to the pilot at 30Hz. The states shown are roll angle, ϕ, and heading angle, ψ, in degrees.

In addition, the actual deflections (δ_A and δ_R) of the control surfaces are displayed as well. The autopilot status is also displayed to the pilot. A software-based control is available as an output for the pilot to toggle the autopilot on or off.

The ordered I/O data structures for the Visualization Module are shown in Figure 5.4.



Visualization Module

+Aileron Deflection [deg]: doubl
+Rudder Deflection [deg]: double
+phi [deg]: double
+psi [deg]: double
+A/P Status [1]: int

(a) Inputs

Visualization Module

+A/P SW Toggle [1]: int

(b) Outputs

Figure 5.4: Visualization Module I/O Specifications

5.4 Data Routing Configuration

In this example, a scripting program performs the Module data mapping configuration prior to Module execution. The scripting program sends the required ASCII-character-based configuration commands to a Daemon through the Configuration Task UI TCP Socket. Module output will need to be verified both during and after Module execution to determine proper routing and data unit and type conversions. Lack of Daemon and Module runtime errors or notices of improper or incomplete I/O mapping are also a sign of full configuration success.

As explained in Section 3 and Section 4, each Module uses the API to send its output directly to the simulation network. When a specific Module requires the output of another as input, the local Daemon fetches the data, converts data types and units if necessary, and sends it to the Module. Figure 5.5 shows which Module outputs are required and configured through a Daemon to be relayed as input to other Modules for this example setup. The inputs and outputs are shown with the given **data type / units / name** for each. The *Joystick Module* has only outputs which are the joystick and throttle axes, the joystick buttons, and the joystick hat positions. All joystick axes are output as floating point numbers and all buttons and hat positions are output as integers. The inputs for the *Aicraft Simulation Module* are the Roll Axis coming from the *Joystick Module* Roll Axis, the Yaw Axis coming from the *Joystick Module* Yaw Axis, the Autopilot (A/P) Hardware (HW) Toggle button coming from the *Joystick Module* Button 8, and the A/P Software (SW) Toggle button coming from the *Visualization Module* A/P SW Toggle. The outputs for the *Aicraft Simulation Module* are the lateral-directional aircraft states as floating point numbers, the control surface deflections as floating point numbers, and the A/P Active Status as an integer. All angular outputs for the *Aicraft Simulation Module* are in radians or radians/second. The inputs for the *Visualization Module* are the control surface deflections as double precision numbers in degrees, the roll and yaw Euler angles as double precision numbers in degrees, and the A/P Active Status as an integer. The outputs for the *Visualization Module* are an A/P SW Toggle signal for activating/deactivating the autopilot from the pilot's display.

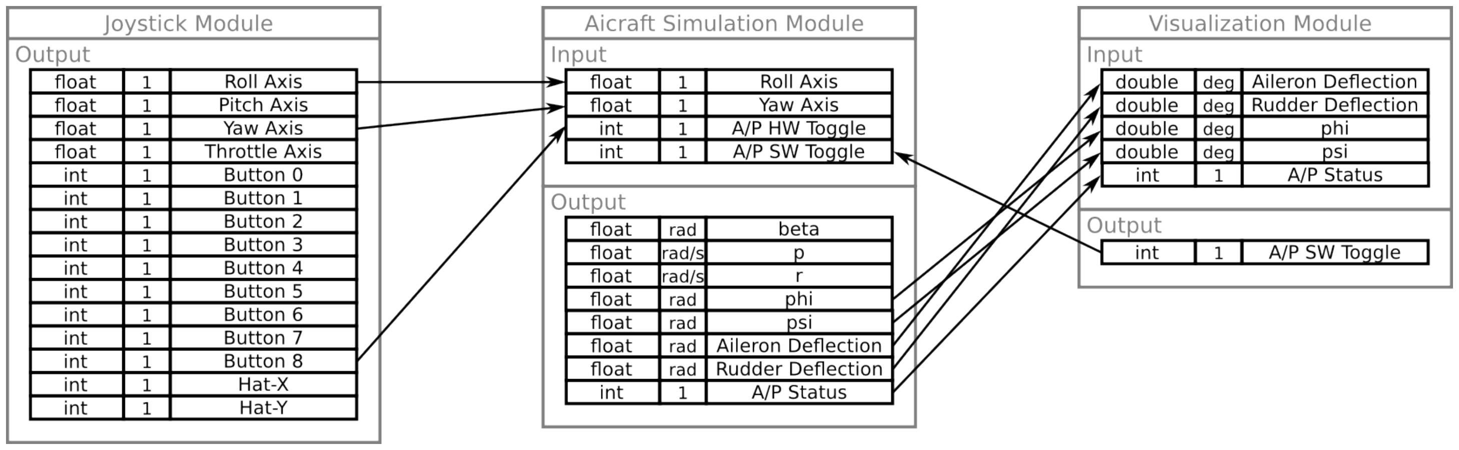


Figure 5.5: Example Module Data Routing

6. EXAMPLE IMPLEMENTATION RESULTS

A successful implementation of Daemons and Modules will meet transmission deadlines allowing for seamless sampled-data distributed simulation performance. For the example setup described in Section 5, a series of data and other architecture parameters were internally sampled from each program during run-time execution on a network in order to test the success criteria. Daemons were used to configure the I/O mapping for the example implementation prior to Module execution.

The success criteria for configuration and information transmission between Daemons are if the I/O between Modules is correctly received, converted, and transmitted. This will verify that the I/O mapping and other I/O CFG information was shared between Daemons correctly. Because each configuration is unique, the implementation human operator must check that data I/O for each Module is correct. The example implementation is designed to allow the human operator to easily and visibly determine if there is an error in I/O transmission.

Section 3.1 describes clock synchronization between Daemons on a simulation network. In this example, *Daemon 1* on Computer 1 is the master clock and *Daemon 2* on Computer 2 is a slave and receives a clock offset from the master. The clock offset between the Daemons/Computers in this example for Computer 2 is that it must add 0.056636 [s] to its own clock for all scheduled events to match with the master clock. Once clocks are synced, all data can be compared between Daemons/Computers according to the time that it was processed and logged on that computer. The time axis on all time history plots in this section are displayed as time elapsed from the initial start execution time for the Modules. The start execution time is prescribed by the human operator and relayed automatically by the Daemons.

It can be seen and compared in Figure 6.1, Figure 6.2, and Figure 6.3 that the Roll Axis, Yaw Axis, and Button 8 outputs from the *Joystick Module* match respectively and directly in units and output with the Roll Axis, Yaw Axis, and A/P HW Toggle inputs from the *Aircraft Simulation Module* as prescribed in the Module Data Routing from Figure 5.5. More unused Joystick output time histories can be found in Appendix A.

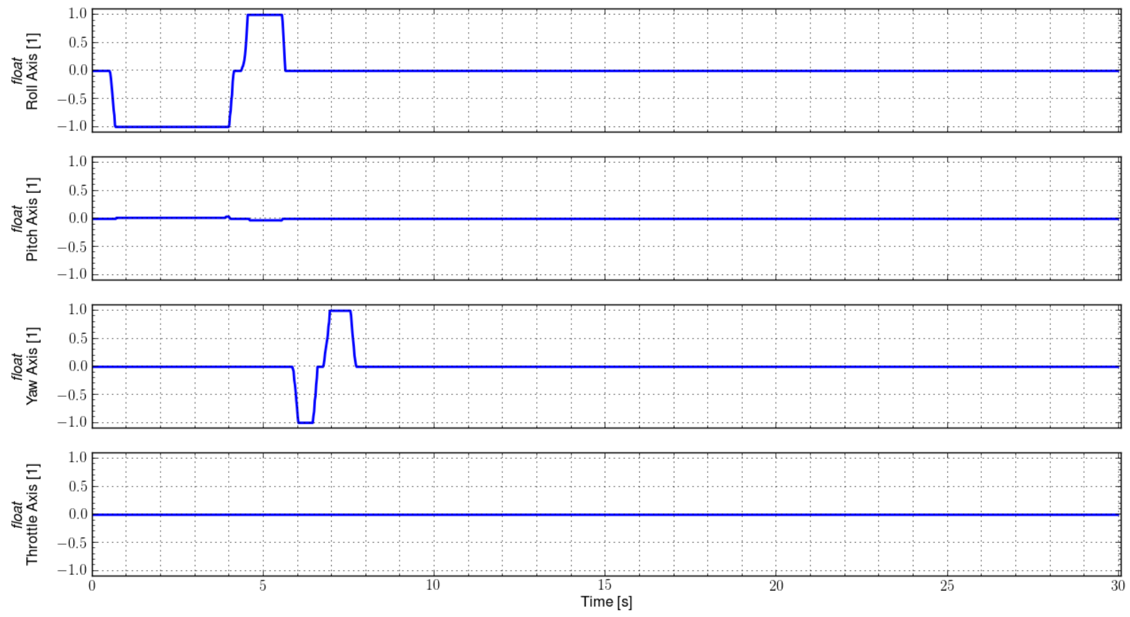


Figure 6.1: Joystick Module Outputs Axes Time History

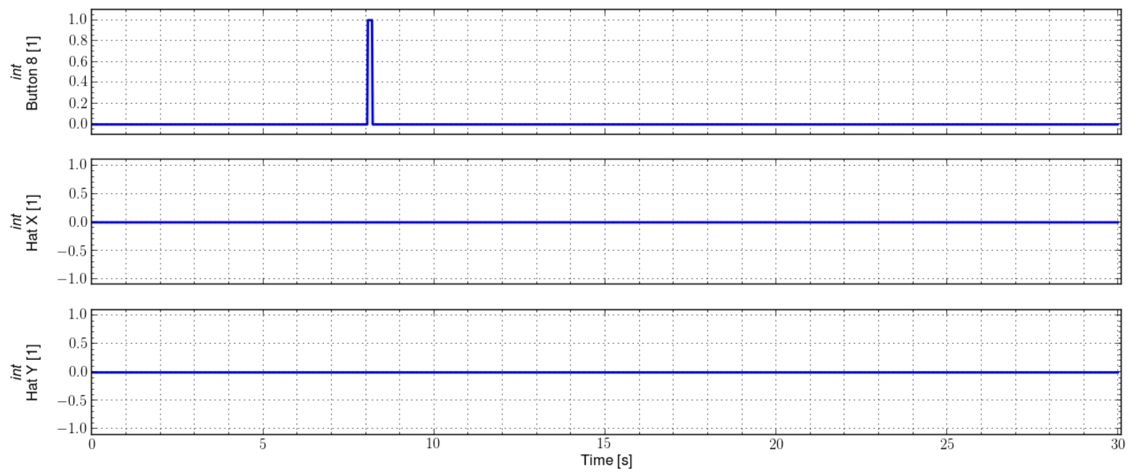


Figure 6.2: Joystick Module Outputs Button 8 and Hat X-Y Time History

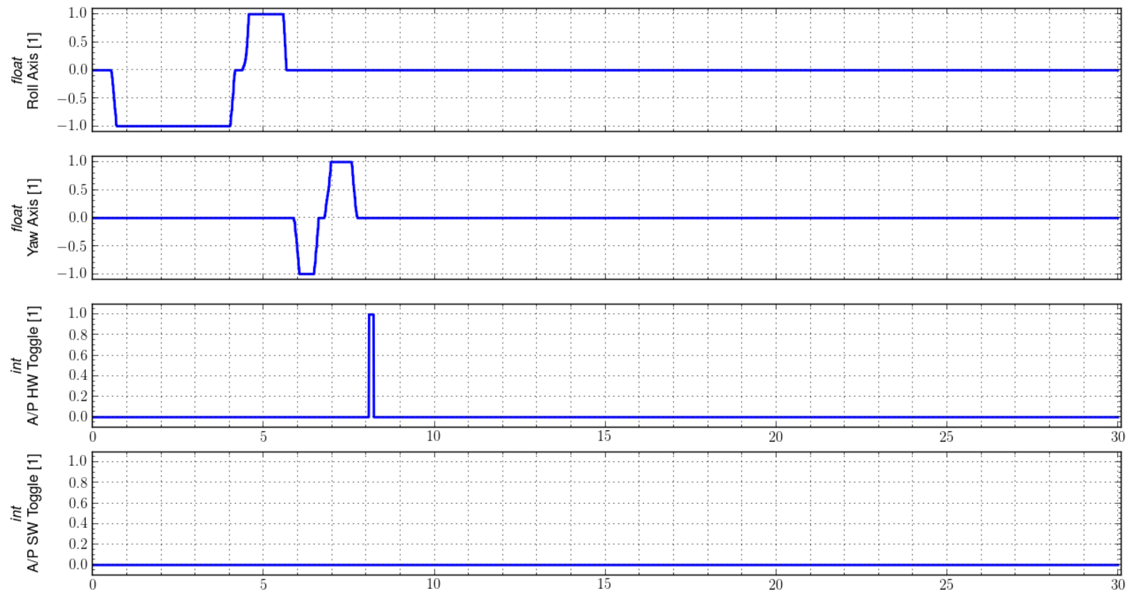


Figure 6.3: Aircraft Module Inputs Time History

It can be seen and compared in Figure 6.4, Figure 6.5, and Figure 6.6 that the Aileron Deflection (δ_A), Rudder Deflection (δ_R), Roll Angle (ϕ), Heading Angle (ψ), and A/P Status outputs from the *Aircraft Simulation Module* match respectively and directly with converted data types and units with the Aileron Deflection (δ_A), Rudder Deflection (δ_R), Roll Angle (ϕ), Heading Angle (ψ), and A/P Status inputs from the *Visualization Module*. Radian outputs are converted to degrees as inputs.

When the autopilot is off, the Roll and Yaw axes from the *Joystick Module* in Figure 6.1 match by a scaling factor with the Aileron Deflection (δ_A) and Rudder Deflection (δ_R) in Figure 6.5 as described in the *Aircraft Simulation Module* Section 5.2. Also, the Autopilot Toggling method described in Section 5.2 can be verified by comparing Figure 6.1 and Figure 6.3 for Toggle Button transmission and Figure 6.5 for A/P Status output.

Figures 6.4-6.6 also verify that the LQR controller of the *Aircraft Simulation Module* described in Section 5.2 is working correctly based on the inputs fed from remote Modules. The Heading Angle, ψ, and Yaw Rate, r, are regulated to 0 when the autopilot is active.

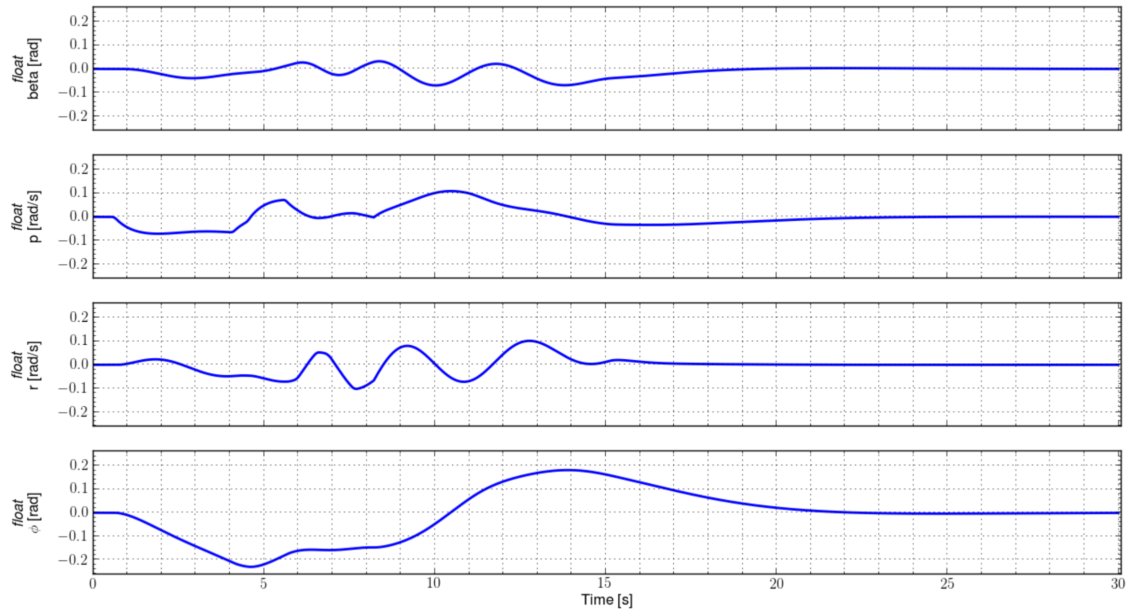


Figure 6.4: Aircraft Module Outputs 0-3 Time History

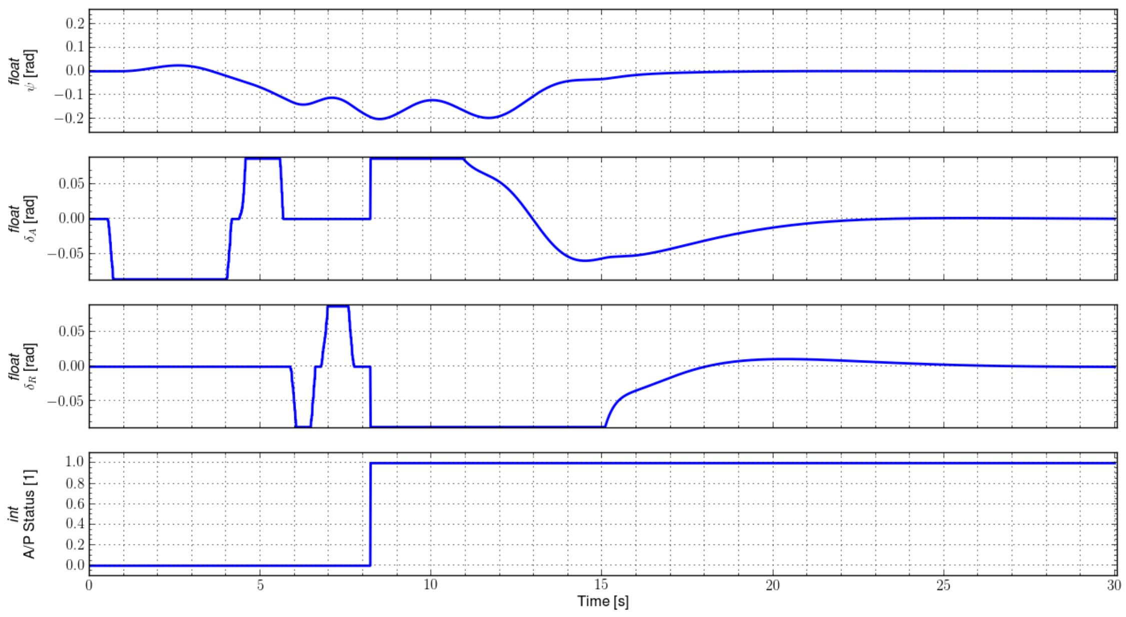


Figure 6.5: Aircraft Module Outputs 4-7 Time History

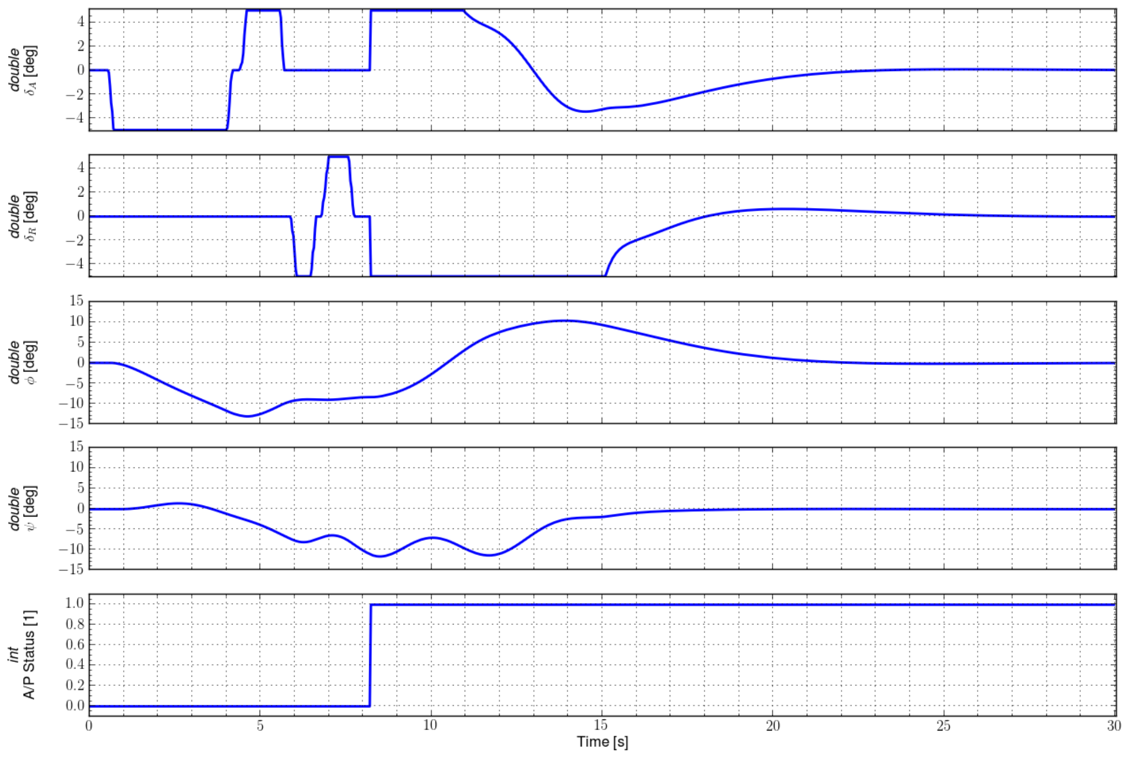


Figure 6.6: Visualization Module Inputs Time History

Because hADES executes Modules in a sampled-data manner, the outputs of one module are not available until the next time that Module is scheduled to loop. This allows each Module to use the entire loop time if necessary to do complex calculations and send and receive large data structures. Figure 6.7 shows when an Autopilot Toggle button press is sent across the network, remote Modules will not receive or react to that data until the next time the *Joystick Module* is scheduled to execute and similarly for transmitting A/P Status information. Figure 6.7 also verifies the Autopilot Toggling method described in Section 5.2. Note that all three modules execute at different rates. In Figure 6.7, blue dots represent when outputs are generated, the red dotted line represents the loop time for that Module, and the red triangle represents when the generated outputs will be available elsewhere on the network.

Sampled-data operation also allows Modules to continue to run even if there are delays or errors in data packet transmission. The fallback action when data is expected to be received, but isn't, is to use the previous timestep data. In the example trial presented in this section, only one packet transmission delay was detected. *Daemon 2* on Computer 2 expected, but did not receive, data from the *Aircraft Simulation Module* on Computer 1 at 29.94 seconds into the 30 second trial. After this packet loss was detected, data was received for the remainder of the simulation trial time.

It is also important to execute distributed simulation implementations using networking hardware able to provide the required data throughput in a timely manner. A throughput test on consumer-grade wireless 802.11g hardware empirically resulted in significant packet transmission delay when sending multicast simulation data. For 100Hz data packet transmission, groups of 4-6 packets were delayed in transit at the networking hardware level and then released at one time to the computers on the net-

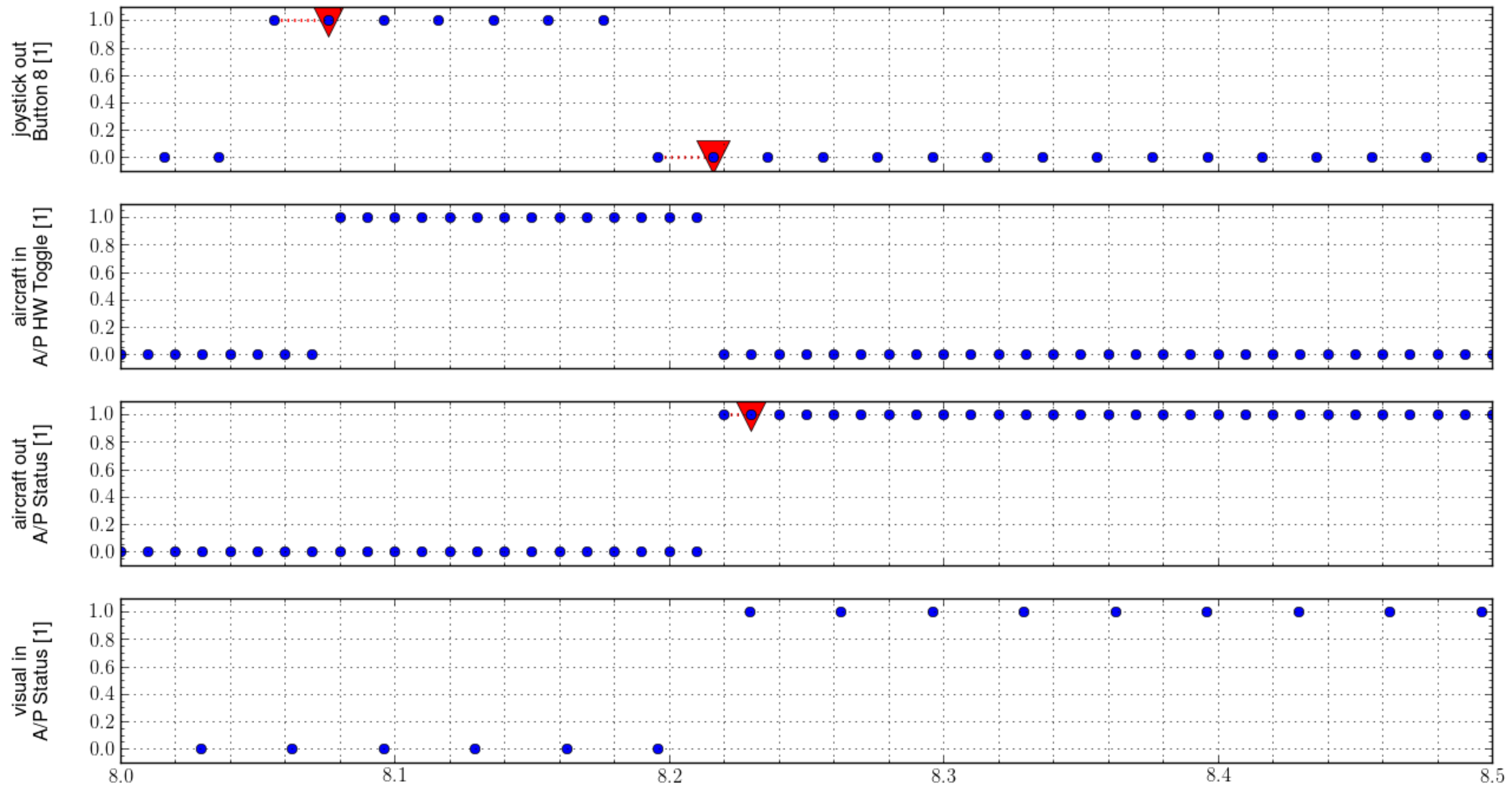


Figure 6.7: Autopilot Toggle Across Network Time History

work where the Daemons on each computer reported receiving the multiple packets during a single scheduled loop. Subsequent trials, including the one presented, were executed on a consumer-grade wired 100Mbit/s Ethernet network which resulted in significantly less to no packet transmission delays. End-user implementation of hADES should be executed on industry or commercial-grade 1Gbit/s Ethernet networking hardware allowing for faster data transmission, more data throughput, and fewer packet delays.

7. CONCLUSIONS

hADES is able to mitigate or remove the complex interdependence between the codebases of distributed simulation programs that communicate with one another. The hADES Daemon does this by modifying data packets in transit between Modules to avoid having to statically compile and/or verify that transmission data structures match between distributed Module I/O mappings. Module API configuration information is shared with the Daemons allowing for a human operator to define the I/O mappings between Modules from anywhere on the simulation network.

The Module API libraries allow Module designers to supply necessary information to the Daemons for extensible configuration and remove the requirement for Module designers to write distributed communication code and logic. The API libraries standardize methods for implementing distributed simulations using the defined library functions.

Performance was evaluated for an example system in Section 6 and determined to be acceptable, comparable, and as good as performance in legacy VSCL architectures. Recorded performance data for the example system in Section 6 shows that time deadlines are almost always being met by hADES. Data packets are being received, converted, and transmitted correctly. For all of the Modules for the entire 30 seconds of run-time, only 1 packet out of 5400 was transmitted late and missed the scheduled deadline giving a transmission/reception success rate of 99.98%.

The example system emulates a use-case scenario and shows that hADES is an acceptable method for connecting distributed simulation in the VSCL and other similar laboratories. The codebases for the different modules were developed independently of the others and the Daemon handled data and unit conversion flawlessly.

7.1 Recommendations

Based on the success of the example implementation, current and legacy Modules can and should be converted for use in a hADES network to allow greater flexibility and to foster future Module development. Performance can be tested on a case-by-case implementation basis for any set of connected, distributed Modules.

The delay time for reading network data, described in Section 3.3, should be empirically determined for a planned implementation network or defined on a case-by-case basis. Future implementations could define a “smart” delay that changes based on network performance, but the logic for that was not explored in this thesis.

The database of unit conversions is included in the Daemon program. This database can and should be expanded to include every set of units to be used a an implementation facility. A secondary extension may be to allow new units and unit conversions to be added using the user interface in Figure 3.1. The command set for adding units and unit conversions will need to be defined in the Daemon as shown in Section 3.2.3.

For public use, documentation for an hADES end-user and Module implementer will need to be created to the level and as would be expected of any commercial or sufficiently advanced open-source project. Upon completion of all requirements for this thesis and related graduate work and finalizing all developer manuals and documents, the hADES codebase will be publicly available with an appropriate free software license at the following url: `https://github.com/jimmayjr/hADES`.

REFERENCES

[1] Ippolito, C. A. and Pritchett, A. R., "Software Architecture for a Reconfigurable Flight Simulator," *Proceedings of the AIAA Modeling and Simulation Technologies Conference and Exhibit*, AIAA, Denver, CO, 14-17 August 2000.

[2] IEEE Computer Society, "IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Framework and Rules," IEEE Std. 1516-2000, The Institute of Electrical and Electronics Engineers, Inc., 3 Park Avenue, New York, NY 10016-5997, USA, 21 September 2000.

[3] Painter, J., Valasek, J., and Ward, D., "Integrating the Evolving Modern Cockpit," *Proceedings of the AIAA Guidance, Navigation, and Control Conference and Exhibit*, AIAA, Montreal, Canada, 6-9 August 2001.

[4] Dartmouth College, Hanover, NH, *BASIC: A Manual for BASIC, The Elementary Algebraic Language Designed For Use With The Dartmouth Time Sharing System*, 1964.

[5] International Business Machines Corporation, 590 Madison Avenue, New York, NY, *The FORTRAN Automatic Coding System for the IBM 704 EDPM: Programmer's Reference Manual*, 15 October 1956.

[6] Wirth, N., *The Programming Language Pascal*, Swiss Federal Institute of Technology, Zurich, Switzerland, 2nd ed., November 1971.

[7] Hawley, P. A., "An Object-Oriented Simulation Architecture," *Proceedings of the AIAA Modeling and Simulation Technologies Conference and Exhibit*, AIAA, Providence, RI, 16-19 August 2004.

[8] Kernighan, B. W. and Ritchie, D. M., *The C Programming Language*, Prentice Hall, Englewood Cliffs, NJ, USA, 2nd ed., April 1988.

[9] Ritchie, D. M., "The Development of the C Language," *Proceedings of ACM History of Programming Languages II*, Cambridge, MA, 20-23 April 1993.

[10] International Business Machines Corporation, 590 Madison Avenue, New York, NY, *FORTRAN II for the IBM 704 Data Processing System : Reference Manual*, 1958.

[11] Stroustrup, B., *The C++ Programming Language - Reference manual*, AT&T Bell Laboratories, Murray Hill, NJ 07974, November 1984.

[12] Gosling, J. and McGilton, H., *The Java^{TM}Language Environment: A White Paper*, Sun Microsystems, Mountain View, CA, May 1996.

[13] Python Software Foundation, "About Python," 14 November 2012, http://www.python.org/about/.

[14] Doebbler, J., Rong, J., Ding, Y., Spaeth, T., and Valasek, J., "Design and Implementation of a Distributed Multi-Pilot Engineering Flight Simulation Facility," *Proceedings of the AIAA Modeling and Simulation Technologies Conference and Exhibit*, AIAA, San Francisco, CA, 16 August 2005.

[15] Moore, G. E., "Cramming more components onto integrated circuits," *Electronics*, Vol. 38, No. 8, 19 April 1965, Archived at `http://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf`, accessed on 14 November 2012.

[16] Simon, W. G., "Transistor Count and Moore's Law - 2011," Web resource: `http://en.wikipedia.fcorg/wiki/File:Transistor_Count_and_Moore%27s_Law_-_2011.svg`, accessed on 13 October 2012.

[17] Kurzweil, R., "Moore's Law, The Fifth Paradigm." Web Resource: `http://en.wikipedia.org/wiki/File:PPTMooresLawai.jpg`, accessed on 5 July 2005, Released for use by Kurzweil Technologies, Inc. `http://www.kurzweilai.net`.

[18] Rong, J., Spaeth, T., and Valasek, J., "Small Aircraft Pilot Assistant: Onboard Decision Support System for SATS Aircraft," *Proceedings of the AIAA 5th ATIO and 16th Lighter-than-Air and Balloon Systems Conferences*, AIAA, Arlington, VA, 26-28 September 2005.

[19] Rong, J., Ding, Y., Valasek, J., and Painter, J. H., "Intelligent System Design with Fixed-Base Simulation Validation for General Aviation," *Proceedings of the 2003 IEEE International Symposium on Intelligent Control*, IEEE, Houston, TX, 5-8 October 2003.

[20] Ding, Y. and Valasek, J., "Feasibility Analysis of Aircraft Landing Scheduling for Non-Controlled Airports," *Journal of Guidance, Control, and Dynamics*, Vol. 30, No. 1, January - February 2007, pp. 252–255.

[21] Rong, J. and Valasek, J., "Onboard Pilot Decision Aid for High Volume Operation (HVO) in Self-Controlled Airspace (SCA)," *Proceedings of the 23rd Digital Avionics and Systems Engineering Conference*, AIAA, Salt Lake City, UT, 24-18 October 2004.

[22] Doebbler, J., Gesting, P., and Valasek, J., "Real-Time Path Planning and Terrain Obstacle Avoidance for Aircraft," *Proceedings of the AIAA Guidance, Nav-*

igation, and Control Conference, AIAA, San Francisco, CA, USA, 5-18 August 2005, pp. AIAA–2005–5825.

[23] May Jr., J. and Valasek, J., "Extensible Software Architecture for a Distributed Engineering Simulation Facility," *Proceedings of the AIAA Modeling and Simulation Technologies Conference and Exhibit*, AIAA, Toronto, Ontario, Canada, 2 August 2010, pp. AIAA–2010–8102.

[24] Ward, D. T., Alcorn, W. P., Hull, J., Miller, A., Robbins, A. C., Shandy, S. U., and Yu, R., "EFS System Description, Version 1.1," Technical report, Texas A&M University Department of Aerospace Engineering, College Station, TX 77843-3141, November 30, 1998.

[25] Roza, M. and van Gool, P., "Simulating Free Flight in HLA Federations," *Proceedings of the AIAA Modeling and Simulation Technologies Conference and Exhibit*, AIAA, Denver, CO, 14-17 August 2000.

[26] IEEE Computer Society, "IEEE Standard for Local and Metropolitan Area Networks," IEEE Std. 802-2001, The Institute of Electrical and Electronics Engineers, Inc., 3 Park Avenue, New York, NY 10016-5997, USA, 7 February 2001.

[27] Braden, R., *RFC 1122 Requirements for Internet Hosts - Communication Layers*, Internet Engineering Task Force, October 1989, Web resource: `http://tools.ietf.org/html/rfc1122`, accessed on 11 October 2012.

[28] Waitzman, D., *RFC 1149 A Standard for the Transmission of IP Datagrams on Avian Carriers*, BBN STC, 1 April 1990, Web resource: `http://tools.ietf.org/html/rfc1149`, accessed on 10 October 2012.

[29] IEEE Computer Society, "IEEE Standard for Information Technology - Protocols for Distributed Interactive Simulations Applications. Entity Information and Interaction," IEEE Std. 1278-1993, The Institute of Electrical and Electronics Engineers, Inc., 3 Park Avenue, New York, NY 10016-5997, USA, 22 March 1993.

[30] May, J., Doebbler, J., and Valasek, J., "Simulation Architecture Development of a Distributed Multi-Pilot Engineering Flight Simulation Facility," *Proceedings of the AIAA Modeling and Simulation Technologies Conference and Exhibit*, AIAA, Honolulu, HI, 18-21 August 2008.

[31] Gusella, R. and Zatti, S., "The Accuracy of the Clock Synchronization Achieved by TEMPO in Berkeley UNIX 4.3BSD," *IEEE Transactions on Software Engineering*, Vol. 15, No. 6, July 1989, pp. 847–853.

[32] Gralla, P., *How the Internet Works, Chap. 9*, Que Publishing, Indianapolis, IN, September 1998, p. 41.

[33] Rao, S., Vin, H., and Tarafdar, A., "Comparative Evaluation of Server-Push and Client-Pull Architectures for Multimedia Servers," *Proceedings of NOSSDAV'96*, Zushi, Japan, 23-26 April 1996, pp. 45–48.

[34] Cotton, M. et al., *RFC 6335 Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry*, Internet Engineering Task Force (IETF), August 2011, Web resource: `https://tools.ietf.org/html/rfc6335`, accessed on 11 October 2012.

[35] Rivest, R., *RFC 1321 The MD5 Message-Digest Algorithm*, Massachusetts Institute of Technology Laboratory for Computer Science and RSA Data Security,

Inc., April 1992, Web resource: `http://tools.ietf.org/html/rfc1321`, accessed on 10 October 2012.

[36] Cerf, V., *RFC 20 ASCII Format For Network Interchange*, University of California, Los Angeles, October 1969.

[37] Valasek, J., *Linear Aircraft Models - Block 5-2 - Aero 625*, Texas A&M University, College Station, TX, December 2012.

APPENDIX A

ADDITIONAL RESULTS

This appendix includes unused Joystick output time histories from the example simulation in Section 6 to provide a complete set of time histories combined with the other figures in the section.

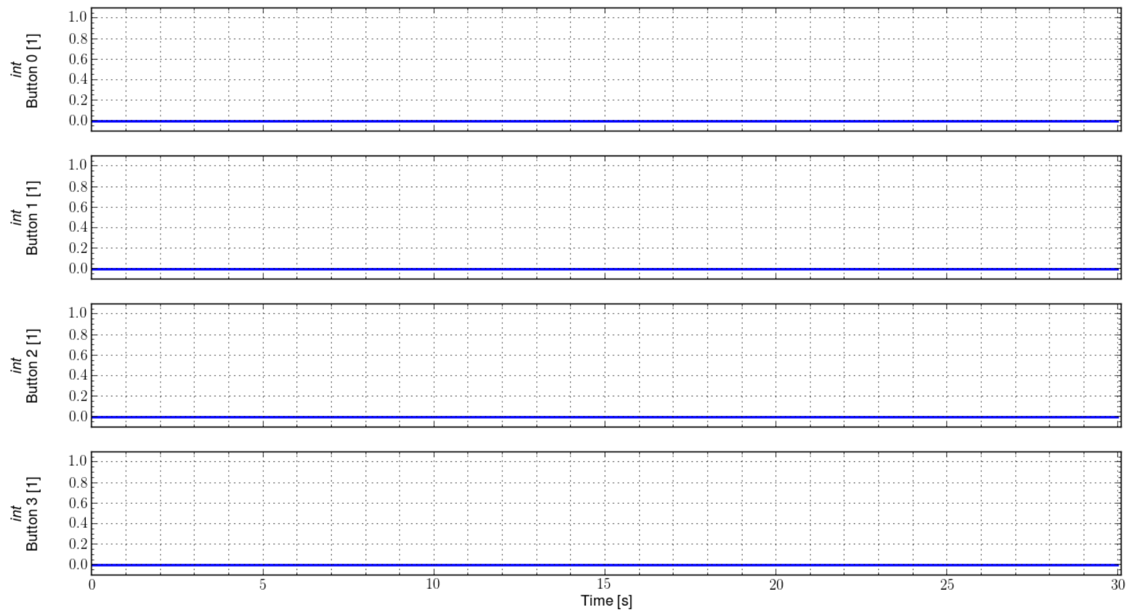


Figure A.1: Joystick Module Outputs Buttons 0-3 Time History

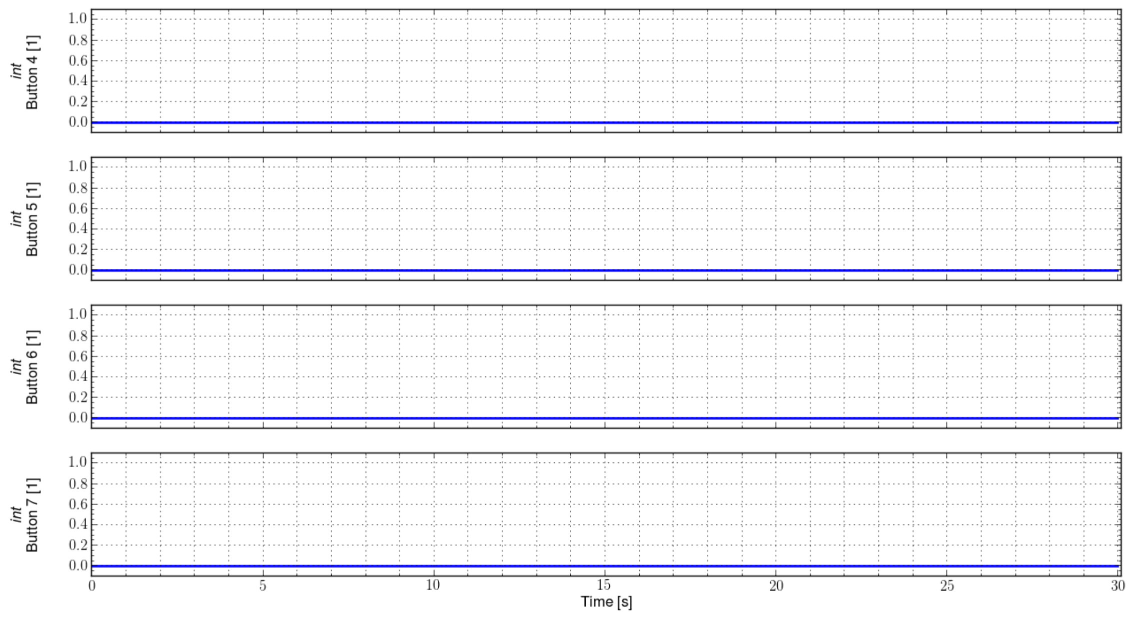


Figure A.2: Joystick Module Outputs Buttons 4-7 Time History