Extensible Software Architecture for a Distributed Engineering Simulation Facility

Jim May Jr.* and John Valasek†
Texas A&M University, College Station, Texas, 77843-3141, U.S.A.

This paper discusses the design and implementation of a software architecture for an Ethernet and wireless IEEE 802.11 network-based distributed simulation and experiment facility. 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 codes have extended the capabilities of educational research facilities and opened the way for the development of the architecture presented in this paper. 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.

Nomenclature

- API Application Programming Interface
- COTS Commercial Off-The-Shelf
- CPU Central Processing Unit
- DIS Distributed Interactive Simulation
- EFS Engineering Flight Simulator
- GUI Graphical User Interface
- HDD Head-Down Display
- HLA High Level Architecture
- I/O Input/Output
- LCD Liquid Crystal Display
- OOP Object-Oriented Programming
- PC Personal Computer
- RTI Run-Time Infrastructure
- SGI Silicon Graphics Incorporated
- TCP Transmission Control Protocol
- UDP User Datagram Protocol
- VSCL Vehicle Systems and Control Laboratory

I. Introduction

Simulation has become a major tool in engineering research, design, testing, and education. However, the time and resources required to develop new simulations can sometimes be large and require personnel with skills in many areas outside their major field of study.¹ The spread 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 are highly proprietary and unique to the problems being studied as they are developed by private companies and engineering firms. A result of simulation architectures being unique to each system is the lack of a simple and common method of communication between modules and for minimizing interdependence between the codebases of networked modules. Without high-level multi-platform communication codes, practical application of network distributed simulation systems, especially in an academic environment, remains near the the level of bits and bytes and requires a development team made of of students from Aerospace Engineering, Electrical Engineering, Computer Science, and other related disciplines.²

---

*Graduate Research Assistant, Vehicle Systems and Control Laboratory, Department of Aerospace Engineering, Student Member AIAA, jim.may.jr@gmail.com.
†Associate Professor and Director, Vehicle Systems and Control Laboratory, Department of Aerospace Engineering, Associate Fellow AIAA, valasek@tamu.edu.
This paper discusses the design of a simulation architecture to provide a simple and common method of communication between distributed simulation modules. The main feature developed in this architecture is a communication backbone program called the simulation Daemon. A second feature developed is a set of application programming interface libraries for allowing simulation software modules to communicate with one another through the simulation Daemon. A broad third feature are the set of individual software modules themselves and the methods for their integration into a distributed system. The simulation Daemon and API libraries are designed specifically to meet the educational and research requirements of the VSCL. In addition to meeting VSCL requirements, the API libraries are designed to minimize integration efforts of new and legacy simulation modules. The evolution of engineering simulation has led to the simulation software modules seen today with both limitations and advantages to be addressed.

I.A. Engineering Simulation Evolution

The original simulation technologies were developed using hardware and software that were extremely primitive by today’s standards. Several major advancements in simulation technology have led to the hardware and software architectures seen today.

I.A.1. Software

Early simulation codes were restricted to the monolithic structures of procedural architectures and primitive programming languages, such as the original versions of BASIC, Fortran, and Pascal. 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. Variables in procedural architectures have no protection and much care is needed to ensure the integrity of the 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. Some initially procedural programming languages, such as C and Fortran, had function and subroutine capabilities upgraded. Scalability is an issue with modular architectures when moving from simulations with one body or system to simulations hosting many bodies or systems because the data for each is not encapsulated. Data variables 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 if not already in an array. If data variables are already in an array, they will then 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 like functions and data can be encapsulated into objects using object-oriented programming languages such as C++, Java, and Python. 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 is a base aircraft class which has position, velocity, acceleration, mass, inertia, force, and moment variables. A fighter aircraft class may be derived from the base aircraft class adding 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 will 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.

I.A.2. Hardware

The advent of PCs expanded engineering simulation use as well. Past high-fidelity simulations were only achieved using high-end and proprietary hardware. While powerful, high-end hardware traditionally has very high purchase, support, and maintenance costs. Simulations that once required proprietary, specialized, and expensive computer mainframe hardware are now able to be executed using common and cheaply procured personal computers with a much wider support base. The ability for simulations to be executed on PCs derived from major increases in computing power, performance, and availability of PCs.
Simulations began to exhaust the resources of a single PC, multiple network-distributed computers were used for single simulations. The expansion to multiple PCs was due to simulation of large numbers of complex simulated agents, considerably complex and high-fidelity environments and vehicles, and other features and capabilities expanded upon from and by previous generations of architectures. Distributed simulations increase scalability, flexibility, and reconfigurability of a simulation environment promoting exchange, reuse, and inter-operation between multiple simulation components. Distributed simulations in the architecture presented in this paper are simulations spread across computers strictly on commonly used Ethernet and wireless IEEE 802.11 networks.

II. Vehicle Systems and Control Laboratory

The Vehicle Systems and Control Laboratory of the Department of Aerospace Engineering at Texas A&M University is an aerospace vehicle research, simulation, and education facility. The VSCL has a series of distributed networked PCs for simulation. These PCs are used for manned vehicle simulations, unmanned vehicle simulations, machine learning and control, cockpit displays, and many other aerospace-related uses. 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 display and systems class designs and tests displays in the VSCL. Aircraft control students 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 the actual test aircraft. Many of these classroom activities use the same software modules but may require a few more or a few less sets of data from the simulation system than others.

The initial hardware architecture of the VSCL in 1998 (then called the Flight Simulation Laboratory) included only a single engineering flight simulator. The main EFS used a SGI Onyx Reality 2 computer for scenery generation and six degrees of freedom aircraft model dynamics calculations in a single compiled binary executable using a mixture of procedural and modular architecture paradigms. The SGI computer had the aforementioned high purchase, support, and maintenance costs of high-end simulation hardware. The single SGI computer also created a single point of failure in the ability to run simulations for the laboratory. The EFS hardware was 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. The glass display interfaces were generated using Windows 98 PCs. 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.

The next upgrade to the VSCL in 2005 distributed the main EFS computing to multiple PCs, added three PC-based pilot stations, upgraded the Windows operating systems to Windows XP, as well as installing touch-screen LCDs to replace the cathode ray tube displays and input buttons of the previous HDDs. 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 consist 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 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.

Figure 1 and Figure 2 represent the current user-hardware interface of the EFS and pilot simulation stations respectively. Every simulation computer serves a double purpose: being used for manned and unmanned real-time flight simulation or simulation development and being used as a single or distributed computation machine for any other simulation or calculation required by the VSCL’s various educational and research activities. The VSCL simulation network architecture is shown in Figure 3.
Figure 1: VSCL EFS c. 2009

Figure 2: VSCL Pilot Station Simulators c. 2009

Figure 3: VSCL Simulation Network Architecture
III. Current Architectures

III.A. VSCL Architecture

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. An architecture that is flexible to changes in I/O data packet formats is needed to avoid recompilation of source code of connected simulation modules and manual analysis and confirmation that all I/O data packet formats match upon modification of one simulation module in a distributed system.

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

Another factor in the design of a new simulation architecture is that a majority of the people working on simulations in the VSCL are Aerospace Engineering students with little previous programming experience. Minimizing the amount of time these students currently require to configure and code network communication is essential.

III.B. Other Distributed Architectures

Other distributed simulation architectures, such as the High Level Architecture, already exist. The main component of the HLA is the Run-Time Infrastructure. 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 too time consuming due to the number of specifications required in relation to the time constraints and limited number of students available in the VSCL. A second consideration for not choosing the HLA is requiring students to write a federate program to the HLA specifications. As mentioned before, the limited programming experience and time constraints of students will severely prohibit laboratory 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 VSCL simulations.

Distributed Interactive Simulation is another open standard for real-time distributed simulations. This standard is designed for large scale wargames. Although a detailed description of the DIS is outside the scope of this paper, reasons for not choosing and building an architecture according to this standard closely mirror those for the HLA.

IV. Extensible Architecture

The proposed software architecture features three distinct functional components. The first component of the proposed architecture is a simulation Daemon program that runs on each simulation computer to handle routing and distribution of simulation data. This Daemon is to be implemented using the Python programming language. Though a scripting language which has deficiencies when handling massive amounts of CPU-bound processes, Python is very capable of handling the amount of I/O-bound processes the simulation Daemon will require. The simple and ordered syntax of Python will decrease the learning curve of using and maintaining the system by future students and researchers in the VSCL. The second component of the architecture is an application programming interface to allow simulation modules to connect to and communicate with other modules through the daemon. The third component is the set of distributed simulation modules to be integrated into the architecture.

IV.A. Daemon

The simulation Daemon handles all intermodule communication. It 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 other Daemons on the network. The simulation Daemon is a parallel to the Run-Time Infrastructure of the High Level Architecture. 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.

On each simulation computer, the Daemon can create OOP objects for each simulation module. Module objects are always created for modules local to that machine and also for any remote modules while data is required from them. The module objects contain methods for communication with the module, information about the API being used by the module, and data objects for each data member to be received from and sent to the software modules. Where previous distributed architectures were plagued by hard-coded data packet formats between directly connected software modules, this Daemon 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 to create constant faux data when no input is available. Modified data packet formats consist of new data added to a packet, data removed from packet, changing data type, changing data units, or combinations of the previous. Network data packets are filled with actual or type converted variable data from any of the data objects. An example of a module object containing data objects and some of the available member functions is shown in Figure 4.

![Figure 4: Example of objects within the simulation Daemon](image)

Simulation modules can communicate with the local Daemon over TCP and UDP sockets. Simulation data between Daemons is also sent and received using TCP and UDP sockets. To prevent a module or Daemon from receiving erroneous data, distribution of ports will be handled by a port server. Daemons will request ports to use when a new module connects to the Daemon. The port server will give a Daemon ports for communication with the local module and a port for network distribution of module data. When a module is no longer used, the Daemon will let the port server know that the ports are no longer in use and can be distributed to other Daemons. The Internet Protocol Suite has ports 49152–65535 reserved as dynamically available ports meant to be used on an as-needed basis by clients and servers on a network. These ports are to be distributed by the port server. Also, 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 unique multicast addresses.

A method for configuring the data paths between software modules is needed for a flexible system. A human user oversees simulation configuration and will need a method for defining configurations as well as visual confirmation that the configuration is correct. There are many entry points into a distributed simulation, many of which are capable of handling system configuration. Configuration of data paths between modules will be handled by the Daemon through a graphical user interface. The GUI is a human-readable abstraction of TCP and UDP communication between Daemons on the network. The interface will be able to display module data values, types, names, and units from all Daemons on the network. The GUI will also be able to send commands from one Daemon to another in order to create module objects and send start and stop commands to modules on remote computers.

Communication of simulation data from the Daemon and sampling of data into the Daemon is executed in the Daemon event handler. The purpose of the event handler is to create soft real-time performance of simulation data transmission at a desired rate. The creation of a module object in the Daemon also adds an event object to the Daemon event handler. An event object contains a list of the data objects to be sent either to the network or to a local module. The event object also contains a list of the data objects values
that are to be updated. Also included in the event object are the addresses and ports to send and receive data to and from, allowing for an instance of a generic event class to communicate with both local modules or remote Daemons. The event handler has a single event loop. Once per event loop, the event handler checks to see if an event is due to execute. If an event is due to execute, the event handler sends, receives, and updates data in data objects. If the event is not due to execute the event handler will simply pass to the next event in the list.

**IV.B. Application Programming Interface**

The simulation Daemon API is a set of programming libraries used by simulation modules for registering input and output data packet formats with the Daemon, communicating with the Daemon, and allowing for executive control of module main loop execution. Communication methods using the simulation Daemon API standardize intermodule communication. 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 and UDP protocols. The Daemon receives the the expected I/O data packet formats for each module using the API before a simulation begins, allowing the Daemon to be able to handle potential differences in I/O data packet formats between software modules. Configuration and executive control of simulation modules will be handled through TCP socket communication with the Daemon. An example of the process of a new simulation module connecting to the Daemon and executing is shown in Figure 5.

![Diagram](image)

**Figure 5: New module configuration process**

**IV.C. Modules**

Simulation modules are the programs that execute the actual dynamical simulations. 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 data names. The modules also have a desired rate for the main loop to receive data, execute one step of the simulation, and send data. This set information is used by the Daemon API 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 Daemon API.

Some modules may be developed from COTS software. Integrating COTS software into a simulation system has many advantages and disadvantages. Some advantages are that simulation developers can put emphasis on how software works rather than how it was put together, it can significantly reduce development costs, and software vendors maintain and add extra capability to the software. Some drawbacks are that reoccurring costs can be high, a custom interface “wrapper” is required for COTS software to communicate with other modules, and in many cases, the actual code implementation of the COTS software is completely unknown.

V. Conclusion

The advancement of programming paradigms and increases in computing hardware power have enabled large-scale, complex distributed simulations; an architecture capable of handling them is needed. Preliminary analysis of the efforts required to implement existing distributed architectures deemed the development time to be too great. From this point, the needs of the VSCL were reevaluated and a new system was developed for configuring and sharing distributed simulation information.

Where previous upgrades to the VSCL software architecture allowed modularity and distribution of simulations, the architecture developed in this paper mitigates previous restrictions between distributed modules, eases and standardizes the methods of creating and integrating new simulation modules, and promotes module development through the use of simulation Daemon API libraries. Modularity is emphasized while fostering reconfigurability of and compatibility between software modules. Innovative use of object-oriented programming capabilities allows for this type of flexibility. Deconstruction of data packets into the data objects inside the Daemon allows for speedy and extensive reconfiguration of simulation modules while decoupling the input and output data packet formats between separate simulation modules.

Each simulation configuration is unique and performance must be individually evaluated. The simulation Daemon acts as an intermediary between software modules and simulation computers. Latency due to processing overhead and network transfer overhead is added each time data passes through the Daemon. Achieving all simulation configurations and module loop rates will not be possible; however, these problems are inherent to any distributed architecture.

V.A. Future Work

Many independent multi-system simulations may be running on the architecture network. The simulations may be faster than real-time batch simulations, slower than real-time batch simulations, or true real-time simulations. Where and how a centralized timing mechanism is located in the architecture, and if one is actually needed, will need to be evaluated for each simulation configuration. Another issue in developing this system for the VSCL is the execution of real-time simulation on non-real-time computing systems. A soft real-time approach must be used where a simulation task is attempted to be executed at a specific rate, but may be interrupted by the operating system to allow higher priority tasks to be completed. If delays are small or infrequent, the user will not perceive any performance degradation. However, each simulation module, operating system environment, and computer system combination is unique and soft real-time performance must be independently examined.

References


