DEVELOPING SCADA SIMULATIONS WITH C2WINDTUNNEL

By

Andrew Davis

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Computer Science

May, 2011

Nashville, Tennessee

Approved:

Professor Gabor Karsai

Professor Yuan Xue

ACKNOWLEDGEMENT

This work was supported in part by TRUST (Team for Research in Ubiquitous Secure Technology), which receives support from the National Science Foundation (NSF award number CCF-0424422) and the following organizations: AFOSR (#FA9550-06-1-0244), BT, Cisco, DoCoMo USA Labs, EADS, ESCHER, HP, IBM, iCAST, Intel, Microsoft, ORNL, Pirelli, Qualcomm, Sun, Symantec, TCS, Telecom Italia and United Technologies.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENT ii

LIST OF FIGURES iv

Chapter

I. INTRODUCTION 1
SCADA Systems 1
SCADA Security 2
Security Threats 3
Simulating SCADA 5

II. BACKGROUND 7
SCADA Systems 7
Architecture 8
C2WindTunnel 12
GME 12
Matlab/Simulink 13
OMNeT++ 14
Portico 14
The High Level Architecture 15
History and Motivation 15
System Architecture 16
Data Types 17
Data Flow 18
Publishing Data 19
Subscribing to Data 20
Time Management 21

III. IMPLEMENTATION 25
OMNeT++ Integration 25
Time Management Policy 25
Interaction Management 33
Network Routing 36
GME Additions 40
Windows Support 42

IV. CASE STUDY 45

V. CONCLUSION 52
Future Work 52

LIST OF FIGURES

Figure		Page
1	Typical SCADA architecture	9
2	Field device control loop	10
3	HLA architecture	17
4	Time advancement state diagram	23
5	Event-driven federate scheduler code	26
6	Time-stepped federate scheduler code	28
7	OMNeT++ scheduler-module barrier workaround	30
8	Time-stepped module-based federate code	32
9	Data flow for a simulated network packet	33
10	GME network interface	37
11	GME network interface with multicast	38
12	Ambiguous GME network model	38
13	Ambiguous GME network model workaround	39
14	Changes to HLA metamodel	41
15	Modeling interactions in GME	46
16	Modeling federates in GME	47
17	Modeling federation deployment in GME	48
18	Plant model in Simulink	49
19	Controller model in Simulink	49
20	Liquid level setpoint	51

CHAPTER I

INTRODUCTION

Industrial control systems form the backbone of countless industries affecting nearly every basic service modern society requires. These large networked computer systems are used to manage the production and distribution of electric power, the treatment and disposal of sewage, and the production of food and pharmaceuticals, among countless other vital tasks [12]. Their prominent and increasing importance in modern life makes them and important asset whose safety and security must be protected. Unfortunately, securing these systems is a complex and difficult task and one to which too little attention has been paid in the past. In light of the growing prevalence of cyber attacks on the computer networks and systems in the infrastructure of major nations, an understanding of the vulnerabilities of industrial control systems and an investigation of appropriate and effective mitigation techniques is of vital importance.

SCADA Systems

The computer systems used to monitor and control major infrastructure are known by various names, among the most common Supervisory Control and Data Acquisition (SCADA) systems. The system's name reflects its basic functions: it must provide data related to the operating state of the system and allow operators to remotely control the distributed system. By utilizing these services, system operators can effectively respond to changes in the process operating conditions or adapt to evolving production goals or changing corporate directives.

The rising prevalence of SCADA systems in infrastructure is a result of the variety of benefits such systems can provide to the businesses that operate them. By shifting away from purpose-built hardware towards more flexible full-featured hardware running operation-specific software, development of control systems can be accom-

plished more quickly and at lower cost. Specialized interfaces can be designed for system operators that minimize the difficulty of management and allow rapid and effective reactions to changing process conditions. The use of SCADA systems allows high-level management of the industrial process by merging data from the many distributed portions of the process. This can help enhance the robustness and reliability of the system. Finally, flaws in the design of the control system can be more easily addressed and operators may receive maintenance and support from vendors of SCADA software and hardware. Taken as a whole, these benefits provide a powerful incentive to migrate to SCADA solutions for control of complex distributed processes [17].

SCADA Security

In past control systems, security concerns were considered to be of less importance and commanded less attention and investment of resources than safety concerns. The networks employed to interconnect the many field devices required for process operation were typically well isolated from any outside interference. Connections to corporate networks or other external networks were not prevalent [12], which minimized the risk of outside interference in system operation. Intrusion could only have been accomplished with physical access, so physical security measures were sufficient to repel computer-based attacks. As these systems began to change and became more interconnected, however, operators began to recognize that cyber security was a real concern that must be addressed in order to maintain the safety and reliability of process control.

A number of changes in modern SCADA systems drive the increasing interest and investment in techniques to manage cyber security threats. The shift from purpose-built hardware to more flexible off-the-shelf processors running embedded operating systems and operation-specific software is a boon for flexible and cost-effective de-

sign and deployment of control systems. However, this flexibility results in enhanced capabilities available to attackers who can easily gain access to the same hardware and software used by the control system. An increase in complexity also exposes modern SCADA systems to a larger number of implementation flaws which may be be exploited by attackers to gain control over the system. Increasingly, SCADA systems are connected to corporate or other external networks. This allows the business to operate more efficiently and remain competitive by enabling business leaders to track and control production in real time and react quickly to evolving production goals and changing market conditions. However, such increased connectivity exposes control systems to an enlarged attack surface. Penetrating the control network could be accomplished without physical access to the system by remote attackers exploiting vulnerabilities in the gateway between the corporate and control networks. Finally, the increasing use of commercial off-the-shelf (COTS) and open-source software decreases development and deployment costs, but means that attackers need to acquire less insider knowledge of the operation of the control system than was required when proprietary hardware and protocols were more prevalent [14]. The rapidly evolving landscape of SCADA systems warrants an increased understanding of and focus on their protection from cyber attacks.

Security Threats

Additional investment in SCADA security is further justified by the existence and rising incidence of network attacks on control systems. The British Columbia Institute of Technology began collecting data related to security attacks on industrial control systems in the mid-1990's. The Industrial Security Incident database was able to capture a marked increase in cyber attacks in the decade it collected incident reports [13]. Two well-known attacks on SCADA systems further illustrate the need for security investment: the Maroochy water breach and the Stuxnet worm.

The operators of the Maroochy water services system in Queensland, Australia began noticing issues with the operation of their wastewater pumping stations in March 2000. These stations were behaving erratically, were not responding to the control signals of operators, and were failing to issue appropriate alarm signals. Operators were initially unable to determine the source of the disturbance. It was not until the results of three months of monitoring the communications and behavior of the pumping station that the cause was determined to be a cyber attack. During that time, a disgruntled former employee was able to release one million liters of untreated sewage into the environment [26]. The difficulty operators experienced in detecting and diagnosing this attack is a compelling example of the damage that a can be caused by a knowledgeable insider acting maliciously.

The Stuxnet worm provides a more recent example of why protecting critical infrastructure is an important part of national security. This complex malware was first identified in June 2010 and was likely targeted at SCADA systems in Iran [18]. Stuxnet was able to exploit four zero-day vulnerabilities in Windows, allowing it to infect field devices controlling centrifuges in a Natanz nuclear facility. The malware was then able to maliciously vary centrifuge speeds to force them outside normal operating conditions and sabotage the system [19]. This attack is significant because it illustrated the capabilities of cyber attacks on critical infrastructure and highlights their potential use in cyber warfare.

Although investment in cyber security for industrial control systems is well justified, it is not immediately obvious that SCADA security represents a new challenge that has not previously been addressed by traditional network security research and practice. SCADA security differs from traditional network security in a number of important ways, however. Unlike most corporate networks, SCADA systems interact with the physical environment and control systems that are critical to the safety and productivity of the community they serve. Malfunctions and loss of availability of

these systems can cause massive economic damage or even loss of life. This interaction with the physical environment also provides opportunities to researchers in SCADA security, however. Because the physical environment responds to the control system in predictable ways, computer models of the physical process can be used to help detect network attacks. SCADA systems usually require very high availability, making security patches a significant challenge rather than the minor inconvenience they represent to the typical corporate network. In addition, SCADA systems are often deployed with decades-long expected lifetimes meaning securing legacy hardware from modern cyber attacks is an unavoidable part of SCADA security. Finally, SCADA systems typically employ a stable and predictable network topology and communication pattern, meaning intrusion detection systems designed with SCADA topologies in mind have the potential to be particularly effective [14]. Because SCADA security differs from traditional network security so widely, with a variety of additional challenges and new opportunities, it cannot be approached solely from the perspective of currently available network security research, but must be investigated as its own research area.

Simulating SCADA

The differences in SCADA security and traditional IT approaches mean that even reliable and trusted solutions cannot be applied without significant testing. The potential damage of malfunctions and loss of availability of critical infrastructure further necessitate thorough testing. However, testing new solutions for SCADA systems is not easily accomplished. Live systems clearly cannot be used because of the potential damage unintended consequences could cause. Developing parallel but inactive systems for the purpose of testing is an approach that is often viable for testing network security, but would be prohibitively expensive for testing complex infrastructure installations. Instead, the complexity of SCADA systems calls for a

thorough software simulation to help uncover the benefits and consequences of novel security solutions.

Simulating SCADA solutions is a complex and difficult task, however. Because the development of a single-purpose simulation that captures the behavior of only a single system would be inefficient and costly, simulations should be composed of simple and reusable simulation components. Simulators dealing with the industrial process, the controller software, and the intervening network could be combined to form a simulation of the SCADA system as a whole. This requires coordinating a variety of simulation engines, each with a different set of internal data and with varying approaches to the progression of time and events. The task of coordinating these diverse simulations is the goal of the C2WindTunnel project.

The C2WindTunnel platform was designed to facilitate the evaluation of novel approaches to military command and control [22]. It is able to coordinate a variety of heterogeneous simulations in disparate problem domains to form a simulation of greater detail and broader scope. The C2WindTunnel software architecture is readily applicable to the problem of coordinating heterogeneous simulations for the purpose of SCADA security evaluation. My work focuses on modifying and enhancing C2WindTunnel to reflect the requirement of SCADA simulation.

Chapter II provides background information for SCADA systems and the C2WindTunnel platform. Chapter III describes the implementation of the C2WindTunnel extensions created to facilitate SCADA simulation. In chapter IV, a case study is presented, demonstrating the design of a SCADA simulation with C2WindTunnel. Finally, chapter V provides conclusions and a discussion of possible future work.

CHAPTER II

BACKGROUND

This section provides background information related to the architecture and operation of SCADA systems, the C2WindTunnel project used to coordinate heterogeneous simulations, and the software platform on which C2WindTunnel is built, the High Level Architecture.

SCADA Systems

A wide variety of industrial processes are managed via computerized control systems, and their diverse purposes mean that industrial control systems themselves are diverse in implementation.

The term SCADA is most frequently used to describe systems whose assets are highly distributed geographically. The control of electrical grids and oil and gas pipelines, for instance, involves aggregating sensor measurements from hundreds of widely dispersed field devices so that operators can use a centralized control interface to manage the whole process in real time. Field devices are located physically close to the portion of the process that must be controlled, and monitor sensors and drive actuators connected to the process. They are connected to the SCADA control center via a wide area network which may use a variety of topologies and protocols and be wired or wireless. Such systems must typically take into account the low bandwidth and relative lack of reliability of the networks in use, perhaps employing fault-tolerant hardware and algorithms. In addition, they must typically contend with legacy hardware and protocols since widely dispersed hardware devices are difficult and expensive to upgrade [14].

Much smaller scale operations, such as chemical manufacturing plants and pharmaceutical processing facilities, are also examples of SCADA systems. These ge-

ographically localized processes may reside entirely within a single plant floor and are sometimes differentiated from geographically dispersed SCADA systems with the term Distributed Control Systems (DCSs) [14]. These systems use field devices that are located physically close to the portion of the process under control and are connected to the master control center via the control network. The control of the whole process is modularized with the use of local controllers to provide fault tolerance and reduce the impact of a malfunction at a single field device. DCSs typically use a highly reliable and relatively high bandwidth LAN to connect field devices with the control center. In addition, physical security may be more effective since a geographically centralized system is less difficult and expensive to protect.

Although the systems that employ SCADA are widely varied in topology, scale, and purpose, they are unified by a single type of architecture. The recognition of their fundamental similarities is important to the research of SCADA security, since it allows researchers to make use of general models of the class of the class of all SCADA systems. This general model is composed of four major parts: the process to be controlled, the field devices physically connected to it, the centralized control center, and the network that connects the controller and field devices. The relationship between these components is shown in figure 1.

Architecture

The process is the physical phenomenon that operators seek to control. This portion of the system will be distinct in all SCADA systems. The process typically can be broken down into a number of smaller control problems. For instance, a plant producing a particular chemical in a reactor may need to control the temperature and pressure of the reaction as well as the volumes of the reactants. Each of these may be considered separate control problems, with local controllers engaged in the maintenance of each variable within established operating limits. However, these

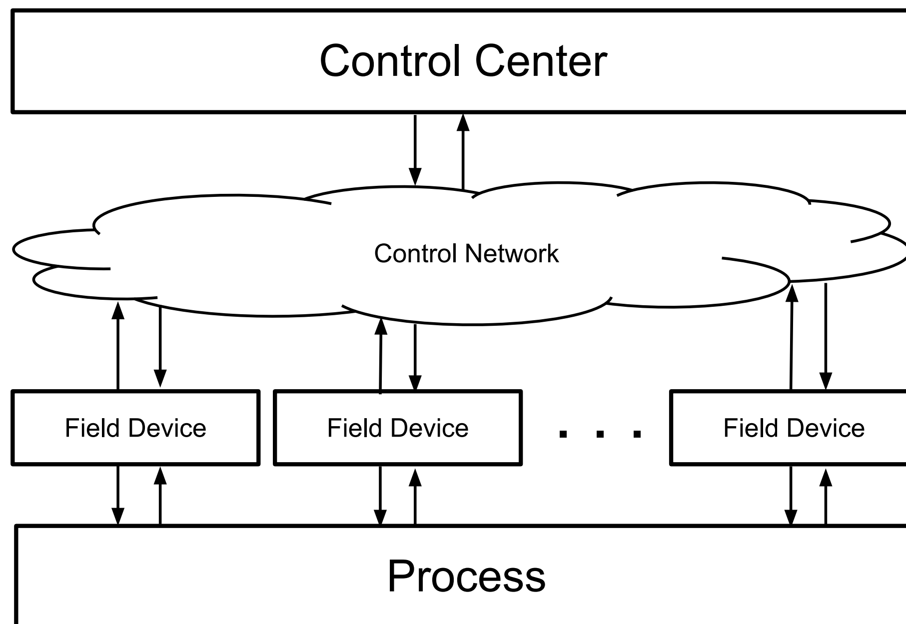


Figure 1: Typical SCADA architecture

variables are interrelated; the pressure and volume of reactants of the reactor affect its temperature and vice versa. Local controllers performing localized tasks cannot effectively maintain the high level operation of the system, necessitating a centralized master control system to perform this task.

Field devices interact with the process via sensors and actuators. They are sometimes termed Programmable Logic Controllers (PLCs), reflecting the fact that they act as controllers on a local level. Field devices deal with a localized control problem, but also send updates and receive commands from the master controller so that their local control loop can be operated in accordance with the overall process control strategy. For instance, a field device controlling the liquid level in a tank may receive liquid level readings from a sensor and be able to maintain the appropriate level by using an actuator that controls a runoff valve. Its local control problem would be to maintain the liquid level in the tank within some tolerance of a set value. Because this setpoint value is likely affected by other factors in the process, however, the field

device would receive commands to set this value from the centralized control center. Because the state of the local control problem likely affects other the state of the process as a whole, the field device would send regular sensor updates or alarms to the control center. This forms a high level control loop that drives the lower level localized control loops. The local control loop's operation and relationship to the rest of the SCADA system is diagrammed in figure 2. Field devices may connect to a single sensor or actuator or may be connected to a large network of sensors and actuators and maintain a complex local control loop.

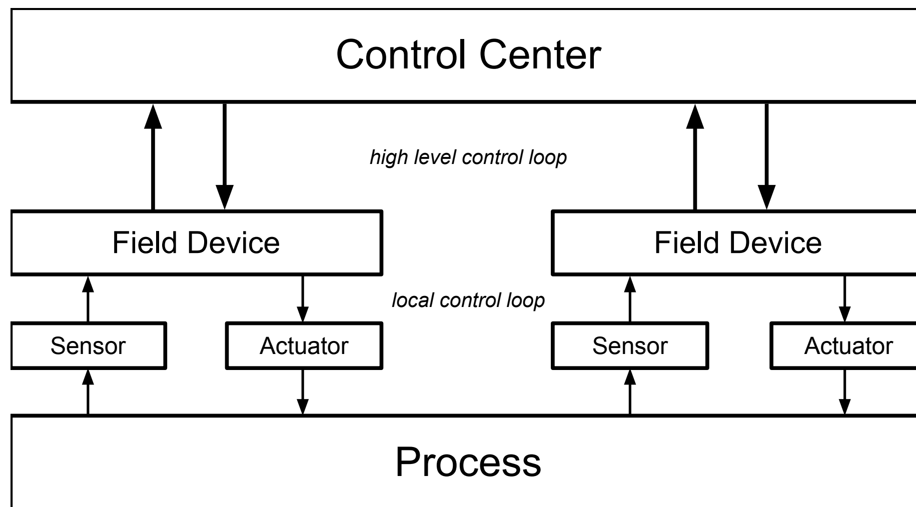


Figure 2: Field device control loop

The control center acts as the master controller, maintaining the high level operation of the process. Many field devices are employed by SCADA systems to operate local control loops, each affecting a single control problem, but in a atypical process these control problems are interrelated. For instance, the control system operating a canal may use a large number of field devices controlling the water levels in a system of locks. Because the control strategy of one lock directly affects the control strategy of its neighbors, a high level strategy must be employed to ensure correct operation. The control center sends control commands and receives sensor updates from the

field devices to allow this high level control. Depending on the SCADA deployment, control centers may operate automatically or rely on the intervention of human operators. The control center provides the interface to the human operators of the system. This interface is called the Human Machine Interface (HMI) and allows the operators to see an aggregated view of the state of the process and provides the means to send control commands to field devices in order to maintain correct operation. A control center may include several HMIs, each reflecting the requirements of its users. For instance, administrators and business managers require a different set of data and controls than system engineers. The control center is connected to field devices via the control network, and may also be connected to a corporate network or WAN to allow remote access to engineers and business administrators.

The connection between the control center and field devices is provided by the control network. This may be a wired or wireless network and may operate with a variety of network protocols. Some control networks use TCP/IP while others use fieldbus protocols, which are simple protocols designed around the sensor update and control command communication patterns of SCADA networks [23]. Depending on the process, it may be important to provide real time guarantees or provide fault tolerant or redundant networks.

Communication on control networks typically consists of control commands from the control center and sensor updates from the field devices. The communication can be asymmetric, with sensor messages being larger and more frequent than control messages. Sime field devices communicate at fixed intervals while others use alarms to communicate only significant events. Prioritization of important control messages over bulk senor readings is a typical communications requirement, as is some guarantee on the timeliness and stability of message delivery [23].

C2WindTunnel

C2WindTunnel is a software platform designed to make the design and implementation of large-scale multi-part simulations easier and more efficient through the use of model-based design techniques [22].

C2WindTunnel was designed to aid in creating detailed simulations for the military command and control domain. Command and control environments are complex and cannot be simulated by a single monolithic platform. Instead, C2WindTunnel employs the HLA framework to integrate multiple simulations into a more detailed multi-part simulation called a federation. In addition, C2WindTunnel uses the GME modeling tool to allow rapid federation design in a graphical modeling environment. Like command and control simulations, SCADA simulations are complex and can benefit from the techniques used by C2WindTunnel. For this reason, my work focuses on adapting the C2WindTunnel platform for SCADA simulations. In this section, background related to the components of C2WindTunnel is provided.

GME

Developing a federation is a complex task involving the definition of a large hierarchy of data classes and specification of an array of federates, each with its own timing and data flow information. Designing and maintaining such a large system can be a challenging task, so the C2WindTunnel platform makes use of graphical modeling techniques to help ease this burden. The Generic Modeling Language (GME) [1] is employed to provide users of the system with an easy-to-use graphical interface for modeling complex federations.

GME is a toolkit for creating domain specific modeling languages (DSMLs) with a graphical interface based on UML class diagrams. With GME, a model designer can specify the entities of interest in the problem domain as well as define the set of valid relationships among them. GME allows the creation of a DSML specifically tailored

to the needs of system designers working in a particular problem domain. The ease of use of GME and its convenient interface allow the modeling language to evolve over time as the needs of the system designers become more clear [1]. These DSMLs, called paradigms in GME, allow domain experts to work within the domain of their particular problem, reducing complexity and increasing efficiency when compared to the difficulty of using a general purpose modeling language.

GME also allows paradigms to be extended with tools called model interpreters. Model interpreters are software components that parse user-created models in order to provide additional domain-specific functionality. For example, a model interpreter could generate executable code based on the user-create model.

The C2WindTunnel platform provides a GME paradigm for modeling HLA federations and includes a set model interpreters that generate federate code and other artifacts needed to run the federation execution.

Matlab/Simulink

Matlab[4] is a programming language and numerical computing environment designed to support a wide range of problem domains. Matlab allows users to write efficient code for numerical operations and provides a large library of functions for mathematical computing and data visualization. In addition, Matlab can interface with other programming languages, especially C, C++, and Java code. Simulink[8] is a Matlab package that adds support for model-based design of dynamic systems. With Simulink, users can design a continuous time simulation using a graphical interface based on connecting elements called blocks. Blocks may perform simple built-in functions, such as adding values or plotting variables, or may be drawn from the extensive libraries provided with the Simulink package to perform more complex domain-specific tasks. In addition, users can create custom blocks with code written in Matlab, C, or C++ and can import and use Java packages. Matlab and Simulink are widely used

in the control theory domain, making them useful simulation engines for SCADA simulations. Specifically, plant and controller federates can be designed in Simulink with HLA support provided by C2WindTunnel.

OMNeT++

OMNeT++[6] is a discrete event simulation framework focusing on computer network simulations. OMNeT++ is an open source tool and allows users extensive low-level control over its scheduling algorithm used and other simulation details. Although the tool is not limited to network simulation, and provides primitives that can be used to create arbitrary discrete event systems, an extensive set of libraries are available to handle the details of popular networking protocols such as TCP and IP. For example, the INET library allows network modelers to include commonly used networking nodes like routers and TCP/IP hosts. The simulated network can be designed using the provided graphical interface or by manipulating simulation files in OMNeT++'s custom scripting language, NED. Network modelers must define the nodes in the network, the messages types that can be passed among them, the connections between nodes, and any configuration parameters needed by the included nodes. In addition to using nodes provided by OMNeT++ and its libraries, users can write modules in C++ that can be configured with the NED scripting language, allowing implementation of custom behavior. SCADA simulations can use OMNeT++ to model a control network in a SCADA system, with HLA integration support provided by C2WindTunnel.

Portico

C2WindTunnel is baed on the HLA architecture, which handles the coordination of time and data passed between federates. The Runtime Infrastructure (RTI) is the software component that implements this functionality. Since the HLA is a popular standard, a number of RTIs are available, both commercial and open-source.

Portico[7] was chosen for the C2WindTunnel platform because it is open-source, implements a significant portion of the standard, and provides both C++ and Java bindings. Portico has the additional advantage that it does not require a centralized daemon to be started before the federation can begin. Instead, federates can be started at any time and will coordinate federation setup in a distributed manner.

The High Level Architecture

The High Level Architecture (HLA) is the underlying software architecture used by C2WindTunnel to facilitate the sharing of data and coordination of time among the many diverse simulators that make up a SCADA simulation.

History and Motivation

The High Level Architecture (HLA) was developed to allow many independently developed simulations to be combined into a larger and more complex simulation. Modern simulations are rarely so simple that a single simulator, designed for a single problem area and used solely for one task, will be a cost-effective investment. Many modeling and simulation challenges involve the composition of a variety of domains, so when a simulator is created to address the narrow focus of a single challenge, significant effort is duplicated and resources are wasted. The HLA was designed to alleviate this problem by allowing diverse simulators, each designed for a particular problem domain, to be interconnected.

The U.S. Department of Defense recognized simulators as significant defense assets, and recognized that new simulators could not repeatedly be created to meet constantly changing user needs and to incorporate frequent technological advances. In addition, it recognized that no single simulation could be complex and detailed enough to meet the diverse array of modeling and simulation challenges presented to the department. For this reason, in 1994 the Department of Defense began efforts to

create a simulation architecture that would facilitate simulator composition and reuse [16]. In 1998, the Defense Modeling and Simulation Office released the first completed specification, HLA 1.3, and produced an implementation later that year [20]. Since that time, many commercial and open-source implementations of the specification have been produced. The HLA became an IEEE standard in 2000 [24], with the most recent revision of the standard having been completed in 2010 [11].

System Architecture

A simulation designed to use the HLA consists of a set of simulators known as federates that are interconnected into a larger simulation known as the federation. The set of federates may include simulators in distinct problem domains with diverse internal representations of simulation data and may even use different models of time. The HLA is flexible enough to allow discrete event simulators to interoperate with continuous time simulators, for instance. The task of the HLA is to coordinate the diverse set of federates and allow them to communicate while maintaining a meaningful progression of time for the overall federation. A single run of the federation, in which all federates execute together, is a federate execution.

The basic architecture of the HLA is composed of three parts: the Runtime Infrastructure (RTI), the Federation Object Model (FOM), and the set of federates used in the federation. The RTI is the underlying software that facilitates communication between the federates. Federates are not directly connected to one another, but instead are all connected to the RTI. The RTI implements the logical connection between the federates by passing messages between them when requested and assuring that no single federate strays irrevocably from the common federation time it maintains. The FOM is an object model that represents the data types available to the federation. The FOM is common to each federate and allows data objects sent from one federate to be received and interpreted by another [20]. The relationship

between the components in the HLA architecture is shown in figure 3.

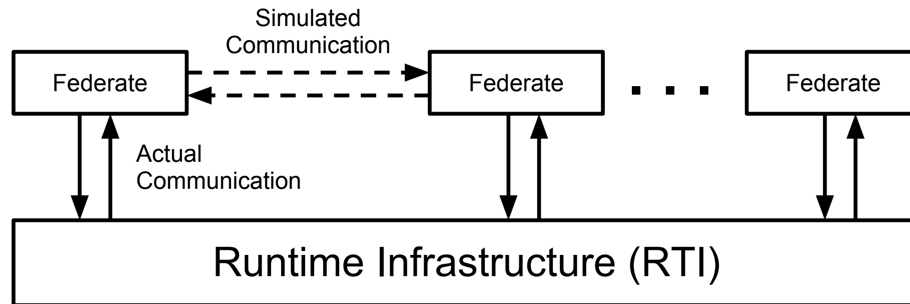


Figure 3: HLA architecture

Data Types

Federates written to comply with the HLA interface are often intended to be interoperable with a large variety of other federates so that they can be reused in multiple varied simulation challenges. In order to facilitate interoperability and reuse, the means and type of data that will be exchanged by the federate, its external interface, must be agreed upon. The HLA uses the FOM to specify this interface. The HLA standard does not constrain the content of the FOM but defines the format with the Object Modeling Template specification [10]. Using the OMT, the federation's FOM can be written so that it captures the necessary exchange of data and control information within the federation without the assumption of any particular federate implementation. By describing only the sharable information in the federation, a well-designed FOM promotes interoperability and reuse by remaining implementation-agnostic [16]. The data included in the FOM includes two types: interactions and objects.

Data that is persistent in the federation is modeled by the object data type. The federation's FOM describes the set of object classes that can exist in the federation. Each object class has a unique name and is positioned in a single-inheritance tree

rooted at the base object class, ObjectRoot, which must be included in all FOMs [10]. Objects include a set of data items called attributes that each have a name and value. Attributes are not typed; they are simply sequences of bytes. Objects, which are instances of an object class, are persistent data items that can be created, destroyed, observed, and modified by the federates in the federation. As the federate execution progresses, the federates may change an object to represent the evolution of an element in the simulation domain. Because many federates may observe and modify the persistent object, the object's evolution can be broken into many smaller phenomena, each controlled by a smaller and simpler simulator than would be required to simulate the several effects in aggregate.

Interaction data types are distinct from objects because they do not persist in the federation, but instead are shared between federates at a single point in time. One federate sends an interaction and one or many federates instantaneously receive the interaction. The data has no duration and does not continue to exist in the federation after it is sent and received. Like objects, interactions are members of the set of interaction classes defined in a federation's FOM. All interaction classes are named and include a set of data members called parameters. Parameters are analogous to an object's attributes and are made up of a name and value, and like attributes are untyped. The federation's FOM defines a single-inheritance hierarchy for interactions that is rooted at a base interaction, InteractionRoot, which must be included in all FOMs. Since interactions are discrete events that do not persist in time, they are often used to model discrete events in the simulation domain [20].

Data Flow

The federation FOM describes the content of the data shared between the federates in a federation, but not the source and destination or the frequency of the flow of data. In order to promote encapsulation of concerns and federate reusability, the HLA

uses a publish-subscribe model to facilitate the sharing of data among federates. In a publish-subscribe system, the sender of the data is not directly responsible for its destination. Instead, the sender, known as the publisher, publishes a data object by making it available to any receiver that registers to receive data of that type. Receivers, known as subscribers, subscribe to the data by alerting the publish-subscribe architecture that they wish to be forwarded any future data objects that are published with their appropriate type. In this way, the sender need not concern itself with the routing of the data, but may instead focus on the content and frequency of its publications. Federates may choose to receive publications or not, depending on the concerns unique to their own simulation domains. The publisher is therefore shielded from directly interacting with the data's consumers. Instead, federates interact only with the shared federation state, which consists of the interactions and objects defined in the federation's FOM.

The publish-subscribe architecture is ideally suited to the requirements of the HLA because it is designed to support decoupling of the producers and consumers of data [15]. Because federates are not interconnected with the consumers or producers of the data they use in their simulations, they minimize external dependencies. Because other federates are relied upon only indirectly, a federate will not be broken when others are replaced or improved. As long as the federation's FOM is not changed, the federate remains usable, which allows it to be used in a large variety of simulation problems alongside a diverse set of other federates.

Publishing Data

Federates prepare to publish objects and interactions by first declaring their intention to the RTI. When a federate joins a federation, it must publish all object and interaction classes that it may send to the RTI over the course of the federation execution. Because object attributes can be updated on an individual basis, the federate must

indicate a subset of the object class's attributes that it wishes to have the ability to update during the federate execution. This is not necessary for interactions; only the interaction class must be specified. Objects and classes are specified by name and must be among the data classes defined in the federation's FOM.

Federates can then create object instances in the federation by registering an object. Once the object has been created, it can be modified by updating its attributes. A subset of an object's attributes can be modified and the RTI will guarantee that they are modified simultaneously. Objects can also be deleted, which will remove the object from the shared federation state and notify subscribers of its removal. Creation, modification, and deletion of objects can all be parameterized with a time argument, allowing the RTI's time management services to affix the event at a point in the federation's progression of time.

Federates can also send interactions. Like object updates, interactions may include all or a subset of the parameters defined on the interaction class. Since interactions occur at a single point in time and do not have duration, they cannot be created or destroyed [20].

Subscribing to Data

In order to receive interactions and object attribute updates, a federate must first declare its interest in a particular interaction or object class. The federate subscribes to an interaction class or a set of object attributes to indicate interest. Later a federate can unsubscribe to stop receiving notifications related to those data classes. When an object is registered by a federate, all federates in the federation that have subscribed to attributes of the appropriate object class are notified by the RTI. These federates discover the new objects. Upon object deletion, subscribed federates are notified by the RTI with the remove callback function. When an object's attributes are updated, all subscribed federates are notified with a reflect callback function, which includes

the set of changed attributes and their updated values. Because interactions do not persist in time, federates do no need to discover them, but simply receive interactions when they are sent [20].

Time Management

In addition to coordinating data flow, the RTI is also responsible for time management within a federation. A federation is composed of many different simulations, each with its own logical time, so the RTI must ensure that any date passed from one federate to another occurs at a time that is appropriate for each simulation involved. The RTI preserves the causality of the federation, ensuring that no simulation receives an event that occurred in the past relative to its own logical time. This requirement is complicated by the variety of time management strategies available to federates in an HLA federation.

Federates must choose their level of involvement in the federation's time management by registering in the initial setup phase. A federate may choose to be time regulating, time constrained, both or neither. A federate that is time regulating is said to regulate the advancement of time of other federates, meaning other federates cannot advance their logical times beyond its own. Similarly, a federate that is time constrained is constrained by the advancement of time regulating federates, meaning its own time cannot advance beyond other federates. Federates may choose to be neither time regulating or time constrained, meaning they do not participate in time regulation, but proceed at a rate independent of other federates. Typically, however, federates are both time regulating and time constrained meaning they participate fully in the federation's time management. They cannot advance time without the coordination of other time regulating federates and cannot be left behind by time constrained federates [20].

In addition, federates may choose one of two general strategies for time advance-

ment: time stepped or event based operation. Time stepped federates proceed in discrete intervals independent of the arrival of events from the RTI. Event based federates advance time until the next event is received and continue simulation at the time of the event. Federates may choose to employ both of these strategies, although applications for this style of time management are rare. Typically, properties of the underlying simulation will suggest the most appropriate time management strategy.

Time regulation in the HLA is centered around the exchange of data among federates. Because they are the only opportunity for the causality of the federation to be violated, the sending and receiving of interactions and object updates must be regulated. Federates may request that events are not time sensitive, however, by designating them Receive Ordered (RO) events. The RTI will ensure that they are received in the order they were sent, but does not ensure that they arrive at any particular time. Time-stamp Ordered (TSO) events, on the other hand, are subject to time regulation and can only be received by time constrained federates.

In order to accomplish time management, the RTI requires another parameter from the federates at registration time, the lookahead. The lookahead value places a restriction on the time that a federate can send events. If a federate's logical time is t and its lookahead value is l, then it cannot send an event until time $t_s > t + l$. This restriction is employed to prevent a possible deadlock scenario. If the lookahead restriction were not in place and a federate were allowed to receive an event and send out another at the same time, the order of events would be ambiguous. The RTI could not allow any federate to advance because doing so would risk a federate receiving events in its past. Because the lookahead value is used, the RTI can guarantee that some time will elapse between a federate's reception and sending of an event and that causality is not violated [21].

Lookahead selection is an important part of federate setup. Small lookahead values may constrain a federate to essentially sequential execution, decreasing parallelism

and this performance. However, large lookahead values decrease the accuracy of the federation execution since federates may have to wait until a later than desired time to send out interactions or object updates.

Lookahead selection can be guided by some properties of a federation. In the case of a time-stamped federate, for instance, no event will we sent until a time step has elapsed, so the lookahead can be safely set to the value of the time step. For other federates, the minimum response time of the simulation should be taken into account. For example, a network federate may receive events, simulate their transmission through a network, and then publish an event when this has completed. If passage through the network involves some minimum delay, then the lookahead value can be set to this delay without danger of degrading the accuracy of the federation execution.

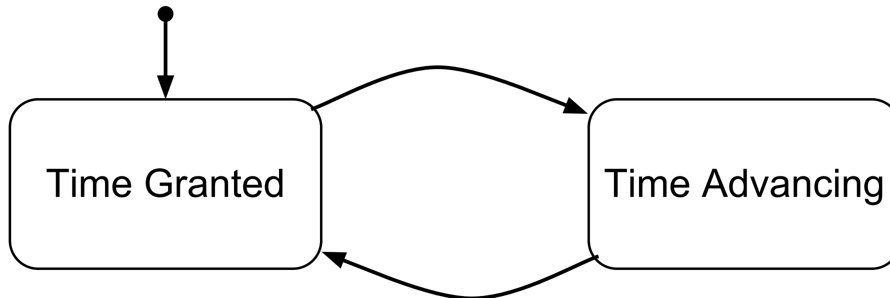


Figure 4: Time advancement state diagram

A federate's time advancement occurs in two alternating phases: the time advancing and time granted phases. The progression of a federate through these phases is shown in figure 4. A federate begins the federation in the time granted phase with an initial logical time. It will then request to advance its time with either the time advance request or next event request calls to the RTI, depending on whether it uses a time stepped or event based timing strategy. Each call has a time associated with it, and each federate has a registered lookahead value. The RTI receives requests from all time constrained and time regulating federates and chooses the least of these

times. Federates cannot send events until their requested time has been reached, so choosing the federate with the least time request value ensures that no event will occur in the past relative to any federate's logical time. The RTI then sends a time advance grant callback to the chosen federate, returning that federate to the time granted state. The federation proceeds in this manner, with the federate with the least requested time allowed to advance at each step, until the federation execution completes [20].

CHAPTER III

IMPLEMENTATION

This section provides implementation details related to the SCADA extensions added to C2WindTunnel, including integration with the network simulator OMNeT++ and Microsoft Windows support.

OMNeT++ Integration

The first major barrier to using C2WT for SCADA simulation was the lack of full support for a network simulator, a key component in any SCADA setup. Support for a network simulator would allow modeling network effects on control algorithms and would allow testing the consequences of network attacks on process stability.

Many network simulators are available that could be modified to support HLA and the C2WindTunnel platform. The open source network simulator OMNeT++ was selected by the C2WindTunnel developers because of the ease with which it can be modified and its modular architecture which makes replacing the event scheduler a straightforward task [22]. OMNeT++ is a packet level simulator, an important detail for SCADA simulations in which network effects on individual packets can impact the operation of the entire process. In addition, OMNeT++ provides a graphical interface for network design, allowing quick and easy-to-use network setup.

Time Management Policy

An important decision for the initial implementation of any federate is its time management policy. This policy controls the rate at which a federate requests time advances from the RTI and the timing of the receipt of RTI events. A network federate such as OMNeT++ can be reasonably implemented in two ways: as a time-stepped or event-based federate. The C2WindTunnel implementation of OMNeT++ network

federates uses a time-stepped time management policy as a compromise between timing accuracy, development effort, and simulation flexibility.

Federates that use an event-based time management policy achieve greater timing accuracy with respect to RTI events than do time-stepped federates. Unlike time-stepped federates that must advance time only in discrete intervals, event-based federates may advance time to a specific point in the future or until the next RTI event is received, whichever occurs first. This means that event-based federates are able to access and respond to RTI events, such as interactions and object updates, at the same time that they occur. This can be implemented by modifying the simulation scheduler in OMNeT++ so that a next event request is called before the local simulation time is allowed to advance.

```
// get simulation time of next OMNeT++ message
cMessage *nextMsg = sim->msgQueue.peekFirst();
double nextMsgTime = nextMsg->getArrivalTime();

// advance time until next simulation message or RTI event
// a callback will add a new message to queue if necessary
getRTI()->nextEventRequest(nextMsgTime);

// next message may have changed if RTI event occurred
nextMsg = sim->msgQueue.peekFirst();

// allow next simulation event to run
return nextMsg;
```

Figure 5: Event-driven federate scheduler code

A simplified example of an OMNeT++ scheduler modified to implement an event-based federate is shown in figure 5. The purpose of this method is to remove the next event, known as a message in OMNeT++, from the simulation's event queue and process it. When the message is returned, the simulation's time is advanced to the arrival time of that message. In order to integrate the federate with the HLA, the OMNeT++ simulation's time must be kept in sync with the advancement of time

in the federation. This is accomplished with the `nextEventRequest` method, which requests that the federation time be advanced to the time of the next message or the time of the next RTI event, whichever occurs first. The OMNeT++ federate must then handle a callback from the RTI that will inform it of the newly advanced time. If an RTI event has occurred before the next simulation message, the federate must advance the simulation time to the time specified by the RTI. This is done by simply creating a new message at the appropriate time and placing it on the simulation's event queue. If no RTI event has occurred, the scheduler can proceed as normal, allowing the next simulation event to occur.

Federates that use a time-stepped time management policy advance time only in discrete intervals called steps. These federates use the time advance request call to advance federate time forward, but unlike event-based time management, this call does not halt advancement upon receipt of an RTI event. Instead, time-stepped federates are not able to respond to RTI events such as interactions or object updates, until the next step has been reached. This constrains the time accuracy of time-stepped federates by making the step the effective minimum time resolution. This time step parameter is user-defined, so by decreasing the step size, time-stepped federates can be made to approach the accuracy of event-based federates at the cost of simulation performance. Still, the use of discrete time intervals makes time-stepped federates inherently less accurate than event-based federates.

A simplified version of an OMNeT++ scheduler modified to implement a time-stepped federate is shown in figure 6. This implementation is similar to the event-based version, with the primary difference being the use of the time advance request method instead of the next event request method. The time of the next OMNeT++ message is checked and the simulation is advanced step-by-step until that time is reached. The federate will receive a callback from the RTI after each time step. The callback must check for new interactions and introduce them into the network

```
// get the next message in OMNeT++'s event queue
cMessage *nextMsg = sim->msgQueue.peekFirst();

// advance time step-wise until the arrival time is reached
while (nextMsg->getArrivalTime() > getRTI()->getTime()) {
  // advance time one step
  getRTI()->timeAdvanceRequest(getRTI()->getTime() + TIME_STEP)

  // next message may have changed if RTI event occurred
  nextMsg = sim->msgQueue.peekFirst();
}

// allow next simulation event to run
return nextMsg;
```

Figure 6: Time-stepped federate scheduler code

simulation if necessary. The receipt of RTI events may cause a message to be added to the head of the simulation's event queue, so the arrival time must be recalculated at each step. Finally, the scheduler may proceed and allow the next simulation event to execute.

OMNeT++ allows access to the simulation scheduler via subclassing the default scheduler and overriding the relevant methods, making scheduling modifications easy to accomplish. The scheduler class is not a first-class OMNeT++ module, however, so there are some restrictions on its use. Scheduler subclasses do not have access to the `omnetpp.ini` file, the standard OMNeT++ interface for assigning user-defined parameters at runtime. The `omnetpp.ini` file is convenient because it allows simulation variables to be changed between executions without the need to recompile the simulation source code. Implementing HLA integration requires a large number of parameters including federation and federate names, the FOM file location, time management data, and publish and subscribe relationships. For this reason, the inability to use the standard OMNeT++ parameter assignment interface is a significant limitation. This can be worked around in two ways: hard-coding the

parameters in the scheduler source code or using a data retrieval method external to OMNeT++. Hard-coding the necessary federation parameters was the approach taken by the C2WindTunnel team in the original implementation of OMNeT++ support. Source code for the scheduler class was generated from the C2WindTunnel federation model and then compiled. This method has the limiting side effect that any change to the federation parameters requires regenerating and recompiling the OMNeT++ scheduler source code. This requirement hampers development by making even minor tweaking of parameters a time-consuming task. Using an external data retrieval method is an acceptable alternative since it does not require regenerating and recompiling the scheduler source code, but it does introduce an additional dependency. Since OMNeT++ already includes an elegant method of assigning user-defined parameters, a solution that makes use of this default interface would be preferable.

OMNeT++ imposes additional restrictions on the internal exchange of data between simulation modules and the scheduler. Modules are able to pass messages to other modules internal to the simulation and to themselves by using the `send` and `scheduleAt` method calls. These methods cause messages to be placed in the simulation's event queue so that they can be scheduled and dispatched. Simulation modules do not have direct access to the event queue, however. Conversely, scheduler classes have direct access to the simulation's event queue, but they do not have access to the `send` method. The scheduler and modules were designed to operate independently, so public methods for direct communication between them are not provided. This presents a problem for scheduler classes that implement HLA integration, since RTI events must be passed to the modules that make up the network simulation in order for network traffic to be simulated.

A reasonable workaround to the scheduler-module barrier in OMNeT++ is diagrammed in figure 7. The scheduler class can access the simulation data to find a pointer to an instance of a module class with a pre-defined name, SchedulerInterface

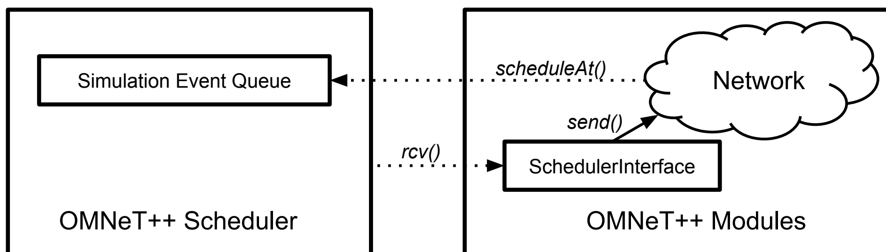


Figure 7: OMNeT++ scheduler-module barrier workaround

in this example. Using this pointer, a specially designed method can be called on the instance to pass data from the scheduler to the module. This `rcv` method passes the newly received interaction and its destination in the network and is used as an alternative to standard message passing. Once the data path from the scheduler to the module instance has been established, the designated module can forward the data to other modules using the normal `send` method. Establishing a data path back from the modules to the scheduler can be accomplished by adding a specially designed message to the simulation queue. This can be achieved indirectly with the `scheduleAt` method. A module can call this method to add an event to the simulation's event queue which can be directly inspected by the scheduler. The event contains the modified message that has crossed the simulated network.

While this approach does provide a solution to the scheduler-module communication problem, it bypasses the recommended means of data transfer in OMNeT++. Special care must be taken to ensure that the module referred to by the scheduler's pointer is not deallocated and the OMNeT++ ownership management system must be alerted when messages are referenced in this manner. In addition, development effort must be spent implementing a data forwarding system between the designated module and others in the network, essentially replicating a feature already available in OMNeT++. OMNeT++'s documentation discourages the use of these direct method calls, both for its fragility and inelegance, so a solution that did not make use of this

approach would be preferable.

To achieve Windows compatibility with the 1.0.1 version of Portico, the RTI implementation used by C2WindTunnel, my development of HLA integration for OMNeT++ makes use of Portico's Java bindings. This requires embedding Java method calls within OMNeT++'s C++ code, a facility provided by the JSimpleModule library for OMNeT++. The reasoning for and consequences of the use of Portico's Java bindings is further discussed in the Windows Support section. JSimpleModule provides support for Java calls from within modules but not from within a scheduler class. Although previously described scheduler-module data passing techniques could be employed to circumvent this limitation, JSimpleModule's lack of scheduler support provides additional incentive to seek a module-based scheduling solution.

Since a scheduler-based implementation of HLA integration in OMNeT++ is subject to a number of restrictions, it is fortunate that a time-stepped time management policy can be achieved from within a module class using an unmodified scheduler. An analogous event-based policy with an unmodified scheduler is not practically achievable, however. Although event-based time management is clearly a superior policy for a network federate, the opportunity to implement time management from within a module provided a compelling reason to choose a time-stepped policy. Finally, since much of the initial C2WindTunnel code assumes the use of a time-stepped federate, a time-stepped implementation could reuse a significant amount of existing code. For these reasons, a time-stepped policy was chosen as a compromise between simulation accuracy, flexibility, and development effort.

A simplified version of the code used to implement a time stepped time management algorithm in an OMNeT++ module is shown in figure 8. The module initializes at the simulation's initial logical time and schedules itself to run every time step. This means that the network federate will only interact with the RTI once per time step. The HLAInterface module manages two queues: one for incoming and another for

```
// convert and route incoming interactions
Interaction *inIntr;
while (intr = incomingQueue.front()) {
  cMessage *inMsg = convertToMsg(inIntr);
  sendToNetwork(inMsg);
}

// convert and publish outgoing interactions
cMessage *outMsg;
while (outMsg = outgoingQueue.front()) {
  Interaction *outIntr = convertToIntr(outMsg);
  publish(outIntr);
}

// advance time by one step
getRTI()->timeAdvanceRequest(getRTI()->getTime() + timeStep);

// schedule module to run at the next time step
scheduleAt(heartbeat, getTime() + timeStep)
```

Figure 8: Time-stepped module-based federate code

outgoing interactions. If an event occurs in the network that necessitates an outgoing interaction or if an incoming interaction is sent at a time between time intervals, these interactions must be queued and wait until the next time step. The HLAInterface module is able to operate only once per time step without loss of accuracy because a time-stepped federate only interacts with the RTI at this frequency anyway.

An event based HLAInterface could not be implemented in the manner because it requires information about the simulation's event queue. An event based implementation must call the next event request method with the time of the next simulation event. If it uses a time later than the next simulation event, that event could be sent in the past relative to the simulation's logical time. Without access to the simulation's event queue, only a time stepped version can be implemented.

Interaction Management

The primary task of an HLA network federate is to receive RTI events representing network traffic, simulate their progression through the network, and publish them to be received by the destination federate. Network messages can be represented in the HLA FOM as either interactions or objects, but they map most naturally to interactions since they are sent and received at discrete times and do not persist in the federation once they have completed transmission.

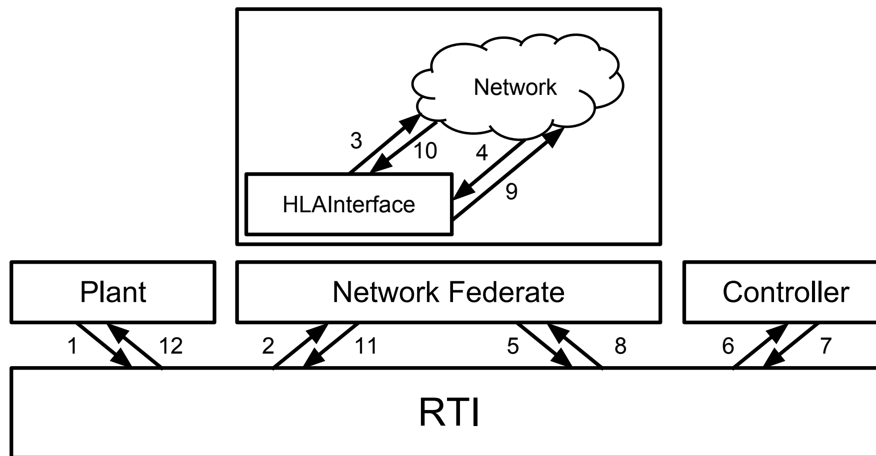


Figure 9: Data flow for a simulated network packet

The path taken by data simulating a network message transmitted from a sensor federate to a controller federate in a SCADA system is shown in figure 9. A sensor federate must first publish an interaction containing the message data to the RTI. The interaction type and parameters are specified in the federate's FOM, which is generated from the federation model specified in GME. The interaction is received by the network federate, specifically the HLAInterface module that implements the method calls and callbacks from the RTI. The HLAInterface module must complete the setup of the network federate by registering to publish and subscribe to interactions of the relevant types. To do this, the module needs information about the federation name,

its own federate name, and a list of the interactions relevant to its execution. The federate and federation names are passed to the module via the `omnetpp.ini` configuration files. The initial version of this file is generated by C2WindTunnel's federation model interpreter. Network modelers may modify the configuration file to further configure the network. The interactions and parameters used in the federation are defined in the FOM, but not the federate's publish and subscribe relationships. For this reason, an annotated version XML version of the FOM is generated that includes this information. This version uses a more common format than that used by the FOM so that XML parsing and generation libraries, which are widely available, can be used. The annotated FOM is used in the message conversion and routing steps as well. The FOM and annotated FOM are stored in a predefined location dependent on the federation name so that they can be located with the configuration parameters provided.

The next step is the conversion of network interactions into network packets for the purpose of simulating their progression across the network. OMNeT++ uses user-defined message classes to represent network packets, so the HLAInterface module must instantiate and populate these classes upon receipt of each interaction. The message classes are generated by the C2WindTunnel federation model interpreter and included in the network simulation's source folder. C2WindTunnel uses interaction names to associate interactions with messages, so an HLA interaction of type SensorUpdate will map to an OMNeT++ message of type SensorUpdate. OMNeT++ provides introspection methods on message classes so the HLAInterface can use the list of message parameters to perform a field-by-field copy from the HLA interaction to the OMNeT++ message. Once the message is created, it can be included as the payload in a network packet and passed to the routing algorithm. Because different network simulations may require different packet types, the HLAInterface class can be subclassed and the packaging method overridden. The details of the routing

algorithm are discussed in the next section.

The next step is the simulation of the packet crossing the network. The routing module takes care of the introduction of the packet into the network at the appropriate point and the OMNeT++ simulation handles its progress across the network. The task of the network modeler is to include the network topology as well as the behavior of its participants, including attackers. Once the passage across the network is complete, the packet is passed back to the HLAInterface module for conversion and to be published to the RTI.

One issue with the message conversion strategy used in this network federation implementation is the performance cost of the incoming and outgoing copy operations. A cheaper option would be to store incoming interactions and simply pass a pointer to the interaction across the network. This approach works well when the packet does not need to be modified. In more complicated simulations, such as a man-in-the-middle attack, however, the message may need to be modified by modules in the network. Using a pointer to an interaction complicates the task of the network modeler who would then be required to have knowledge of both the network domain and the underlying federation platform used. For this reason, the message conversion approach was chosen, despite the performance penalty.

After the message transmission has been simulated, the HLAInterface module simply performs the reverse process of its initial conversion step. However, publishing the interaction is complicated by the name of the outgoing interaction. The HLAInterface must be aware of the naming convention used to differentiate incoming and outgoing interactions and translate accordingly.

A network federate cannot publish and subscribe to identical interactions because it would then be unable to distinguish between incoming and outgoing data. Because it would publish and subscribe to the same interaction type, the network federate would receive every interaction it sent. Two simple options are available to work

around this problem. The first is to include a parameter in all network-bound interactions that specifies whether they are going to or coming from a network federate. The parameter could be updated by the network federate when simulation of the packet was completed. Including this parameter requires all federates interacting with the network to be aware of it, however, and would require code changes in each of them. An alternative approach is to use two interaction types to specify a network message, one each for messages inbound to and outbound from the network federate. These interaction types would be identical in structure and be differentiated only by name. C2WindTunnel uses the convention of inbound interactions being prefixed with "Send" and outbound interactions being prefixed with "Receive". With this approach, federates wishing to subscribe to sensor updates that had just come from the network could subscribe to ReceiveUpdateOut whereas federates wishing to publish sensor updates to be sent across the network could publish to SendSensorUpdate. With this approach, federates must only be aware of the naming convention and do not require code changes.

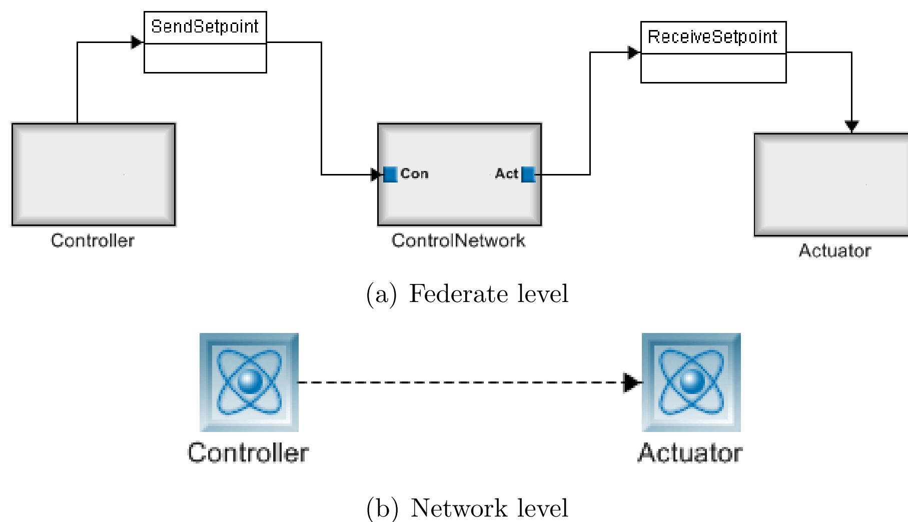
Now that the flow of data shown in figure 9 has passed from the plant federate through the network federate to the controller federate, the controller may send a control command in response. The path is taken in reverse to deliver the control command, and the control loop is closed.

Network Routing

A key component of any network simulation is the routing of data from its point of origin to its destination. In a typical network simulation, individual hosts will determine the destination of each packet and use a routing protocol like IP to ensure it reaches its destination. Because C2WindTunnel allows network modelers full control over the operation of the network via OMNeT++, manually coded routing solutions are of course supported. This would be the preferred approach for implementing a

complex network simulation with a dynamic topology or routing algorithm. In many cases, however, all that is necessary is a static topology with constant points of entry and exit. To support this common use case, a default routing mechanism is generated from the federation model via C2WindTunnel's model interpreter.

In C2WindTunnel, network packet classes are created from interaction classes specified in the federation's FOM. The HLAInterface module, which contains the RTI calls and callbacks, converts interactions into messages based on the interaction class of the incoming data. The interaction class is also important for specifying network routes. The federation model allows federates to be connected with arrows signifying publish and subscribe relationships. This mechanism is extended to allow federate modelers to define message routing paths as well. Network federates can include Endpoint components that represent points of ingress and egress on the simulated network. By connecting an interaction class with a network federate's Endpoint component, the federation modeler can define the default point of entry and exit for a class of interactions. Routes through the network can then be defined by connecting Endpoint components within the network federate.



(a) Federate level

(b) Network level

Figure 10: GME network interface

An example of the GME interface provided is shown in figure 10. This represents

a simple but common use case in which a controller sends command setpoints across a control network to an actuator. The federate publish and subscribe relationships and interaction entry and exit points are specified in figure 10(a), at the federate level of the model. Users can drill down into the network level, shown in figure 10(b), to specify the routes within the network.

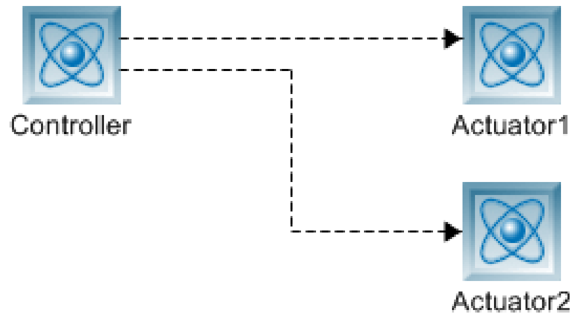


Figure 11: GME network interface with multicast

The federation model also supports multicast messages, as shown in figure 11. In this example, a controller sends commands to replicated actuators. The generated routing code supports multicast messages by replicating the message and sending it to all specified destinations.

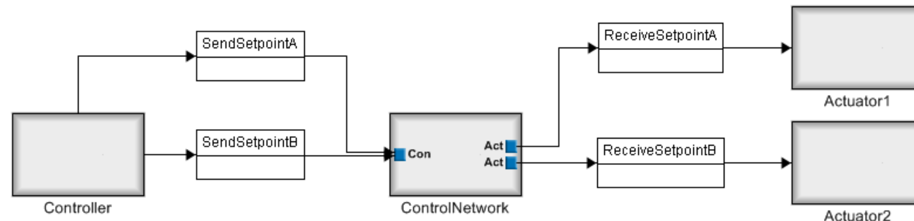
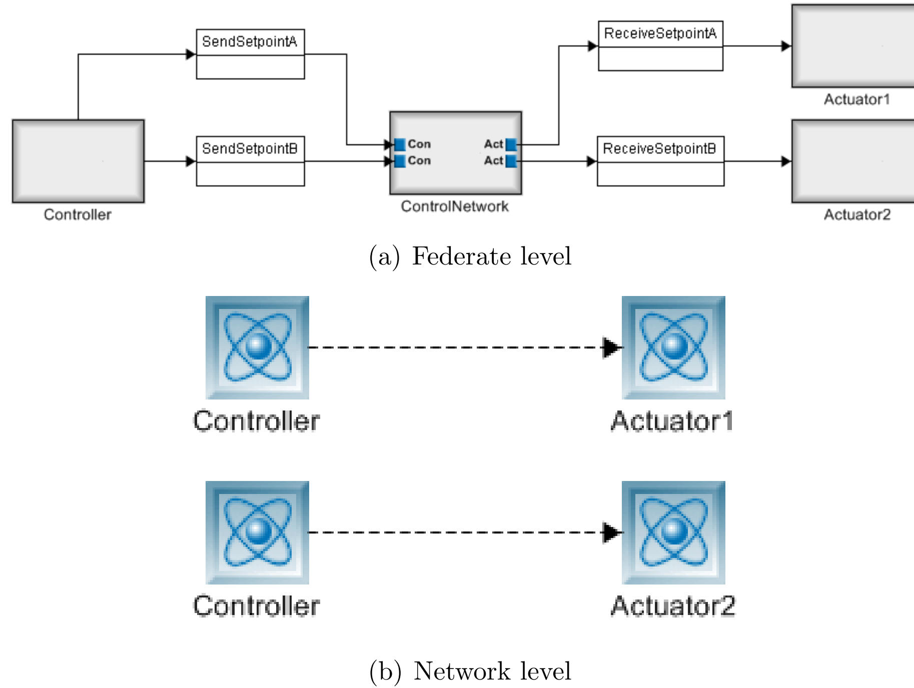


Figure 12: Ambiguous GME network model

The federation model does have some weaknesses when representing complex scenarios, however. In figure 12, a single controller routes two different types of command messages, each to a different actuator. Which messages to route to which actuator is ambiguous in this model, however. The generated code will send both messages to both actuators, but in this case that is not the desired behavior.



(a) Federate level

(b) Network level

Figure 13: Ambiguous GME network model workaround

A method of working around this ambiguity is shown in figure 13. Here the controller endpoint is included twice and a single route is defined with each Endpoint. The interpreter will condense Endpoints of the same name into a single component but will preserve the routing information defined in the original model.

Each Endpoint specified in the federation model is included in the OMNeT++ network as a standard INET module, a StandardHost, running a simple routing application. The generated nodes will have the same name as the Endpoint component used to model them in GME.

The routing information is included in the annotated FOM and is parsed by the routing module in OMNeT++. The routing module creates a simple routing table from this data, associating message types with their points of entry and exit on the network. During the setup phase of the network simulation, the routing node distributes this routing information to the routing applications running on the generated network endpoints. These applications will create their own simple routing

tables. Entry points into the network will keep entries associating message types to their network destination and exit points from the network will a list of message types that should be forwarded to the HLAInterface module to be converted and published. A single endpoint can act as both an entry and exit point without issue.

As long as the generated endpoints are used, the network that connects them can be set up to use any topology. New messages can be created and existing messages can be dropped or destroyed and the changes will be reflected in the interactions published by the network federate. This routing setup provides a good baseline for the common use case of a static topology and static routes. Although dynamic network topologies are rare in SCADA systems, the default routing solution can be extended or replaced to support this and other simulation goals.

By default, C2WindTunnel's SCADA interpreter generates an OMNeT++ network file that includes the set of specified network endpoints and an HLAInterface module connected to the default routing module. Network modelers wishing to bypass the default routing algorithm entirely can simply remove the routing module altogether. It can be replaced with a more specialized routing module or the HLAInterface can be connected directly to a node in the network if only one entry point is desired. To simply tweak the included routing code, users can subclass the routing module or routing applications used by the endpoint nodes and extend them to satisfy a more complex routing goal.

GME Additions

In order to support a higher level of detail in the federation model, several components were added to the HLA metamodel. In addition, the original C2WindTunnel federation interpreter was extended and specialized for SCADA experiments.

Figure 14 shows the changed portion of the HLA metamodel in GME. The largest change was the addition of an OmnetFederate model that inherits from the Federate

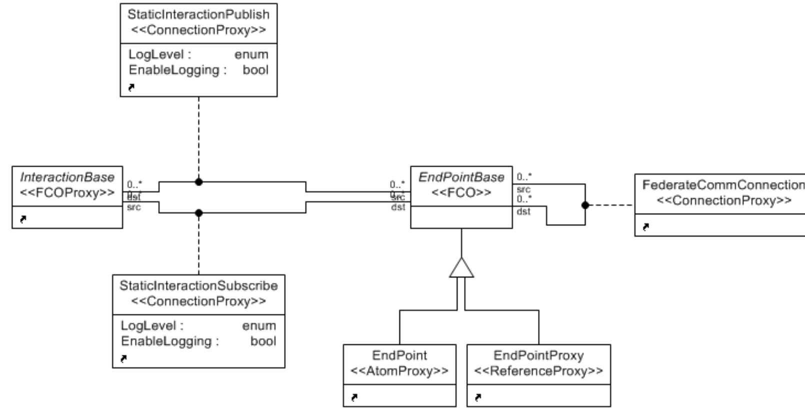


Figure 14: Changes to HLA metamodel

model. The OmnetFederate inherits the ability to define publish and subscribe relationships by being connected to an interaction object. It has been extended to allow interaction connections to specify network routes. OmnetFederate models can contain Endpoint models that can be connected to each other and to interactions. Endpoint models are exposed as ports, meaning they are visible as child components within an OmnetFederate and can be connected to objects external to their containing model. Connections from interactions to Endpoints determine the type of incoming interaction and the location of its entry point in the network. Connections from endpoints to interactions determine the type of outgoing interaction and the location of its exit point. Connections between endpoints define the path that the simulated packet will take and connect incoming interaction types to their outgoing counterparts. It is only the entry and exit points in the network that are defined at the federation model level. It is up to the network modeler to fill in the remaining components in the network using OMNeT++. In addition to the changes to support an OMNeT++ federate, an attribute was added to the Federate model to define a federate's time step. Previously this value had been hard-coded into the federate source.

C2WindTunnel's included interpreter was also changed to reflect the additional

network support added to the HLA metamodel. In addition to the FOM file that was generated by the original C2WindTunnel interpreter, the SCADA interpreter also generates an annotated FOM file. This file is in XML format, allowing federates that parse it to use widely available XML libraries. The annotated FOM contains the list of federates in the federation and the interaction and object hierarchies present in the original FOM, but also contains information about publish and subscribe relationships and network routes used by the OMNeT++ federate. The SCADA interpreter also generates the files needed to setup an OMNeT++ federate. A network file is generated that includes the HLAInterface and routing nodes as well as the endpoint nodes and their routing applications. A configuration file is generated that provides the federation and federate names to the HLAInterface module. Finally, any message classes that will be used by the simulation are generated and placed in the `msg` subfolder. When the OMNeT++ makefile is run, C++ source will be generated for each message class and compiled. No additional source code needs to be generated for an OMNeT++ simulation. Instead, behavior for the HLAInterface, routing nodes, and routing applications is provided by reusable library code.

Windows Support

Support for Microsoft Windows was an important target for maintaining and improving the usability of the C2WindTunnel platform. C2WindTunnel relies on GME for its federation modeling features, but GME is only available on the Windows operating system. Since GME is used for code generation and does not interact directly with a running federate execution, developing and deploying a federation on different machines could be accomplished without issue. However, any modeler wishing to develop and test an application on the same machine would be required to use a Windows machine. Furthermore, providing Windows support broadens the pool of developers that are able to use C2WindTunnel.

C2WindTunnel was able to run on Windows prior to my involvement, but adding OMNeT++ support while maintaining Windows compatibility proved to be challenging. The 4.0 version of OMNeT++ includes a packaged and preconfigured compilation environment based on MinGW[5], a C++ compiler for Windows based on the GNU compiler toolset. Using this compiler allowed C2WindTunnel to avoid depending on Microsoft's Visual Studio C++ compiler and was a desirable choice since its use in OMNeT++ requires no additional configuration by the user. Unfortunately, the C++ bindings for Portico, the RTI implementation used by C2WindTunnel, do not support the MinGW compiler. The 0.8 version of Portico did provide MinGW support, but the recent 1.0.1 version dropped MinGW as a compatible compiler.

To work around this issue, the OMNeT++ code used to interface with the RTI uses Portico's Java bindings instead. OMNeT++ code is written in C++, so using the Java bindings required employing the JSimpleModule library[3], which allows OMNeT++ modules to be written in Java. JSimpleModule accomplishes this by wrapping the OMNeT++ simulation library using SWIG[9]. This allows the HLAInterface module to be written in Java, allowing Portio's Java bindings to be used without interrupting the operation of the other network nodes. The decision to use JSimpleModule also influenced the selection of a time management strategy, as discussed in the High Level Architecture section.

Although Windows support complicated the development of a C2WindTunnel-compatible OMNeT++ federate, the operating system also provides the opportunity to create an installer application to aid in platform setup. C2WindTunnel is composed of many separate programs and libraries, so setting the system up by hand is a complex and time-consuming endeavor. Using a Windows installer reduces setup time by automating some of the tedious and error-prone installation tasks involved. The installer was built using the Inno Setup installer creation tool[2]. It automates the setup of the C2WindTunnel directory structure, defines several required environ-

ment variables, and sets registry entries required by the included GME components, resulting in a single installation executable. Although additional setup steps are required beyond the running of the installation file, much of the tedious work is done for the user.

CHAPTER IV

CASE STUDY

In this section, a description is provided of the process of designing and running a sample SCADA simulation. The design of the federation using GME, the generation of federate code, and the implementation of its component federates in Simulink and OMNeT++ is discussed.

The SCADA system modeled in this case study is based on a widely used control theory challenge problem based on a multi-reactor process used by the Tennessee Eastman company [25]. The chemical process used is a simplified version of the originally presented challenge problem, and serves to exercise the C2WindTunnel framework by providing a realistic control problem upon which to base the simulation. The process exposes 10 sensor variables and 4 actuator setpoints that will be operated by the controller federate. This study will examine the effects of control network disruption caused by a distributed denial of service attack on an intermediate router in the network.

Federation design begins in GME where a new model is created using C2WindTunnel's HLA metamodel. Federation modelers have three main tasks: defining the object and interaction hierarchies, listing the federates and their publish and subscribe relationships, and defining the deployment setup.

First, the data that will be used by the federation is defined. The HLA requires that all interactions be members of a single-inheritance tree rooted at the InteractionRoot interaction. Figure 15 shows the interaction tree used in this scenario. The HLA metamodel provides interaction models that can be connected with arrows indicating inheritance. Two main interaction types are defined: Control and Observation. These represent control commands sent by the control federate and sensor updates sent by the plant federate. Interaction models can contain atoms representing inter-

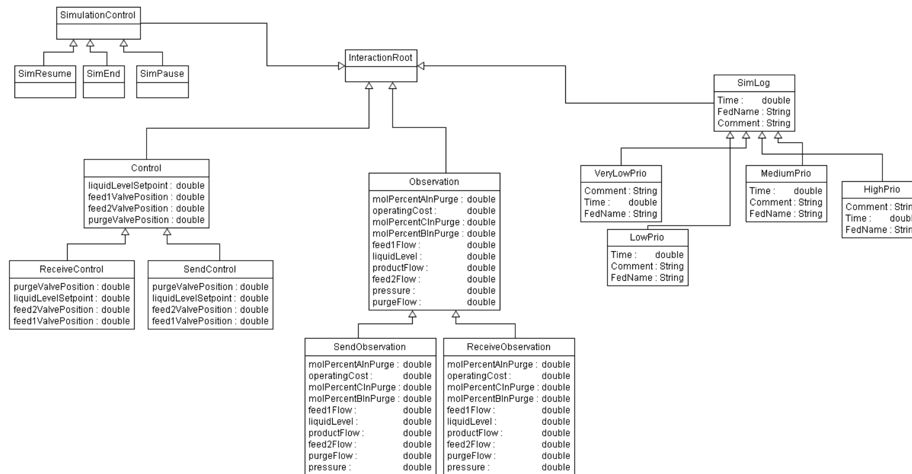


Figure 15: Modeling interactions in GME

action parameters, each with a name and data type. The Observation interaction includes ten parameters, each a double value representing a sensor reading, and the command interaction includes four parameters, each a double value representing a control setpoint. Because both interactions will be sent across a network federate, they must have “Send” and “Receive” variants. By subclassing from a parent class that defines the interaction parameters, they can have the same parameters without having to redefine them for each variant.

This federation contains three federates, each added as a Federate object in the federation model. The plant and controller are represented by Matlab federates and the control network that connects them is represented by an OmnetFederate object. The plant in this scenario sends a single type of message that includes the values for all sensor readings and responds to one message type that includes the values for all actuator setpoints. Because interactions mapping to network messages in C2WindTunnel must use different interaction types for incoming and outgoing messages, four interaction types are shown in figure 16: SendObservation, ReceiveObservation, SendControl, and ReceiveControl. The plant federate uses connections to the SendObservation and the ReceiveControl interactions to show that it publishes

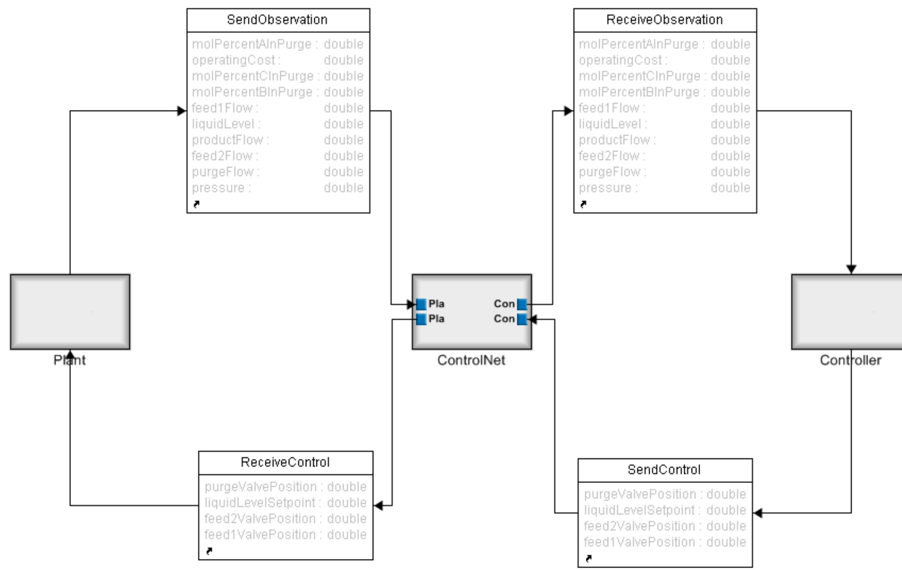


Figure 16: Modeling federates in GME

sensor data and subscribes to control commands. Similarly, the controller federates shows that it publishes control commands and subscribes to sensor updates. The interaction objects are connected to endpoint ports in the OMNeT++ federate to define the entry and exist points of messages through the network. The control network in this case in simple, containing endpoints for the plant and the controller. The endpoint components of the OMNeT++ model demonstrates that the plant and controller federates will participate in the simulated network. Only the network nodes that will be communicating with the other federates in the federation via interaction publishing or subscribing need to be included in the network model. In this case only the plant and controller need to be included with the rest of the network being defined in OMNeT++.

Finally, the deployment information can be filled out for this federation. C2WindTunnel supports large-scale federations using many machines on a LAN. For this case study however, only a single machine was used to run all federates. This deployment model uses three components to model the deployment configuration of a federation execu-

Figure 17: Modeling federation deployment in GME

tion. The Network component contains nodes that lists the machines that will be used in the federation execution to run federates. In this case study, a single `localhost` node is used. The Experiment component allows the modeler to list the federates that will be used in the federation execution. This could be useful if a particular execution wanted to use only a subset of the federates, possibly omitting a logging federate, for instance. In this case, all three federates will be used and as such are included in the Experiment model. Finally, the Deployment model is used to map federates to the machines on which they will run. In this execution, all federates will run on `localhost`, and each federate in the Deployment component is connected accordingly.

At this point, the federation modeling step is complete and federate code can be generated using the model interpreter. This generates a subfolder in the C2WindTunnel directory containing the FOM and annotated FOM, the OMNeT++ files for each network federates in the model, the Matlab files for each network federate in the model, and scripts for starting and ending the federation.

Now the Simulink federates can be created. For each Simulink federate in the federation, the SCADA interpreter creates Matlab blocks that handle publishing and subscribing to data from the HLA. These simulations can simply include the publishing and subscribing blocks and connect them to blocks that implement the

functionality of the federate.

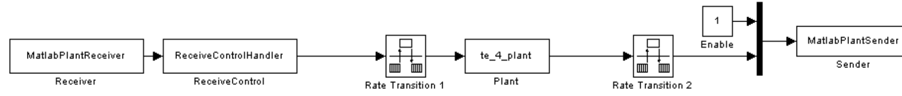


Figure 18: Plant model in Simulink

To create the plant federate, shown in figure 18, a Simulink model of the Tennessee Eastman plant had to be created. A simplified version of the Tennessee Eastman challenge problem was published as Fortran code, so this code was translated into C and used as a Simulink S-Function in the plant model. Incoming control command updates act as input to the plant and the output from the plant is published as a sensor update interaction. The publish and subscribe functions are added to the Simulink federate by including and connecting the MatlabPlantSender and MatlabPlantReceiver blocks that were generated by the SCADA interpreter.

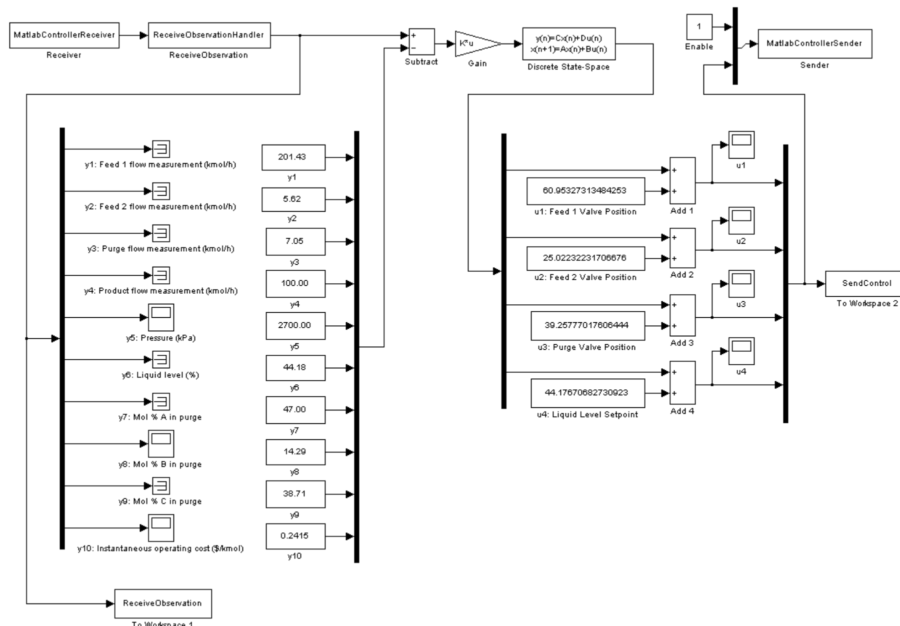


Figure 19: Controller model in Simulink

The controller federate, shown in figure 19 is created in a similar way. The con-

trol algorithm is implemented by a discrete state space block, and scope blocks are included so that control commands and sensor updates can be graphed over the life of the federate execution. As in the plant federate, the MatlabControllerSender and MatlabControllerReceiver blocks provide the interface to the HLA.

The next step is to model the control network in OMNeT++. First, a new OMNeT++ project is created using the generated directory. Nodes representing the plant and controller are already included, so the remaining network nodes and the attacker must be added. In this scenario, a distributed denial of service attack will be simulated. The controller and plant are connected with three routers, which can be added with the OMNeT++ GUI or by editing the `Network.ned` file manually. A subnet of attacking hosts executes the attack by sending a high volume of traffic across the network to a dummy receiving host. The intent of the attack is to overload the middle router in the network, forcing it to drop packets sent by the plant or controller. A simple application is written for the attack nodes that floods their destination with traffic and begins and ends at a user-defined time. Once this is written, the simulation can be compiled, resulting in a single executable in the project directory.

At this point the federation is ready to be run. A bash script is generated to start the federates and the federation manager, a small federate with a simple GUI to allow starting and stopping the federation. Once the federation execution is complete, the results can be viewed in the OMNeT++ and Simulink GUIs. The OMNeT++ logger view allows the user to view statistics about network traffic such as dropped packets while the Simulink scope views allow the user to view the sensor readings and command values.

The liquid level in the plant plotted across the life of the simulation is shown in figure 20. The liquid level is seen to sharply drop throughout the duration of the attack, and then slowly recover when the attack is completed. This is a demonstration of the potential negative effects of negative attacks on SCADA systems. More impor-

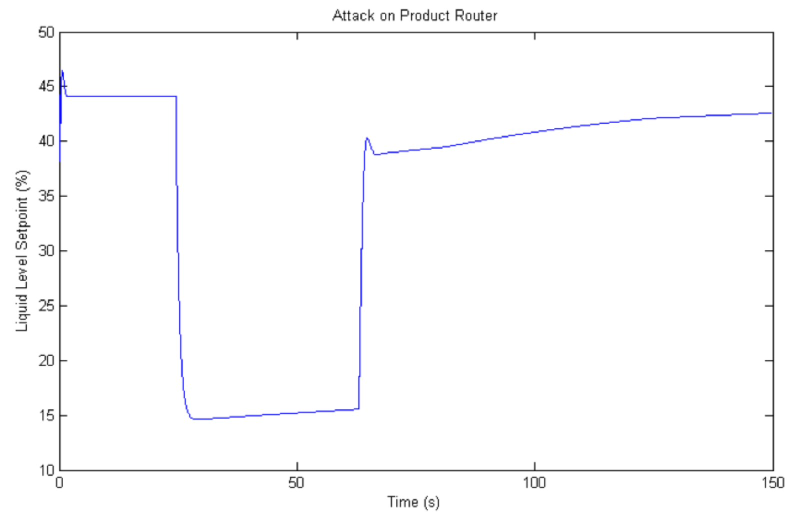


Figure 20: Liquid level setpoint

tantly, though, this case study provides a demonstration of the rapid development of SCADA simulations that can be achieved with the C2WindTunnel platform. By using C2WindTunnel, modelers of SCADA systems can focus their efforts on modeling in the relevant problem domains and not on cumbersome integration issues.

CHAPTER V

CONCLUSION

SCADA system security is an area of growing interest and one in which researchers face significant challenges in developing and testing solutions. In response to these challenges, this thesis details extensions to C2WindTunnel that allow the tool to be used for rapid SCADA simulation development and deployment. Improvements to the C2WindTunnel platform focused on extending the federation modeling environment to provide support for control and sensor messages sent across a control network. To support the expanded capabilities of the modeling environment, integration of the OMNeT++ network simulator is implemented. To allow rapid development of network federates in OMNeT++, automatic interaction conversion and routing is provided by generated user-configurable code. Finally, Windows support for OMNeT++ federates and a Windows installer is provided to allow the tool to be used seamlessly on the Windows operating system, facilitating development and testing on a single machine. The result is a platform that promotes model-based design of SCADA systems so that novel control algorithms and network attack mitigation strategies can be rapidly developed and tested.

Future Work

The C2WindTunnel platform could be extended in a number of ways to provide SCADA system modelers with a more robust modeling environment. In this work, an elegant solution for providing event-based time management in the network federate while maintaining Windows compatibility was not found. This is a clearly desirable property of network federates, however, and a solution would enhance the accuracy of message timing in the federation. This work focused on TCP/IP control networks, although many legacy SCADA systems use fieldbus protocols instead. Support for

fieldbus-based networks would allow more realistic SCADA systems to be developed and tested. To aid in testing the robustness of control algorithms to network based attacks, a library of common attack scenarios could be provided by C2WindTunnel. These could be implemented with a library of host applications, each performing a common attack strategy such as denial of service or man in the middle attacks, and could be added to the network modeler by simply including the host applications. Each of these proposed extensions follows the theme of reducing the design effort required to model SCADA systems by providing modelers with ready to use solutions to commonly encountered scenarios. This furthers the primary goal of C2WindTunnel, to allow rapid development in a model-based environment that hides the underlying federation coordination details.

REFERENCES

[1] GME. www.isis.vanderbilt.edu/Projects/gme/.

[2] Inno Setup. www.jrsoftware.org/isinfo.php.

[3] JSimpleModule. www.omnetpp.org/pmwiki/index.php?n=Main.JSimpleModule.

[4] Matlab. www.mathworks.com/products/matlab/.

[5] MinGW. www.mingw.org.

[6] OMNeT++. www.omnetpp.org.

[7] Portico. www.porticoproject.org.

[8] Simulink. www.mathworks.com/products/simulink/.

[9] SWIG. www.swig.org.

[10] IEEE standard for modeling and simulation (M&S) high level architecture (HLA) – object model template (OMT) specification. *IEEE Std 1516.2-2000*, pages i–130, 2001.

[11] IEEE standard for modeling and simulation (M&S) high level architecture (HLA) – framework and rules. *IEEE Std 1516-2010 (Revision of IEEE Std 1516-2000)*, pages 1–38, 18 2010.

[12] Markus Brandle and Martin Naedele. Security for process control systems: An overview. *IEEE Security and Privacy*, 6:24–29, 2008.

[13] Eric Byres and Justin Lowe. The myths and facts behind cyber security risks for industrial control systems. In *VDE Congress*. VDE Association for Electrical, Electronic Information Technologies, oct. 2004.

[14] Alvaro A. Cárdenas, Saurabh Amin, and Shankar Sastry. Research challenges for the security of control systems. In *Proceedings of the 3rd conference on Hot topics in security*, pages 6:1–6:6, Berkeley, CA, USA, 2008. USENIX Association.

[15] Angelo Corsaro. Quality of service in publish/subscribe middleware. Technical report, SELEX-SI - Roma, 2006.

[16] Judith S. Dahmann, Richard M. Fujimoto, and Richard M. Weatherly. The department of defense high level architecture. In *Proceedings of the 29th Conference on Winter Simulation*, WSC '97, pages 142–149, Washington, DC, USA, 1997. IEEE Computer Society.

[17] A. Daneels and W. Salter. What is SCADA? *Int. Conf. on Accelerator and Large Experimental Physics International Conference on Accelerator and Large Experimental Physics Control Systems*, 1999.

[18] Nicolas Falliere, Liam O Murchu, and Eric Chien. W32.stuxnet dossier. Technical report, Symantec, 2011.

[19] James P. Farwell and Rafal Rohozinski. Stuxnet and the future of cyber war. In *Survival*, volume 53 of *1*, pages 23 – 40, January 2011.

[20] Judith Dahmann Frederick Kuhl, Richard Weatherly. *Creating Computer Simulation Systems*. Prentice Hall PTR, 1999.

[21] Richard M. Fujimoto. Time management in the high level architecture. *SIMULATION*, 71(6):388–400, 1998.

[22] Graham Hemingway, Himanshu Neema, Harmon Nine, Janos Sztipanovits, and Gabor Karsai. Rapid synthesis of high-level architecture-based heterogeneous simulation: a model-based integration approach. *SIMULATION*.

[23] Vinay M. Igure, Sean A. Laughter, and Ronald D. Williams. Security issues in SCADA networks. *Computers Security*, 25(7):498 – 506, 2006.

[24] Katherine L. Morse and Mikel D. Petty. High level architecture data distribution management migration from DoD 1.3 to IEEE 1516: Research articles. *Concurr. Comput. : Pract. Exper.*, 16:1527–1543, December 2004.

[25] N. Lawrence Ricker. Model predictive control of a continuous, nonlinear, two-phase reactor. *Journal of Process Control*, 3(2):109 – 123, 1993.

[26] Jill Slay and Michael Miller. Lessons learned from the Maroochy water breach. In *Critical Infrastructure Protection '07*, pages 73–82, 2007.