Development of the Virtual Flight Deck – Real-Time Simulation Environment

by

Kin Wing Tsui, B.Eng.

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Applied Science

Ottawa-Carleton Institute for
Mechanical and Aerospace Engineering

Department of Mechanical and Aerospace Engineering
Carleton University
Ottawa, Ontario, Canada

April 2009

Your file Votre référence
ISBN: 978-0-494-52033-8
Our file Notre référence
ISBN: 978-0-494-52033-8

Your file Votre référence
ISBN: 978-0-494-52033-8
Our file Notre référence
ISBN: 978-0-494-52033-8

Abstract

Historically, modelling and simulation has supported research and development in safety-critical areas of shipboard operations. One common feature between these simulations is the generation of ship motion. A simulation environment, the Virtual Flight Deck – Real-Time (VFD-RT), was created to assemble existing and new ship motion models. The VFD-RT simulation environment project seeks to develop a suite of applications dedicated to running a variety of flight-deck and related simulations, for a range of simulation goals including research, engineering analysis, and product development.

The VFD-RT environment operates on Windows XP, and makes use of the integrated Windows message system for communication between component applications. The environment architecture has three layers: a layer of input data providers, a middle core application layer, and a layer of output data clients. The core application is responsible for time and data management for the simulation, while providers generate simulation data and clients manipulate the data for analysis, final output, or feedback into the simulation. A set of "use cases" guided the development of the various components of the core application. The providers and clients developed include two ship motion providers (using the SHPR prescribed frequency-domain method and an interactive time-domain method based on ShipMo3D), a ship energy analysis client, a replenishment-at-sea analysis client, a TCP connection client, and the DynamicsViewer 3-D visualization client.

To ensure reliability of the simulation, a set of verification and validation tests were completed. The environment software was tested at the code level through unit testing, at the system level through performance testing, and at the deployment level through validation with three research and development projects.

Dedicated to my Lord and Saviour Jesus Christ, who sustains me and gives me joy in all circumstances, and to my family, who always supports and encourages me wherever I go.

Acknowledgments

First of all, I would like to thank my supervisor, Prof. Rob Langlois for his guidance, patience, and encouragement throughout this project. He has been to me a mentor of excellence and of generous service. His time and attention to this thesis project is immensely appreciated. I would also like to thank Nick Bourgeois for his support with the Flight Deck Motion Display system contributions, and Kevin McTaggart at Defence Research and Development Canada for providing the ShipMo3D software, data, and integration support.

My work on this project would not have been possible without the support from the Department of Mechanical and Aerospace Engineering in the form of a teaching assistantship, from the Natural Sciences and Engineering Research Council Scholarship, and in part from the FDMD Collaborative Research Project funded by the Ontario Centres of Excellence and General Dynamics Canada.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Safe ship flight-deck operations consist of many safety-critical aspects. Examples include helicopter launch, recovery and on-deck manoeuvring, personnel postural stability, and securing of on-deck equipment. Mathematical modelling and computer simulation have supported research and development in these areas through engineering analysis, hardware-in-the-loop equipment development, and other applications. One commonality between the simulations in these areas is the requirement for ship motion corresponding to prevailing seaway parameters. To improve the reusability of the common ship and sea models, the Virtual Flight Deck – Real-Time (VFD-RT) simulation environment software is developed with a vision for a suite of models relevant to research in shipboard operations. The simulation environment was designed with consideration for modularity and applicability to research and development.

This thesis presents an overview of the VFD-RT simulation environment, its design, implementation, and development process. The software executes on the Windows operating system, and has capabilities for real-time and non-real-time simulations, for a variety of applications.

1.1 Background

Dynamic analysis of ship deck operations, aimed at improving safety and operational capability, has been an active field of on-going research in the Applied Dynamics Research

Group at Carleton University. Projects to date have been undertaken in various key areas. Dynamic interface analysis has contributed to the definition of helicopter and equipment securing requirements on ships [1]. Mathematical models have been developed and validated for helicopter manoeuvering on deck [2, 3, 4]. Postural stability research is in progress to increase safety and efficiency of on-deck activities [5]. Blade sailing research has led to comprehensive modelling capability and potential solutions to fuselage and tailboom strikes during rotor engagement and disengagement [6, 7]. Ship motion studies have produced dedicated flight deck motion monitoring equipment [8].

Supporting this variety of research areas are ship modelling and simulation. The common requirement for these areas is an appropriate representation of the seaway, the ship motion, and the environment. One example of a ship and seaway representation is the implementation of a Monte Carlo simulation environment in the Applied Dynamics Research Group, called the Virtual Flight Deck (VFD) [1]. The initial intent for the VFD was to use a probabilistic approach to sea state modelling, combined with deterministic helicopter and ship models, to define mechanical design requirements for shipboard aircraft securing equipment. The spectra of helicopter-ship interface variables relevant to securing were produced by the simulation and analyzed for design and operation purposes, as well as for fatigue life prediction. The simulation time of the VFD is managed by an event coordinator and runs at a faster-than-real-time rate, in order to simulate design life of systems such as the securing equipment.

Efforts to consolidate and reuse common ship and seaway models, combined with a need for a real-time ship simulation to extend support capabilities for on-going research, led to the concept of a real-time version of VFD with the ability to connect existing mathematical ship motion models with existing and subsequently-developed application models, in a way that avoids a repeated implementation of the common models. The time management model in the VFD is revised, leading to the new project of the Virtual Flight Deck – Real-Time (VFD-RT).

Time management is fundamental to simulation, such that key changes affect the entire simulation design. The addition of a timer in the VFD-RT for keeping track of real time

required a complete redesign of the existing simulation environment. The new design is the subject of this thesis.

With regards to the real-time characteristic of the simulation environment, existing requirements for research in shipboard activities have been for "soft" real time - a small degree of tolerance is allowed if the simulation time deviates slightly from true time. True real-time simulation with currently available computers is difficult, if not impossible, since by design computations are performed in processing cycles, and therefore in discrete time. Events that occur between time steps are either ignored or delayed, but cannot be captured in true time. However, with increasing computer processing power, especially in commercially available systems, soft real-time programming is made possible for a variety of applications, including ship simulations.

1.2 Flight Deck Simulation

The VFD-RT simulation environment was developed to model helicopter and ship operations with an architecture tailored for small-scale distributed computing. The development of such an environment must be supported by an operating system capable of precisely measuring the passage of time, while performing all necessary dynamics calculations for all of the bodies in the simulation. Depending on the research area, simulation infrastructures are implemented to emphasize different research priorities. The following literature review covers infrastructures used for ship and flight deck simulation, including an overview of the operating system options for real-time simulations and existing simulation standards, as well as examples of other published real-time simulation projects.

1.2.1 Simulation Infrastructure

Operating Systems

One of the first decisions to make when considering constructing a real-time simulation environment is the choice of an operating system. Some specialized commercially available operating systems are designed for real-time software operations. These dedicated systems

have a high precision in terms of their ability to produce ticks consistently at a given time interval. Their tasks are performed with a high degree of control over the use of the central processing unit (CPU) of the computer, working close to the hardware level. However, these systems are noticeably more expensive than the more widely-available operating systems like Windows. The trade-off, then, is between precision and cost. An investigation was conducted into the literature regarding the possibility of using a widely-available operating system, including Microsoft Windows and Linux, that would provide an optimal trade-off point for a real-time simulation. These operating systems are found to have similar characteristics in their configuration for real-time simulations, particularly the necessity to schedule tasks onto one specific CPU in a multi-core system and to acquire an external hardware timer for real-time operations below the nanosecond level. The findings led to the conclusion that, with a careful choice of software configuration, Microsoft Windows has the relevant balance of precision and cost for a millisecond-precision real-time software simulation [9]. Moreover, this operating system is widely available and can be easily obtained.

Within Windows, there are two software timers available for use in real-time simulations: the multimedia timer and the high-performance counter. The multimedia timer provides millisecond-precision timing, while the high-performance counter gives nanosecond-level precision. However, on multi-core systems where the time values can be retrieved from different cores, there exists a possibility of a difference in the time between two cores, such that two successive timer readings might lead to time appearing to run backwards. Another issue to consider are process priorities. A Windows process is an instance of a program. It can be given a priority value, such that the CPU would perform the tasks from higher priority processes first. On multi-core systems, the simulation applications can be given real-time priority, since other cores can handle other running processes. However, on a single-core system, real-time priority for the simulation timer would prevent other processes, like the user interface, from running properly [9, 10].

Standard Simulation Architectures

Along with operating system considerations, the choice of the simulation architecture is a significant design decision. Specifically, structures that support distributed simulations are becoming increasingly important as the multi-core computer is growing in prominence. In distributed simulations, tasks are scheduled between the different cores such that they are optimized for processing speed. Three existing distributed simulation architectures are most commonly used: High-Level Architecture (HLA), Common Object Request Broker (CORBA), and Remote Method Invocation (RMI) [11].

The HLA is a general purpose architecture with a standardized specification, defined under IEEE standard 1516 [12]. Programs with specialized interface features, called federates, communicate through the Real-time Infrastructure (RTI). The RTI [13] is a program to which federates can subscribe and publish model data in order to coordinate with other parts of the simulation. The HLA federates are language-independent. With the proper programming interface defined by the standard, programs can act as a federate to the simulation. CORBA [14], a more recent development, uses the Interface Definition Language (IDL) to construct the network of simulation parts. The IDL has a syntax similar to that of C++, but defines interfaces rather than implementations. Therefore, CORBA is also language-independent. CORBA uses object request brokers (ORB), software residing in the simulation machines, to manage objects in the simulation. Finally, RMI [15] is a Java-based architecture. Its implementation uses a Java package and consists of invoking methods from another object in the simulation. Java programs have the advantage of being cross-platform, capable of running on different operating systems, through the use of a virtual machine that interfaces the user applications with the computer-specific hardware.

Both CORBA and RMI allow for direct communication between objects. As well, both of them use TCP/IP as a communication protocol, such that they can communicate across the internet. HLA, on the other hand, does not specify the communication protocol, which varies between RTI vendors. These standardized architectures have a common disadvantage; added infrastructure is imposed in order for the architecture to have generic features for

general purpose distributed computing simulations. HLA and CORBA run an additional program to facilitate communication between machines, adding to the overhead cost. RMI is restricted to Java programs. Although Java is a platform-independent language, a virtual machine is used, also adding to the overhead cost.

1.2.2 Shipboard Operation Simulation Systems

Based on design goals and requirements, various systems and frameworks have been developed within the research community and industry over the past years to effectively construct and run simulations related to shipboard operations, including helicopter launch and recovery. A review of these simulation systems was conducted to investigate aspects of their implementation relevant to their versatility in executing different simulations. Five systems, from both academia and industry, were found and are discussed in this section.

HELIFLIGHT

In 2001, the Flight Simulation Laboratory at the University of Liverpool acquired a HELIFLIGHT flight simulation system built by Motionbase plc/Advanced Rotorcraft Technology Inc. The system consists of a flight dynamics modelling software called FLIGHTLAB, a six-degree-of-freedom motion platform with four axes of dynamic control, three computer monitors to display the simulation visualization, which is powered by five computers with 3D Voodoo 3 graphics cards, and one more monitor for the computer-generated instrument panel. The modelling software, FLIGHTLAB, is developed by Advanced Rotorcraft Technology Inc. for rotorcraft simulation using the Blade Element Method. It contains tools to generate non-linear multi-body models with little software coding. The implementation details of FLIGHTLAB are not disclosed due to the proprietary nature of the software. HELIFLIGHT is an immersive flight environment and has been used for a number of academic research projects [16].

MoSART

The Modeling, Simulation, Animation, and Real-Time Control Environment (MoSART) is an educational and research tool for a variety of dynamic systems such as robotic systems, aircraft, helicopters, and other six-degree-of-freedom systems. The system runs on Microsoft Windows, using Visual C++ for the software programming and Direct-3D for the visualization, and can be combined with MATLAB and Simulink for enhanced features. The environment consists of five modules:

- a user interface module for interaction with the user;
- a stand-alone simulation module which solves ordinary differential equations of the system and is implemented within MATLAB/Simulink;
- a Simulink module providing a set of Simulink block diagrams and models related to each of the dynamic systems that can be represented by MoSART;
- a graphical visualization and animation module to display animations, graphs, and data to visualize the simulation; and
- a hardware module allowing communication with connected hardware, if present, through a data acquisition board.

The control system models represented have been developed for general six-degree-of-freedom systems (fixed-wing aircraft, helicopters, unmanned air vehicles (UAVs), missiles, submarines, spacecraft, and satellites) and for various cart-pendulum and robotic systems with instructional and research purposes. Due to its close integration with MATLAB, classical and modern control analyses can be performed using the available MATLAB control toolboxes. The environment uses Windows multimedia timer with a one-millisecond resolution, so that soft real time can be achieved [17, 18].

MulTiSim

MulTiSim is an object-oriented framework for implementing real-time simulations. As a framework, it can be implemented in any number of object-oriented languages that have facilities for encapsulation, data abstraction, inheritance, and dynamic binding. All objects in the simulation are derived from a single "Top Object" and fit into a hierarchy of simulation objects. Each processor in the simulation has a database which contains the information about all the objects in the simulation. When a new object is created locally in one database, a single message is broadcast to the network of processors, with only the information necessary to create this object, so that a copy of that object is created in the database of the other processors. When an object is destroyed, the memory is not deleted for future reuse and to avoid the complexities of garbage collection facilities. This technique is useful for sufficiently simple simulations. However, the accumulation of destroyed objects in memory, serving no immediate purpose in the simulation, can affect simulation performance negatively over time.

The communication packages sent between the processors are independent of object classes so that different object types can communicate with each other. The packages have "structural transparency," meaning that a complex message can be broken down for handling by the separate components of this object. A compound object, built from the communicating entities, is responsible for the message interpretation. Thus objects of different classes can communicate with each other. In terms of data storage, structural transparency also means that access to data in a component can be done either through the compound object or directly to the component itself. One consequence of this property is that the attribute of a component can be viewed as an attribute of the compound object itself, such that a complex compound object can be broken down into its components without affecting the interface with the rest of the simulation. In the simulation environment, a scheduler keeps track of all of the events so that the simulation can be run in real time or non-real time, in continuous or discrete time [19].

FLSIM

The FLSIM package is a low-cost commercial-off-the-shelf (COTS) simulation system designed to be customizable by a non-programmer end user. The architecture consists of three processes: user interface, simulation, and visualization. These processes communicate with each other through an Ethernet connection with the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP), allowing distributed simulation over a network. The processes are also modular, such that they can be independently replaced. Flight dynamic model data used by the package makes the distinction between aerodynamics and programming. The user creates aerodynamic coefficient curves graphically, based on control and stability derivative data. Simulation is driven by the curves created by the user, so that the user does not need to program code for the simulation. The FLSIM package is software, and is deployable with any COTS hardware and visualization, provided that the interface is correctly implemented [20].

DIMSS

The Dynamic Interface Modeling and Simulation System (DIMSS) was developed as part of the Joint Shipboard Helicopter Integration Process (JSHIP) in the U. S. military, with the purpose of defining and expanding the wind-over-deck (WOD) envelope of safe helicopter launch and recovery, through the use of simulation. Due to the critical application of the simulation, the system uses the Vertical Motion Simulator (VMS) at NASA Ames Research Center as part of an extensive verification, validation, and accreditation process. In particular, the UH-60A Blackhawk and the helicopter-carrying amphibious assault ship were chosen as models for this process.

DIMSS did not receive the accreditation for developing the WOD envelope that it first set out to obtain due to a lack of favorable test conditions; instead, the system is currently being used for training [21]. Many of the research results were used in a variety of applications and developments such as pilot workload [21, 22], fidelity standard development [23], and airwake model development [24].

1.2.3 Summary

This review has revealed some details useful for the implementation of the Virtual Flight Deck – Real-Time simulation environment. In terms of distributed simulations, the Windows operating system, using its available multimedia timer, can produce millisecond-precision simulations. Compared to Linux, Windows is more widely available and the required setup for a real-time environment requires approximately the same effort. Existing simulation architectures like HLA, CORBA, and RMI have some merit, but the drawback of the overhead cost makes them less suitable for small scale networks and for expandability. Many existing simulation projects do not use these architectures, but are developed for specialized purposes, including for analysis. Some interesting features to note from these simulators are the concept of structural transparency of simulation objects, separation of the dynamics modelling and programming modules, and use of TCP and UDP for inter-module communication.

1.3 Objectives

Considering the need in the Applied Dynamics Research Group for a simulation environment capable of supporting a variety of research projects in flight deck and related operations, the objective of the Virtual Flight Deck – Real-Time (VFD-RT) simulation environment project is to develop a suite of applications that serves as a versatile control centre for real-time simulation. Specifically, this project aims to:

- design a real-time simulation infrastructure capable of connecting mathematical models of ships and shipboard operations to form the basis of a real-time simulation;
- implement the infrastructure as a simulation environment composed of a set of applications relevant to flight deck operations; and
- verify, validate, and evaluate the environment through cases that demonstrate the real-time performance of the environment and its applicability to a variety of research and development projects.

The outcome of this project is the delivery of a simulation environment software capable of supporting engineering analysis, training simulation projects, and development work resulting from research into shipboard activities.

1.4 Thesis Overview

This chapter has provided an overview of the VFD-RT project and has reviewed existing simulation architectures, their implementation, and design considerations. The objective of the VFD-RT project was described. In Chapter 2, the simulation environment design and implementation is presented in detail. The most significant design considerations and implementation challenges are explained, particularly relating to the real-time characteristic of the simulation. Chapter 3 details the verification and validation process, including an evaluation of the real-time performance of the simulation. The process is outlined from code-level testing to the demonstration of applicability in the research and development setting. Chapter 4 contains concluding remarks and recommendations for future work.

Chapter 2

Design and Technical Implementation

This chapter details the design of the VFD-RT simulation environment and provides a technical description for the implementation. The discussion covers the operating system selection, environment architecture, and simulation design considerations including time management, data management, component communication, and user interfaces.

2.1 Programming Framework

Before designing the simulation environment, a framework is set up to support the development work. This programming framework gives an initial direction to the project and defines the context in which the simulation software is developed. The choices of an operating system, a programming language, and a modelling system are described in this section.

Within this section, and throughout the thesis, the term "application" most often refers to a computer program with a user interface that interacts with the operating system to perform some task for the user. The other usage of this word refers to the use of something in a particular case; research and development, for example, are application areas of simulation in this sense. The context will clarify which definition is appropriate.

2.1.1 Operating System

An investigation was carried out to guide the selection of the operating system for the development of the VFD-RT from among the existing systems, including Microsoft Windows, Unix, and specialized real-time operating systems. The operating system of choice is Microsoft Windows XP. Several factors influenced this selection: event-based execution, availability, a well-known user interface, and time management capabilities.

One main driver for the selection of Windows is its event-based execution of programs. This property enhances performance and control of the VFD-RT simulation, as the user can interact with the user interface, without significant hindrance to the simulation execution. Consequently, the simulation can be manually paused or stopped during the execution. More importantly, the event-based environment allows the passage of time to be controlled by a single application among the many applications comprising the simulation, so that time management is encapsulated. As it monitors the passage of time, the central VFD-RT application can trigger an event in the other applications of the simulation to perform certain operations. Event-based simulation is a key concept of the operation of the VFD-RT simulation environment, which implicates a number of supporting applications that depend on messages from the central application.

Microsoft Windows XP is the operating system in use for all current experiments and projects in the Applied Dynamics Research Group for which the VFD-RT simulation environment is initially designed. Developing under Windows is a cost-effective solution and reduces the chance of complications arising from the integration of the existing experiments and projects with the simulation framework. Windows is also widely used in the industry, so the VFD-RT environment can be effectively integrated on many existing computer networks. According to a study of the operating systems market in March 2009, Windows makes up 88% of the market share [25]. Though designed on Windows XP, the VFD-RT environment is built on the Win32 application programming interface common to all 32-bit Windows versions, ensuring compatibility with a majority of existing Windows versions. Therefore, it can be executed on earlier versions of Windows such as Windows 95, as well as

the newer Windows Vista and the upcoming Windows 7. Furthermore, the design has the flexibility to interoperate with other systems; if some parts of the VFD-RT are required to run on an operating system other than Windows, the simulation can include applications with specialized network communication protocols such as the internet protocol (IP).

As a result of the availability of Windows across the industry, the potential external users of the VFD-RT would have had experience using a Windows program. The use of software buttons, drop-down menus, text edit boxes, and the interaction with the Windows system itself such as closing, maximizing, or minimizing the program, have become common skills for a vast majority of computer users. Even if the user is not familiar with Windows, the user interface appeals to the intuition, with buttons and lists analogous to their real-life counterparts.

In terms of the capabilities of running real-time simulations, real-time deterministic simulations have been shown to be possible on Linux systems without specialized external timers [10]. Moreover, comparative studies between Unix-based operating systems, such as Linux, and Windows reveals that, with careful implementation and system configuration, both systems perform equally well in terms of their time management characteristics in real-time applications – Windows has been shown to have the capabilities for deterministic simulations [9]. The suitability of Windows to real-time operations avoids the need for the acquisition of a costly real-time operating system or external hardware.

2.1.2 Windows Messages

The Windows operating system runs on a message-driven architecture. Every execution of Windows applications depends on the event messages exchanged with the operating system. When a user clicks on a part of the screen, for example, the Windows operating system sends a message to the application which owns that part of the screen, telling the application that the user has clicked a certain button or a certain box. A typical Windows application runs a loop to listen to these messages, until a quit message is received [26].

The "window procedure" is a function that processes the messages in a continuous fashion with a series of conditional statements. When a message is sent to an application,

it is placed in a message queue. The messages are processed within the execution loop one at a time by the window procedure. Depending on the parameters within the message structure, this procedure then calls the relevant entities in the application to perform the task that is commanded by the received message.

2.1.3 Inter-Process Communication Method

The VFD-RT simulation environment employs multiple applications running concurrently and requiring constant communication between them. Consequently, an appropriate inter-process communication (IPC) method must be selected. A process is an instance, or a copy, of the application; Windows can run several applications concurrently by running them on different processes. Windows offers several IPC methods, including the commonly used Clipboard. However, after eliminating some of these methods based on the requirement for the simulation to run in real time, three methods remained: the data copy message, the Dynamic Data Exchange (DDE), and the window socket. The last two methods require the configuration and use of a specialized infrastructure and a particular communication protocol [27]. The clearly-defined protocol and infrastructure make these methods readily implemented, but with a cost to efficiency and scalability, as the overhead can become prohibitive with growing simulation size. On the other hand, the first method, the data copy message, integrates directly with the window procedure that is already running with the application. Since this method does not adhere to a particular protocol apart from the typical structure of a Windows message, protocol design and implementation are flexible. Therefore, the data copy message is chosen for the VFD-RT simulation environment to communicate between the different applications of the simulation.

The data copy message, appearing as *WM_COPYDATA* in the code, is the only Windows message that crosses process boundaries and is designed specifically for a direct transfer of custom data between two Windows applications. This message takes in three parameters:

- the handle, or address, of the destination window;
- the handle of the sending window; and

- a data copy structure, *COPYDATASTRUCT*.

The data copy structure itself contains:

- an integer specifying the type of data being sent;
- the size, in bytes, of the data to be sent; and
- a pointer to the data to be sent.

Communication between the applications in the VFD-RT simulation environment is accomplished through the data copy message with parameters set to different values depending on the purpose of the communication.

2.1.4 Windows Application Programming Interface and Programming Language

The programming language used to develop the VFD-RT simulation environment is Visual C++, using Visual Studio .NET 2003 as the development environment. This decision is driven by the need for a language that has a combination of direct coupling of the program source code with the window procedure, a high-degree of control over the processing of Windows messages, and the availability of tools for the design of a simple user-interface. Compared to other popular languages like Java and Python, which make use of virtual machines, C++ has relatively low overhead cost. In particular, since the simulation environment is to be designed for a Windows system, the need for a cross-platform language is unnecessary and can degrade real-time performance. If parts of the simulation must run on a different operating system, provisions are available for communication to these parts, with some extra overhead cost. The Visual Studio development environment is also already available within the Applied Dynamics Research Group, so it is also a cost-effective solution.

For programming in the Windows environment, the Windows application programming interface (API) is extensively used. The Windows API contains a number of functions and structures related to the Windows operating system, including Windows messages. Since

applications process messages from the operating system, the API is necessary for running applications. Additionally, the API simplifies the graphical user interface implementation, so that elements like buttons and lists can be manipulated using messages.

2.1.5 Unified Modelling Language

The development of each VFD-RT simulation environment application, from initial definition to source code, follows a few common steps.

1. Define the purpose of the application in relation to the VFD-RT simulation environment.
2. Define the design cases of the application.
3. Develop a component-level software architecture of the application.
4. Refine the architecture by breaking down components into object-oriented classes.
5. List the related classes for each design case defined.
6. Design a sequence of function calls for each design case to accomplish the purpose of the case.
7. Refine the model of classes and sequences for efficiency and performance.
8. Validate the model with the intended purpose of the application.
9. Interpret the model into Visual C++ source code.

These steps are not necessarily sequential, as later steps may require an iterative process or the re-execution of earlier steps.

To provide a framework for this modelling development, the unified modelling language (UML) is used. UML offers a formal specification of the application model, and has a number of advantages. Among them are: distinguishing models for structures from models for activities, facilitating model validation, and ensuring that all design cases are covered.

UML is well suited for object-oriented programming, as the language is designed with the object-oriented philosophy.

UML specifies certain diagrams for particular development tasks, such as:

- use-case diagrams, to list the design cases considered for the application;
- class diagrams, to summarize the application architecture in terms of object-oriented classes; and
- collaboration diagrams, to model activities between classes.

These diagrams are defined in the UML specification produced by the Object Management Group (OMG) [28]. The VFD-RT simulation environment uses UML version 2.0 for its modelling. The open-source software StarUML was used to produce the UML diagrams for the VFD-RT environment applications. StarUML has a library of templates for these diagrams, and has the ability to generate code from a class diagram. Furthermore, it is an inexpensive and simple tool suitable for the modelling needs of the VFD-RT environment.

2.2 VFD-RT Simulation Environment Overview

The VFD-RT simulation environment is designed to perform real-time simulation for a variety of engineering purposes, particularly related to shipboard helicopter operation and associated tasks. The environment can include mathematical models for maritime weather and ship motion, together with a number of simulation data output devices, such as a motion platform, video projection, and specialized equipment. Connecting these components, at the core of the simulation environment, is the VFD-RT application, which manages the data flow throughout the simulation. The VFD-RT application requests data input, formats the data, controls the passage of time, and coordinates the data distribution to output devices. Further, the VFD-RT supports tight coupling of models.

2.2.1 Concept of Operation

The general software architecture is divided into three layers. The input layer provides data using existing mathematical models of the simulated objects and environment. These models are developed independently. They are adopted into the VFD-RT environment through the use of wrapping programs that handle the connection with the core application, and execute the model using available simulation data. Real-time inputs, such as human commands, would be accepted through the input layer. The middle layer, the VFD-RT application, keeps track of the passage of time and coordinates data distribution to the output layer. The output layer relays the results to the user through various media, including the user interface, video projectors, motion platforms, or other data readers. Figure 2.1 shows these layers as they are set up for a flight deck simulation.

The connection between the various layers is initially established through a request from the application downstream of the data flow. Thus, the middle layer application is responsible for making the initial connection request to all of the input layer applications in order to receive data from them, and every output layer application is responsible for making the connection request to the VFD-RT application so that the output application can receive the simulation data. The main reason for an upstream-connection scheme is to give connection control to the applications that use the data, so that they can disconnect or connect as they require. A more detailed explanation is provided in Section 2.3.5.

Time passage is an essential element to control in real-time simulations. Therefore, the centralized simulation architecture, with the VFD-RT application in the middle, is advantageous over a distributed architecture in that only one component keeps track of the time, so as to avoid the need for clock synchronizations between all of the simulation components. In the centralized simulation architecture, all components are linked to a central software that can keep track of time; on the other hand, a distributed simulation architecture requires that each component communicates its state, including information about its internal simulation clock, with the other simulation components individually. Clock synchronization requires messages extra to those carrying simulation data, increasing message traffic

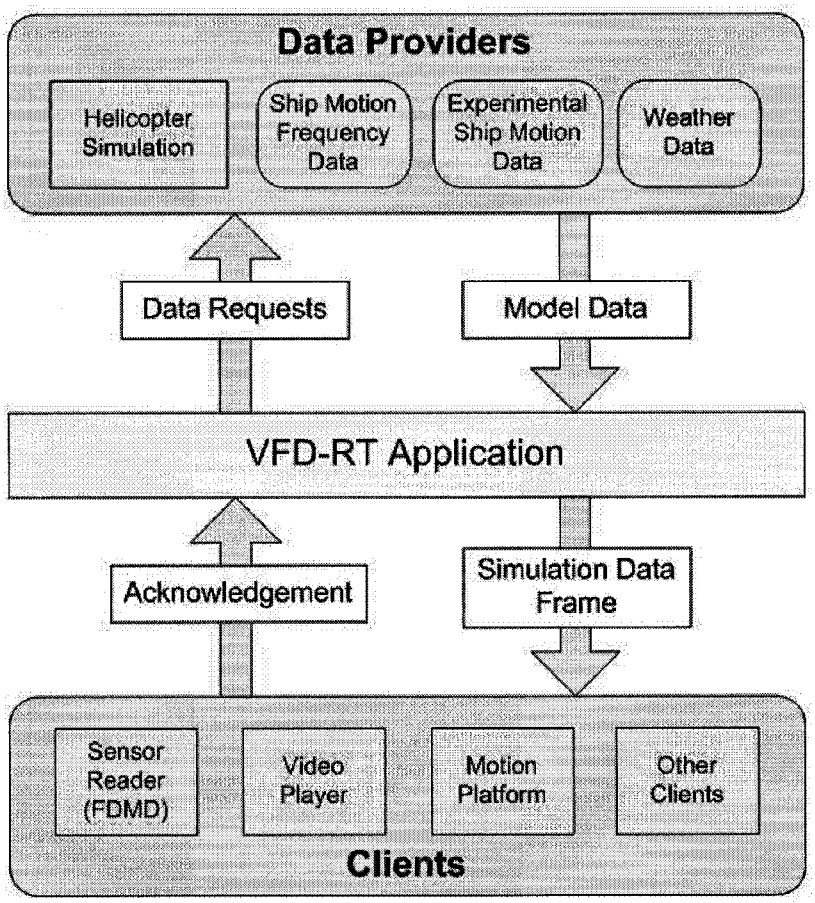

Figure 2.1: Example showing the general layered architecture of the VFD-RT simulation environment.

and decreasing real-time performance. Moreover, a centralized architecture offers a single location to control all entities during the simulation so that the whole simulation can be stopped quickly if required.

The choice of having separate input and output layers is based on the concept of separating data requests from data services. Output applications only receive data requests, while clients receive only the data served by the VFD-RT application. This distinction reduces unnecessary traffic, in a way that each data request to the input layer does not need to carry unused simulation output data as well. It should be noted that one application can be both in the input and output layers – the distinction between layers is in the data structure that is exchanged. An application can process both data requests and data services; the modules within this application that handles data requests would be considered as part of the data input layer, whereas the modules that receives data services would be part of the data output layer.

2.2.2 Data Input Layer

The input layer of the VFD-RT simulation environment is composed of applications that produce simulation data from mathematical models and from the simulation time value given by the VFD-RT application. Each of the input applications, called "data providers", waits for a data request from the VFD-RT application before performing any calculations. The data request contains a timestamp, which the data provider applies to its internal model and produces the appropriate output data. One example of a data provider is the Ship Motion Provider (SMP), which generates ship motion through applying the simulation time to a time-domain ship motion evaluation algorithm based on frequency-domain ship motion response data, where the response spectrum data are read from an external file.

For initiating and closing communication between the data provider and the middle layer, the VFD-RT application is responsible for requesting the connection to the data providers. Once the request is made and acknowledged, the connection is established.

In referring to the relationship between the input layer and the VFD-RT application, the term "provider" is used to distinguish the data source from the data destination, which

is the "receiver", referring to the VFD-RT application. The provider-receiver relationship is associated with the input layer and is distinct from the "server-client" relationship that is associated with the output layer.

2.2.3 Data Output Layer

The output layer of the VFD-RT simulation environment is composed of data clients that pass on the VFD-RT application simulation data to output devices or to other applications. Each client waits for the connected middle layer to send an output data copy message, which contains the simulation data. The client then formats the data as necessary and sends the data along to output devices, such as the user interface, a video projector, or a motion platform. An example of a VFD-RT data client is the DynamicsViewer, an application designed to visualize the simulation data through 3-D modelling of the environment and moving objects within the environment. Simulation data can also be applied directly to mathematical models in real time, or can be recorded to a file for further analysis. For example, as the simulation is running, the values for the kinetic and potential energy of a ship can be calculated, using the ship motion output of the simulation, and then saved in a file.

In referring to the relationship between the output layer and the VFD-RT application, the term "client" is used to distinguish the data source from the data destination, which is the "server", referring to the VFD-RT application. The server-client relationship is associated with the output layer and is distinct from the "provider-receiver" relationship that is associated with the input layer, to avoid confusion between the interactions with the different layers.

An application that is in both the input and the output layers typically performs some operation on the simulation output and feeds this data back into the simulation. Such mixed "provider-client" application uses internal memory to pass data from its client module to its provider module, with each of the modules performing its corresponding task of either receiving simulation output or responding to data requests.

2.2.4 Simulation Execution

To start a simulation, the user must perform the following steps in the VFD-RT application.

1. Start all necessary providers.
2. Start the VFD-RT application.
3. Connect the VFD-RT application to all the providers, one at a time.
4. Start all necessary clients.
5. Connect each client to the VFD-RT.
6. Set the simulation frequency and other options in the VFD-RT application main window.
7. Press the "Start" button in the main window.

These steps can be reordered, with care, at the convenience of the user without detrimental effects on the simulation run. However, the design of the application was based on the above sequence of steps.

Once the simulation has started, the VFD-RT application sends a data request to all the providers and waits for a reply. The data is transferred through the Windows data copy message. At each time frame, when the simulation has determined that the next set of data should be sent out based on a predefined simulation frequency, the application runs through the following steps.

1. Sends the simulation data in the current frame to all the clients.
2. Sends out a data request to all the providers with the timestamp of the next frame.
3. Increments the frame number.

Although designed for real-time simulations, the VFD-RT has the capability of running faster or slower than real time.

A number of providers and clients – some generic, others more specialized – were developed, along with the VFD-RT application. To demonstrate the implementation of providers and clients, the following sections discuss the VFD-RT core application, the Ship Motion Provider (SMP) application, and the DynamicsViewer client.

2.3 VFD-RT Application

The VFD-RT application is the central element of the VFD-RT simulation environment and acts as a control centre for the simulation. It is designed as a Windows API application that runs on the Windows XP operating system. It performs four main tasks in the simulation:

- request input data from data providers;
- arrange the input data into a coherent data frame;
- manage the passage of time so that the simulation runs properly; and
- send the data frame to clients.

The execution of the VFD-RT application is event-based, as in any standard Windows application. Events are triggered by the user interface, by the timer, or by other applications, through Windows messages. Communication to and from other applications, such as data providers or clients, is accomplished via the inter-process data copy message, a simple and flexible Windows communication method.

In the framework of the simulation environment, the VFD-RT application controls the execution of the simulation, by receiving and serving data at a predefined frequency. Besides simulation management, the application is responsible for requesting connection to data providers, requesting data from the providers, and maintaining a list of connected providers and clients. It has the ability to disconnect providers or clients, by the press of a button in the user interface.

2.3.1 Windows Management

Since the VFD-RT application uses object-oriented programming, modularity is a key driver in deciding the division of responsibility between various windows in the application. A Windows application can launch more than one window, some of which can be hidden. By using hidden windows, the application can dedicate some of them to certain tasks. The VFD-RT application operates using two windows: one, the application window, is a hidden window responsible for receiving messages from providers and clients; the other, the main window, handles all messages related to the user interface.

The application also runs other windows at various times during its operation, in the form of dialog boxes. A dialog box is a window template that comes with the Windows API and that simplifies the design of the graphical user interface. Dialog boxes are specifically designed to interact with the user. A user interface template can be designed graphically using tools such as Visual Studio and can then be applied to a dialog box.

In its implementation, the main window of the VFD-RT itself uses a dialog box. Other dialog boxes appear in response to particular commands, namely for:

- selecting providers to which to connect;
- receiving user input of the ship motion parameters when generating new ship motion; and
- showing the ship motion generation progress on a progress bar.

The hidden application window and the visible main window of the VFD-RT application are distinguished primarily through their different window procedures. They will be discussed in the following sections.

Application Window

The VFD-RT core has an application window which serves as the communication module. To accomplish this task, the application must register a user-defined window procedure with the operating system, in order to create windows and receive messages. Once registered,

the application creates a hidden window titled "Virtual Flight Deck – Real Time." This window is the application window responsible for exchanging messages with providers and clients.

The window procedure in the VFD-RT application window defines the code to process three specific messages. They are:

- the data copy message for communicating with other applications;
- the close message for closing the window; and
- the destroy message for terminating the application.

The last two are essential to the normal operation of the application. Therefore, interpreting the data copy message is the specialized task of the application window. When receiving the data copy message, the procedure interprets it by calling a function that determines how to process the message data, depending on whether it is a connection request or some input provider data.

Main Window

The main window for the VFD-RT application is made for interacting with user commands to the application, such as simulation control or provider or client connection. It is comprised of a window procedure and the user interface detailed in Section 2.3.2.

The main window is a modeless dialog box, which means the window uses the dialog box template of the Windows API but has a customized window procedure that can run concurrently with the window procedure of the application window. In contrast, the window procedure of a modal dialog box must end before the application window procedure is allowed to receive further messages. The modeless dialog box is useful, not only because it allows both the application window and the main window to run at the same time, but also because the function to create such a box also provides a window handle, which is the address of that window, so that the rest of the application can also access and manipulate the dialog box elements – for displaying data, for example. Visual Studio provides facilities

for designing the dialog box templates graphically with ease, which is another reason for using a modeless dialog box for the user interface of the VFD-RT application.

Three messages are processed by the window procedure of the main window. They are:

- the dialog initialization message for initializing the dialog box;
- the command message for handling interactions with the user, including button presses and drop-list selection changes; and
- the close message for closing the dialog box.

As suggested by the messages, the main window is designed for the purposes of interacting with the user through responding to messages coming from the user interface.

Class-based Window Procedures

When Microsoft first released Windows API, applications had a global window procedure, not belonging to any class structures. However, in the object-oriented environment of the VFD-RT application, the window procedures should be placed inside class structures of the application code. This concept of encapsulation is part of the object-oriented programming philosophy, to organize the source code more coherently. Some additional codes were necessary to allow this placement.

The window procedure must be declared as a static class method, meaning that only one copy of that window procedure can exist for that class, even though multiple instances of the class may be in use. By declaring the procedure static, the same copy of that class method would exist in all the class instances. This feature is required for use with the Windows API.

To allow the static window procedure to access the members and methods of the class to which it belongs, a pointer to this class is stored in extra window memory, using the *SetWindowLongPtr()* function of the API, and is retrieved at every message processing, using the *GetWindowLongPtr()*. Since the same copy of that window procedure exists for all class instances, the procedure cannot distinguish between the various instances. Therefore,

a pointer to the class is stored in extra window memory when the window procedure is created, and is retrieved when a message is processed. The pointer would point to the instance of the class which is using that particular window procedure. This pointer is stored in memory along with the handle to the window corresponding to that class instance, so that the window procedure can differentiate between the various instances.

A consequence of declaring the window procedure static in a class is that the procedure can only access public members and methods of that class. Therefore, all the methods called by the window procedure in its class are declared public, rather than private. Without the private declaration, these methods are allowed to be called by other classes, making them vulnerable to misuse. The documentation of the code must hence provide clear warning. As well, the code is designed with that potential danger in mind. Data checks are included to ensure that the values of the variables to be used fall within the appropriate range.

The window procedures for both the application window and the main window are static methods located in the application root class, as they both require a top-level access to all of the features of the VFD-RT application.

2.3.2 User Interface

The user interface of the VFD-RT is divided into 3 sections:

- the simulation group on the left, which controls the execution of the simulation;
- the provider group in the middle, which maintains a list of connected providers; and
- the client group on the right, which maintains a list of connected clients.

The user interface and its various parts are shown in Figure 2.2.

Simulation Group

The simulation group is dedicated to controlling the execution of the simulation, as well as providing additional simulation-related features. It is positioned on the left third of the

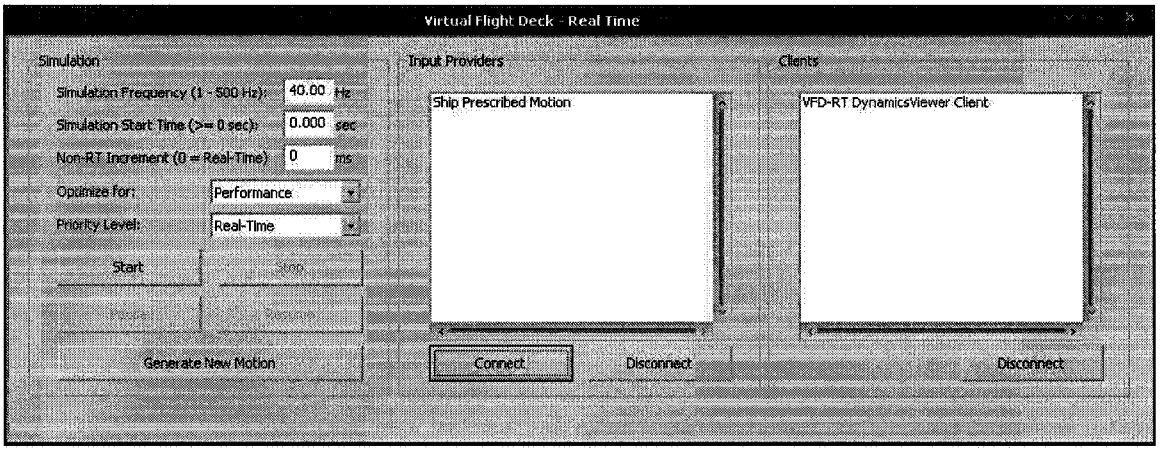

Figure 2.2: VFD-RT application user interface.

window and consists of five buttons, three edit boxes and two drop-down lists, enabled or disabled appropriately at various stages of the simulation.

Four of the buttons control the simulation run: start, pause, resume (from pause), and stop. The simulation will not start until at least one provider is connected, since the simulation cannot be run without input data. Initially, the start button is enabled while the other three are disabled. Upon starting, the start button disables accordingly, while pause and stop become the available options. Resume only enables during pause and, at any point in the simulation execution, paused or not, the stop button is available.

An additional button provides the choice of requesting new ship motion to be generated from a selected provider of ship motion data. A ship motion for a particular ship and wave setting may initially be loaded in the provider; if a simulation run requires a new ship heading, for example, this button can be used to generate a new set of ship motion data. Upon pressing the button, a dialog box will appear asking for several parameters, namely wave height, ship heading relative to the principal wave direction, and ship speed, and also for the provider name, selected from a list of connected providers. Special care is taken so that the drop-list of connected providers does not overlap any buttons in the dialog box, to avoid unintentional pressing of the buttons during selection.

The first of the three edit boxes in the simulation group sets the target simulation frequency, ranging from 1 Hz to 500 Hz. The simulation frequency determines how often a

data frame is sent to the clients. The edit box accepts floating point values, and will set the target value to the closest boundary if the value exceeds the limits (1 Hz or 500 Hz). Due to the resolution of the multimedia timer and processing limitations of the application, a target simulation frequency higher than 500 Hz may not be accurately achieved. A more detailed discussion of the timing limitations can be found in Section 2.3.3.

The second edit box sets the initial timestamp of the simulation run. This parameter is useful when reading, for example, a ship motion file that starts with a non-zero timestamp, or when attempting to reproduce a simulated situation around a particular time frame. The box takes in a floating point value.

The third edit box sets the time increment at every simulation loop for non-real-time simulations. The VFD-RT is capable of running non-real-time simulations, such that time is incremented by a user-defined value at every simulation loop. The third edit box sets this time increment value. If the value is zero, the VFD-RT assumes real-time simulation. If the value is non-zero, the simulation frequency edit box is ignored, and time is incremented at every simulation loop. The actual loop speed for non-real-time simulation will depend on the computer used, since time is incremented as soon as one simulation loop is completed.

Two drop-lists provide additional options for the simulation. The first list allows the user to select between optimization for performance or for usability. When optimized for performance, the simulation timing is valued over the user interface operation, so that Windows messages related to the simulation are processed first before messages from the user interface. Thus, performance is increased, but the user interface will be noticeably slow to respond during a simulation run. On the other hand, if the simulation is optimized for usability, the user interface responds quickly to inputs, and therefore allows for a higher degree of control over the simulation run through manually pausing or stopping. The second list gives the option of setting the application priority to normal or real-time, which affects the speed at which a command from the application is processed by the computer processor. Priorities are discussed in further details in Section 2.3.3.

Provider Group

The provider group maintains a list of the providers connected to the VFD-RT application. It is located in the middle of the window and has controls for requesting a connection to providers that run on the same operating system as the VFD-RT, and for disconnecting providers that are already connected. Providers are designed to be connected before the simulation starts, as part of the initial setup. Once the simulation has started, the connection buttons are disabled. The reason is that the memory for the data frame in the VFD-RT application that holds the incoming provider data is dynamically allocated and sized to the number of connected providers. Dynamically allocating memory during the simulation run would impede the simulation performance. Additionally, if provider disconnection was allowed, any client whose operation depends on the disconnecting provider could experience an unexpected loss in data flow and could compromise the safe use of the client data.

When pressed, the "Connect" button shows a dialog box listing all providers running as an application on the same operating system as the VFD-RT application. The provider titles can then be selected and connected one at a time.

In the main window of the VFD-RT application, the provider can be selected from the list in the provider group, so that it can be disconnected from the simulation through the "Disconnect" button.

Client Group

The client group, on the right-hand side of the main window, maintains a list of the clients connected to the VFD-RT application. The purpose of the client group is to monitor the connection status of the clients to the VFD-RT application. The group has a "Disconnect" button that can disconnect clients. The disconnection of the client is possible at any time during the simulation run. Unlike the providers, the client connections do not affect the size of the data frame, and the disconnection of one client will only affect the data use of that particular client.

In the case of the mixed provider-client applications, each side of the application must

be connected separately: the provider module is connected through the "Connect" button in the VFD-RT main window, and the client module is connected through a button on the provider-client application. In this manner, the same application appears in both the provider list and the client list.

2.3.3 Time management

As a real-time simulation environment, VFD-RT is required to keep track of time with a degree of fidelity to the real passage of time. An appropriate timer must be chosen to accomplish this purpose.

Timer Selection

In the Windows API, three timing methods were considered:

- the Windows timer message that sends a *WM_TIMER* message at regular intervals;
- the multimedia timer, *timeGetTime()*; and
- the high-resolution timer, *QueryPerformanceCounter()*.

To compare the differences, the measure of resolution is used, defined as the time interval between successive ticks of the timer. This measure is chosen because the time between simulation data being sent to clients, and requests to providers, is directly tied to the time interval of the timer, and the resolution value is proportional to the amount of deviation from the actual send or request time. A smaller resolution value means a larger range of simulation frequency.

The Windows timer message is a logical preliminary candidate as all Windows applications have a procedure to handle messages. However, its resolution is approximately 55 milliseconds [29], resulting in a simulation frequency of about 18.2 Hz. This resolution varies greatly [9]. The timer message is thus inadequate for real-time simulations.

The *timeGetTime()* multimedia timer has a resolution of less than 2 milliseconds [30], allowing the simulation frequency to achieve of 500 Hz if necessary. This timer is often used

for determining an elapsed time, particularly of the execution of a piece of code. Monitoring elapsed time is appropriate because the VFD-RT simulation is driven by successive events, namely data sending and data requests, separated by a user-defined fixed time interval. One drawback is that the timer stores time in a 32-bit value, such that the timer resets to 0 after approximately 50 days of continuous execution. For the VFD-RT simulation environment, however, which is not expected to run for that length of time, the multimedia timer is a potential candidate for VFD-RT time management.

The *QueryPerformanceCounter()* high-performance timer works directly with hardware devices, such as the motherboard and the central processing unit (CPU), to obtain a time with resolution in the tens of microseconds [30]. The resolution below millisecond levels is an attractive feature. Like the multimedia timer, the high-performance timer is designed for benchmarking elapsed time to execute a piece of code. However, there is a performance issue related to this method on multi-processor or multi-core processor systems: if the CPU core timers are offset due to a bug, two consecutive queries to the timer may yield a negative time change. With increasing use of multi-core processor systems, the VFD-RT must be designed against such conditions. A solution to the problem is to ensure that the time is retrieved from only one particular CPU core, which in turn may limit extensibility of the VFD-RT to use multiple CPUs.

For the VFD-RT, the multimedia timer was chosen because it is simpler and less limiting in its implementation than the high-performance timer, and the maximum simulation rate is sufficiently high. Furthermore, a wrapper class, called *Timer*, can be written for the multimedia timer, following the object-oriented encapsulation standard of the application.

Process Time Considerations

Apart from the timer selection, the time to process application messages must also be considered. Specifically, the number of providers and clients connected to a running VFD-RT application affects the simulation performance. Windows API offers a *SendMessage()* function that tells the recipient to process the message and return immediately. This method ensures message delivery and processing, but the application will hold at the *SendMessage()*

function until the message is successfully returned. To minimize delay, by default a provider or client that has received a message from the VFD-RT application makes a local copy of the necessary data from the data copy message and returns before further data processing, in order to free the VFD-RT application from waiting too long for a return. Still, the designer of the provider or client has the option of processing the data immediately if it is deemed necessary. In this way, the processing time for a data copy message is limited.

The implementation in the VFD-RT application employs a variant of the *SendMessage()* function, the *SendMessageTimeout()* function, which returns if the message recipient takes longer than a given amount of time. This amount of time is set to 1 millisecond, the minimum possible. Because of this feature, a provider or client that fails to run normally will not completely disrupt the running of the rest of the simulation. The implementation on the side of the provider or client, located in the window procedure, consists of the following steps. When a data copy message is received from the VFD-RT core, the application makes a local copy of the message content, posts or sends itself a data reception message, and then returns. After returning, the provider or client reads the data reception message from its message queue and performs tasks with the recently-stored local copy of the data. When referring to the data reception message, sending means using the *SendMessage()* function, while posting means using the *PostMessage()* function, which only puts the message in the message queue without waiting until the message is returned by the recipient. By default, the posting function is used, but the provider or client designer can modify this convention as required.

To further enhance simulation timing performance, the VFD-RT application has the option of setting its thread and process priority to real-time level during the simulation run. In Windows, the priority value affects the scheduling of tasks for the CPU, such that if two applications, one with normal and another with higher-than-normal priority, have tasks to be processed by the CPU, the task from the higher-than-normal application will be processed first. Real-time priority is the highest level of priority. Because running the simulation at that level can possibly cause other programs and even the operating system in some cases to be unresponsive, VFD-RT provides the option of lowering the simulation

run thread and process priority to normal, which can cause some delays depending on the tasks present in the system.

Additionally, efforts were made so that the time required for processing any message is less than one simulation period. After every data send or request, the variable *mNextSendTimeInMilliseconds* is updated with the value of the next data send time. The timer is then looped and checked against this variable until the time is reached or passed. Consequently, send or request events may occur slightly later than the ideal event time. However, this looped checking involves only tens of CPU cycles, depending on the CPU, and was timed by the high-performance timer to be in the microsecond level for each loop on a nominally 1.6-GHz CPU – a significantly smaller time value than the simulation period is allowed to be achieved. Any delay of event occurrences also does not accumulate, since it is reset at every simulation loop, and has no significant detrimental effects on the simulation execution for the foreseeable implementations, so the current time management method is adequate.

2.3.4 Data management

The data in a VFD-RT simulation are generated at the providers. The generated data proceeds to the VFD-RT application and ends at the clients, which performs specific tasks with the simulation data or feeds the data back into the simulation through a provider module. When it reaches the VFD-RT application, the data must be organized in a structure in a way to maintain traceability to the data source. A simulation data frame is the structure that organizes the data and defines the packet that is sent in a message to the clients of the simulation.

Data Frame

The simulation data frame is a block of memory in the VFD-RT application dedicated to keeping the provider data in an organized manner and allowing an efficient data sending process to clients. When provider data arrives in a message, an identifier in the message gives access to the specific location within the data frame where the data from that provider is to be copied.

The data frame is made up of a header and a finite number of blocks, bundled in *State* classes. Each *State* class consists of 20 values: one 32-bit value for the data type identifier, an array of 32 8-bit characters for the data source name, and 18 64-bit double-precision data values, totaling 1440 bits per *State*. The data type identifier corresponds to a ship, a helicopter, weather, or a generic data packet. Depending on the data type the client may interpret the data differently, since for example velocities and accelerations are appropriate for vehicles, but not for weather. The data source name describes the nature of the data; it may be the name of a specific ship being modelled, or a description of the associated generic data. The 32-character limit allows enough variation of data descriptions while limiting the amount of data sent to minimize the amount of data copying between applications. The use of dynamically-allocated memory to vary the length of this character string was considered, but deemed less efficient. A varying name string size requires detailed string parsing on the part of the client, a logically more complex procedure, involving more steps, than a direct data copying followed by a data cast to a fixed-size structure so that the location of specific data is already known. Therefore, a static 32-character data source name is determined to be sufficient. The 18 double-precision values are designed for the position, velocity and acceleration of a 6-degrees-of-freedom simulated rigid body like a ship or a helicopter.

These *State* classes are part of the VFD-RT data frame, of which Table 2.1 shows the format.

Table 2.1: Structure of the VFD-RT simulation data frame.

Section	Variable Type	Size	Variable Name
Header	Handle to window	64 bits	Server handle
	Handle to window	64 bits	Client handle
	Double-precision value	64 bits	Timestamp
	Integer	32 bits	Frame number
	Integer	32 bits	Number of states (n)
Data	States	n × 1440 bits	Simulation states

The purpose of the header section is to provide additional data surrounding the *State*

structure, as well as information to check that the message is routed to the correct recipient. The data frame is implemented so that, in each simulation loop, the provider data is copied into the corresponding *State* structure, and then the data frame is sent to each client. The header data can therefore be used by the client for a message routing check, by comparing the message destination client handle with its own handle. As well, the client can make use of the data surrounding the *State* class, such as the frame number or the name of the server that can be retrieved using the server handle. The window handle is a data type defined in the Windows API representing a memory address of the window, and is 64 bits long. The timestamp is a double-precision value coming directly from the simulation timer. The frame number and the number of states are signed integers to allow for the possibility of building in error codes in the negative range.

The frame variables are accessed by declaring pointers to the corresponding fixed locations in the memory block. This frame structure is maintained when the data is sent to clients.

Data Flow

During a simulation run, the generated data flows through the VFD-RT application in several steps. Between the data arriving into the VFD-RT application as a message and the data being copied to a client, the key steps are:

1. to copy and send the data frame to all the clients via the data copy message;
2. to request a data message from providers for the next data frame;
3. to check for message routing errors by verifying that the provider and receiver information in the message header corresponds to the information determined by the initial connection messages;
4. to verify that the size of the provider data is as expected, depending on the data type; and
5. to copy the data values to the corresponding location in the simulation data frame.

These steps are looped during the execution of the simulation. The data sending proceeds first before the data requests to providers because the data sending is more time-sensitive. The provider generates data depending on the timestamp rather than the actual time it received the request message, so the task can be performed anytime before the next data sending time. As it is, then, data requests are not sensitive to small delays within one simulation loop. On the other hand, clients may expect data to come at a given frequency for real-time simulations, so sending data must take precedence to minimize delays between the desired and actual sending times.

2.3.5 Communication Management

The applications in the VFD-RT simulation environment communicate through Windows messages. They are defined in the Windows API and are the basis for programming in Windows. While most messages are passed from the operating system to the application, applications can send data to another application through the data copy message, *WM_COPYDATA*. A communication protocol has been developed around this message for the VFD-RT simulation environment.

As a convention, the applications in the simulation that are downstream of the data flow are responsible for requesting data from, or connection to, upstream applications. In general, therefore, the applications that receive simulation data initiate communication with the data source applications. The reason for having the downstream of the data flow responsible for the connection is because those are the applications that will require the data. If this application no longer needs new simulation data, or requires data from a different source, it can disconnect from the original source, without depending on operations from that source. Ultimately, since the simulation is used to produce data useful to the user, the user is the last downstream "client" of the simulation, so this upstream-connection scheme gives the connection control to the user.

Data Copy Message Structures

The data copy message requires three parameters: a handle to the destination application, a handle to the sending application, and a *COPYDATASTRUCT* data structure which contains the simulation data, data size in bytes, and a user-defined integer indicating the type of data structure being sent. This user-defined integer can represent client connection, client data, provider connection, provider data, or new ship motion requests. This message is sent through the *SendMessageTimeout()* function of the Windows API, in the same way messages are sent to window elements through the operating system.

When compiling the data structures using Visual Studio .NET 2003, the default data structure alignment is 8 bytes. In other words, each value in the data structure must fit into a memory space with a size that is a multiple of 8 bytes; otherwise some padding would be added to fill in the space. To minimize the data structure size, the data structure alignment is required to be set to 1 byte for all applications interacting with the VFD-RT application as well as the VFD-RT application itself, in order to reduce data copying time and, consequently, maximize simulation performance.

Connection Protocol

Two similar but distinct protocols are used to connect to the VFD-RT application: one for providers, one for clients.

In connecting with a provider, the VFD-RT application is a receiver of the data and is responsible for initiating communication with the provider. When a provider connection or disconnection is requested through the user interface, a data structure of type "provider connection" is sent in a data copy message. The provider connection data structure is summarized in Table 2.2.

When the provider receives this message, it processes the message in the window procedure. The receiver and provider handles are used to check for proper message routing. The "connect or disconnect" integer is set to the appropriate value (1 for connect, 2 for disconnect) by the VFD-RT application. The other variables – data type, data name, provider

Table 2.2: Breakdown of the provider connection data structure.

Variable Type	Size	Variable Name
Handle to window	64 bits	Receiver handle
Handle to window	64 bits	Provider handle
Integer	32 bits	Data type
String (32 characters)	32 × 8 bits	Data name
String (32 characters)	32 × 8 bits	Provider name
Integer	32 bits	Connect or disconnect
Integer	32 bits	Message status

name, and message status – are filled in by the provider after receiving and checking the message. The provider name is the 32-character description that shows up on the list of providers in the user interface of the VFD-RT application. The data type and name refers to the information that will be encountered in the *State* structure sent from this provider.

The provider confirms the connection with a data copy message containing the same structure, with the message status set to 1 indicating successful connection and the other values filled in. Once the confirmation is received, the VFD-RT application then prepares for interactions with the provider. The same procedure is used with disconnecting a provider from the VFD-RT application. Figure 2.3 illustrates the connection and disconnection protocol with providers. In the figure, the VFD-RT application sends a *ProviderConnection* data copy message to the provider, which sets its internal state to send data to the given VFD-RT application handle upon data requests, and then sends a filled-in *ProviderConnection* data copy message back to the VFD-RT.

In connecting with a client, the client is being served the data and is responsible for initiating communication with the server. When a client connection or disconnection is requested, a data structure of type "client connection" is sent via the data copy message from the client to the VFD-RT application. The client connection data structure is summarized in Table 2.3.

When the VFD-RT application receives this message, it processes the message in the

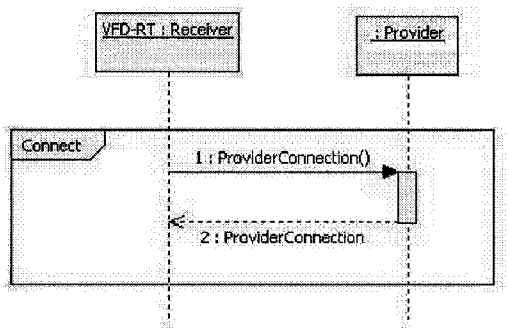

Figure 2.3: Diagram of the protocol for connecting or disconnecting the VFD-RT application to or from a provider.

Table 2.3: Breakdown of the client connection data structure.

Variable Type	Size	Variable Name
Handle to window	64 bits	Server handle
Handle to window	64 bits	Client handle
Integer	32 bits	Client index
String (32 characters)	32 × 8 bits	Client name
Integer	32 bits	Connect or disconnect
Integer	32 bits	Message status

window procedure. The server and client handles are used to check for proper message routing. The client handle is also stored as the address to which to route the data frame during the simulation run. The client index is assigned by the VFD-RT application to identify the client. The client name is a 32-character description of the client, which will also show up on the user interface in the VFD-RT application. As with the providers, the "connect or disconnect" integer is set to the appropriate value (1 for connect, 2 for disconnect) by the VFD-RT application. The message status indicates success or failure of the connection attempt.

One can note that the provider index is absent compared to the client connection structure; that is because the provider index is only assigned after the VFD-RT application receives the confirmation message from the provider, and therefore after the provider has finished processing the connection message.

When receiving a client connection message, the VFD-RT application adds the client information to its list of clients, so that data will be served to the client at the next data-sending event. The VFD-RT application then sends a confirmation message containing the same structure as the one received, with a specific client index and the message status set to 1 indicating successful connection. The same procedure is used for disconnecting a client from the VFD-RT application; the client is removed from the list of clients to be served simulation data. Figure 2.4 illustrates the connection protocol with clients. In the figure, the client sends a *ClientConnection* data copy message to the VFD-RT application, which prepares itself to send data to the given client handle upon data requests, and then sends a *ClientConnection* data copy message back to the client.

Although connection is directional by convention, if the simulation is not running, providers and clients can be disconnected from either end of the connection, to permit the removal of applications that may hinder the simulation performance, as in the case where an application falls into an infinite loop, for example. When closing the VFD-RT application, a disconnection message is to be sent out to all connected applications, so that connections can be properly terminated.

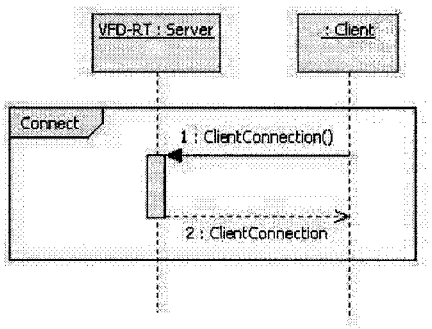

Figure 2.4: Diagram of the protocol for connecting the VFD-RT application with clients.

Data Transfer Protocol

Because the VFD-RT application controls the passage of simulation time, on which data transfer depends, the application is responsible for transferring data from the provider and to clients. The VFD-RT application sends a data request to the provider during every simulation loop. This request is in the form of a data copy message containing a data copy structure detailed in Table 2.4.

Table 2.4: Breakdown of the provider data copy structure.

Variable Type	Size	Variable Name
Handle to window	64 bits	Receiver handle
Handle to window	64 bits	Provider handle
Integer	32 bits	Data type
String (32 characters)	32 × 8 bits	Data name
Integer	32 bits	Provider index
Double-precision value	64 bits	Timestamp
Double-precision values	18 × 64 bits	Simulation data

The receiver and provider handle, the data type, and the data name are used for checking message routing errors. The provider index is mainly used by the VFD-RT for efficient indexing of the provider data in the data frame. The timestamp is used to inform the provider of the simulation time at which data is requested. The simulation data is in the form of 18 double-precision values, which will be placed directly into the *State* structures in the data

frame when the VFD-RT application receives this message. Similar to the provider connection data structure, the provider fills in the 18 values and returns the structure. Therefore, the same structure is used for requesting provider data and for sending the provider data to the VFD-RT. The case of using two different structures, one for requesting data and one for sending data back to the VFD-RT, was considered, to avoid sending the 18 64-bit values that are essentially undefined during data requests. Although sending the initial 18 undefined values to the provider would increase the size of the data to copy, and therefore be slightly less efficient, the provider now does not need to create a new structure on every simulation request but can rather simply fill in the existing structure, compensating for the data copying inefficiencies. Furthermore, having an 18-value simulation data structure gives a clear interface for the provider designers and avoids complications of accidentally including too much or too little simulation data than what the VFD-RT application expects.

So when requesting data, the VFD-RT application sends the data copy structure with the simulation data left undefined. The provider fills in the simulation data values and returns the data copy structure to the VFD-RT application, which copies the data to its appropriate location in the data frame. One request is made for each provider during every simulation loop. Figure 2.5 illustrates the data transfer protocol for providers. Similar to the connection diagram in Figure 2.3, the VFD-RT application sends a data request in the form of a *ProviderCopyData* data copy message to the provider. The provider applies the timestamp to its internal model, produces simulation data, and sends a filled-in *ProviderCopyData* data copy message back to the VFD-RT.

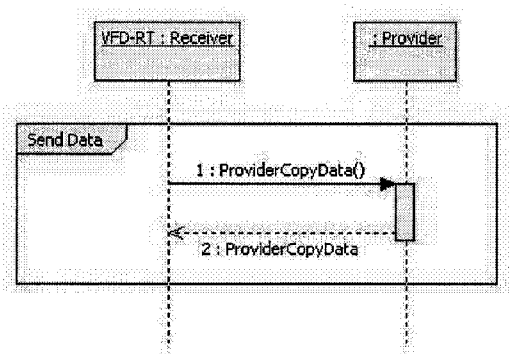

Figure 2.5: Diagram of the protocol for requesting and receiving data from a provider.

At the beginning of every simulation loop, the VFD-RT application sends out simulation data to clients. Because the VFD-RT application controls the passage of time, the VFD-RT application initiates the data transfer. The client is not required to keep track of time; it produces output only when data is received.

The data copy message sent to the client, on the other hand, contains the data frame structure, as shown in Table 2.1. This design allows a fast and direct data transfer to clients, without extra overhead for reorganizing the data. Section 2.3.4 explains the breakdown of the data frame, which is the data copy structure sent to clients.

The VFD-RT application does not require a confirmation message from the client for a data frame sent, since clients do not contribute simulation data. Figure 2.6 illustrates the data transfer for clients.

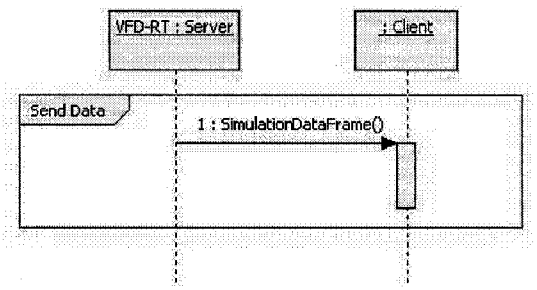

Figure 2.6: Diagram of the protocol for sending simulation data to a client.

New Ship Motion Case A request to generate new ship motion is a special case of the communication protocol, whereby the VFD-RT application sends a command to a ship motion provider. The corresponding data copy message has data structured as shown in Table 2.5. The provider uses the receiver and provider handles to check for message routing errors, and uses the desired wave height, the ship heading, and the ship speed to modify its internal model to produce the desired motion when requested. A message status, as seen previously, is used to confirm a success or failure of the new ship motion command.

In the VFD-RT application user interface, when the "Generate new motion" is pressed, if a provider is connected, a dialog box appears requesting the desired wave height, ship heading, and ship speed, as well as the provider to which this new motion applies. As the

Table 2.5: Breakdown of the new ship motion data structure.

Variable Type	Size	Variable Name
Handle to window	64 bits	Receiver handle
Handle to window	64 bits	Provider handle
Floating-point value	64 bits	Wave height
Floating-point value	64 bits	Ship heading
Floating-point value	64 bits	Ship speed
Integer	32 bits	Message status

dialog box is completed and closed at the command of the user, a new modal dialog box appears to show a progress bar and wait for a confirmation from the provider. A modal dialog box differs from the modeless ones in that the modal box suspends the operation of its parent application – in this case, the VFD-RT – until it is closed. The purpose of the modal dialog box is to avoid starting a simulation while the provider is changing ship motion models, which may produce undesired sudden changes in simulation data values. Because of the modal nature of the waiting dialog box, the confirmation message sent by the provider cannot be processed by the suspended application window procedure. And so the dialog box cannot receive its cue to close through the parent application. The modal dialog box, therefore, contains its own window procedure that waits for a confirmation message from the provider, through the data copy message, that indicates that the motion generation was a success. The modal dialog box then closes and the VFD-RT application resumes its normal operation.

The dialog box also displays the progress of the connection; however, if no problem is encountered, the ship motion generation completes before the waiting dialog box can be fully drawn on the screen.

2.3.6 Logging

Logging in the VFD-RT application is currently used for code debugging. The logging class is a singleton, which means only one copy of the class exists for the VFD-RT application

and can be accessed by any other classes in the application code. The VFD-RT logging class also has a double-log feature, with one log recording all entries and the other recording only entries marked as high-priority. The higher priority log is used for performance measurements, so that the normal logging operations can be turned off to avoid interference with the measurements.

Every time the VFD-RT application runs, a new entry is added to the log. Each entry of the log stores the time of the entry. Additionally, when the VFD-RT application starts, if logging is enabled, a separator and the current date are entered in the log. By default, logging is disabled.

2.3.7 Software Architecture Design Specification

The design specification of the VFD-RT application is defined in UML. The VFD-RT application is driven by design use cases. The architecture is designed to handle all of the use cases necessary for the operation of the simulation environment.

Use Cases

To ensure that the VFD-RT application accomplishes its desired goals, a list of use cases is considered and developed in order to guide the design of the software architecture. Figure 2.7 shows a use-case UML diagram depicting the use cases of the VFD-RT application, the actors, which can be human or computer software, and their dependencies. A solid line represents a direct involvement, while a dashed arrow represents a dependence relationship of the cases.

There are a number of ways the VFD-RT application responds to user commands. It can execute a simulation run, request connection to providers, generate new ship motion, and disconnect clients and providers, the latter being dependent on an existing connection. To the client, the VFD-RT application can connect or disconnect it from the simulation, and can serve the simulation data to it, provided that the simulation is running. The provider is involved in its connection or disconnection with the VFD-RT, the receiving of data dependent upon connection, and, if it is a ship motion provider, the generation of new

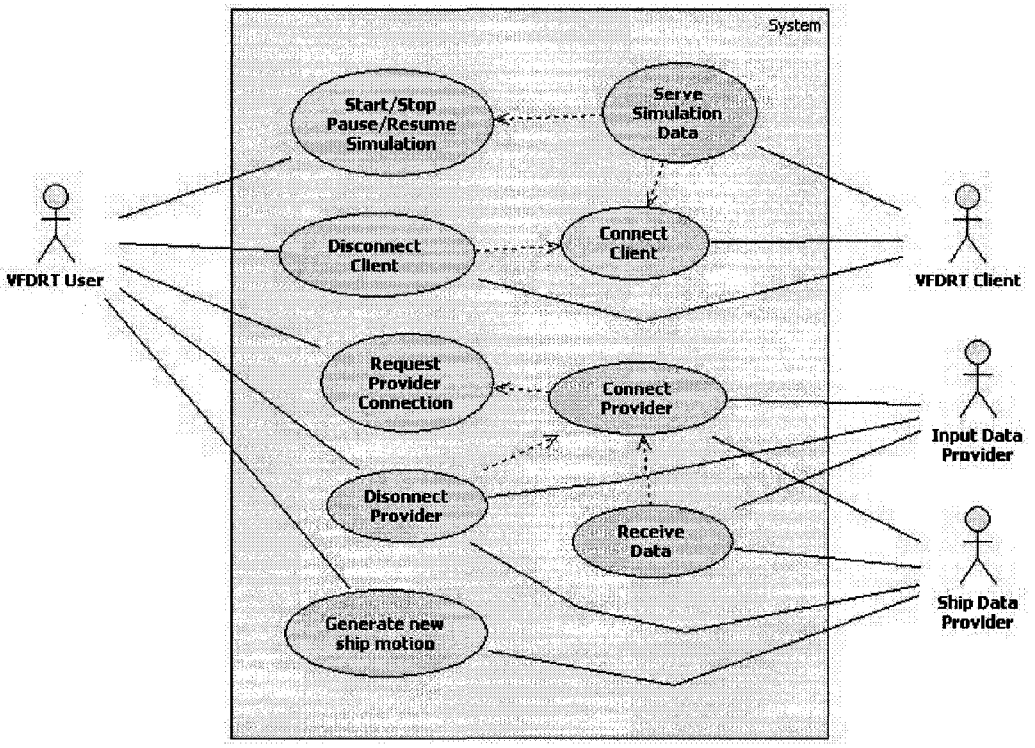

Figure 2.7: Use case UML diagram for the VFD-RT application.

ship motion. The VFD-RT is designed with these cases in mind.

Class Design

From the use cases, and using object-oriented programming, a general architecture is designed with class structures and their relations to each other. Figure 2.8 shows a class UML diagram designating the various dependencies of the classes. The various symbols indicate the type of class relationships:

- a solid diamond: composition of an instance of one class in another;
- a hollow diamond: class aggregation, which means that a class contains a pointer to an instance of another class;
- a hollow arrow: class generalization, which implements as a concept of inheritance in object-oriented languages; and
- a line arrow: accessibility of one class to some elements of another.

The diagram can be divided into two sections: the left side, all connecting to the *MainWindow* class, are the classes associated with the user interface; the right side is associated with data manipulation. At the top, the *Application* class has a solid diamond, meaning that it is composed of instances from 6 classes: the *Timer*, the *MainWindow*, the *ClientManager* which manages a list of *Client* class instances, the *InputManager* for a list of *InputProvider* class instances, the data *Frame* containing *State* class instances, and finally the *ShipMotionGenerator* which is responsible for managing the dialogs to generate ship motion and for processing all of the operations related to ship motion generation. This generator class has a pointer to the *InputManager* in order to communicate with the appropriate ship motion provider. The *InputManager*, in turn, has a pointer to the data *Frame*, so that *InputProviders* can update data within the data *Frame* at every simulation loop. It is important to note that the *Client* and *InputProvider* classes are not the actual clients and providers themselves, but are rather components of the application which manage the connection with the client or provider associated with that class instance. Each instance

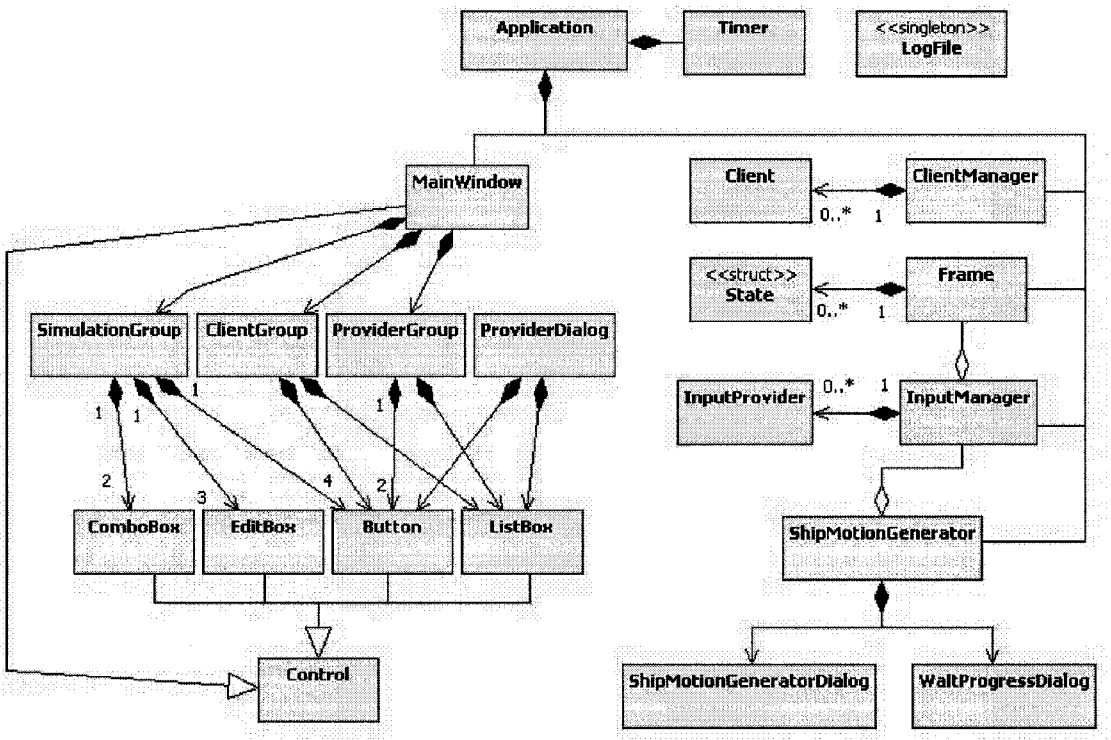

Figure 2.8: Class UML diagram for the VFD-RT application.

of a *Client* class holds the information to communicate with one client, as it is for the *InputProvider* class instances.

On the *MainWindow* side, one can note that the *Control* class is the base class inherited by a majority of the other user interface classes. This *Control* class has general variables and functions used by the Windows API, such as the handle, and enable or disable functions. The *MainWindow*, the *ComboBox*, also known as the drop-list, the *EditBox* or text boxes, the *Button*, and the *ListBox* are all derived from this base class. Moreover, the *MainWindow* is divided into three sections called groups, as described in the user interface section of this document, Section 2.3.2: the *SimulationGroup* controlling the simulation run, and the *ClientGroup* and the *ProviderGroup* listing the connected components and giving certain control over connection and disconnection via buttons. The *ProviderDialog* is separate from the *MainWindow* since it is created only when the user requests to connect to a provider.

In addition, the *Timer* class wraps the simulation timer, as described in Section 2.3.3 dealing with time management. The *LogFile* is the singleton class that can be accessed by all the other classes to output messages to a file for debugging or other monitoring purposes.

Collaborations

For each of the use cases of the VFD-RT application, collaborations between instances of the classes have been defined. They are summarized in collaboration UML diagrams. These diagrams describe the sequence of calls that one class makes to a function within the called class, to which the arrow points.

Simulation Start

When starting a simulation, the user presses the "Start" button. The button sends a message to the application window procedure, which handles it by executing Step 1 of the collaboration illustrated in Figure 2.9.

After Steps 1 and 2, which checks for connected providers, the application retrieves start time information from the user input (Steps 3-5) and uses that time to set the timestamp in the data frame (Steps 6). The request for data is then made to all the connected

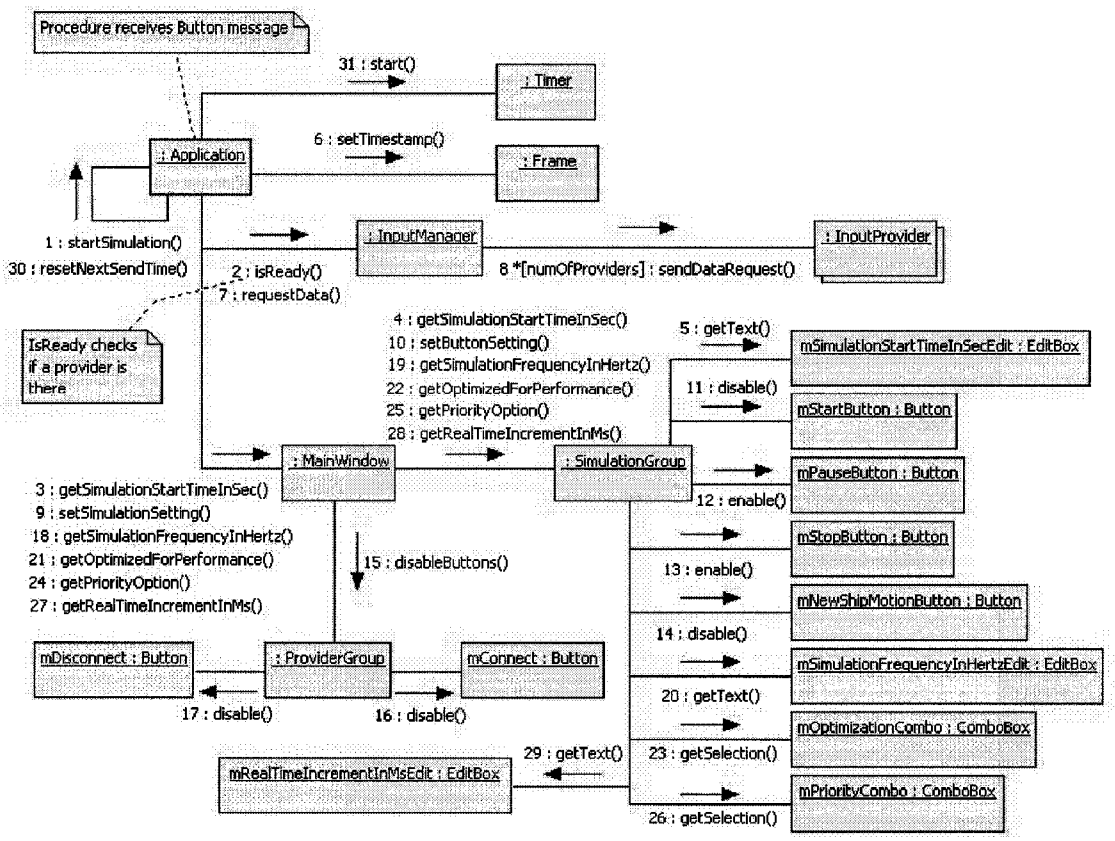

Figure 2.9: Start simulation collaboration UML diagram.

providers (Steps 7-8), and the user interface is updated – buttons are disabled or enabled as appropriate (Steps 9-17). The user selected simulation options are then retrieved and set by the application, from the simulation frequency to the simulation timer increment for non-real-time simulation (Steps 18-29). The time of the next data sending event is set based on the start time and the simulation frequency (Step 30). Finally, once the initializations are completed, the timer starts. At this point, the provider should have returned with the generated data stored in the VFD-RT data frame, waiting to be sent to the client at the indicated start time.

Simulation Stopping

When stopping the simulation, the user presses the "Stop" button. The button sends a message to the application window procedure, which handles it by executing Step 1 of the collaboration illustrated in Figure 2.10.

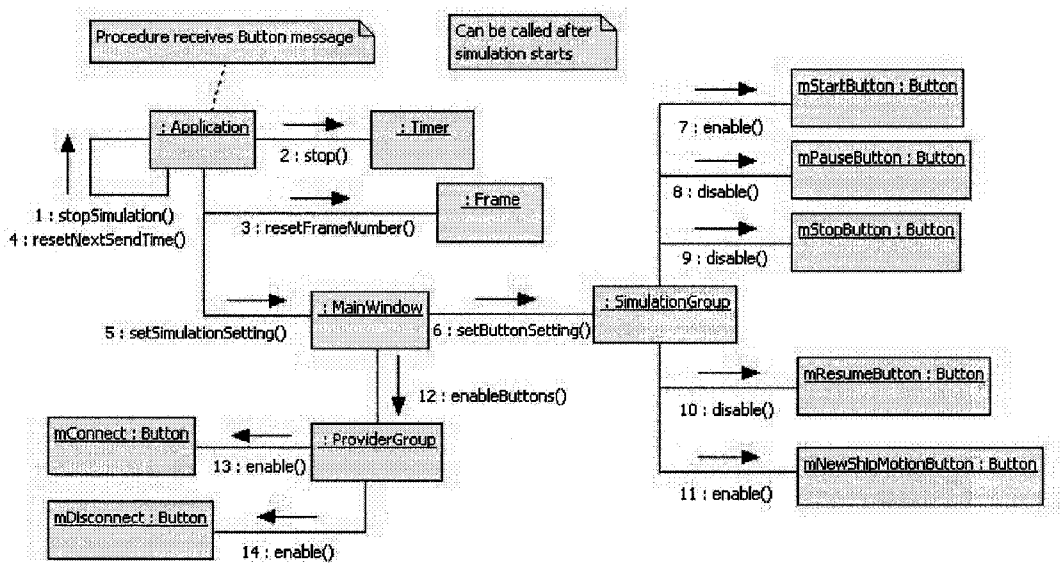

Figure 2.10: Stop simulation collaboration UML diagram.

At the command, the *Timer* is stopped, the data frame number is reset to 0, and the time for sending the next frame is also reset (Steps 2-4), since these are no longer needed and the resetting ensures that the variables have the correct value at the next simulation

start point. The rest of the collaboration involves resetting the user interface (Steps 5-14).

Simulation Pausing and Resuming

When pausing or resuming the simulation, the user presses the "Pause" or "Resume" button. The button sends a message to the application window procedure, which handles it by executing Step 1 of the collaboration illustrated in Figure 2.11. This collaboration essentially consists of pausing or resuming the *Timer* (Step 2) and setting the user interface buttons.

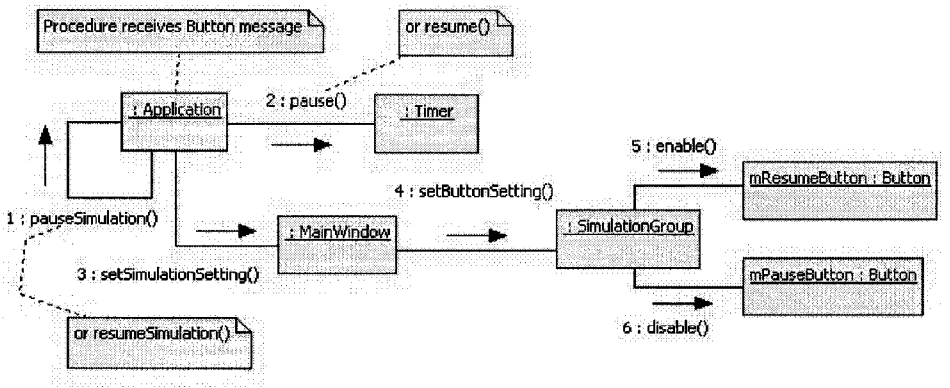

Figure 2.11: Pause/resume simulation collaboration UML diagram.

Data Receiving

When a provider sends data to the VFD-RT application, the data copy message is processed by the application window procedure, which handles it by executing Step 1 of the collaboration illustrated in Figure 2.12. The application asks the *InputManager* to read the received data into the data frame (Step 2). But before proceeding, the *InputManager* checks for any message routing or data errors such as data content size (Steps 3-7). Finally, the data frame is called to read the data directly into the *State* class within *Frame* according to the provider index that is part of the message. Section 2.3.5 of this document describes the data contained in the provider data copy message.

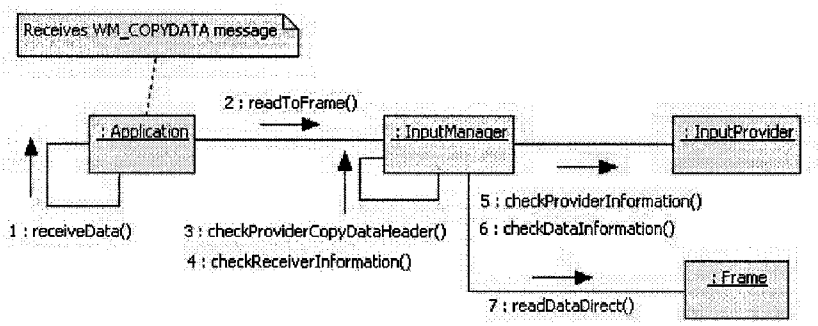

Figure 2.12: Receive data collaboration UML diagram.

Data Serving

When the application is not sending or receiving data, the simulation loops continuously until the time indicated by the timer reaches the next sending event. At that point, the VFD-RT application executes Step 1 of the collaboration illustrated in Figure 2.13. The application retrieves the current time from the timer to be set as the timestamp of the current frame. This step must be completed first because the time at the beginning of the current simulation loop marks the timestamp of this frame and is used for calculations by the provider models; performing this step later would introduce delays caused by sending messages to clients and providers, and by other operations. Following the retrieval of the time, the next most urgent operation is to send the data frame to each client (Steps 3-4), since they are designed to depend on the timing controlled by the VFD-RT application. The new *Frame* timestamp is set to prepare the data frame to receive provider data. Data requests are sent to the providers (Steps 6-7). Finally, the frame number and the next send event time is incremented (Steps 8-9).

Provider Connection Request

When requesting connection to a provider, the user presses the "Connect" button under the provider group section of the VFD-RT main window. The button sends a message to the application window procedure, which handles it by executing Step 1 of the collaboration illustrated in Figure 2.14. Upon requesting connection, the application retrieves the handles

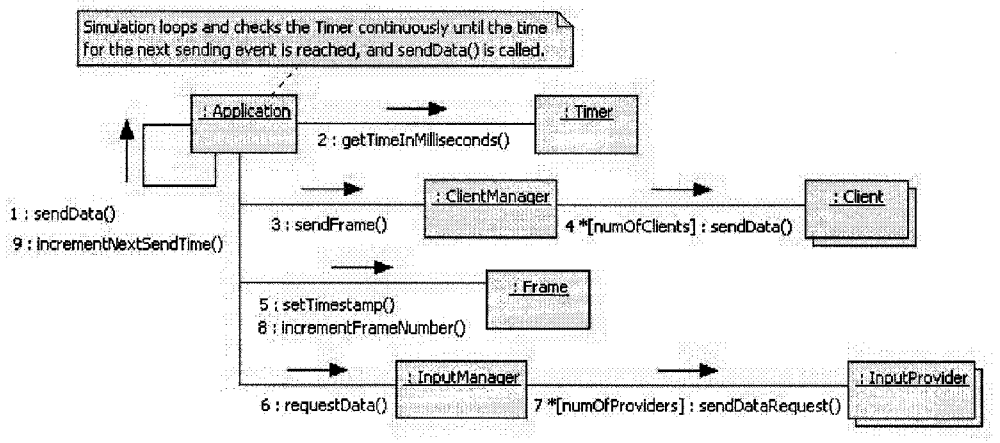

Figure 2.13: Serve data collaboration UML diagram.

of the connected providers, searches the operating system for all the providers currently running, eliminates from that list the ones that are already connected, and shows this list in the *ProviderDialog* (Steps 2-4). This *ProviderDialog* is where the user chooses to which unconnected provider the application should connect. Upon choosing "Connect" for a particular provider, the handle of the selected provider is passed on to the *InputManager* for verification and request sending (Steps 5-8). This collaboration mainly uses provider handles, rather than provider names, since handles are unique to each application while names may not be. The provider connection request collaboration is performed before the simulation starts, so the steps are not time critical and their ordering is based on logic, as explained in Section 2.3.2.

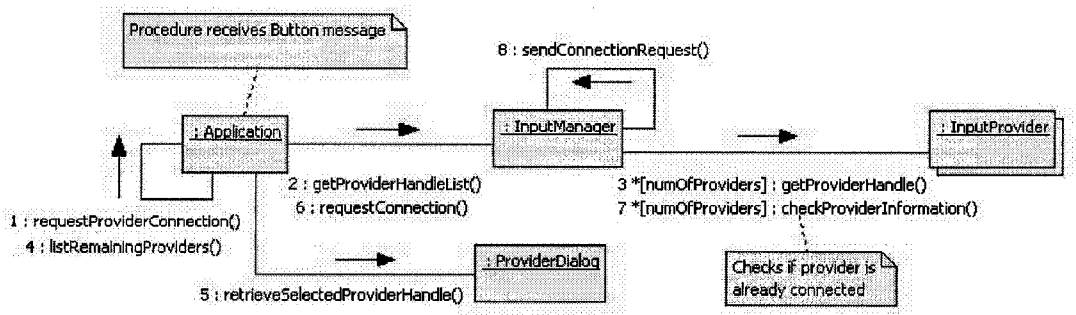

Figure 2.14: Provider connection request collaboration UML diagram.

Provider Connection

After the request has been sent, a provider returns with a confirmation message to the VFD-RT application, and the data copy message is processed by the application window procedure, which handles it by executing Step 1 of the collaboration illustrated in Figure 2.15. Upon receiving the data copy message, the data is passed on to the *InputManager*, which attempts to connect by checking the data content and ensuring that the provider is not already connected (Steps 2-5). If not connected, the provider is added to the list of providers in the *InputManager*, which creates a new *InputProvider* instance that holds the connection information to that provider (Steps 6-8). The data *Frame* is updated by the addition of a *State* to hold the data from the new provider (Steps 9). The other steps are all related to updating the user interface: retrieving the list of the data names of the providers and updating the list in the provider group of the user interface (Steps 10-16). The provider connection collaboration is performed before the simulation starts, so the steps are not time critical and their ordering is based on logic.

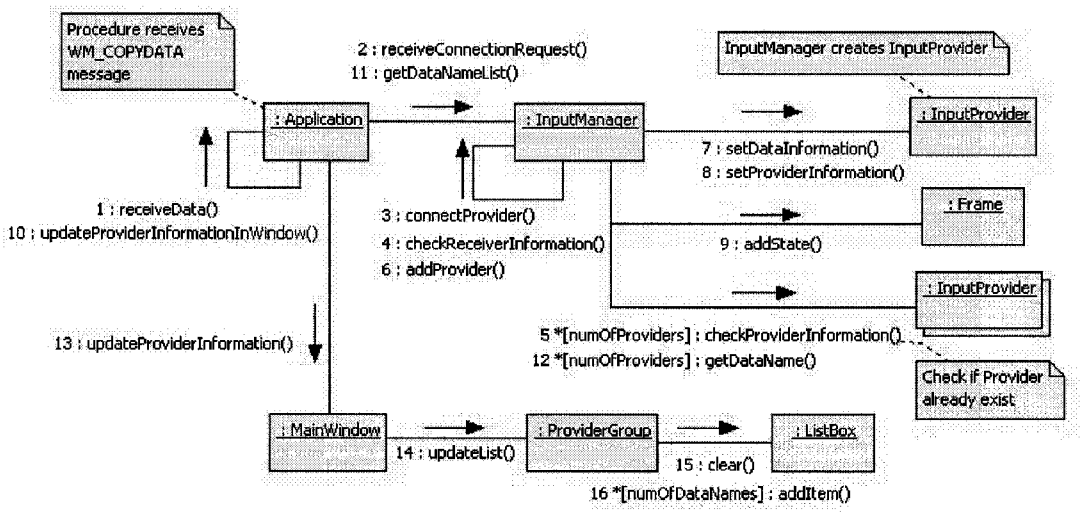

Figure 2.15: Provider connection collaboration UML diagram.

It should be noted that requesting connection does not guarantee a connection, since the provider application may not be ready to connect or may be operating abnormally.

Provider Disconnection

When a provider closes or when the VFD-RT application disconnects a provider through the "Disconnect" button, Step 1 of the collaboration is executed as illustrated in Figure 2.16. The application calls for the disconnection of the provider through the *InputManager* that extracts and removes it from the provider list, removes the associated *State* in the data *Frame*, and uses the extracted copy to send the provider a disconnection message (Steps 2-5). Extracting a copy is necessary because the disconnection message needs the destination handle stored in the *InputProvider* instance that is to be deleted. Moreover, the disconnection message is sent last because if an error is encountered when removing the *State* from the *Frame*, the provider is not successfully disconnected and the confirmation can be prevented from being sent. The user interface is then updated, as with the collaboration for connecting a provider. Like provider connection, the ability to disconnect is disabled during the simulation run, so this collaboration is performed before the simulation and is not time-critical.

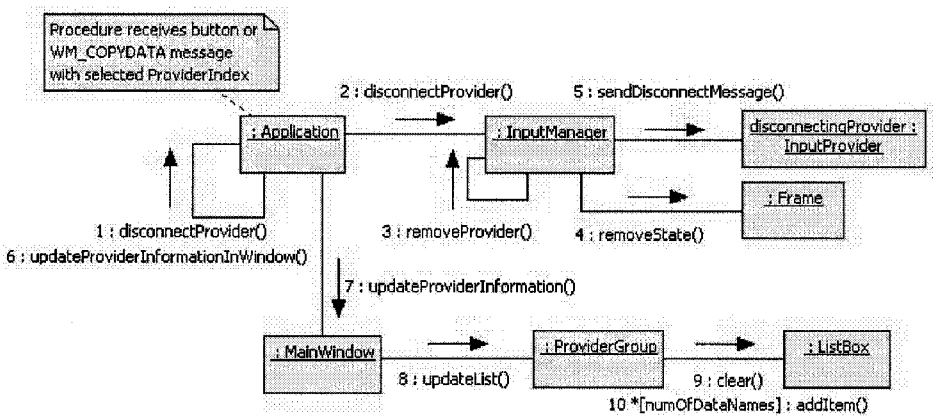

Figure 2.16: Provider disconnection collaboration UML diagram.

Client Connection

In connecting a client, the client initiates the connection by sending a data copy message to the VFD-RT application window procedure, which handles it by executing Step 1 of the

collaboration illustrated in Figure 2.17. Similar to connecting a provider, the application passes the received data to the *ClientManager* that checks the data content for errors, checks if the client is already connected, and if not, creates a new *Client* class instance with the given information and sends a confirmation to it (Steps 2-7). Following the client creation, the *ClientGroup* in the user interface is updated with the new list of client names (Steps 8-14). Connecting a client can be performed after a simulation run has started, but is not recommended because of the extra processing required to update the user interface. Nevertheless, if necessary, adding clients to the simulation is possible during simulation.

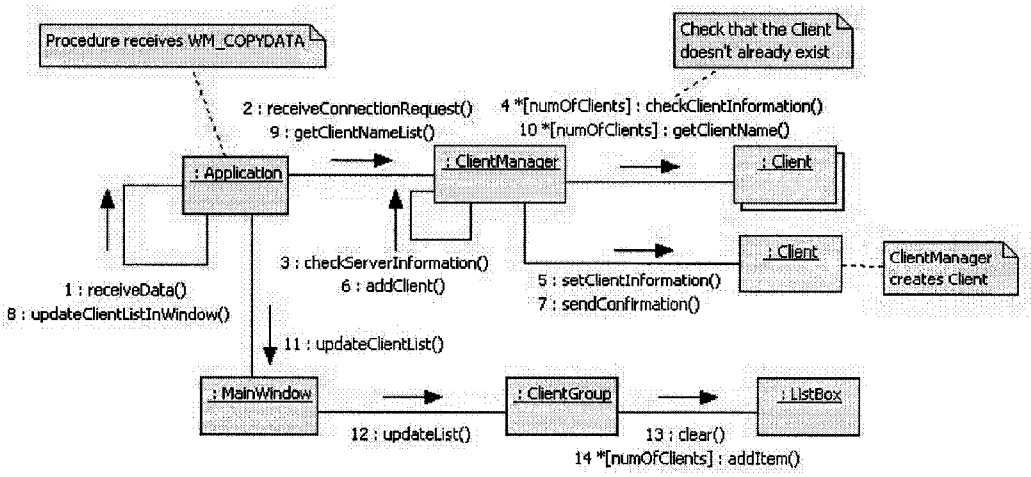

Figure 2.17: Client connection collaboration UML diagram.

Client Disconnection

When a client closes or sends a disconnection data copy message, or when the VFD-RT application disconnects a client through the "Disconnect" button, Step 1 of the collaboration is executed as illustrated in Figure 2.18. This collaboration is similar to disconnecting a provider, with checks done on the message data content, confirmation sending to the disconnecting client, removal of the *Client* instance from the client list in the *ClientManager*, and updating the list in the main window user interface. If the disconnection is triggered not by a data copy message but a button press by the user, data checks are not necessary, since the client information used is that which is already stored in the *Client* instance. Compared

to disconnecting a provider where a copy of the *InputProvider* instance is made, no copy needs to be made of the disconnecting client, because the *Frame* class is not involved and so cannot cause errors as in the provider disconnection. Therefore, a disconnection confirmation message can be sent at any time during this collaboration, and its position in the order of function calls is chosen to avoid the need to make a copy of the *Client* instance.

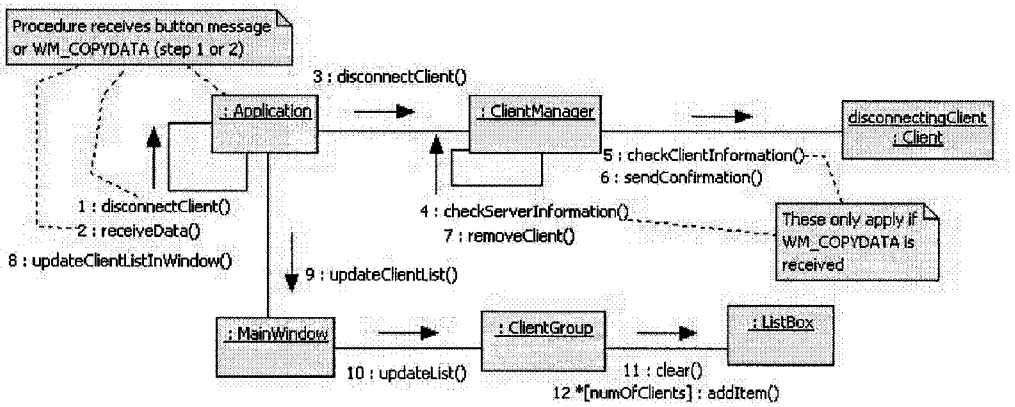

Figure 2.18: Client disconnection collaboration UML diagram.

New Ship Motion Request

When the "Generate New Ship Motion" button is pressed, the VFD-RT application executes Step 1 of the collaboration illustrated in Figure 2.19. The application calls the *ShipMotion-Generator* to generate, which is followed by the *ShipMotionGenerator* executing a dialog for the user to input the desired ship motion (Steps 3-4), retrieving of the user inputs (Steps 5-6), preparing a ship motion generation message using the data from the *InputManager* (Steps 7-8) and opening a wait dialog, with a progress bar, to send the message and wait for confirmation (Step 9). The wait dialog closes as soon as it receives a confirmation data copy message from the provider.

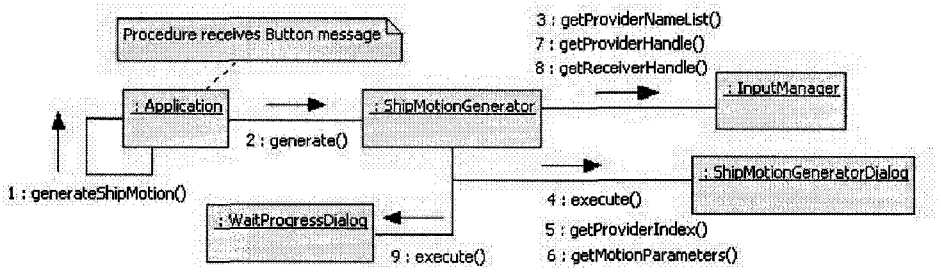

Figure 2.19: New ship motion collaboration UML diagram.

2.4 Ship Motion Provider

A provider lies in the data input layer of the VFD-RT simulation environment and is responsible for providing data when requested by the simulation through the VFD-RT core application. To better demonstrate the implementation of this concept of operation, the example of the Ship Motion Provider (SMP) is presented. The SMP is a VFD-RT provider application that calculates and provides ship motion data to the simulation, based on user-defined parameters, simulation time, and the built-in ship motion model. Using a ship model and code developed on a previous project in the Applied Dynamics Research Group, the SMP calculates 18 values of the ship state, which are the position, velocity and acceleration in six degrees of freedom of the ship.

2.4.1 Motion Generation

The model produces ship motion through commonly-used conventional linear ship response theory [31]. It uses a library of ship motion spectra, generated from a combination of a sea spectrum model and the ship frequency responses to unit amplitude waves evaluated with the *SHIPMO07* ship hydrodynamics code [32]. The ship motion spectrum can then be converted to time-domain data through the inverse Fourier transform. The library in current use represents significant wave heights of 1 to 6 metres in increments of 1 metre, ship heading of 0° through 360° in increments of 15°, and ship speeds from 5 to 25 knots, in increments of 5 knots.

This model is implemented in the VFD-RT environment through a FORTRAN function,

called *SHPR* developed by Dr. Rob Langlois [1], that employs the library and applies the inverse Fourier transform. The function internally extracts the library file name, which contains the wave height, ship heading and ship speed information, from an external file, "shpr.inp." The function then reads the corresponding ship motion frequencies from the library file and produces the time-domain data. The input to the function is a time value, and the output is the 6 degree-of-freedom ship position, velocity, and acceleration.

The SMP wraps this FORTRAN code in a Windows application in order to exchange time and ship motion data with the VFD-RT simulation. The "wrapping" method is preferred over translating to C++ since if the FORTRAN code is translated, the implementation would require verification with the SHPR model. By keeping the model implemented in FORTRAN, the verification process is avoided. The integration of FORTRAN code into the C++ program of the SMP consists of compiling the C++ and FORTRAN code separately, and then linking the resulting object files into one executable file. This method demonstrates how new or existing source code can be integrated into the VFD-RT environment, regardless of the programming language or coding standards.

2.4.2 User Interface

The SMP user interface is designed to be small and simple, as it does not require user input during its operation. Figure 2.20 shows the user interface.

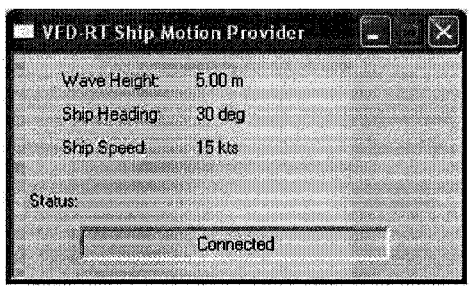

Figure 2.20: SMP user interface.

The top section of the interface shows the parameters defining the ship motion: wave height, ship heading, and ship speed. They can be modified through the VFD-RT application by choosing "Generate New Ship Motion" from the simulation group. The bottom

section of the interface is a box indicating the status of the provider. The interface is simple without requiring any direct user input through buttons or other window controls, except for the program close button at the top right of the window. That is because all of the required operations – connection, disconnection, and generation of new ship motion – are performed by the VFD-RT application, which is downstream of the data flow and is therefore responsible for connecting and disconnecting from the data source, as mentioned in Section 2.2.

2.4.3 Data Management

Since the SMP mainly interacts with the VFD-RT application, the data storage structure in the SMP is identical to the provider data copy structure, as detailed in Section 2.3.5. When ship motion data is calculated, at every data request, the resulting values are stored directly into the provider data copy structure. The data is then copied to the message and returned to the core application.

The data that is contained in the *COPYDATASTRUCT* parameter of the data copy message is structured in the same way as discussed in Table 2.4 of Section 2.3.5. Within that data structure, the 18 double-precision values are organized in the following order:

- position in the three translational degrees of freedom: surge (translation along the ship's longitudinal axis), sway (translation along the ship's lateral axis), and heave (translation along the ship's vertical axis);
- position in the three angular degrees of freedom: roll (rotation about the ship's longitudinal axis), pitch (rotation about the ship's longitudinal axis), and yaw (rotation about the ship's vertical axis);
- velocity in the three translational degrees of freedom: surge, sway, and heave;
- velocity in the three angular degrees of freedom: roll, pitch, and yaw;
- acceleration in the three translational degrees of freedom: surge, sway and heave;

- acceleration in the three angular degrees of freedom: roll, pitch, and yaw.

The positions are measured in metres (m) or radians (rad), velocities in metres per second (m/s) or radians per second (rad/s), and accelerations in metres per second squared (m/s^2) or radians per second squared (rad/s^2).

The SMP also processes the provider connection structures described in Section 2.3.5, which it uses to check for message routing error by storing the handle of the receiver, the VFD-RT application. As with the VFD-RT application, the SMP has two windows: one visible main window displayed on the screen, and one hidden application window, responsible for processing incoming data copy messages.

2.4.4 Software Architecture Design Specification

The architecture is designed to handle the three main use cases encountered by the SMP: connecting to or disconnecting from the VFD-RT application, sending ship motion data, and generating new ship motion. The design specification of the SMP is defined in UML. Figure 2.21 presents a class UML diagram for the SMP classes.

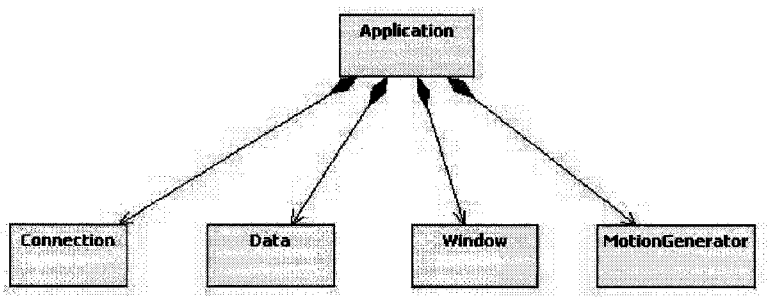

Figure 2.21: SMP class UML diagram.

The *Application* class contains the application window procedure and one instance of each of the four other classes of the application: *Connection*, *Data*, *Window*, and *Motion-Generator*. The *Connection* class, as the name suggests, holds the variables necessary for the message error checking, data parsing, and message sending operations. The *Data* class imports and uses the *SHPR* function to generate simulation data that is then stored, and

the *Window* class updates the status box and motion parameter values in the user interface. The *MotionGenerator* is the component that changes the motion parameters in response to a new motion generation message from the VFD-RT. The motion parameters are changed by modifying the "shpr.inp" file used by the FORTRAN code to determine which library file to use for its inverse Fourier transform operation. By changing the library file name, which is inside the input file, to one that corresponds to the given new ship motion parameters, the *SHPR* function is redirected to the new library file.

Connection and Disconnection

Connection to the VFD-RT application is triggered by a VFD-RT application request for connection; disconnection is triggered either by a request from the VFD-RT or by the closing of the SMP program. Figure 2.22 shows a collaboration UML diagram of the connection or disconnection events.

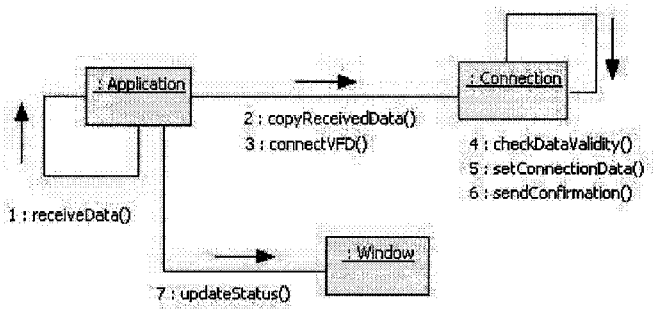

Figure 2.22: Connection and Disconnection collaboration UML diagram for the SMP.

When the application receives a connection data copy message (Step 1), the *Application* class passes the data on to its *Connection* member (Step 2), to check for message routing and data content error, namely that the data content size is as expected, to save the connection information for data transfer, and to send the confirmation back to the VFD-RT application (Steps 3-6). Finally the *Window* is updated (Step 7).

Sending of Ship Motion Data

The sending of ship motion data is initiated by a data request message coming from the VFD-RT application. The request, a *WM_COPYDATA* message interpreted by the SMP application window procedure, triggers the execution of Step 1 in the collaboration diagram shown in Figure 2.23. Similar to the connection collaboration, the data is copied and checked for errors, this time in the *Data* member class (Steps 2-4). The timestamp in the message is then extracted and applied in the *SHPR* function called by the *calculateShipMotion()* method in *Data* (Step 5). The resulting 18 values for the 6 degree-of-freedom motion are then sent back to the VFD-RT application in a data copy structure (Step 6). The window is then updated (Step 7).

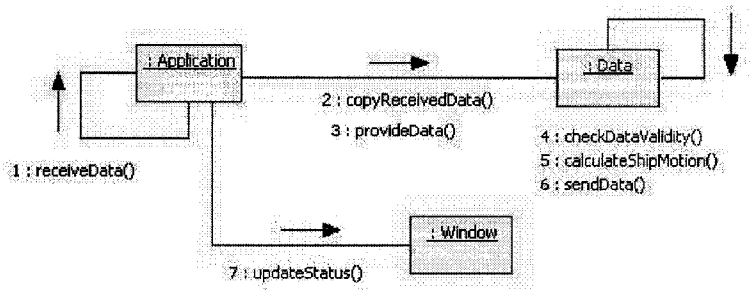

Figure 2.23: Sending of ship motion data collaboration UML diagram for the SMP.

Generation of New Ship Motion

The generation of new ship motion is initiated by a request message coming from the VFD-RT application. The request, a *WM_COPYDATA* message, triggers the execution of Step 1 in the sequence diagram shown in Figure 2.24. Upon receiving the generation message, the SMP sets the window status box to indicate the process has started (Step 2). The data is copied into the *MotionGenerator* member of the *Application*, and the motion generation process, which involves building the name of the library file to use, finding that library file to ensure that it exists, and creating the new "shpr.inp" file to overwrite the existing one with the new library file name (Steps 3-7). The user interface then displays a visual

confirmation that the generation is successful.

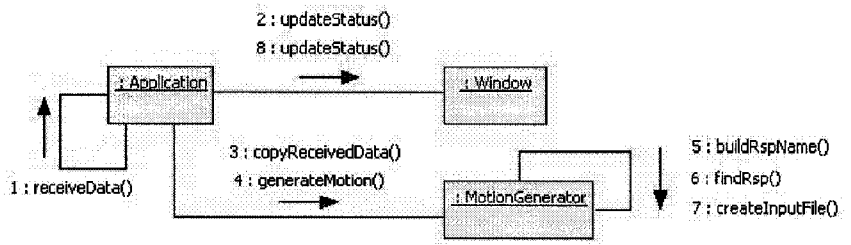

Figure 2.24: New ship motion generation collaboration UML diagram for the SMP.

2.5 DynamicsViewer Client

Clients in the data output layer of the VFD-RT simulation environment are the "users" of the simulation data. There are no restrictions from the VFD-RT application as to how the data can be used by the client, except for the connection protocol described in Section 2.3.5 regarding connection, disconnection, and the data available through the data frame received. The client does not require an internal clock, as the VFD-RT application keeps track of time and sends the data periodically to the client. As data arrives, a local copy is made, on which the client can perform its task. To discuss the implementation of a client, the DynamicsViewer 3-D visualization client is described here. The original DynamicsViewer was developed by the author prior to this thesis project, to assist in the various research projects at the Applied Dynamics Research Group [33]. Using separate data files for the 3-D visualization models and for the model motions in the virtual world, the DynamicsViewer is able to quickly and effectively produce animations valuable for the analysis of multi-body dynamic problems. The application is built on the OpenGL library and in the object-oriented C++ language. Modifications were made to the original DynamicsViewer to integrate it with the VFD-RT simulation environment. Changes to the connection management component involve the receiving and processing of simulation data as a client through the data copy messages. The animation rendering is also modified to allow for incoming data to move the different 3-D models and to set the animation time.

2.5.1 Model Data Management

The 3-D model data used by the DynamicsViewer client is initially stored in a file using the STL standard data format. The STL format stores data for each facet of the model as the coordinates of the 3 vertices of the facet and the normal direction of the facet. By parsing this information, the vertex and normal data can be loaded into arrays in the DynamicsViewer client memory, in instances of the *RigidBody* class. Besides the array of the 3-D model data, the class also contains information about the model colour, centre of rotation, and transformations applied to the rigid body, such as scaling, rotation and translations. Each body is an object in memory that is retrieved when the object is rendered.

The motion data for each rigid body comes directly from the output of the VFD-RT. When a data copy message is received containing simulation data, the DynamicsViewer client makes a local copy of the data, verifies the server information of the message, and parses the message for translational and rotational information for the body associated with that particular message. This transformation information is then applied to the corresponding *RigidBody* class instance.

It should be noted that the *RigidBody* class is designed to represent rigid bodies. It is possible to represent a flexible body as a series of rigid bodies in the DynamicsViewer, as was done for a project involving the study of flexible rotor blades [34].

The DynamicsViewer client has a hidden application window, responsible for processing data copy messages, and a visible main window serving as the user interface, similar to the two windows used in the VFD-RT application.

2.5.2 User Interface

Before the DynamicsViewer client can display the animation, the 3-D models and the motion data must be initialized in the virtual world. For this reason, an initial user interface is used, consisting of a main menu as shown in Figure 2.25.

The "Create New Animation" option creates a new animation for the DynamicsViewer. The set-up windows that subsequently appear allows the user to choose the 3-D models

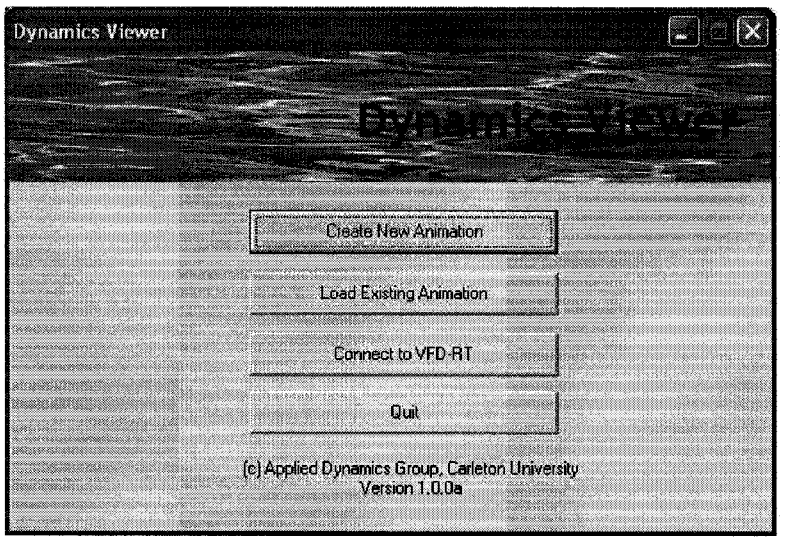

Figure 2.25: DynamicsViewer Client user interface.

to be represented and the accompanying transformations in terms of scaling, rotation, and translation within the virtual world, their colour, and their motion profile. A new animation parameter file that stores this user-input data is created and can be saved for future reference. The user can also load, with the "Load Existing Animation" option, an animation parameter file saved from a previous new animation creation. It allows for quick loading of the animation by simply selecting the animation parameter file. These two options preclude the VFD-RT connection, since the motion profile is loaded from a file, rather than assigned by the data copy messages in real time. They are carried over from the original DynamicsViewer, before integration with the VFD-RT. The third option is the modification that integrates the application to the simulation.

2.5.3 Connection to VFD-RT

This option connects the DynamicsViewer client to the VFD-RT environment. As a client, the DynamicsViewer is responsible for requesting connection to the VFD-RT server. When the "Connect to VFD-RT" button is pressed, the DynamicsViewer client searches for the VFD-RT application and sends a connection request to it. Connection is established when a confirmation is returned from the VFD-RT application and necessary connection data is

stored. An equivalent process is used for disconnection.

However, before the connection can be made, the windows, particularly the hidden one, must be created. To do so, a file-search dialog box appears right after the connect button press, asking for a previously-saved animation parameter file. This parameter file is modified from the ones generated by the "Create New Animation" button, with a parameter for each rigid body, indicating the index of the *State* structure, within the VFD-RT data frame, that describes its motion. Since the order of the *State* structures is the order of their corresponding provider in the VFD-RT application's list of providers, care must be taken to connect providers to the VFD-RT in the order that would correspond to the set values of these *State* index parameters. For example, if a helicopter 3-D model has a *State* index of 0 in the animation parameter file, the helicopter motion provider must be the first provider to be connected to the VFD-RT; if a ship 3-D model has an index of 2, the ship motion provider must be the 3rd provider to be connected.

Once this animation file is selected in the file-search dialog box, the DynamicsViewer client attempts to load the 3-D models, whose file names are recorded in the animation file, and to display the models. If this operation is successful, the windows are created, allowing the handle to the application window to be passed on to the VFD-RT server, which will send data copy messages to that handle.

Figure 2.26 illustrates the process as described above in a UML collaboration diagram of the client initialization. The *DynamicsViewerApp* class, at the root of the client, starts its execution by calling *initializeData()* which triggers a setup in the *Animation* (Step 2). The *DlgMainMenu* shows up as the one seen in Figure 2.25 (Step 3). As the user chooses "Connect to VFD-RT", a dialog box searching for the special animation parameter file, *FileShellDlg*, appears, after which the file name selected is retrieved (Step 4). The animation parameter file is loaded into the *RigidBody* class and other classes not shown (Step 5). The *Connection* class then receives the pointer to the *RigidBody* objects that were just initialized, in order for incoming simulation data to control the motion of the *RigidBody* (Step 6). The *RenderWindow* is given a pointer to the *Connection* class so that if it receives data copy messages, it can send them to the correct class for interpretation

(Step 7). The second phase of initialization starts with creating the windows, both hidden and visible through the *create()* command of the *RenderWindow* (Steps 8-10). Finally, once there hidden window handle is available, the *Connection* class stores the handle and sends a connection request to the VFD-RT server (Steps 11-12). Later, when the VFD-RT server sends back confirmation, the data of that confirmation message is checked for error and used to establish connection.

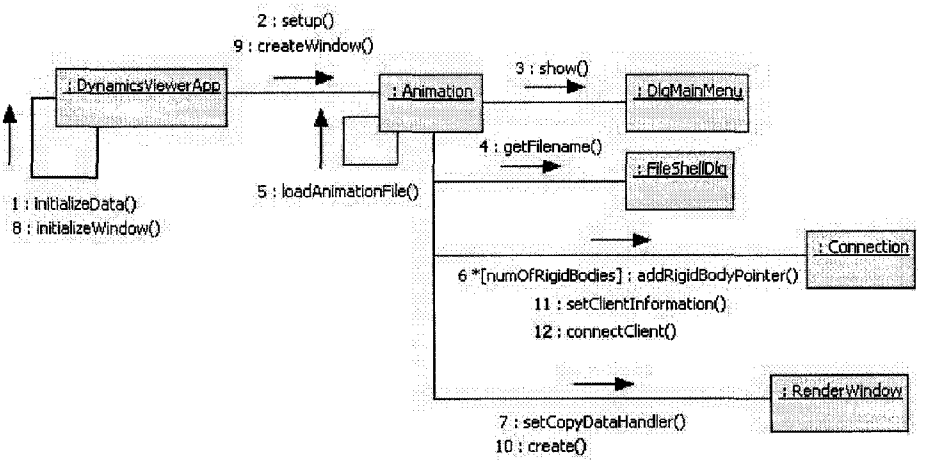

Figure 2.26: DynamicsViewer initialization collaboration UML diagram.

2.5.4 Object Rendering Method

The DynamicsViewer client employs the OpenGL library for rendering. The method used to render objects follows the conventional rendering pipeline: setting of the lighting, camera, and transformations. Initially, lighting and camera are set for the scene. The DynamicsViewer code inherited from the original version does not allow changes to lighting values without recompiling the code. However, for the purposes of the VFD-RT, it suffices to have the default lighting, as a fixed white point light positioned at coordinates (10, 10, 10). Using the Phong lighting model of the OpenGL library, which divides light sources into three components, the ambient light is set to 10%, the diffuse light, 90%, and the specular light, 40%. This light setting was deemed to provide adequate visual cues, such as shadows and surface shades, to observe rigid body motion. The camera is free to roam as controlled

by user input.

Transformations are applied to each body according to the information stored in their properties, after loading the animation file. The transformations that can be applied are scaling, rotation, and translation. Because OpenGL acts as a state machine, to which transformations are applied as matrix pre-multiplications, transformation functions for scaling (*glScalef*), rotating (*glRotatef*), and translating (*glTranslatef*) are called in the reverse order in which they are to be applied in the virtual world. The order in which these functions are called, then, is as follows:

- set the paint brush colour to that stored in the property of the object;
- translate the body to the position given by the received simulation data from the VFD-RT;
- rotate the body by the angles given by the received simulation data from the VFD-RT, applying the rotation about the x-axis, the y-axis and then the z-axis for the Bryant angle convention used in ship motion models such as the one employed in SMP;
- translate the body to the centre of rotation, stored in the property of the object; and
- scale the body by the factor stored in the property of the object.

Once the transformations are applied, the object is ready to be rendered. Several methods are offered by OpenGL to render a set of vertices and normals. However, to draw data stored in an array, a combination of *glNormalPointer*, *glVertexPointer*, and *glDrawArrays* is used. These commands have some optimizations built in to allow for the video card to access the vertex and normal data in a more direct manner.

2.5.5 Animation Rendering for Simulation

Initially, the DynamicsViewer uses the Windows multimedia timer to keep track internally of the passage of time. However, with the integration to the VFD-RT environment, the timer is no longer necessary, as the animation can be timed using the information provided

by the VFD-RT application. Also, the original DynamicsViewer loads a fixed number of frames according to the motion profile loaded. Rather than having a set of pre-determined frames in the animation, the new version now has an undetermined number of frames. Due to these considerations, it was decided to have the client loop one frame during the animation with the data from the most recent data copy message. The dead-reckoning technique can be applied to improve visual fidelity, but for the purposes of demonstrating the capability as a client, the extra set of calculations for dead-reckoning is unnecessary, particularly when a simulation runs at a sufficiently high frequency such that the human eye cannot perceive the "jumps."

When new simulation data is received, the transformations are applied to the bodies and the frame is redrawn immediately, so that the frame is up-to-date. This method precludes the possibility of dynamically adding objects to the scene during runtime, but simplifies the integration task and allows for efficient animation rendering. Additionally, workarounds can be devised if it is known beforehand that an object is to be added to the scene, such as hiding the object from camera view until it is needed.

Chapter 3

Verification, Performance Evaluation, and Validation

The development of the VFD-RT had several stages of testing to ensure that the implementation complies with the design model and with the project objective, which is to develop a suite of applications that serves as a versatile control centre for simulation and that allows independent simulation model development. To deliver a reliable simulation system, the test process must include:

- verification of the code to confirm that the implementation meets design specification;
- performance evaluation of the system to characterize limitations and validate real-time performance; and
- validation of the capability of the environment to execute a variety of simulation requirements.

This chapter is divided into sections that describe each of these stages of testing for the VFD-RT. First, the verification of the VFD-RT code consists of testing each component to ensure that they operate as designed. The method requires careful scheduling and extensive planning of the tests, such that a range of possible input cases is covered in an organized manner. In the second section, the system performance is evaluated on aspects related to the application code and to the communication via Windows messages. These are the drivers

of the evaluations because these aspects pertain critically to the real-time characteristic of the system.

The third part is the validation of the VFD-RT concerning its capabilities as a simulation environment, to execute a variety of ship-related simulations. Cases are presented to validate the environment with simulations at different scales. Some research projects may require a simulation for data generation and analysis; other larger-scale projects may use a simulation with a feedback loop that sustains a complex virtual environment; still other projects apply the simulation for system research and development. The validation of the VFD-RT environment will cover this range of cases.

For all of the testing performed in this project, the computer had the following hardware configuration:

- Genuine Intel duo core CPU, 1.60 GHz (533 MHz front-side bus, 2 MB cache);
- 1.99 GB of RAM;
- 80 GB hard drive; and
- integrated graphics card.

All tests were performed on the Windows XP SP3 Home Edition operating system.

A summary of the verification, performance evaluation, and validation processes are described in the following sections.

3.1 Verification

The formal testing of the VFD-RT application begins with the unit testing of the source code implementation. Unit testing is a method to verify the source code by dividing the code into logical testable units. In the VFD-RT application, these units are the individual classes as described in the class diagram in Figure 2.8. The methods within each class are tested to ensure that their operation matches the specification described by the collaboration diagrams and the expectation from their caller methods.

This subsection uses several terms related to the hierarchy of the application components. The *class* is a logical grouping of variables and functions, within the whole application, with a specific role. For example, the *Frame* class has the role of storing the simulation data. A class may contain *methods*, *members*, and other classes. A *method* is a function inside a class, to accomplish one task to fulfill the role of that class – this is the focus of our tests. The term *function* will be used here to refer to those functions that do not belong to any class. Thus if a "function" belongs to a class, it is a method; otherwise, it is simply called a function. A *member* is a variable within a class. A *unit* can either be a group of one or several methods, or an entire class.

3.1.1 Unit Testing

The main purpose of the unit testing is to ensure that every method of the application operates as expected. For each method in the tested class, a set of input-output pairs are listed and compared to the results produced. The VFD-RT has three types of this black-box style test:

- direct comparison – the *assert()* function is used extensively by most classes throughout the unit testing; *assert()* is a C-language function that stops the application if its input argument is a false Boolean expression, as in the case when an expected output is not equal to the actual output;
- user interface inspect – class methods that affect the user interface, for which *assert()* is not sufficient, are verified through visual inspection; and
- data copy verification – for class methods requiring the use of the data copy message, a small test client is used to display the content of the data copy message so the output can be verified manually.

For each class to be tested, a separate testing class is created containing one or two testing methods corresponding to each method of the tested unit. These testing methods

are where the *assert()* resides. A DOS-like console is included to display the test results [35], as seen in Figure 3.1.

Figure 3.1: Console used to display unit testing results.

3.1.2 Code Verification Procedure

To test all the methods, there needs to be a proper organization on two levels: the unit level and the method level.

On the class level, where each class is considered a unit, the order in which the VFD-RT is tested follows two paths.

First, those connected to the *Application* class, are tested in a top-down fashion: the methods in the top class, *Application*, are tested first, followed by the classes that comprises *Application*. This top class is where normal execution begins and depends on all the other classes. Since the other classes have not yet been tested, method stubs are created to generate only the expected output for a given set of test inputs. These stubs would also print a message to the console to show that they have been called. The top-down test direction is chosen because these top classes were designed through the specification diagrams. Since the sequence of the method calls is the critical design point of these methods, they are

tested for their call sequence through method stubs, in a top-down fashion.

A second direction applies to all the other classes not connected to *Application*: from bottom up. The methods of these classes generally have a specific response to a given input, such that they can be tested independently from the rest of the application. Bottom-up testing avoids the need for method stubs. This testing direction begins with classes that do not depend on other classes; it helps ensure that all the classes being tested use only classes that have been verified previously.

In the VFD-RT, the bottom-up testing starts with a few classes that depend only on their own members and methods. Five classes fall into this category:

- *LogFile* class recording information about the execution of the software;
- *StdString* class implementing a custom object-oriented version of the C++ string;
- *Timer* class which has the two derived classes:
  - *TimerRT* for real-time simulations; and
  - *TimerNRT* for non-real-time simulations.

With a detailed inspection of the source code and a careful ordering based on the dependencies of the classes, most classes can be tested such that they depend only on previously tested classes. For example, the button only depends on the *StdString* class, so the *Button* class is tested after *StdString* and before the *MainWindow* class. Starting with classes that depend only on the 5 independent classes that were tested first, a test ordering can be formed. Of the 28 classes tested, 18 of them can be ordered in this manner, in addition to the five independent classes:

- *InputProvider*;
- *WaitProgressDialog*;
- *Client*;
- *Frame*;

- *Control*;
- *Button*;
- *ComboBox*;
- *EditBox*;
- *ListBox*;
- *InputManager* (depends on *InputProvider* and *Frame*);
- *ShipMotionGeneratorDialog* (depends on *Control* and *ComboBox*);
- *ClientManager* (depends on *Client* and *Frame*);
- *DataGroup* along with *HelicopterDataGroup*, *ShipDataGroup* and *WeatherDataGroup* (depends on *Control*, *ComboBox* and *EditBox*);
- *SimulationGroup* (depends on *Control*, *Button*, and *EditBox*); and
- *ShipMotionGenerator* (depends on *WaitProgressDialog*, *InputManager* and *ShipMotionGeneratorDialog*).

The remaining five classes are close to the top of the class hierarchy:

- *Application*;
- *ProviderDialog*;
- *ProviderGroup*;
- *ClientGroup*; and
- *MainWindow*.

These classes cannot be ordered as they are because their dependency is more complex, with some circularity whereby a class can depend on itself. Therefore, they are subdivided into groups of methods, forming new test units. These units can thus follow the ordering pattern based on their dependence on other units:

1. *ClientGroup 1:* constructor and destructor;
2. *Application 1:* constructor, destructor, *getHandle()*, *setHandle()*, *deinitialize()*, *incrementNextSendTime()*, *listRemainingProviders()*, *resetNextSendTime()*, and *findProvidersProc()*;
3. *ProviderDialog 1:* constructor and destructor;
4. *ProviderGroup 1:* constructor and destructor;
5. *MainWindow 1:* destructor and *close()*;
6. *ClientGroup 2: getSelectionData()*, *setChildrenHandles()*, and *updateList()*;
7. *Application 2: execute()*, *close()*, *generateShipMotion()*, *pauseSimulation()*, *resumeSimulation()*, *startSimulation()*, *stopSimulation()*, *assignApplicationHandleAndName()*, *disconnectAllClients()*, *disconnectAllProviders()*, *initialize()*, *runMessageLoop()*, and *sendData()*;
8. *ProviderDialog 2: initialize()* and *retrieveSelectedProviderName()*;
9. *ProviderGroup 2: disableButtons()*, *enableButtons()*, *getSelectedData()*, *setChildrenHandles()*, and *updateList()*;
10. *MainWindow 2:* constructor, *initialize()*, *setSimulationSetting()*, *show()*, and *assignChildrenHandles()*;
11. *Application 3: disconnectClient()*, *disconnectProvider()*, *initializeWindow()*, *requestProviderConnection()*, *updateClientListInWindow()*, *updateProviderInformationInWindow()*, and *mainWindowDialogProc*;
12. *ProviderDialog 3: providerDialogProc()* and *getSelectedProviderName()*;
13. *MainWindow 3: getSelectedClientIndex()*, *getSelectedProviderIndex()*, *getSimulationFrequencyInHertz()*, *getSimulationStartTimeInSec()*, *getRealTimeIncrementInMs()*, *updateClientList()*, and *updateProviderInformation()*; and

14. *Application 4: getSelectedClientIndex(), getSelectedProviderIndex(), receiveData(), applicationProc()*, and *createApplicationWindow()*.

Each unit depends on one or more methods in the units earlier in the list, or on previously tested classes. Once the unit-level ordering is complete, methods are ordered within each unit. The methods that are completely independent are tested first, followed by those that depend on previously tested methods.

The VFD-RT has 302 class methods and functions. These include

- 1 C++ main function, which has only 4 lines of code and was verified by inspection;
- 5 pure virtual methods, which have no code (used for class inheritance);
- 33 constructors, which are methods called when an instance of the class is created, and which were inspected and tested to ensure that all the class members are properly initialized; and
- 28 destructors, which are methods called when an instance of the class is destroyed, and which were verified for the proper deletion of all class member pointers.

The other functions and methods are tested normally using the unit testing classes.

3.1.3 Verification Remarks

The unit testing brought improvements concerning:

- the automatic disconnection of the providers and clients when the VFD-RT application closes;
- the sorting of the providers and clients on the user interface by order of connection;
- the use of messages to selectively prevent the execution loop from looping unnecessarily when the simulation is not running;
- the optimization for usability; and

- the selection of process priority.

These improvements did not involve major design changes. Concerning the optimization and priority, the collaboration diagram for simulation start (Figure 2.9) was updated to reflect the changes. Overall, the code was successfully tested and its operation verified.

3.2 Performance Tests

In evaluating the VFD-RT simulation environment, the factors affecting the real-time performance can be divided into two categories:

- data transfer through the operating system; and
- function execution dependent on the VFD-RT code.

In the first category, the effects of the operating system on a simulation involves the time to transfer data from one application to another using the data copy message. A data copy message performance evaluation is carried out to determine how the operating system affects the VFD-RT simulation through the data copy message.

For the second category, the performance of the code is examined. The majority of the simulation development time was spent on ensuring that the VFD-RT code runs as close to real-time as possible. However, to characterize the real-time performance, the evaluation must take into account the variation in loading – that is the number of connected applications. Therefore, the VFD-RT is tested under various load conditions. The goal is to determine the limitations of the VFD-RT in terms of the number of provider and client connections without significant deterioration in running the simulation in real-time.

This section describes the data copy message test and the loading test performed, along with their results.

3.2.1 Data Copy Message Performance Tests

The data copy message is a major factor in transfer delays because it carries the largest data structures of all the messages, like the simulation data frame. Other messages sent

to the user interface can be large as well, but can be avoided through design. Therefore, a study of the delays caused by the data copy message is of interest to determine how the operating system affects a simulation cycle of the VFD-RT.

The data copy message is sent through the *SendMessageTimeout()* function in the VFD-RT. When called, the operating system copies data to a memory location on the receiving application. The system then provides to the receiving application a pointer to this new memory in order for it to be read and used. The delay of the data transfer due to the operating system is caused by this data copying operation.

Additionally, within the calling and receiving application, some infrastructure is required to be implemented in order to use the data copy message, the call to the *SendMessage-Timeout()* function being the first. Others lie on the receiving end: the window procedure receives all the incoming messages; then a structure of conditional statements determines the identity of each message. Only once the data copy message has been identified can the application use the copied data. This extra infrastructure delay affects all the VFD-RT applications that make use of the data copy message.

For evaluating the performance of the VFD-RT code, the data copy message delay can be defined as the combination of the operating system data copying delay and the application infrastructure delay. This delay is equal to the total time difference between just before the call to the *SendMessageTimeout()* and just after message identification by the receiving window procedure, inclusively. This message delay cannot be shortened by the code of the VFD-RT. Measuring this delay would provide a theoretical upper limit for the simulation frequency.

Message Test Method

Since the data copy message is sent between processes, two applications were created specifically for this test: a client, and a server that initiates the communication. Both must be Windows applications because they must process messages, and in particular the data copy message. Their design should be such that the communication procedures mimic those between the applications of the VFD-RT.

Timer Considerations To measure the delay of the message transfer, the typical method is to record the computer clock times at two points in the execution and obtain the difference. The Windows API offers two timers for the task, which are the multimedia timer and the high-performance timer. Because the delay is expected to be below 1 millisecond, the millisecond resolution of the multimedia timer is inadequate. Therefore, the high-performance timer is used for the test, using the *QueryPerformanceCounter()* function. With this timer, careful implementation is required in a way that the timing is measured from only one CPU, to avoid any potential discrepancy between CPU times.

One consideration concerning timing the delay is the location of the time measurement code. An initial design places the measuring points on the server and the client: on the server prior to sending the message, and on the client, immediately after the message has entered the window procedure. However, the time in that case is measured on two separate processes, with the possibility that they run on two separate CPUs. Alternatively, a two-way sending configuration can avoid this problem. The first data copy message is sent by the server, and the client sends back a copy of the same message, mirroring the procedure that the server took for the first message. By timing between the first sending and the receiving of the return message, the timer can be kept on the server application alone, while the delay of the data copy message can be evaluated as half of the measured time difference.

Test Operation To better visualize the test procedure, Figure 3.2 shows a sequence diagram of the function calls made within one test cycle. Each application run is illustrated by a vertical dashed line with time increasing down the lines. The operating system is also included in the diagram to distinguish the function calls and to show the operation of the *PostMessage()* function. A shaded band shows when the function call is active and ends by returning control to the caller. The lengths of the bands are not scaled to time, but rather simply show the sequence of the function calls.

When the user presses the start button on the server user interface, the server application sets up the communication by finding the client window handle and creates a test data copy structure. Then the function *sendCopyData()* is called (Step 1), which records the time (2)

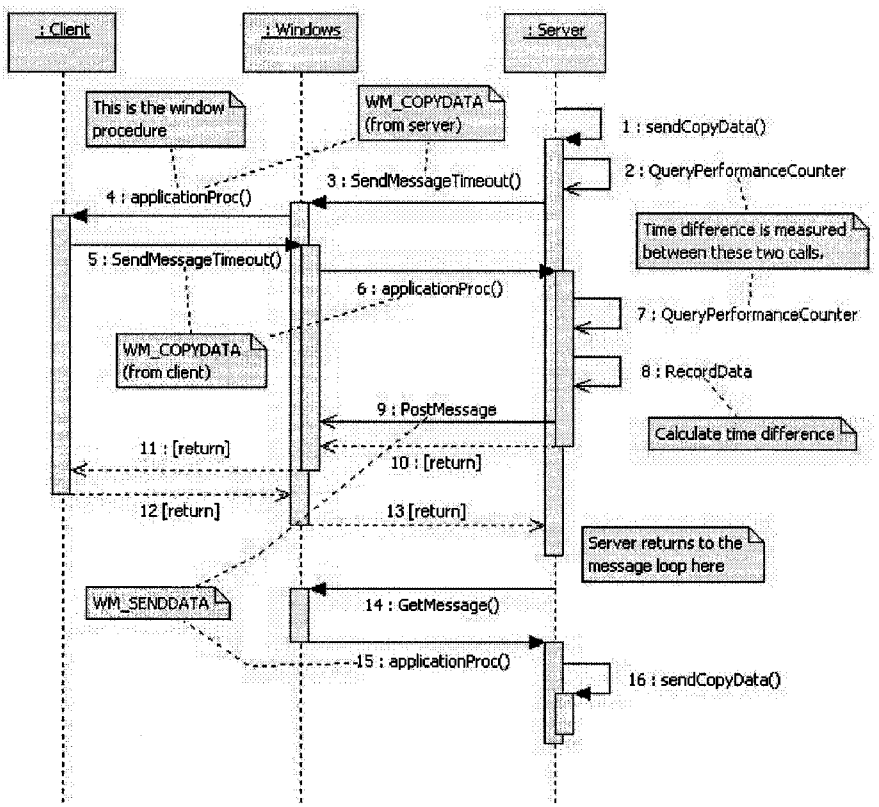

Figure 3.2: Sequence diagram of the data copy message test setup.

prior to sending out the message (3). The operating system passes the message to the window procedure of the client (4), which verifies that it is a data copy message. The client then sends back a data copy message with the received data (5) which is forwarded to the window procedure of the server (6). Upon receiving the message and ensuring that it is a data copy, the server records the time (7), closing the time measurement period. The time difference can then be calculated (8), outside the delay of interest so as to not interfere with the measurement.

If more message tests are required, the server posts a *WM_SENDDATA* message to the operating system (9) so that when all the functions currently active on the server complete and return to the message loop, the server will send another data copy message. As the processing in the server window procedure is completed, the active functions in all three entities return control to their callers (10-13), ending with the server returning control to its execution loop. In the execution loop, the server retrieves the next message in the message queue, which is the *WM_SENDDATA* posted earlier (14). The operating system calls the window procedure of the server to process this data sending message (15), and the cycle restarts.

In a manner similar to the VFD-RT application, the function *SendMessageTimeout()* is used to send the data copy messages. The parameters for this function control how long the function waits until it completes, regardless of the time the receiving application takes to process the message. Like the VFD-RT application, this timeout period is set to one millisecond.

The test data copy structure used within the test message is constructed to simulate the VFD-RT data frame, size being the critical factor in determining the speed of any data copying operation. The test structure size can be set in the user interface, as shown in Figure 3.3. When the start button is pressed, the server creates a data copy structure with the given data size in bytes, populated with a character array. The character array is convenient because one character takes up one byte.

On the user interface, the number of time delay measurements per replication and the number of replications can be set. Each measurement should theoretically be independent

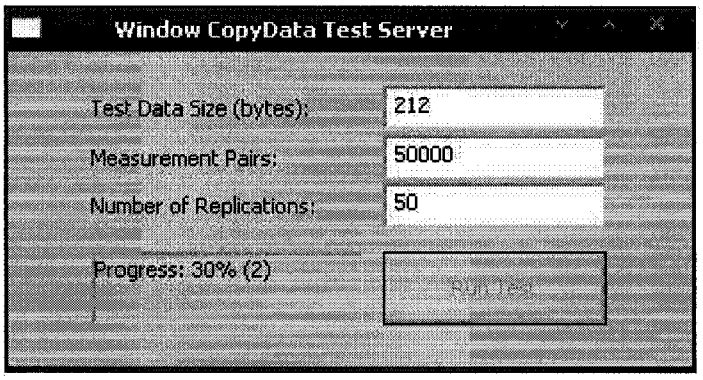

Figure 3.3: User interface of the data copy message test server.

from all others, but in case there exists some dependence between measurements (due to the state of the operating system or the accumulation of messages in the message queue), the test is divided into replications. A two-second wait period is set between each replication to ensure that all queued messages are purged. Each replication produces an independent measurement average, which, by the central limit theorem of statistics, is normally distributed and simplifies statistical analysis of the results.

The status box at the bottom right indicates the progress of each replication in percentage and the replication number, in parenthesis.

The client simply has a blank user interface, except for the system title bar. The only purpose of the client is to return test data copy messages, so no specialized user input is required.

Two-Way Message Delay Calculations The two time measurements are taken between the data copy message sending by the server and the receiving of the a copy of that message in the window procedure, inclusively, as shown in Steps 2 and 7 of Figure 3.2. Each *QueryPerformanceCounter()* call returns a 64-bit integer value equal to the number of CPU performance timer ticks since the last computer start-up. For the CPU on which the tests are run, one second real time is equivalent to 3,579,545 ticks. This conversion value changes from CPU to CPU, and can be obtained by calling the API function *QueryPerformanceFrequency()*.

An issue to consider is the overhead of each *QueryPerformanceCounter()* call, which adds ticks to the measurements. The overhead is the computer processing time needed to read the timer. To remove this overhead bias, after each two measurements, two consecutive calls of the query-counter function are made. The difference in the time of these successive calls reveals the overhead bias.

Finally, then, the two-way message delay can be obtained from measurements as

$$d_i = \frac{t_{2,i} - t_{1,i} - b_i}{f} \tag{3.1}$$

where d_i is the two-way message delay in seconds for the i^{th} test, $t_{1,i}$ and $t_{2,i}$ are the first and second time measurements, in ticks, respectively, b_i is the overhead bias in ticks, and f is the performance timer frequency retrieved from *QueryPerformanceFrequency()*, in ticks per second.

Test Execution Procedure The server and client applications are run for 40 replications of 50,000 pairs of time measurements. These values ensure that there is enough statistical evidence to support the results of the average two-way message delay time. The test data copy structure size is set to match the size of the largest VFD-RT data structure, which is the data frame. The number of connected providers determines the number of *State* structures in the frame, which in turn affects the size of the structure. The data frame is made up of a header that is 256 bits long, or 32 bytes, and a number of *State* structures of 180 bytes each. The size of the test structure, then, starts at 212 bytes for one *State*, and increments by 180 bytes up to a total of 5 providers, which gives a 932-byte frame.

After the test measurements are obtained, a statistical analysis is performed on each replication to obtain the average of the two-way message delay, following the equation:

$$\bar{d} = \frac{1}{R}\sum_{r=1}^{R} \bar{d}_r \tag{3.2}$$

where $\bar{d}$ is the overall average two-way message delay, $\bar{d}_r$ is the average delay of replication r, and R is the number of replications. The standard deviation $s_{\bar{d}}$ of the replication averages

are calculated as follows:

$$s_{\bar{d}} = \frac{1}{R-1}\sum_{r=1}^{R}(\bar{d}_r - \bar{d})^2 \tag{3.3}$$

Histograms were also produced, divided into 200 bins ranging from 0 to 0.1 milliseconds, to provide a visual representation of the distribution of the message delays. For each of the cases corresponding to the 5 different structure sizes, a 95% confidence interval is calculated for the replication averages, assuming negligeable systematic error associated with the measurements. Because there are 40 replications, the z-score is used, along with the level of confidence $\alpha = 0.05$, which, for a two-tailed interval, is taken to be $z_{\alpha/2} = z_{0.025} = 1.95996$. The equation for the confidence interval is

$$[\bar{d} - z_\alpha \frac{s_{\bar{d}}}{\sqrt{R}}, \bar{d} + z_\alpha \frac{s_{\bar{d}}}{\sqrt{R}}] \tag{3.4}$$

Results

The results for the data copy message test are summarized in Table 3.1, in terms of the average and standard deviation of the 40 replication means for each test data structure size. A plot of this data is shown in Figure 3.4 to better visualize the data.

Table 3.1: Average two-way message delay for varying test data size.

Data size (bytes)	Average time (ms)	Standard dev. (ms)	Conf. Interval (%Average)
212	0.024315	2.4656×10^{-4}	±0.31425%
392	0.026324	2.4051×10^{-4}	±0.28313%
572	0.028831	2.4500×10^{-4}	±0.26335%
752	0.032085	3.5525×10^{-4}	±0.34312%
932	0.032925	4.8934×10^{-4}	±0.46058%

The plot in Figure 3.4 shows a roughly linear trend, as expected, since the main factor in delay varying for copying data is the data size. The 95% confidence interval is barely visible, meaning that for the system conditions in which the test is run, the average delay

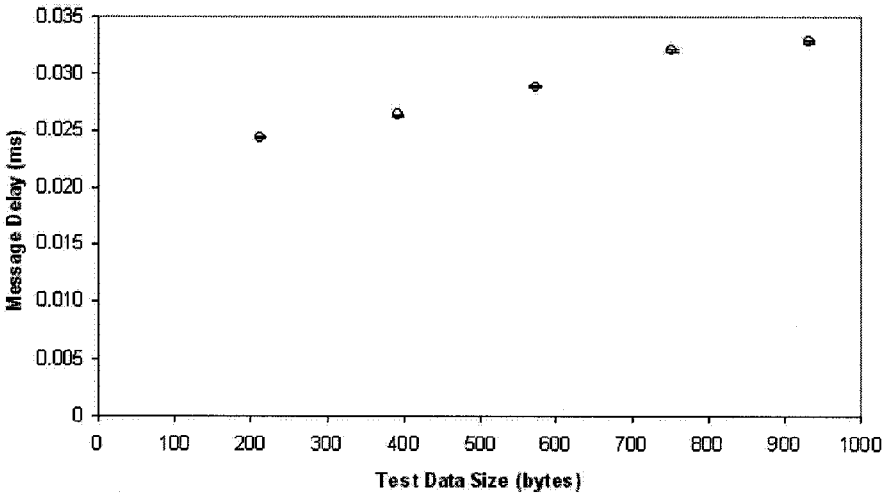

Figure 3.4: Plot of the variation of the average two-way message delay due to varying test data size.

obtained is very likely to be the real two-way message delay. The conclusion that can be drawn from the plot is that, given similar system conditions, there is a roughly linear relationship between the two-way message delay and the data structure size. Moreover, these delays are significantly smaller than the 1-millisecond resolution of the VFD-RT timer and may only lengthen the simulation cycle by a few hundredths or tenths of milliseconds. For example, for a 212-byte structure, representing a simulation data frame with a single provider, the one-way message delay is about half of the measured value in Table 3.1, or about 0.0122 milliseconds.

All the two-way message delays are classified in five 200-bin histograms divided by test data size, shown in Figures 3.5 to 3.9. An observation can be made about the location of the peaks: they shift toward a higher delay value with increasing test data size. As the test data size increased, the variation in the delay increased as well, producing a flatter histogram, from which the peaks are less detectable. The histograms also show, especially for the lower data size tests, two distinct peaks for each test: around 0.024 and 0.026 milliseconds for the 212-byte tests, around 0.026 and 0.029 for the 392-byte tests, and around 0.029 and 0.031 for the 572-byte tests. To investigate this observation, the test runs are plotted with time for the measured message delays, as shown in Figure 3.10. The tests are numbered

successively so the test number increments with time.

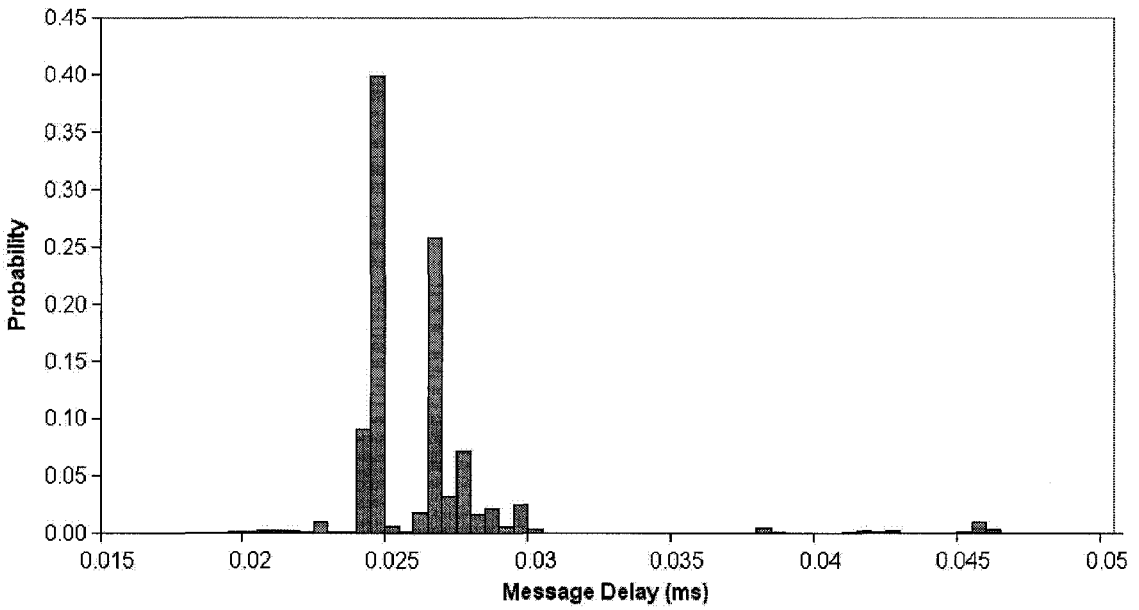

Figure 3.5: Histogram of the two-way message delay distribution for the 212-byte structure test.

This section shows the typical variation of the delays for the 212-byte structure tests. Occasionally due to background operating system functioning, the delay is increased on a test. There is generally no regular pattern for these peaks, and they vary in height up to about 0.256 ms. However, the large peaks occur very rarely, and therefore do not appear on the histogram. Most peaks, as the histogram suggests, are smaller, reaching about 0.026 ms, or approximately 0.002 ms above the most common delay of 0.024 ms. The histogram and the test run section are scaled for clarity; however, the actual difference between the histogram peaks is less than 10% of the average. In terms of the VFD-RT simulation, this variation does not impact significantly the overall effect of the data copy message on the performance of the simulation.

These results are significant only in as far as they are applied to other benchmark performance testing of the VFD-RT simulation on the same computer and with similar background configuration of the operating system. The measured delay values would be different on another system with a different CPU or background conditions. The values of

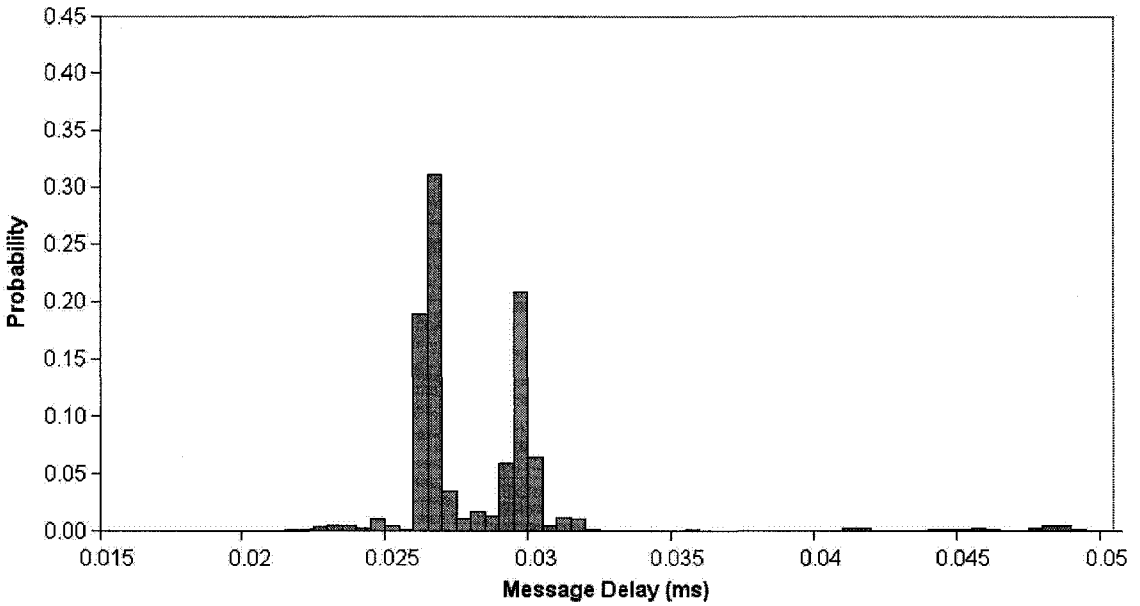

Figure 3.6: Histogram of the two-way message delay distribution for the 392-byte structure test.

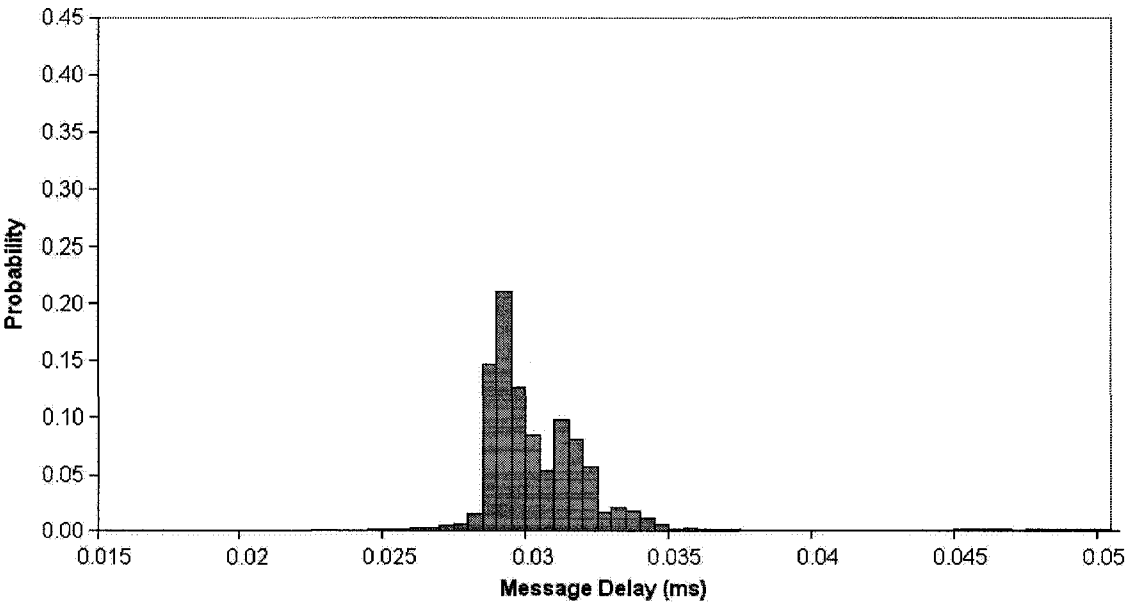

Figure 3.7: Histogram of the two-way message delay distribution for the 572-byte structure test.

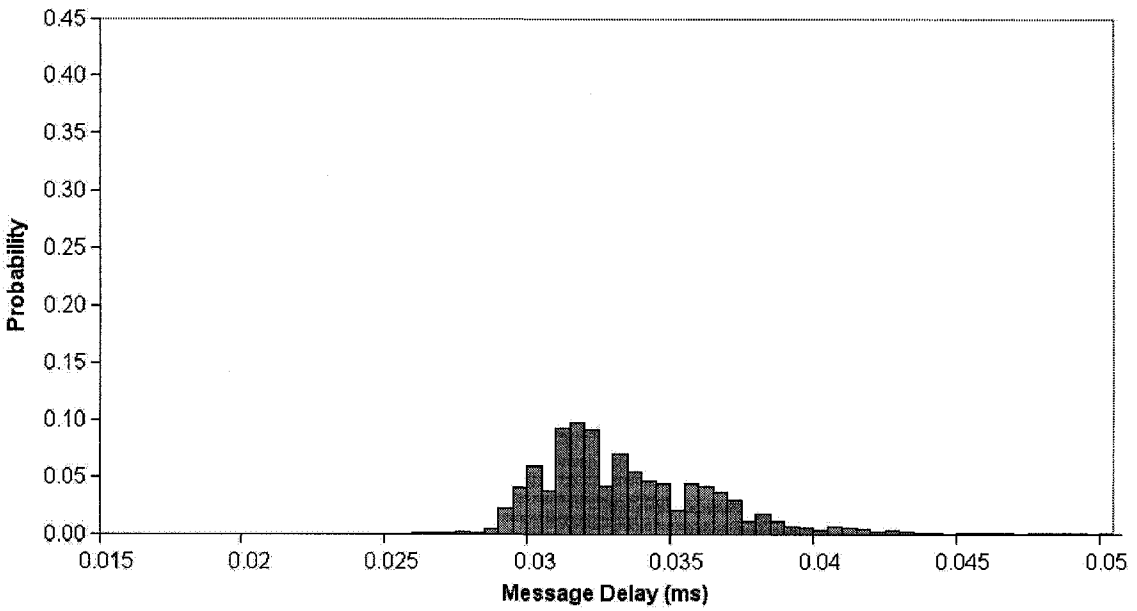

Figure 3.8: Histogram of the two-way message delay distribution for the 752-byte structure test.

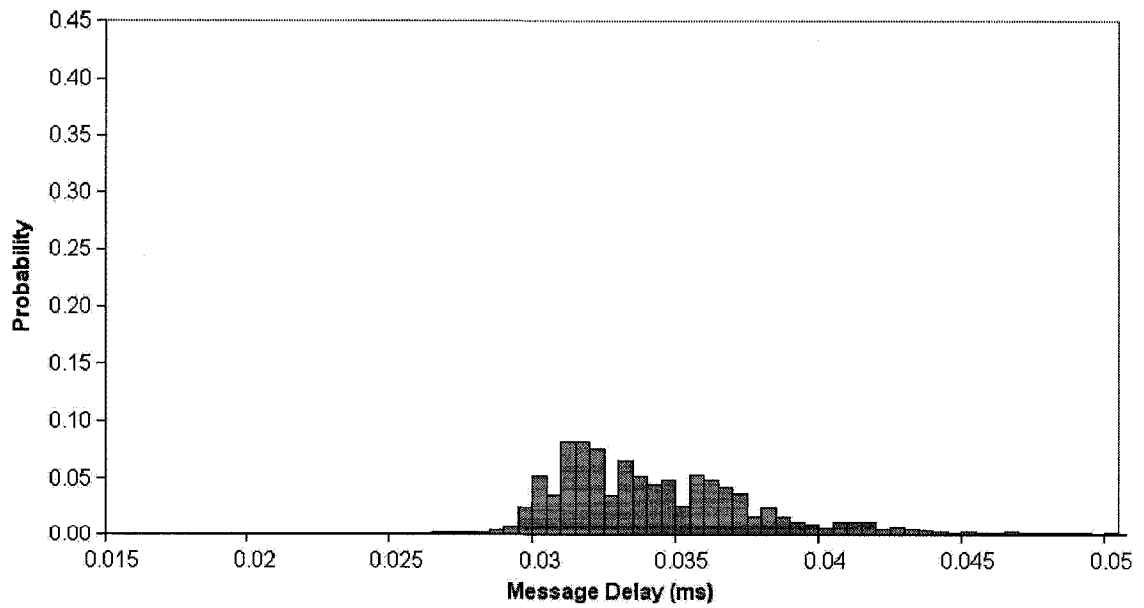

Figure 3.9: Histogram of the two-way message delay distribution for the 932-byte structure test.

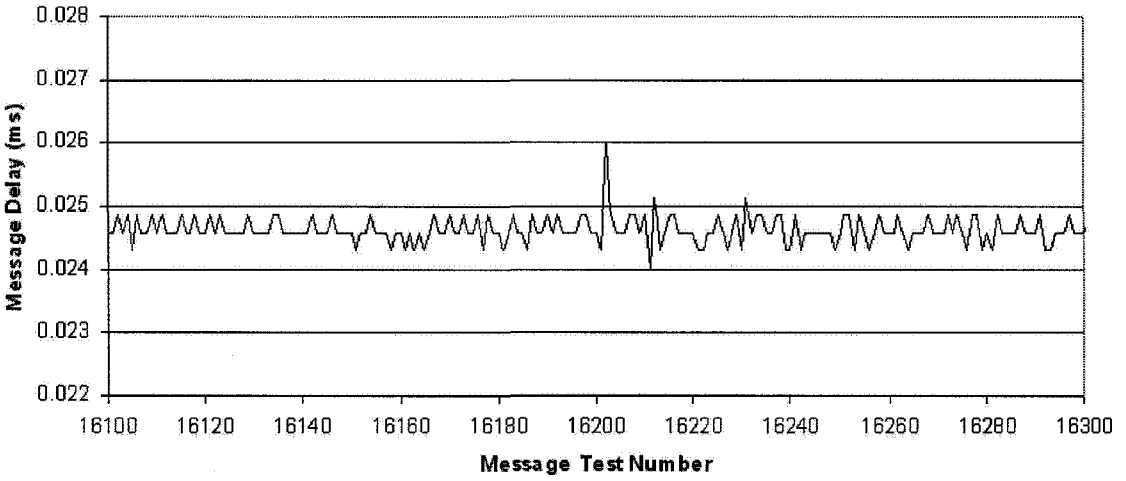

Figure 3.10: A section of the tests run for the two-way message delay measurements.

the message delay found here will be considered in the analysis of the results in subsequent performance tests of the VFD-RT simulation.

3.2.2 VFD-RT Load Performance Test

With each connected provider and client, the VFD-RT application must take time within a simulation cycle to send data through the data copy message. As the numbers of providers and clients increase, a limit is reached where the time to serve these connected applications exceeds the simulation cycle period and causes delays, so that the simulation is no longer running in real time. An evaluation is undertaken to determine the system loading, which is the number of providers and clients, under which the simulation performance begins to degrade.

The real-time performance deteriorates when the processing time for the VFD-RT operations reaches and exceeds the simulation cycle period. The two relevant factors are the processing times and the length of a simulation cycle. The latter is set by the user and can therefore take an arbitrary value within the range of simulation frequencies permitted by the user interface. On the other hand, the processing time depends, to a large extent, on the manner data is being manipulated in the simulation, which relates directly to the source code. Preliminary tests have shown that the variation of processing time with elapsed

time is small. Therefore, the processing time can be a measure of the simulation environment performance. Characterizing this measure would determine the performance limits of the simulation application. The processing time sets an upper bound on the simulation frequency value. As the system loading increases – that is, as the numbers of connected providers and clients increase – the processing time would increase as well, intuitively. The relationship between these two variables is the subject of the load performance test.

The system loading within a simulation cycle in the VFD-RT application is related to two tasks, both described by collaboration diagrams previously: serving data to clients while requesting data from providers (Figure 2.13), and receiving data from providers (Figure 2.12). The time to perform some steps in these tasks, such as incrementing time and frame number, is fixed. The increase in the number of providers would affect the number of provider data requests that are sent out, and the number of message receptions from providers which require an amount of processing. The number of clients is directly related to the number of data frame copies that is to be made and sent using the data copy message.

In addition to the processing within the VFD-RT application, processing of simulation data by the providers and clients also takes up CPU time. Because of the limited number of CPUs, the amount of time for these applications to handle data influences the overall simulation performance. If a client requires a large amount of processing, the CPU would be too busy to process tasks from other clients, thus causing delay. This delay in the processing time is dependent on the provider or client used. The impact from the provider inefficiencies is more prominent than those of the clients, since the simulation is driven by provider data. The client applications processing also consumes CPU time, but their operation does not affect the simulation run directly, and they can run on a separate thread, on a different CPU than the VFD-RT application. An exception is the joint provider-client application, where the provider uses the client data for generating simulation data; in that case, the client processing time has a significant effect on the simulation.

Performance Test Method

The total processing time in a simulation cycle of the VFD-RT application can be divided into two phases. First, the application serves the providers and clients. By convention, a provider or client is designed to make a local copy of the data copy message content and return, before processing the local copy. The VFD-RT application provider and client service time encompasses therefore the sending of the data copy message, and the waiting for its return. The second phase is processing incoming provider data. This operation consists of copying incoming data into the data frame. These operations affect the overall processing time within one simulation cycle in the VFD-RT application.

Service Phase To measure the service phase – sending a message and waiting for the return – two applications, one test provider and one test client, were developed. By connecting the test client as the first of all the clients, and the test provider as the last of the providers, and by choosing appropriate time measurement points within the test applications, an estimate can be made of the total service time. The test client is connected first because, within one simulation cycle, sending data to clients is one of the first operations, as described in the collaboration diagram of Figure 2.13. Particularly, data is sent to clients before sending messages to providers. The test provider, on the other hand, is connected last because provider data requests are one of the last steps of the cycle. Therefore, by enclosing the service phase with test applications that measure time, the total service time can be measured. In this manner, the VFD-RT application, on which tests are performed, does not need to be modified. Figure 3.11 illustrates how this method measures the service time.

In the diagram, each set of 4 arrows, going from the VFD-RT to another application and back, is the servicing of one application. The figure shows an example of a test with one client and one provider, besides the test provider and client. Initially, the test client, being the first of the clients, measures the start time prior to the message returning to the VFD-RT in Step 3. In the end of the cycle, the test provider measures the end time immediately after receiving a data request message from the VFD-RT in Step 14. The difference in the

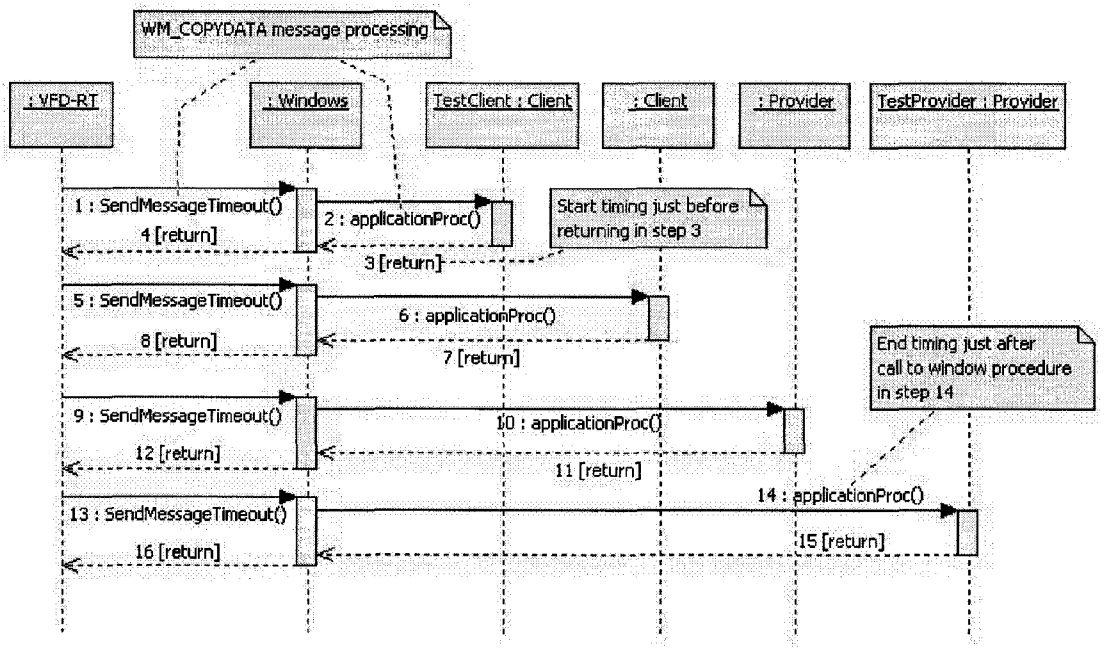

Figure 3.11: Sequence diagram of the system loading performance test setup.

two times gives the total service time plus an overhead delay, in Steps 3, 4, 13, and 14. This overhead can be found by connecting only the test provider and client. Once this overhead delay is identified, it can be subtracted from the measured time difference to obtain the service time.

The test applications use the *QueryPerformanceCounter()* high-resolution timer to obtain microsecond-level resolutions of the time measurements. However, as discussed in the timer selection considerations in Section 2.3.3, the high-resolution timer can cause unexpected differences in multi-core systems, when two CPUs are used to obtain times. Therefore, both the test provider and test client call the *SetProcessAffinityMask()* function of the Windows API to assign a single CPU to process all their tasks. In this manner, all time measurements are made from one CPU.

The collaboration diagram (Figure 2.13) shows the timer call at the beginning of the cycle, and the *Frame* update at the end of the cycle. These are the secondary operations of the service phase. Both the timer call and the frame number increment depend mainly

on the source code within the application, and not on operating system functions. Both of these involve code modifications in the VFD-RT application to include time measurements. These secondary operations are measured separately from the tests involving providers and clients.

The total service time, Δt_s, can then be estimated as the difference of measured time, Δt_m, between the test client and the test provider, from which the measurement overhead, Δt_{m0}, measured with only the test client and provider connected, is subtracted, plus the duration of the secondary operations, Δt_{sec}:

$$\Delta t_s = \Delta t_m - \Delta t_{m0} + \Delta t_{sec} \quad . \tag{3.5}$$

Provider Data Process Phase The second phase is the processing of new simulation data from providers. This phase involves a direct copying of the data to the data frame. The duration of this phase, Δt_{prov}, is measured through the modification of the VFD-RT application code to include time measurements.

Summing the service time and the provider data processing time, the total processing time, Δt_p, within one simulation cycle can be calculated as

$$\Delta t_p = \Delta t_s + \Delta t_{prov} \quad . \tag{3.6}$$

Even though the client also sends an acknowledgement data copy message to the VFD-RT application, the message has no data and is not processed by the VFD-RT, so there is no client data process phase.

Modified Service Phase Thus far, the theory assumes that the provider or the client makes a local copy of the data to be processed later, returning control to the VFD-RT immediately after the local copy is made so that the VFD-RT can serve the other connected providers and clients. The advantage of this design is that the VFD-RT application can continue to send data copy messages to other providers or clients, while, concurrently, the data is being processed by the ones that have already received the simulation data. This

parallel processing takes advantage of the multiple CPUs on certain systems and improves the speed at which all the clients receive simulation data. This convention is implemented through the use of the *PostMessage()* function, which puts a message on the application queue in order to be processed after the current message processing is complete. A typical sequence of operation is illustrated in Figure 3.12. The figure shows a client, but the same sequence applies to providers.

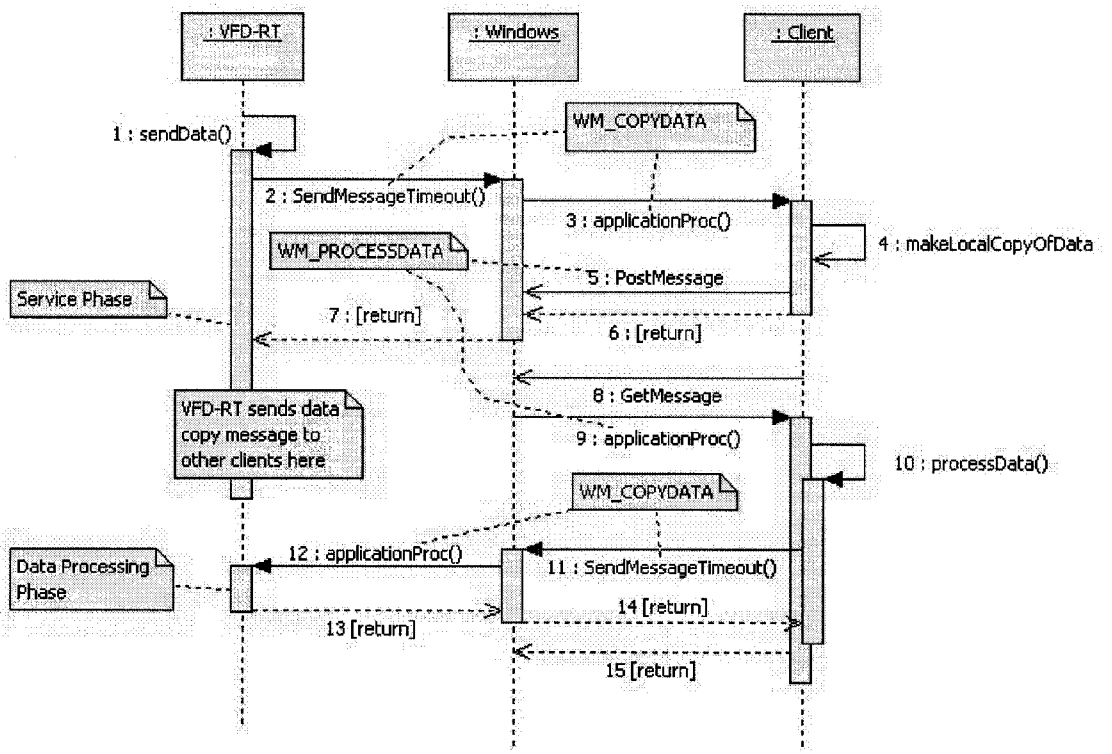

Figure 3.12: Sequence diagram of a typical VFD-RT client using the *PostMessage()* function.

For the VFD-RT application, the service phase begins in Step 1, followed by a data copy message sent to the client (Steps 2-3). After copying the data (Step 4), the client uses a *PostMessage()* call to put the *WM_PROCESSDATA* message on its own queue (Step 5), and then returns (Steps 6-7). The VFD-RT application is now free to serve other clients. Meanwhile, the client gets the *WM_PROCESSDATA* message from the message queue and begins to process data (Steps 8-10). If this were a provider instead of a client, new simulation

data would be sent back to VFD-RT application (Steps 11-12), which triggers the data processing phase. In the end the methods and functions return.

If *SendMessage()* is used instead, all the data processing of one client or provider, during one simulation cycle, is completed before the VFD-RT can proceed with serving the next client or provider. The data processing phase is merged inside the service phase. In real-time simulations, this method is highly inefficient, as the last clients or providers must wait for the earlier ones to complete their task, which may vary in time depending on the task. However, for purposes of testing, this technique is valuable because it merges the data processing with the service phase. Using *SendMessage()* effectively combines the two phases so that only the measured time between the test provider and the test client is necessary to determine the total processing time in one simulation cycle. Figure 3.13 illustrates this *SendMessage()* technique. The figure shows a client, but the same sequence applies to providers.

In equation form, the duration of the modified service phase, $\Delta t'_m$, is

$$\Delta t'_m = \Delta t_m + \Delta t_{prov} \quad . \tag{3.7}$$

such that the total processing time of Equation 3.6 becomes

$$\Delta t_p = \Delta t'_m - \Delta t_{m0} + \Delta t_{sec} \quad . \tag{3.8}$$

The value of Δt_p is the minimum simulation cycle duration, corresponding to the maximum simulation frequency, without deterioration of simulation performance. The maximum simulation frequency (in Hz) that the user can provide is the inverse of the total processing time,

$$f_{max} = \frac{1}{\Delta t_p}. \tag{3.9}$$

All the above time measurements are performed using the *QueryPerformanceCounter()* function. Due to the overhead bias, the actual time difference is given by an equation similar

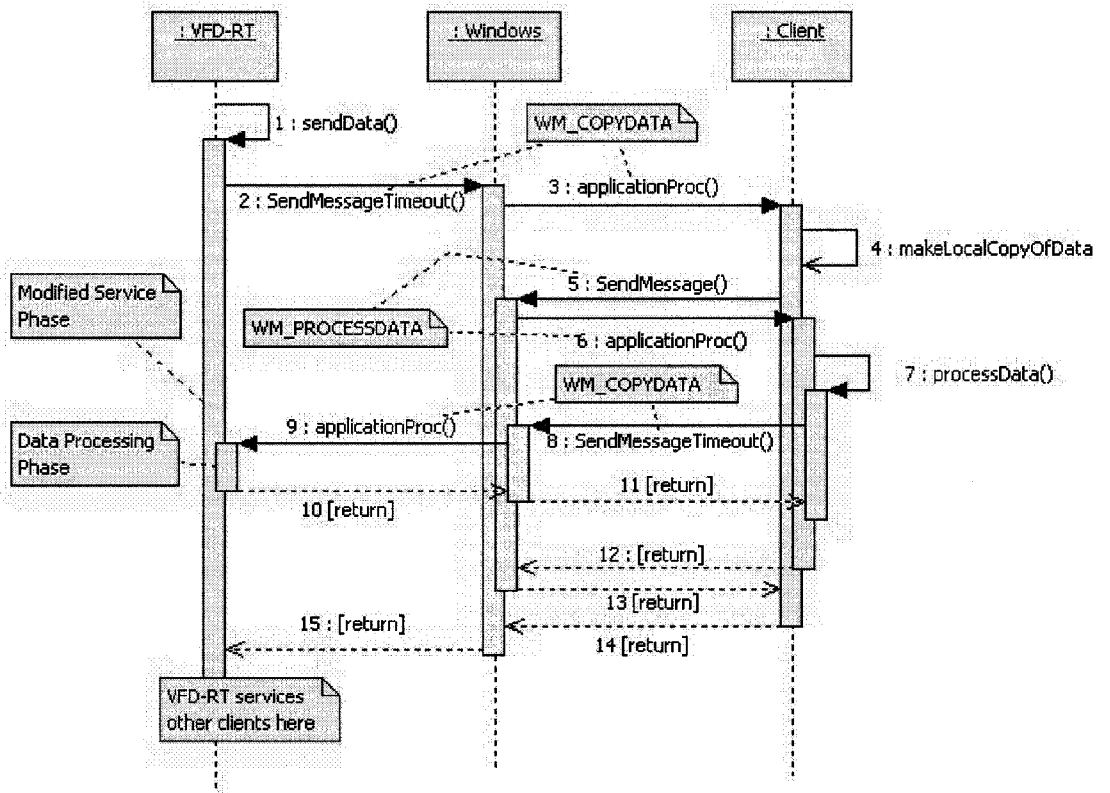

Figure 3.13: Sequence diagram of a modified VFD-RT client using the *SendMessage()* function.

to Equation 3.1:

$$\Delta t_i = \frac{t_{2,i} - t_{1,i} - b_i}{f} \tag{3.10}$$

where Δt_i is the difference in time measurements for the i^{th} test, $t_{1,i}$ and $t_{2,i}$ are the first and second time measurements respectively, b_i is the overhead bias, and f is the performance timer frequency retrieved from *QueryPerformanceFrequency()*.

Performance Test Results

Secondary Operations To measure the duration of secondary operations, which are the timer call and the *Frame* number increment, a total of 50 replications of 100 simulation cycles each were tested. The average of each replication is tallied, and the average of these large-sample averages, with over 40 cycles each, is evaluated. By the central limit theorem of statistics, this average of averages is normally distributed, so a confidence interval can be constructed to provide an estimate of the accuracy of the measurements.

Table 3.2 shows the average, standard deviation, and 95% confidence intervals of the secondary operation time measurements. The confidence interval uses a value of $z_{0.025} = 1.95996$.

Table 3.2: Average secondary operations time measurements.

Operation (bytes)	Average time (ms)	Standard dev. (ms)	Conf. Interval (%Average)
Timer Call	8.01×10^{-4}	7.61×10^{-5}	±2.63%
Frame Increment	4.50×10^{-4}	5.19×10^{-5}	±3.20%
Total	0.00125	9.86×10^{-5}	±2.18%

The total average of secondary operations is therefore approximately 1.25 microseconds, or 0.00125 millseconds. This value is significantly smaller than the data copy message operations seen previously in Table 3.1. Moreover, this value indicates that the resolution of the high-resolution timer may not be adequate. The limits of this timer are made evident when the majority of the 5000 time difference readings for each operation vary discretely

between 0.000279 ms, 0.000559 ms, and 0.001117 ms. Although these averages may have some reliability issues due to the resolution limit, the magnitude of the durations of secondary operations is distinctly smaller than any other procedures involving the data copy message. Therefore, the secondary operations will have minimal effect on the overall time of the service phase.

Modified Service Phase The modified service phase depends mostly on the way the provider or client processes data, since the *processData()* method in the client differs from one client to next, and from one provider to the next. Therefore, this section is divided into three parts: one for the ship motion provider (Section 2.4) as an example of a provider; one for a data display client, used for debugging, whose sole task is to display the received data in a text box on its main window; and the last for the DynamicsViewer client (Section 2.5), as examples of clients. The data display client is included because the DynamicsViewer client, by design, uses the same sequence of operation as the *PostMessage()* method. However, a comparison can still be made.

A total of 40 replications, of 100 simulation cycles each, were run to evaluate the modified service phase duration. Additionally, to test the load on the system with the number of providers and clients, each of the 3 test cases (Ship Motion Provider, data display client, and DynamicsViewer client) are repeated for 1 to 5 copies of the selected provider or client connected to the system. The data display client case is repeated for up to 9 copies to demonstrate that the linearity persists even at higher loads. The expectation is that the measured time difference should increase linearly with increasing copies of the same application connected, for the modified service phase method. The DynamicsViewer will display the 3-D model of a ship and a static sea.

Table 3.3 shows the average, standard deviation, and 95% confidence intervals of the measured time differences between the test client and the test provider, using the modified service phase ($\Delta t'_m$). The confidence interval uses a value of $z_{0.025} = 1.95996$.

SMP stands for ship motion provider, DDC stands for data display client, and DVC stands for DynamicsViewer client. A general observation is that the average time difference

Table 3.3: Load test client and test provider time difference measurements.

Prov./Client (bytes)	Average time (ms)	Standard dev. (ms)	Conf. Interval (%Average)
Baseline	0.02530	0.001992	±2.440%
1 SMP	0.3225	0.003072	±0.2952%
2 SMP	0.6119	0.004915	±0.2490%
3 SMP	0.9277	0.004193	±0.1401%
4 SMP	1.220	0.006168	±0.1566%
5 SMP	1.509	0.002033	±0.4174%
1 DDC	0.3540	0.005607	±0.4909%
2 DDC	1.049	0.002670	±0.7891%
3 DDC	1.335	0.03854	±0.8948%
4 DDC	1.929	0.07938	±1.276%
5 DDC	2.297	0.05016	±0.6767%
6 DDC	2.921	0.05282	±0.5604%
7 DDC	3.255	0.03651	±0.3476%
8 DDC	3.829	0.06071	±0.4913%
9 DDC	4.171	0.03637	±0.2703%
1 DVC	1.278	0.09008	±2.185%
2 DVC	8.501	0.3969	±1.447%
3 DVC	13.85	0.4725	±1.057%
4 DVC	24.06	0.6530	±0.8411%
5 DVC	33.79	0.7272	±0.6669%

increases with the amount of screen output: the SMP has no data-dependent screen output, while each DDC prints the numbers in a text box, and each DynamicsViewer client draws 3-D models of a ship and a flat ocean. For the SMP and the DDC, the measured value is $\Delta t'_m$, while the DynamicsViewer is measured with Δt_m because of its design.

These values can be applied to Equation 3.8 to obtain the total process time, using the secondary operations duration $\Delta t_{sec} = 0.00125$ ms and the baseline case as overhead, $\Delta t_{m0} = 0.02530$ ms. The plots of the total processing time are shown in Figures 3.14 and 3.15 for the SMP and the DDC respectively.

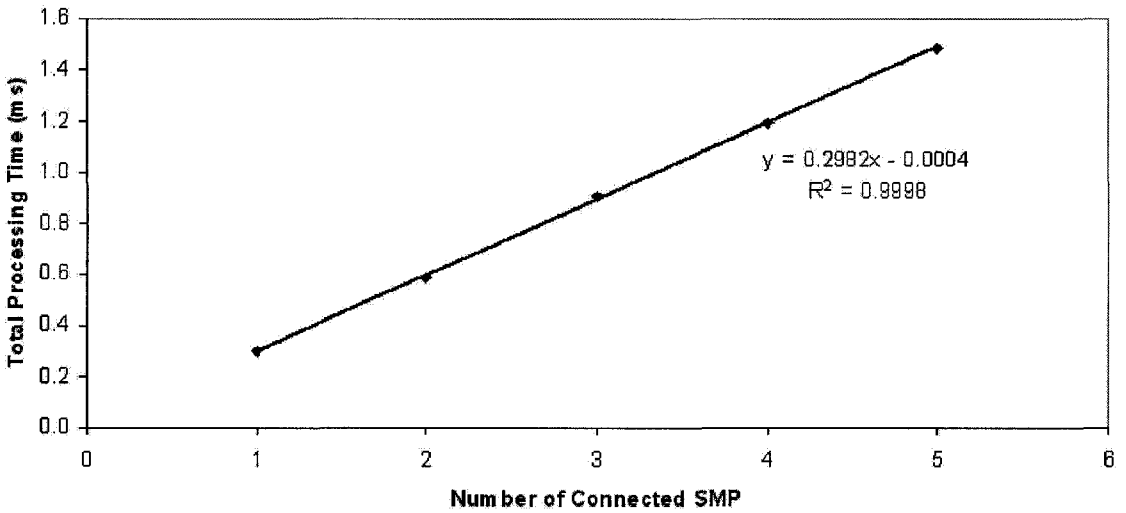

Figure 3.14: Total processing time variation due to increasing number of connected SMP.

Additionally, Figure 3.16 shows the service time for varying numbers of DynamicsViewer in the simulation. The service time is shown because the DynamicsViewer does not use the modified service phase method due to its design.

With strong agreement, the SMP total processing time plot (Figure 3.14) matches the linear regression, showing the linearity of the processing time with increasing load. The y-intercept is also noted to be very close to 0.

The plot for DDC (Figure 3.15) also has linearity, which extends even to the 9th client. One observation that can be made is that there appears to be an oscillation around the

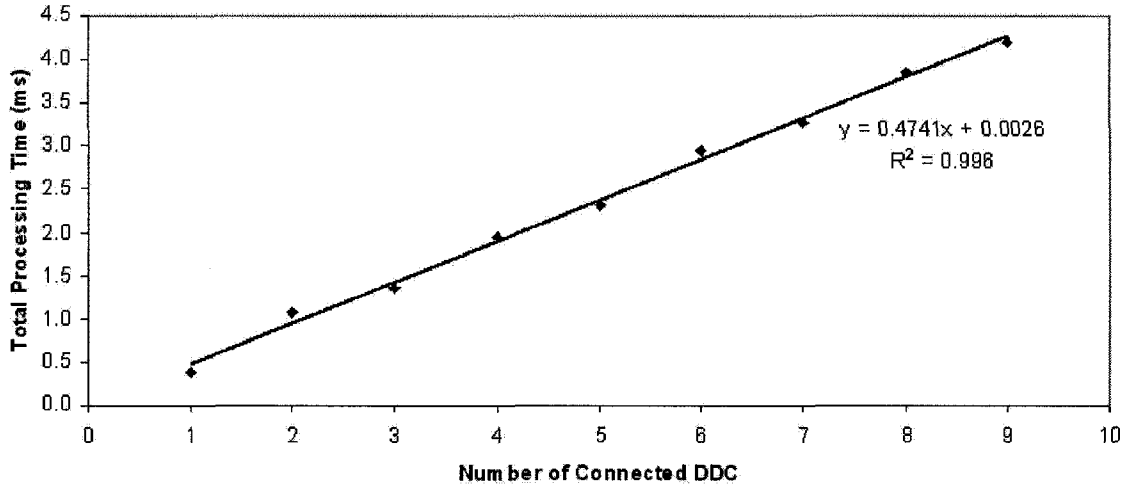

Figure 3.15: Total processing time variation due to increasing number of connected DDC.

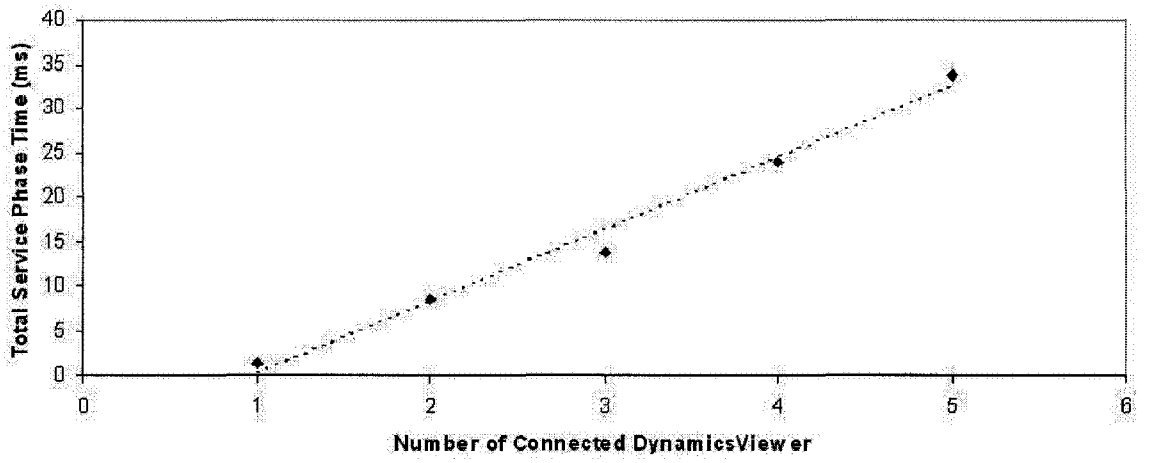

Figure 3.16: Total service time variation due to increasing number of connected DynamicsViewer clients.

best-fit line, with a period of 2 clients. This phenomenon is likely due to the operating system distributing tasks to the two CPUs on the computer. The VFD-RT application runs with high priority, so other tasks assigned to the CPU will only be processed if the VFD-RT does not need the processor. The task of every second client is likely to have been assigned to the CPU running the VFD-RT application, causing a slight rise in processing time. Alternatively, the first connected client can run on the CPU that is free from processing the VFD-RT application tasks, causing the processing time to drop slightly. The same phenomenon can be observed in the SMP plot, but to a very small extent, since comparatively the SMP has less processing required.

The DynamicsViewer plot (Figure 3.16) deviates visibly from linearity, as the dashed best-fit line demonstrates. This non-linearity is due to the design of the DynamicsViewer, which does not use the modified service phase method. The difference is clearer in the plots of the service times for the SMP and DDC cases, Figures 3.17 and 3.18.

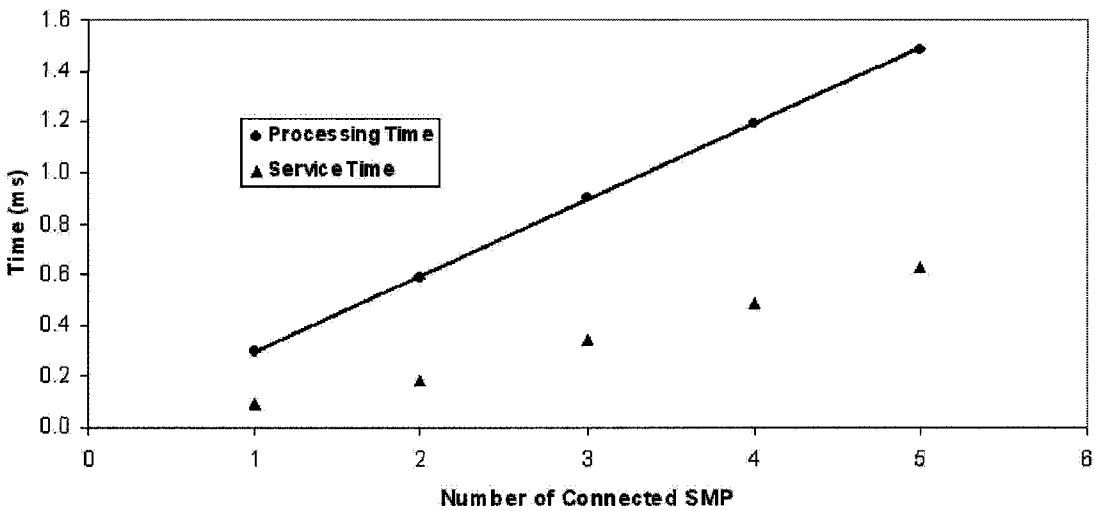

Figure 3.17: Total variation of processing and service times due to increasing number of connected SMP.

The plots show that service time is significantly smaller than the total processing time. The main difference is in the measured time: the service time measures Δt_m, according to the conventional service phase and the total processing time measures $\Delta t'_m$ in the modified

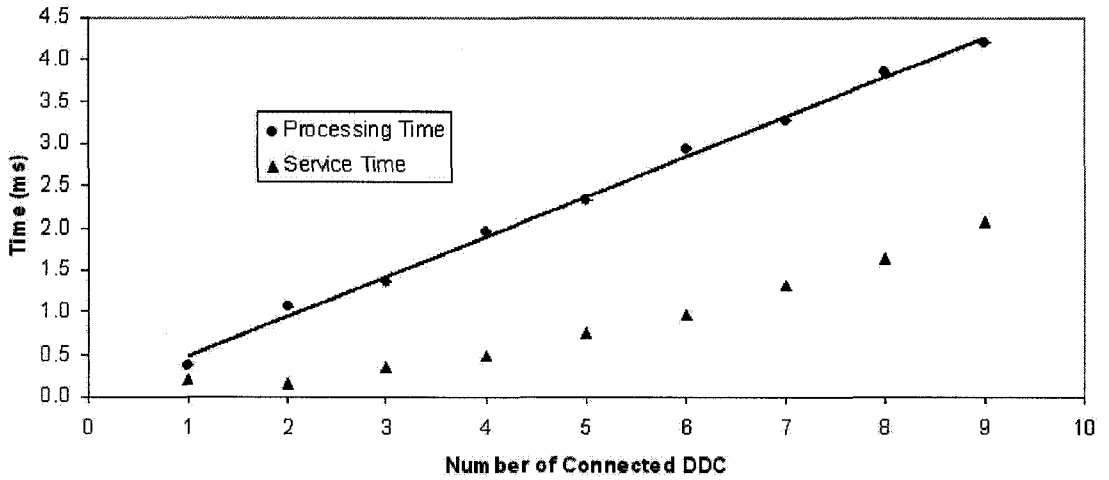

Figure 3.18: Total variation of processing and service times due to increasing number of connected DDC.

service phase method. The total processing time includes all the data processing within the SMP or the DDC, as in Steps 5 through 11 in Figure 3.13. The service time only includes the time when *sendData()* is active (Steps 1 to 7 in Figure 3.12).

As noted, there is a non-linear increase in the service time. With increasing number of providers and clients, the CPU may schedule a few of them to be processed before the service phase in the VFD-RT application finishes. Therefore, rather than increasing linearly, the service time line slope increases with increasing number of clients or providers.

Table 3.4 shows another perspective of the above condition. The list of events within one simulation cycle using 5 DDC is listed with their timestamps with respect to the beginning of the service phase. The sequence of events reveals that the later clients experience delays because earlier clients were processing their data, causing Client 5, for example, to receive simulation data almost 2 ms after the first. Furthermore, this order changes depending on the task scheduling algorithm of the CPU.

To avoid this reordering of events for large numbers of clients (which depend on the CPU), the *SendMessage()* function can be used in the client, so that the *processData()* method can be called before returning the data copy message to the VFD-RT, as in Figure 3.13. However, the *PostMessage()* method is more efficient for smaller number of clients.

Table 3.4: Sequence of events in an example simulation cycle with 5 data display clients.

Event	Timestamp (ms)
VFD-RT send to client 1	0
Client 1 receives data	0.07794
Client 1 returns	0.1048
VFD-RT send to client 2	0.2797
Client 1 processes data	0.3009
Client 2 receives data	0.3160
Client 2 returns	0.3420
Client 1 finishes processing	0.4973
VFD-RT send to client 3	0.5492
Client 3 receives data	0.5685
Client 3 returns	0.5951
Client 2 processes data	0.7585
Client 2 finishes processing	0.9306
Client 3 processes data	0.9619
Client 3 finishes processing	1.241
VFD-RT send to client 4	1.705
Client 4 receives data	1.765
Client 4 returns	1.798
VFD-RT send to client 5	1.965
Client 5 receives data	1.978
Client 5 returns	2.006
Client 4 processes data	2.033
VFD-RT finishes services clients	2.181
Client 4 finishes processing	2.232
Client 5 processes data	2.256
Client 5 finishes processing	2.425

Therefore, for smaller numbers of clients, the messages for processing data in the client should be posted with *PostMessage()*, whereas for larger numbers of clients, for more consistent data receiving time, data processing messages should be sent with *SendMessage()*. This trade-off issue must be considered when designing the client or provider. Furthermore, this finding also implies that the more time-sensitive, less processor-intensive clients should be connected first, so that their data receiving time is more consistent and their processing would not cause large delays for later clients.

Additionally, due to the concurrent nature of the method using *PostMessage()*, the effect of this event reordering becomes less significant if the number of CPUs is increased, so that the VFD-RT can run on one processor as the clients use their own processor to perform their tasks with the received data, thus avoiding the delays in the VFD-RT service phase. The more CPUs a system has, therefore, the lesser this effect is.

Provider Data Process Phase With the service time having been discussed, it is also necessary to investigate the provider data processing phase. In this phase, the VFD-RT application has received new simulation data from the provider, and copies the data into its data frame after some error checking. This value stays relatively constant regardless of the number of connected providers and clients, given enough CPU processing time, since all the required processing is done within the VFD-RT application.

The result from 50 replications of 100 simulation cycles produced an average provider data process phase duration of 0.03612 ms, with a standard deviation of 0.0001737 ms and a confidence interval within 0.1333% of the mean.

Simulation Frequency Limit The theoretical maximum simulation frequency that the VFD-RT application can sustain, in order that all the processing within a simulation cycle completes before the end of the cycle, can be calculated using Equation 3.9 and the total processing time and service time plots (Figures 3.14, 3.15, and 3.16). For example, for a simulation with 1 SMP, 1 DDC, and 1 DynamicsViewer client, the Equation 3.8 becomes,

$$\begin{aligned}\Delta t_p &= \sum \Delta t'_m - \Delta t_{m0} + \Delta t_{sec} \\ &= (0.3225 + 0.3540 + 1.278) - 0.02530 + 0.00125 = 1.9305 \text{ ms}\end{aligned}$$

such that the maximum simulation frequency, as per Equation 3.9, is

$$f_{max} = \frac{1}{\Delta t_p} = \frac{1}{1.9305 \times 10^{-3} s} = 518 \text{ Hz} \quad . \tag{3.11}$$

However, because the DynamicsViewer client processes its drawing task after the VFD-RT service phase, depending on the number of models that are to be drawn, the maximum simulation frequency is decreased. This value, though not giving the exact limit, gives a first approximation of whether a set of providers and clients can achieve near-real-time simulation with the VFD-RT.

3.3 Validation

As a simulation environment, the VFD-RT must demonstrate the ability to execute different ship-based simulations for a variety of purposes. The validation stage employs three cases for this demonstration, representing a range of required simulation complexities as itemized below.

1. Ship energy, a research project requiring a ship motion simulation for the investigation of the time variation of ship total mechanical energy.

2. ShipMo3D-RAS, a case study of a replenishment-at-sea (RAS) operation transferring a payload along cables between two ships, where the simulation accounts for the interdependence between the payload dynamics and the motion of the cable-connected ships.

3. Flight Deck Motion Display (FDMD) development, for which VFD-RT simulation is used throughout the process.

These cases are chosen to represent the range of situations in which the VFD-RT simulation environment can be used to control a simulation. In the first case, the environment

participates in a direct data transformation by transferring data from a ship motion provider to a client that calculates the energy parameters of the ship. Validation in this case shows that the environment has proper data transfer capability and can be used for feed-forward type simulations. In the second case, each ship is modelled to receive external forces and produce ship motion, which is fed into the provider-client model of the RAS payload and gear. This RAS model produces the forces and moments that are the inputs to the ship models. The three models communicate through the VFD-RT in a real-time manner, and demonstrate that a full feedback-type simulation can be correctly produced by the simulation environment. The third case follows the use of the VFD-RT environment through the stages of the FDMD project, from initial software development, to hardware testing, and finally to integrated system evaluation. The case confirms that the environment is applicable in support of both research and development activities.

3.3.1 Ship Energy Calculation

The first validation case compares the VFD-RT simulation output data with those of an existing, validated program. The purpose of this case is twofold:

1. to demonstrate the ability of the VFD-RT environment to modularize a program for simulation, particularly in separating computational models from simulation management components involving time and connection activities; and

2. to validate the data transfer through the simulation.

The program used for this case is for the calculation of ship energy, developed in FORTRAN by Dr. Rob Langlois of the Applied Dynamics Research Group at Carleton University. It uses a simplified principal axes energy model to evaluate the kinetic and the potential energy of the ship given the ship kinematics. The ship is modelled as a rigid body supported by linear and rotational springs representing hydrostatic forces. The kinetic energy is evaluated as the sum of the translational and rotational components

$$T = \frac{1}{2} M \{v_0\}^T \{v_0\} + \frac{1}{2} \{\omega_0\}^T [I_0] \{\omega_0\} \tag{3.12}$$

where M is the scalar mass of the ship, v_0 is the vector of the ship translational velocity, ω_0 is the angular velocity vector in the ship body frame expressed in ship-local coordinates, and $[I_0]$ is the 3 × 3 matrix of the ship mass moment of inertia generally given as

$$I_0 = \begin{pmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{yx} & I_{yy} & -I_{yz} \\ -I_{zx} & -I_{zy} & I_{zz} \end{pmatrix}.$$

The potential energy can be calculated by combining the effects of gravity and of the equivalent springs,

$$U = Mgh + \frac{1}{2}k_z z^2 + \frac{1}{2}k_{\theta_x}\theta_x^2 + \frac{1}{2}k_{\theta_y}\theta_y^2 \tag{3.13}$$

where g is the acceleration due to gravity and h is the vertical displacement of the ship from its equilibrium point. The z, θ_x, and θ_y values are the heave, roll, and pitch components of the ship position that contribute to the potential energy. Their corresponding equivalent spring constants are k_z, k_{θ_x}, and k_{θ_y}, found through the ship-specific heave, roll, and pitch natural frequencies as

$$k_i = M\omega_{n,i}^2 \tag{3.14}$$

where the subscript i represents one of the three components where equivalent springs apply and $\omega_{n,i}$ is the natural frequency of component i. The *testshipenergy.exe* program combines the *SHPR* subroutine, which produces the ship motion, with the *SHIPENERGY* subroutine that implements the energy equations. The time increment is kept by the main program. The output is stored in a text file recording the variation of ship motion and the corresponding energy values through time.

Validation Method

For modularity, the program can be divided into three sections that parallels the VFD-RT input-middle-output layered architecture:

1. the ship motion generator SHPR, implemented in the SMP application of the VFD-RT environment, as described in Section 2.4;
2. the time managing component, which is the VFD-RT core application run in a faster-than-real-time manner; and
3. a ship energy client application that integrates the *SHIPENERGY* FORTRAN subroutine.

The file output function is incorporated in the *SHIPENERGY* subroutine, so the wrapping client application is not required to output any file.

In addition to the three applications, the SMP requires the "shpr.inp" input file along with the ship motion spectrum library. The ship energy subroutine also requires an input file that contains the inertial properties of the ship, the added mass of the ship caused by the motion of the water around the ship hull, and the natural periods of the ship from which the equivalent spring constants are derived.

The specific case tested has the following ship operating conditions:

- wave height of 6 metres;
- ship headings relative to the principal sea direction of 45°; and
- ship speed of 10 knots.

The content of the corresponding "shpr.inp" input file is shown in Appendix A. The first parameter is the library file name and the second is the number of frequency components included in the ship response spectrum file. The ship has the following properties, typical of a frigate:

- ship mass of 4.8 $\times 10^6$ kg;

- roll, pitch, and yaw radii of gyration of 4.5 m, 30 m, and 30 m respectively;
- surge, sway, and heave added mass, normalized by the ship mass, of 0.02, 0.6, and 2 respectively;
- roll, pitch, and yaw added mass, normalized by the ship moment of inertia along the corresponding axes, of 0.15, 1.0, and 0.30 respectively; and
- heave, roll, and pitch natural periods of 5 s, 10 s, and 5 s respectively.

The associated input file, "shipenergy.inp" is shown in Appendix B. The procedure is to run the *testshipenergy.exe* program with the above input files and the following input parameters.

1. Start time at 0 seconds.
2. Increment time by 0.1 seconds at every time step.
3. Stop the simulation after 3000 steps.
4. Output the ship motion and energy parameters at every time step.

Then the VFD-RT simulation is run with:

- the SMP as the provider using the ship motion settings in the same "shpr.inp" file used for *testshipenergy.exe*;
- the VFD-RT as the core application with:
  - start time of 0 seconds;
  - non-real-time simulation increment of 100 ms;
  - optimization for performance and real-time priority;
- the ship energy client using the "shipenergy.inp" file described above; and
- the data display client to monitor the progress of the simulation.

The simulation is manually stopped after 300 seconds of simulation.

Since the same *SHIPENERGY* subroutine is used in both, identical output files would validate the simulation in this case.

Validation Results

A section of the output file for the test validation case is shown below.

time	ship dx	ship dy	...	te
0.00000E+00	0.35580E+00	0.19161E+01	...	0.10375E+09
0.10000E+00	0.35932E+00	0.18917E+01	...	0.96320E+08
0.20000E+00	0.36150E+00	0.18575E+01	...	0.88869E+08
0.30000E+00	0.36238E+00	0.18136E+01	...	0.81528E+08
0.40000E+00	0.36195E+00	0.17602E+01	...	0.74420E+08
0.50000E+00	0.36025E+00	0.16978E+01	...	0.67655E+08

It is found that the output files from both simulations match exactly.

This validation test shows that the data is transferred correctly from the SMP to the ship energy client. The VFD-RT simulation architecture has also proven to be capable of running simulations using models connected directly to the VFD-RT. The advantage of this architecture is that the ship energy program is modular and permits the replacement, addition, or removal of the simulation components without the necessity of code recompilation. It should be noted that a mix of FORTRAN and C++ programming is involved in this simulation, demonstrating the language interoperability of the VFD-RT.

3.3.2 Replenishment-at-sea Simulation

The second validation case demonstrates that the VFD-RT is capable of:

- running a simulation with data feedback; and

- running providers that produce data from current input and internal memory in the form of state variables.

A simulation of a replenishment-at-sea operation is chosen because each of the simulated entities has a complex model requiring data from other entities in the simulation. Replenishment at sea (RAS) is the transfer of goods between two ships operating in a common seaway, and is a common operation among larger ships that are often out at sea for long periods of time. Typically the goods are transferred from a large supply ship to a receiving ship, in seaways with significant wave heights of up to about 5 m. The current equipment on the North Atlantic Treaty Organization (NATO) ships limits the payload mass to a maximum of 2 tonnes. A recent research project [36, 37] used a High-Level Architecture (HLA) distributed simulation to investigate the RAS operation, in support of potential improvement work that may increase the payload mass limit, the transfer rate, the range of sea conditions in which payload transfer is possible, and other aspects. In particular, the simulation of this investigative project produced kinematic and kinetic parameters – that is, the position, velocity, acceleration, and the applied forces and moments – of the payload, the two ships, and the RAS gear.

The RAS gear consists of a system of three cables and their tension control equipment. Figure 3.19 shows the different components of the RAS gear. The three cables are:

- the highline cable from which the payload is suspended;
- the out-haul cable responsible for pulling the payload towards the receiving ship; and
- the in-haul cable responsible for pulling the payload towards the supply ship.

The tension control equipment is located on the supply ship and is composed of a ram tensioner and a set of winches. The tension affects the motion of the two ships, as they are being pulled towards each other. Simultaneously, the RAS gear is also affected by the ship motion. In the HLA simulation, this interaction is modelled by the exchange of force and motion data between the RAS gear model and the ship models. There are five main components to the simulation, which are the RAS model, the two ship motion models,

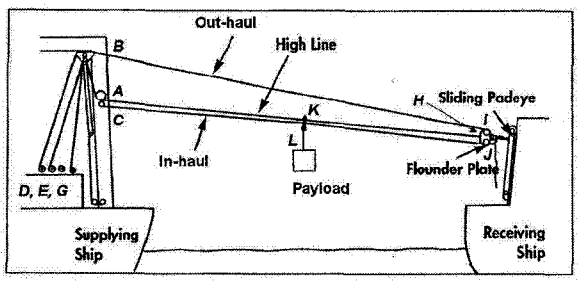

Figure 3.19: Schematic of the RAS gear (adapted from [38]).

and the two corresponding ship helm models. In general, an HLA simulation requires Real-time Infrastructure (RTI) software to manage real-time communication between the various models.

The ship motion models employ the ShipMo3D software, a high-fidelity object-oriented library developed by Defence Research and Development Canada (DRDC) to predict ship motion from hull geometry, seaway data, helm control, and externally applied forces [39]. The software was originally written in the Python programming language, but has now been translated into the C# language. Validation was performed against sea trials data of the tanker Esso Osaka and the Canadian naval destroyer HMCS Nipigon, and against previous comprehensive steered warship model tests [36]. For the HLA simulation, the ShipMo3D library is wrapped in an application capable of communicating with the RTI.

Similarly, the RAS model is wrapped in an HLA-compliant application. The model code is packaged in the *RASEQII* module written in FORTRAN by Dr. Rob Langlois. *RASEQII* uses input ship motion to generate values for the cable tensions, payload kinematics, and the forces and moments acting on each of the ships. The module contains a fourth order Runge-Kutta integrator to solve a system of differential equations [36], such that the algorithm depends on both the current ship motion input and internal state variables. The verification of this module was performed through the *DERAS* development environment that tested *RASEQII* against a range of possible inputs.

Validation Method

To demonstrate the capability of the simulation environment to integrate complex elements, such as data feedback and models with internal state variables, the VFD-RT validation is divided into two parts:

1. to demonstrate through motion plots that the RAS cable forces and the motion of the ships affect each other; and
2. to validate numerically against force and moment data from the *DERAS* environment output.

A simulation architecture similar to that of the HLA simulation is used. The ShipMo3D library is wrapped in a provider-client application which produces ship motion and receives values of equivalent forces and moments about the centre of mass of the ship due to cable tension. The *RASEQII* module is wrapped in a RAS provider-client, which receives ship motion data and provides the values of cable forces applied to each of the ships. The validation setup is shown in Figure 3.20.

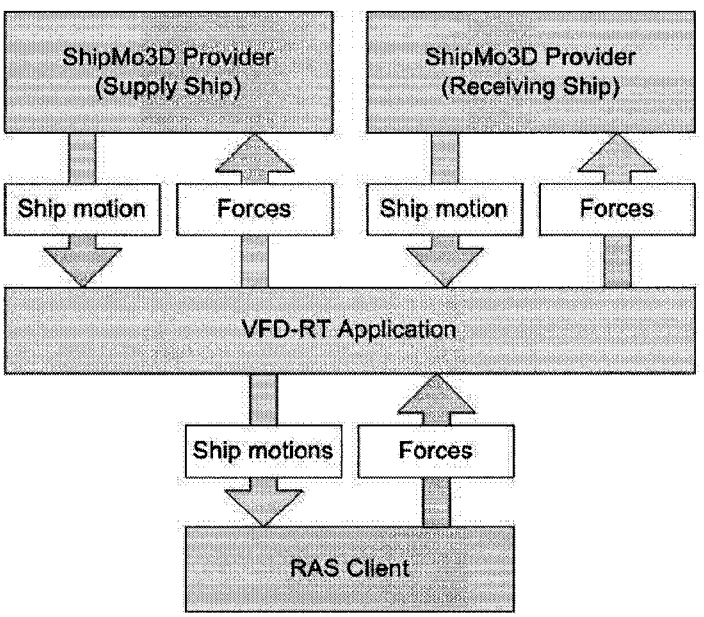

Figure 3.20: Diagram of the validation setup for the VFD-RT RAS simulation.

The implementation of this setup presents another interesting case of language interoperability with the VFD-RT. The ShipMo3D software is packaged in a C# dynamic link library (DLL) file, which is linked to the program only during run-time. In order to use the software, the wrapping provider application code, written in C++, implements references to the functions in the DLL. With the help of the .NET software framework which offers a set of language interoperability features, the C# DLL functions are called by the C++ code when the provider application is executed. Thus C# code can be integrated into a C++ provider. It should be noted that the particular .NET framework feature used requires the provider to be compiled with Visual Studio 2005, rather than Visual Studio 2003 used for the other VFD-RT applications.

Along with wrapping the C# function references, the ShipMo3D provider includes facilities for applying a coordinate transformation on the generated ship motion before sending the data to the VFD-RT core application. ShipMo3D produces ship motion in the coordinates of the Earth-fixed inertial frame, with Euler angles in the ZYX convention. If the simulation requires a different coordinate system or Euler angle convention, like the XYZ Bryant angle convention as used in this validation case, transformation can be applied.

For the purposes of this validation, the parameters related to the seaway, the ship geometry, and the ship helm controls are stored in input files and are fixed for the duration of the simulation.

The FORTRAN *RASEQII* module is wrapped in the RAS client as the SMP does with the *SHPR* module: when the client is built, the FORTRAN code is linked to the final executable. In order to produce validation data, the incoming ship motion values and the output force values are recorded in a text file.

In reference to the ship, the earth-fixed coordinate system used is illustrated in Figure 3.21 [39]. When the ship is at the origin, the x-axis is the longitudinal axis and points towards the bow; the y-axis is positive on the port side of the ship; and the z-axis completes the right-handed coordinate system with positive being up.

Using this coordinate system, the VFD-RT is validated with motion plots when the following observations are made from the simulation output.

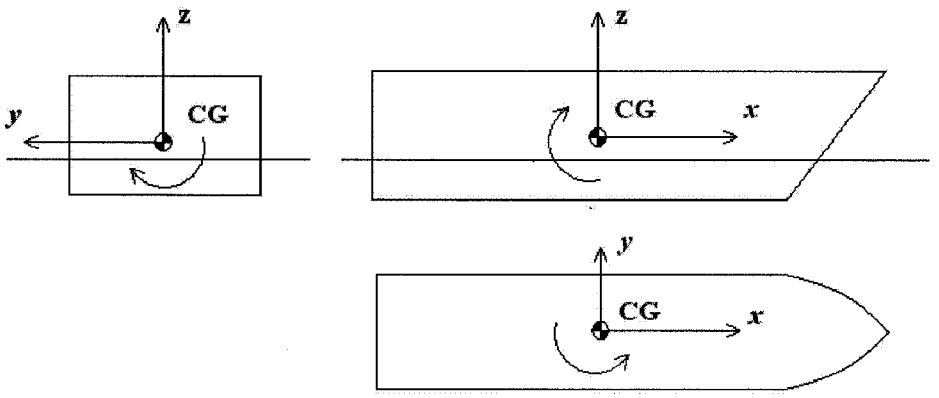

Figure 3.21: Earth coordinate system for describing ship motion 3.21.

- The ships, originally travelling parallel to each other, approach each other over time, such that their distance in the y-axis decreases.
- The ships lean toward each other depending on where the cable attachment points are located, such that the angle of rotation about the x-axis of one ship tends towards positive values, while the other, towards negative values.

Two simulation runs were performed: one without the cable effect fed back into the simulation and another with the cable effects enabled. The two ships are initially travelling parallel to each other, at the same speed. The receiving ship has the characteristics of a Halifax class frigate used by the Canadian Navy, and the supply ship represents the Protecteur class also used by the Canadian Navy. The validation case uses the seaway input files shown in Appendix C, and the ship input files shown in Appendices D and E for the supply and the receiving ships respectively. They contain the following simulation parameters:

- initial distance between ships: 52.25 m, the supply ship being on the port (left) side of the receiving ship;
- commanded ship heading: 0° (straight north) for both ships;

- commanded ship speed: 5.14 m/s (10 knots) for both ships;
- wave type: irregular;
- sea state: 5 (significant wave heights of 3.25 m)
- heading of incoming waves: 330° for both ships;
- payload mass: 1000 kg;
- location of RAS gear on the supply ship: mid-ship, 4.75 m starboard of centerline, 18 m elevation above waterline;
- location of RAS gear on the receiving ship: mid-ship, 4.75 m port of centerline, 15 m elevation above the waterline;
- simulation run time: 80 s; and
- payload transfer start time: 40 s.

The validation of the RAS gear force effects on the ships is followed by the validation of the other side of the feedback loop, which is the effect of the ship motion on the RAS equipment and, in turn, on the forces that they produce. This is achieved through comparing the simulation RAS gear force output against the results from the *DERAS* development environment. As the simulation runs, the positions, velocities, and accelerations in the 6 degrees of freedom of both ships are recorded to a file. These values serve as input to the *DERAS* environment that employs the *RASEQII* module and generates output forces and payload motions associated with the RAS equipment. If the RAS provider-client of the VFD-RT is designed and implemented correctly, the expected outcome is that the output forces from *DERAS* match the output forces as produced within the VFD-RT simulation.

Validation Results

The results are divided into the two parts of the validation process described. The first is to demonstrate that the RAS equipment produces forces that pull the ships closer to each

other. In terms of the simulation output, this phenomenon translates into the decrease of the ship separation along the y-axis. Figures 3.22, 3.23, and 3.24 show the y values of each of the ships, and the distance between the two ships over time.

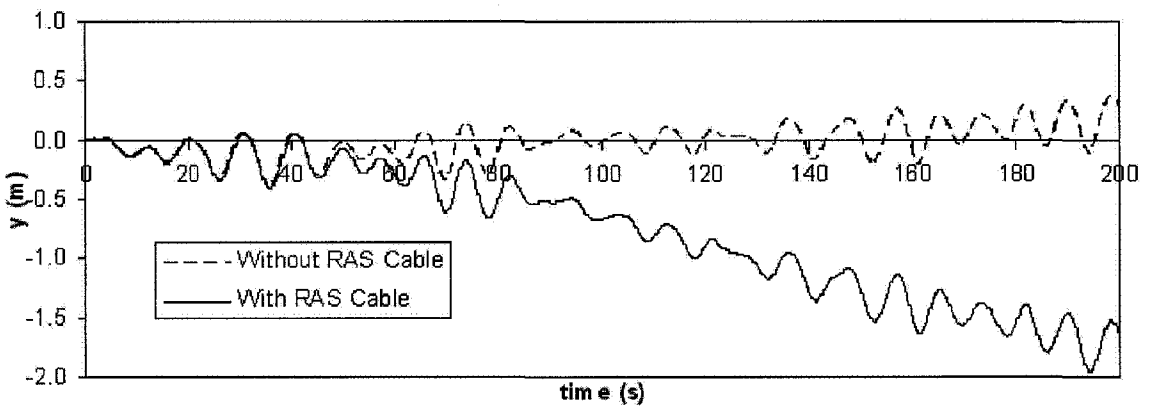

Figure 3.22: Position in the y direction of the supply ship with and without the effect from the RAS cable.

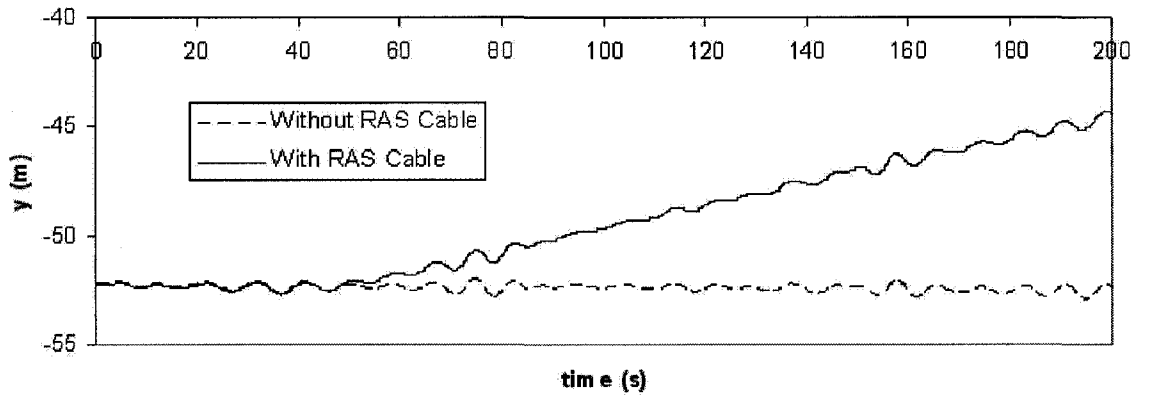

Figure 3.23: Position in the y direction of the receiving ship with and without the effect from the RAS cable.

The dashed curves correspond to the simulation without the use of the RAS cable, while the solid curves correspond to the simulation with RAS cable tensions present. The plots show up to 200 seconds of data to emphasize the differences. The dashed lines in Figures 3.22 and 3.23 stay relatively constant near their initial values, which are 0 m for the supply ship and -52.25 m for the receiving ship, as the RAS operation initiated at 40 seconds. The separation plot confirms that without the RAS cable, the ships run parallel to each other

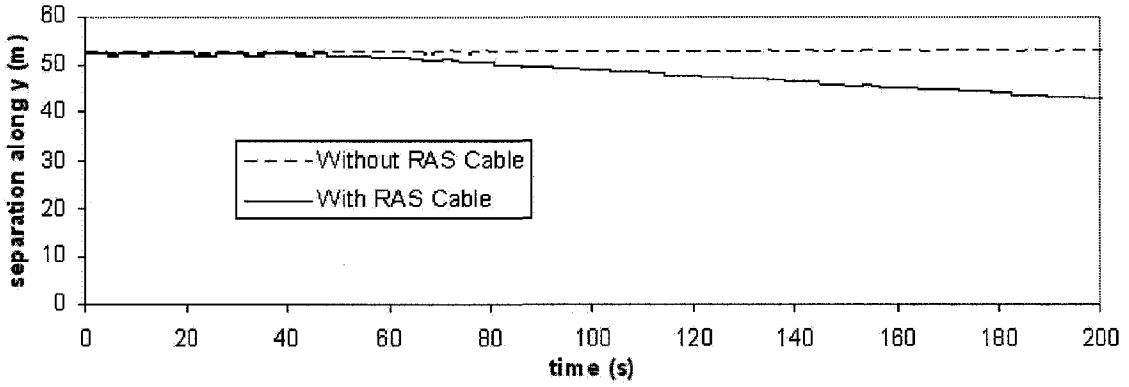

Figure 3.24: Separation of the two ships in the y direction with and without the effect from the RAS cable.

for the duration of the simulation. The solid lines show that the supply ship drifts closer to the receiving ship, having a decreasing y value. On the other hand, the y values of the receiving ship increase over time.

The effects of the RAS transfer can also be observed in the ship roll, which is the rotation about the x-axis, as shown in Figures 3.25 and 3.26. The dashed curve corresponds to ship roll without the effects of the RAS gear, and the solid curve corresponds to ship roll with the RAS gear present.

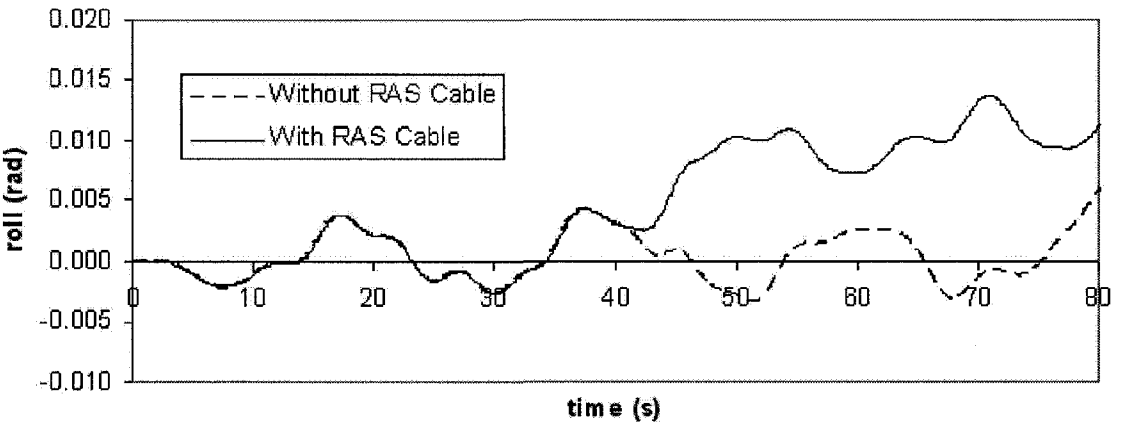

Figure 3.25: Roll of the supply ship with and without the effect from the RAS cable.

The supply ship has a slight positive roll, and the receiving ship, a negative roll, which is expected. The cable attachment points are above the ship centre of gravity, so that the

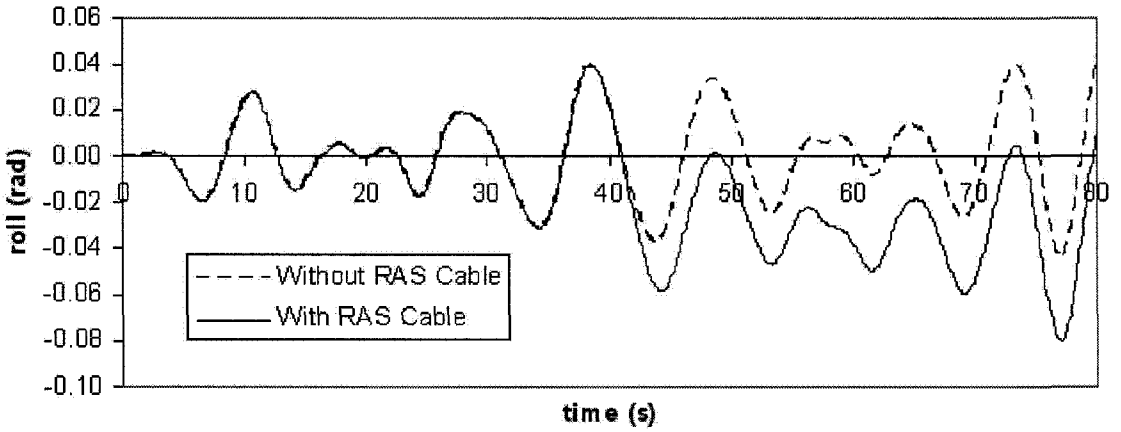

Figure 3.26: Roll of the receiving ship in RAS with and without the effect from the RAS cable.

top of the ships lean toward each other. The simulation has the supply ship on the port side of the receiving ship, so the roll due to the RAS cable is positive as defined by the earth coordinate system. In contrast, the receiving ship has a small negative roll due to the pull of the cable at a point above the ship centre of gravity. Like the y-axis plots, the roll difference starts at around 40 seconds, when the RAS transfer initiates.

The correspondence of the time between the RAS transfer commencement and the deviation of the sway and roll of both ships, along with the direction of displacement shows that the RAS cable affects the ship motion as expected. Therefore, forces are correctly fed back into the simulation from the RAS provider-client to affect the ShipMo3D data.

To give further evidence of proper simulation functioning, the force and moment values generated by the simulation are compared against the data generated from the verified *DERAS* development environment, using the same ship motion as recorded from the VFD-RT simulation. If the RAS provider-client implementation is correct, the forces and moments generated should numerically match the validation data. This was found to be the case. The following series of plots provided in Figures 3.27 through 3.31 attempt to show the agreement between the data. *DERAS* results are indicated by the solid lines, while the VFD-RT results are the squares and circles superimposing the curves.

Figure 3.27 shows the linear displacement of the supply ship in the x, y, and z coordinate

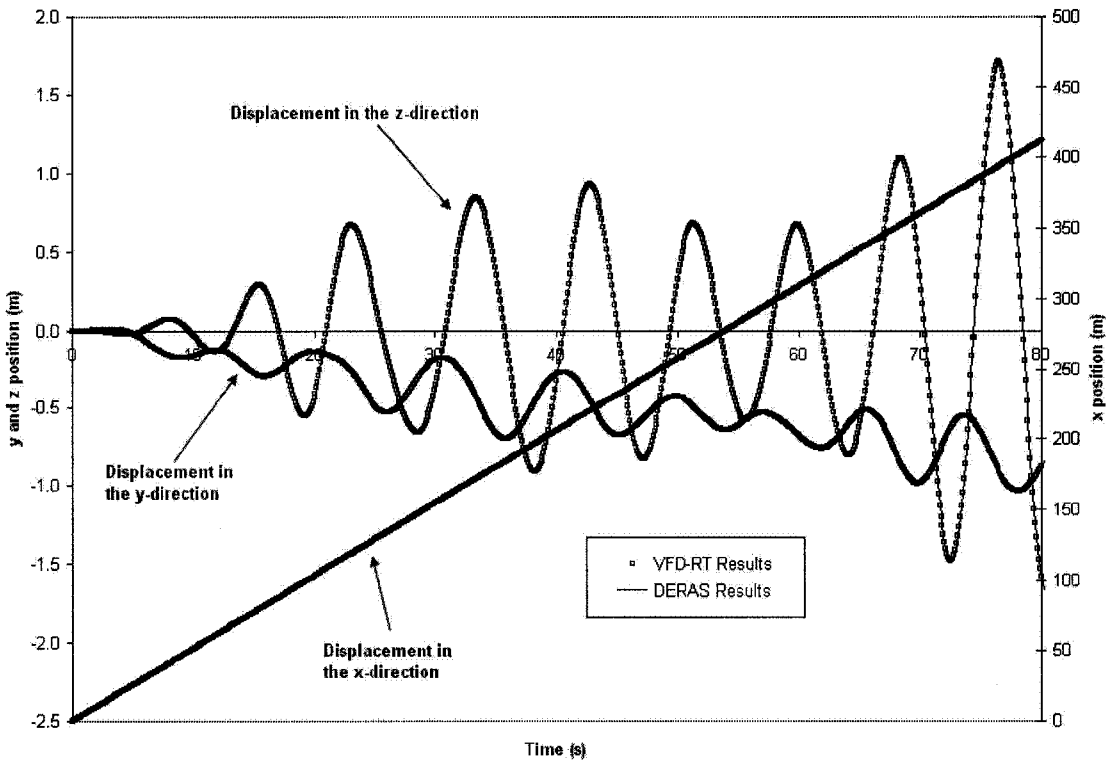

Figure 3.27: Linear displacement of the RAS supply ship in the x, y, and z directions for validation with *DERAS*.

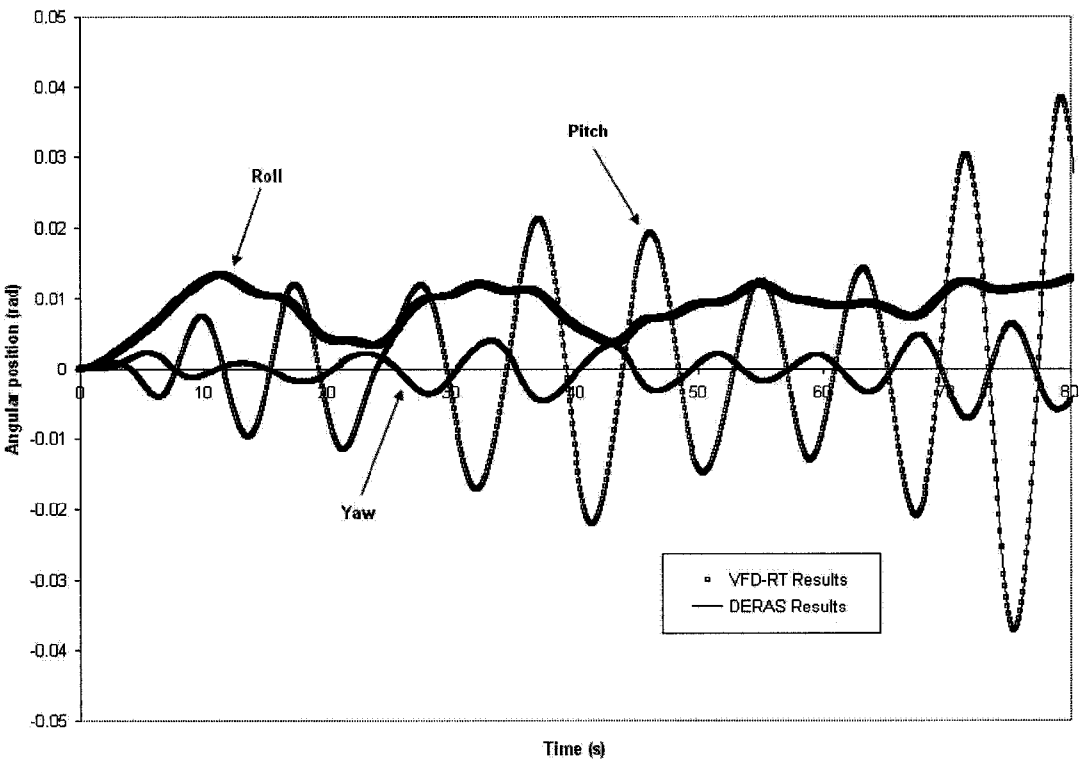

Figure 3.28: Angular displacement of the RAS supply ship in the x, y, and z directions for validation with *DERAS*.

directions during the simulation. The x values are plotted on the secondary axis on the right to show its full range. The y values follow the curve that tends toward the negative, due to the cable tension. The z curve is the vertical ship displacement. Figure 3.28 shows the angular displacement of the supply ship. The roll angle follows the thicker black curve that remains positive, again due to cable tension. The yaw curve has a smaller amplitude than the pitch curve, since the ship heading is kept as close to 0° as possible by the ShipMo3D autopilot. In both plots, there is exact agreement between the *DERAS* output and the VFD-RT simulation results.

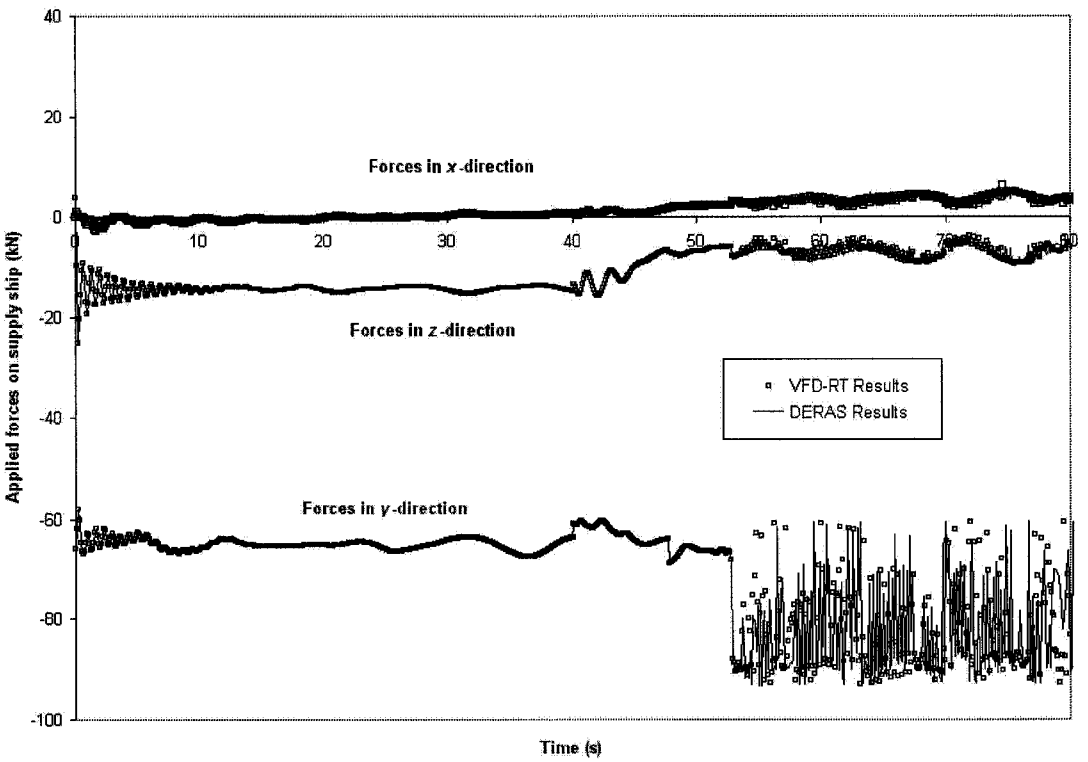

Figure 3.29: Forces acting of the supply ship due to the RAS gear for validation with *DERAS*.

Figure 3.29 shows the plots for, from top to bottom, the forces in the x, z, and y directions applied to the supply ship due to the RAS gear. Similarly, moments applied to the supply ship are shown in Figure 3.30, for the directions of x, y, and z from top to bottom. Both figures show complete agreement of data from 0 up to about 53 seconds,

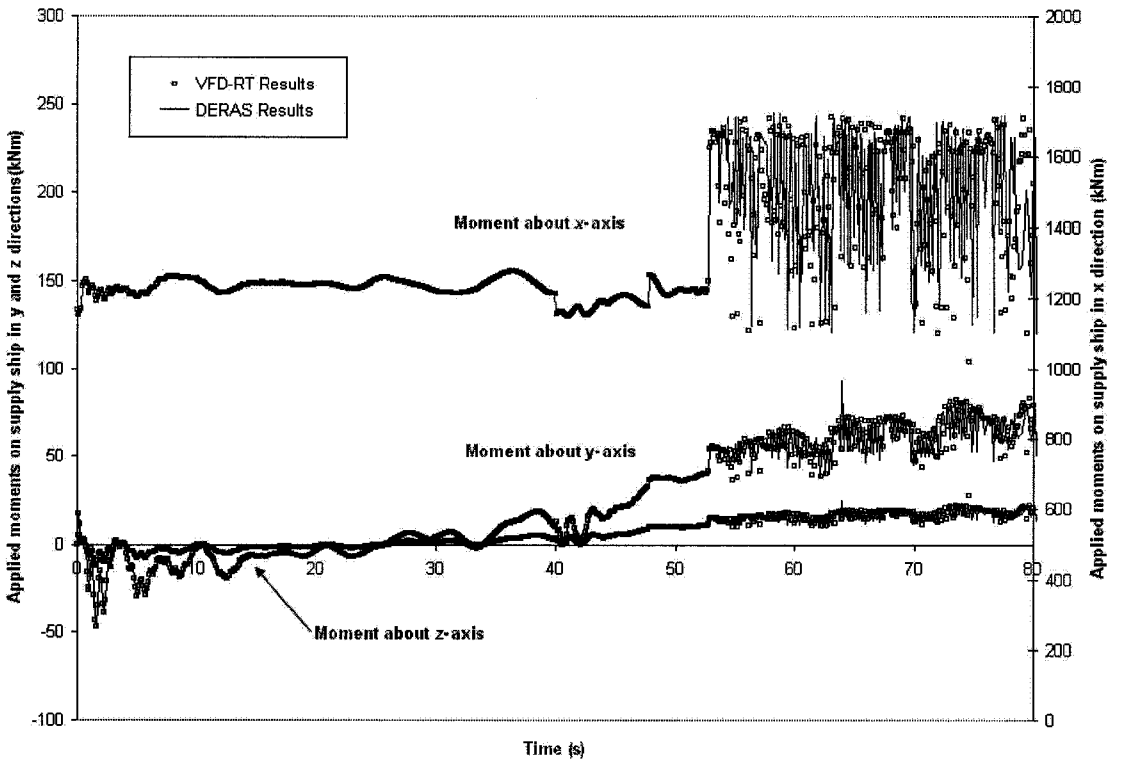

Figure 3.30: Moments acting of the supply ship due to the RAS gear for validation with *DERAS*.

when the RAS payload reaches the receiving ship, after which point the RAS equipment may experience large forces and moments that amplify differences due to numerical round-offs.

A similar observation can be made with the cable tensions in Figure 3.31. The highline cable experiences the largest force, while the out-haul cable has the lowest tension, and the in-haul tension remains constant at 4.45 kN. As with the previous plots, there is strong agreement between the *DERAS* results and the VFD-RT simulation results, particularly during the RAS payload transfer.

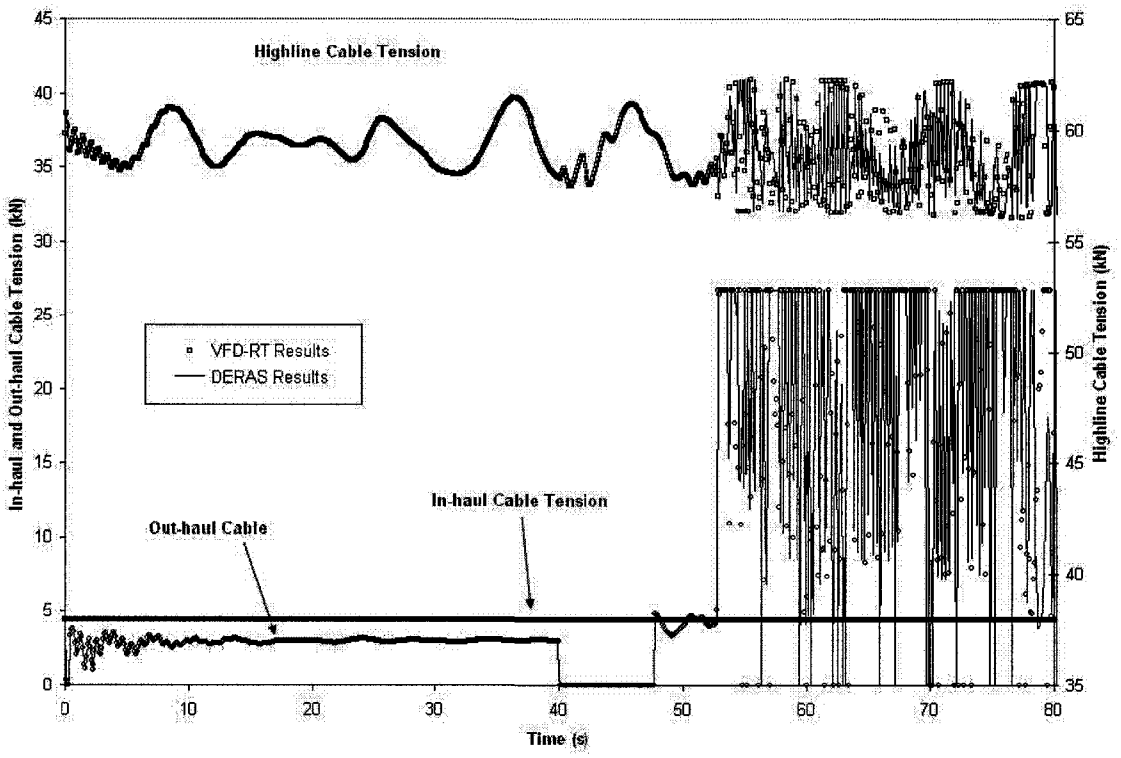

Figure 3.31: Cable tensions of the RAS gear in the validation test with *DERAS*.

With the matching between the *DERAS* results and the VFD-RT, the VFD-RT can be said to be validated for running simulation with data feedback. This validation case also demonstrates the types of simulation that the VFD-RT can support within its environment, including complex models with a number of internal state variables.

3.3.3 Development of a Flight Deck Motion Display

The third validation case demonstrates the applicability of the VFD-RT simulation in laboratory experiments and in the various stages of product development. As part of the toolset used by the Applied Dynamics Research Group, the VFD-RT must be capable of integrating with different stages of the development of a technology, from the conceptual design to final product evaluation. The use of the VFD-RT in the development of the Flight Deck Motion Display (FDMD) is an example of how this simulation environment can be used throughout these stages.

The FDMD is a system designed to provide the landing signals officer (LSO) of a ship with real-time indication of flight deck quiescence. A quiescent period is a brief time segment of calm deck motion when on-deck helicopter operations, such as traversing and landing, can be performed safely. The FDMD follows a draft requirement specifications released by Department of National Defence for a new deck motion monitoring system [40].

The design of the FDMD system consists of two computers, each connected to a sensor on the ship, that displays the flight deck motion graphically. One of the computers, with a specialized rugged display system, is used by the LSO on the deck to assist with helicopter landing, in a way that shows an indicator in the peripheral vision of the officer. The design of this indicator is critical as it provides safety-related information to the officer.

The conceptual design of the FDMD was first formulated during the spring and summer of 2007; it was evaluated by operators in the spring of 2008, and the final version of the prototype was completed in summer 2008. The VFD-RT was planned to support the FDMD project at all stages of development, which included software development, hardware development, and user interface evaluation.

Software Development

Once the initial design was complete, the development of the FDMD moved into an initial implementation phase in September of 2007. The implementation was minimal, containing only the necessary components for communicating with a data production device, recording

data, and displaying data on the user interface [8]. The FDMD communication module uses the transmission control protocol (TCP), which is part of the internet protocol (IP) suite that exchanges data packets between two IP addresses. Figure 3.32 shows a diagram of the configuration used to develop the initial software implementation of the FDMD.

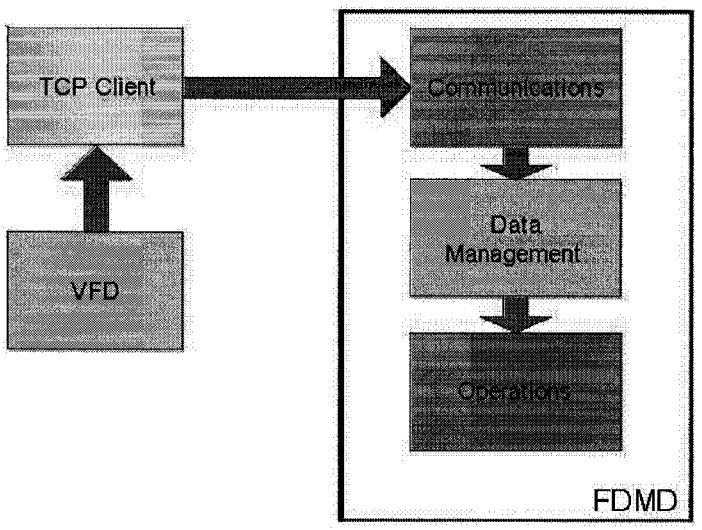

Figure 3.32: Use of VFD-RT in the initial implementation and development of the FDMD software.

The final FDMD system is designed to work with an inertial sensor mounted on a ship. In this configuration, the VFD-RT acts as a mock sensor until the basic FDMD functions have been validated and the real inertial sensor was acquired. On the right of the diagram is the FDMD software that has been implemented: a communications module receiving data, a data management module logging the software activities, and an operations module to display the data on the user interface. Driving the FDMD, on the left, is the VFD-RT, connecting to the FDMD through a TCP client. Not shown is the ship motion provider, the SMP discussed in the previous chapter (Section 2.4), as the input to the VFD-RT which generates the prescribed ship motion required by the FDMD.

The TCP client is designed to convert data copy messages into TCP data packets. The application is divided into four classes, as shown on Figure 3.33. The *ServerConnection* class relates to communication with the VFD-RT, the *TcpConnection* class relates to sending and receiving TCP packets, and *Window* controls the user interface.

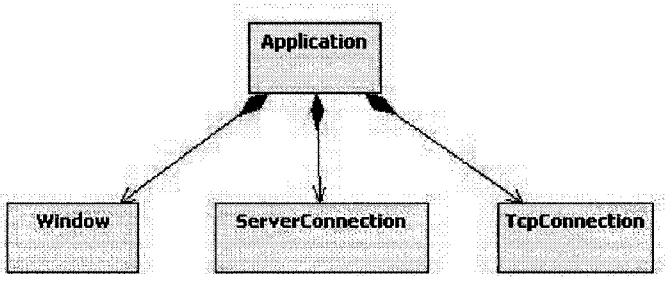

Figure 3.33: Class UML diagram for the TCP client application.

When a data copy message is received by the client from the VFD-RT, a sequence of function calls follows to send the data as a packet through TCP/IP, as illustrated in Figure 3.34. As per convention, the client makes a local copy of the data and returns, posting a *WM_RECEIVE_DATA* message to the operating system to be retrieved later (Steps 1-3). The *Application* retrieves that message and begins processing the copied data, by first checking its validity with saved server information, then by creating a TCP packet, and by sending the packet to the specified IP address. The user interface is then updated.

The data transferred by the TCP packets are the ship linear accelerations, angular rates, and angular positions, because the inertial sensor is expected to provide this data. Since the SMP produces data for the ship motion at its centre of mass, the TCP client also includes the capability to apply a transformation to produce ship motion at any arbitrary point on the ship. This data is then recorded and displayed by the FDMD software. They are sent via the Winsock library version 2.0, a set of Windows API functions dedicated to the connection and transmission of data through TCP/IP.

The picture of this configuration in Figure 3.35 demonstrates the physical connections between the systems. On the left is a laptop computer running the VFD-RT simulation with the ship motion provider, the DynamicsViewer client, and the TCP client. Connected through an Ethernet cable is the FDMD system on the right, which runs the FDMD software. Initially, the FDMD was run on a second laptop until the acquisition of the specialized rugged display with bezel buttons shown. The display system runs on the Windows XP operating system, such that only the software is being tested at this point in the FDMD

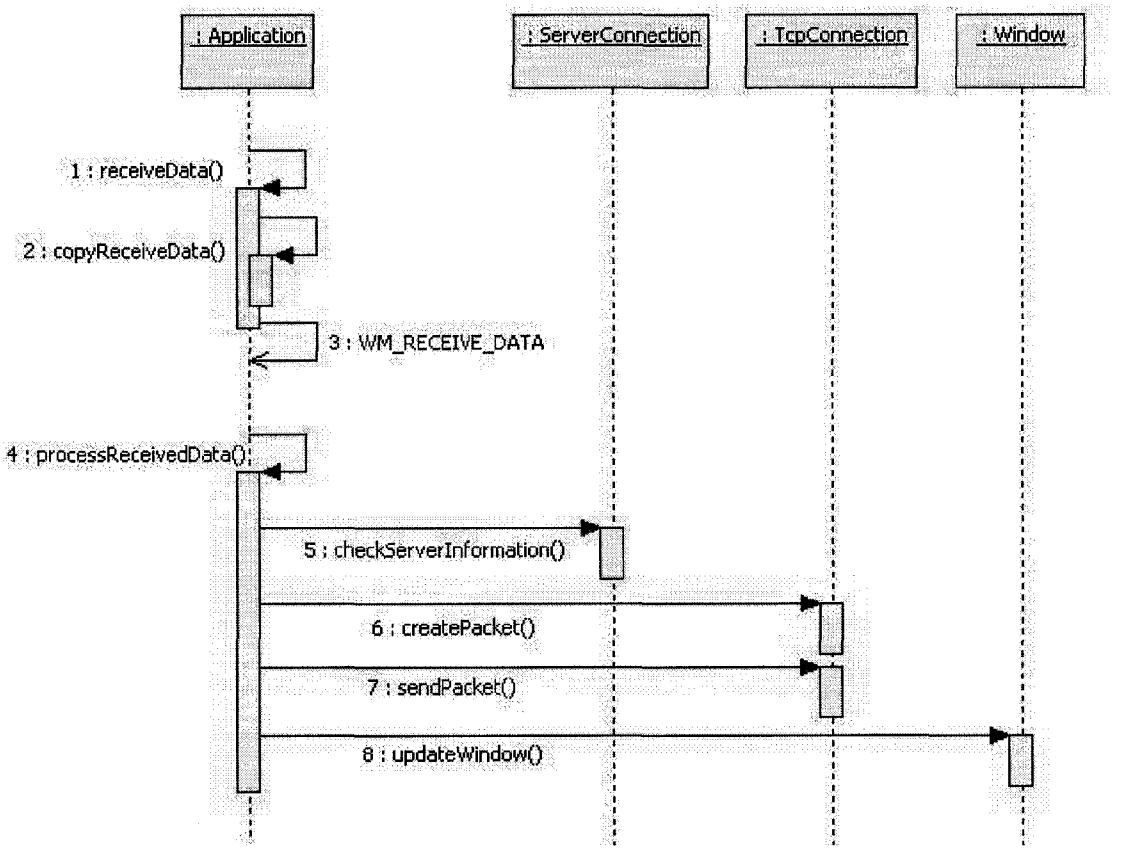

Figure 3.34: Sequence UML diagram for the TCP client application data processing.

system development.

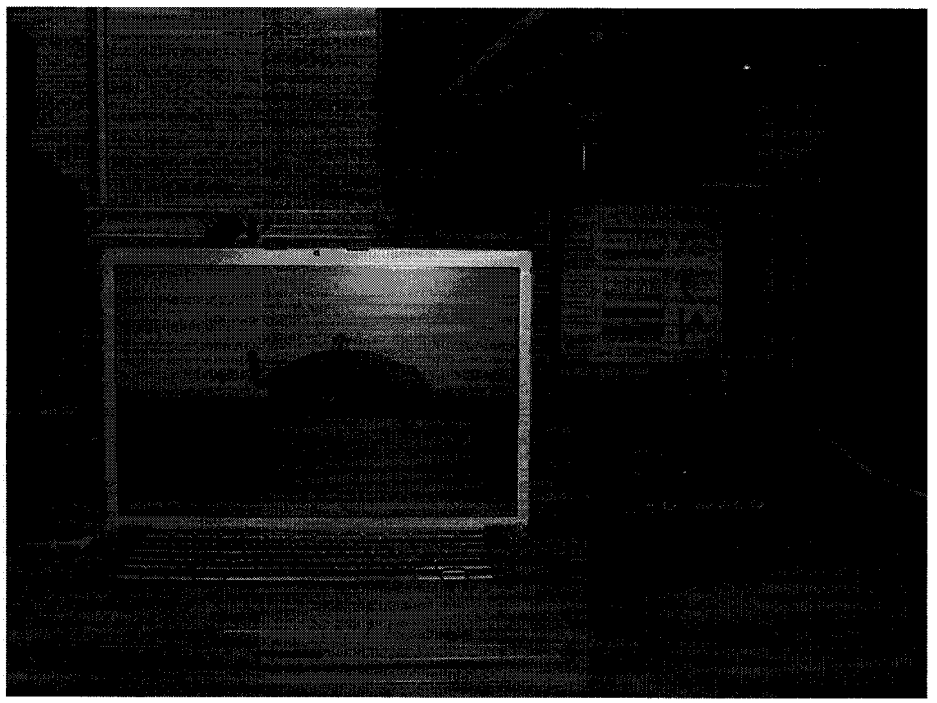

Figure 3.35: Photo of the FDMD software development configuration using the VFD-RT simulation as a mock sensor.

One particularly major contribution of the VFD-RT to the software development is in the design of the visual indicator to simultaneously convey the quiescence status (temporary perceived calmness) of the various ship motion parameters. This user interface design issue has met some technical challenges related to human factors and to data visualization methodologies [8]. However, the process is simplified by the use of simulated ship motion produced through the VFD-RT, such that design problems could be visually detected as the development progressed. The simulation produced ship motion data, using the Ship Motion Provider, that gives a sample of the on-deck visual experience before actual ship trials. Furthermore, the inertial sensor, after its acquisition, was integrated successfully with only minor restructuring of the communication module.

Additionally, the FDMD system is designed to have a second station to back up data and to perform more computationally intensive operations beyond the ability of the specialized rugged display computer itself. Each computer has its own sensor. Therefore, two TCP

clients were used to simulate the two sensors, as depicted in Figure 3.36

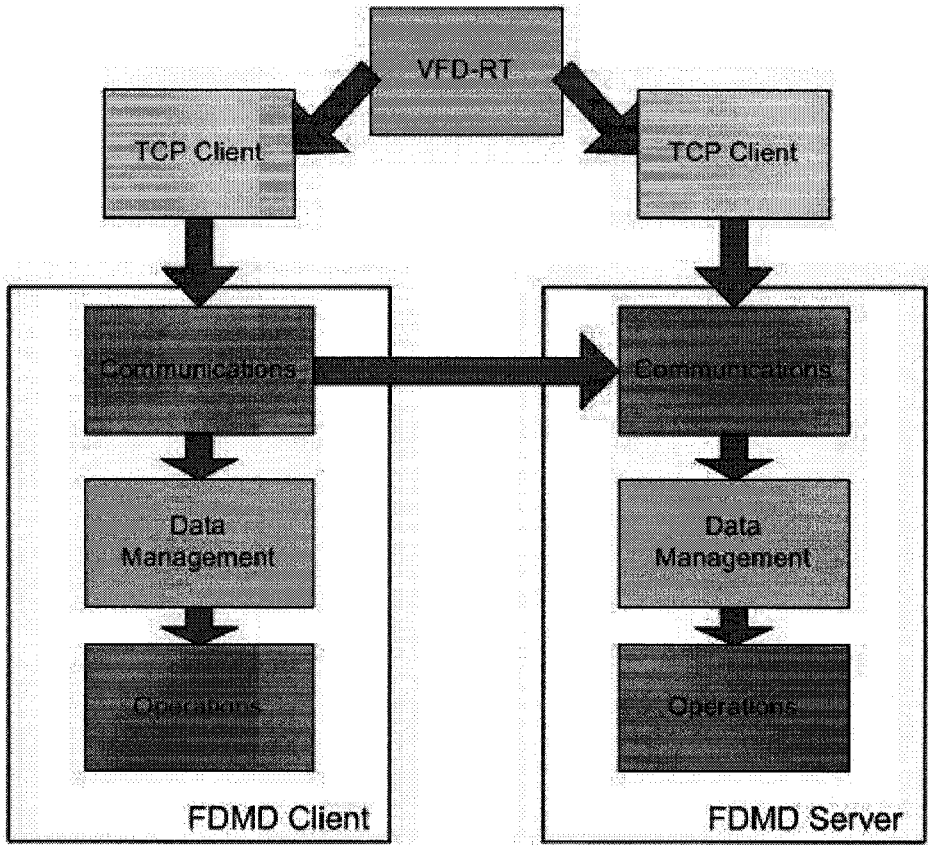

Figure 3.36: Use of VFD-RT in developing the data processing abilities of the FDMD software.

This configuration allows both the data display and computations of the FDMD software to be tested, as well as the communication between the different components of the system.

Hardware Development

In the Spring of 2008, a new 6 degree-of-freedom Stewart motion platform [41] was installed in the Applied Dynamics Research Group laboratory for its various projects [1, 2, 3, 4, 5, 6, 7, 8], including the development of the FDMD. The platform was used in the validation of sensor data during sensor performance tests. The test configuration is illustrated in the diagram in Figure 3.37 [8].

The ship motion is generated using the SMP application as the data provider. The VFD-RT simulation passes the ship motion data to two clients: the data logging client

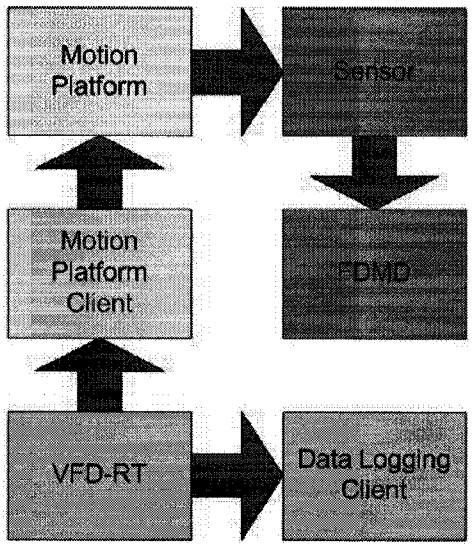

Figure 3.37: FDMD sensor data validation test configuration using the VFD-RT.

which, upon receiving a data copy message, directly writes the received data to a file; and the motion platform client, which has a structure similar to the TCP client. Rather than a TCP connection, the motion platform accepts motion data through the User Datagram Protocol (UDP), a similar but simpler protocol than the TCP. The organization of the motion platform client is the same as the TCP client, except that the *TCPConnection* class is replaced with an equivalent structure that uses UDP which also uses the Winsock 2.0 library. The motion data from this client drives the motion platform, on which the inertial sensor is mounted, as pictured in Figure 3.38; the small box behind the net under the test helicopter blade, near the middle of the picture, is the sensor.

The sensor is connected to the FDMD system, pictured on the bottom left, which runs on the laptop computer along with the VFD-RT. As the motion platform is moved by the ship motion from the VFD-RT, the sensor sends data to the FDMD which records it in a file on the laptop computer. At the same time, the VFD-RT logs the ship motion output on another file. After applying some data transformation due to the position of the sensor relative to the motion platform centre, the data from the two sources are compared. Thus the FDMD inertial sensor was tested and validated [8]. This test configuration also demonstrates the VFD-RT capability to include hardware in the simulation loop.

Figure 3.38: Photo of the FDMD sensor evaluation using the VFD-RT simulation as the motion platform controller.

FDMD User Interface Evaluation

In order for the user interface to gain acceptance from the final users, which are the landing signals officers (also pilots in Canada), the FDMD was evaluated with relevant military personnel from 12 Wing Shearwater in June of 2008, in Halifax, Nova Scotia. The purpose of the evaluation was to obtain user feedback and recommendations for improvement, based on the results [8].

The test required the participant to monitor colour changes in a projected 3-D visualization of a helicopter, while simultaneously pressing a button on the FDMD display when a quiescent period is observed. The purpose of the colour changes were simply to keep the participants focused on the helicopter, glancing at the FDMD display only occasionally as it was designed to be operated. The data is then analyzed, along with user feedback, to suggest improvements to the system. The system configuration for the test is shown in Figure 3.39.

The setup is similar to the configuration used for the initial implementation phase of

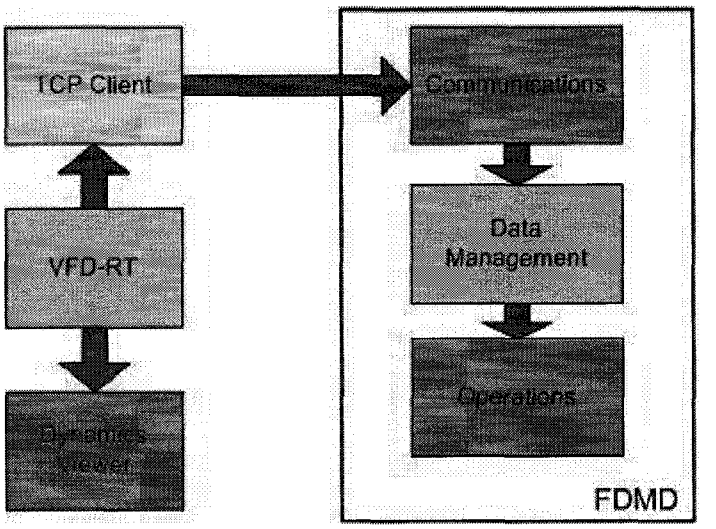

Figure 3.39: System configuration to evaluate the FDMD user interface using a VFD-RT simulation.

the FDMD project, with the addition of the DynamicsViewer client (Section 2.5). The SMP acts as the provider of simulated ship motion for the evaluation. The TCP client, as designed previously, is used to send simulated ship motion to the FDMD. At the same time, the DynamicsViewer client receives the same ship motion and displays it in the form of a 3-D visualization. Some modifications were added to the DynamicsViewer in order for the viewpoint to be fixed at the LSO compartment on the ship deck. As well, a feature was added whereby the application can read in a file containing data for colour changes of the simulated objects, depending on the timestamp received from the VFD-RT. Through the graphics card, the output from the DynamicsViewer is routed to a projector, which enlarges the image for the test.

The final setup is pictured in Figure 3.40, where the officer is operating the FDMD while observing ship motion and looking for colour changes of the helicopter on the image above.

Some valuable results were obtained from this evaluation. For example, the officers were asked to press a button to mark a quiescent period, with different FDMD screen settings. Figure 3.41 shows the results of this particular part of the evaluation. Out of the 4 screen settings and 2 test conditions (SS5 and SS6), screen setting 4 produced the best results as the officers did not misplace any quiescent period markers, in either test conditions.

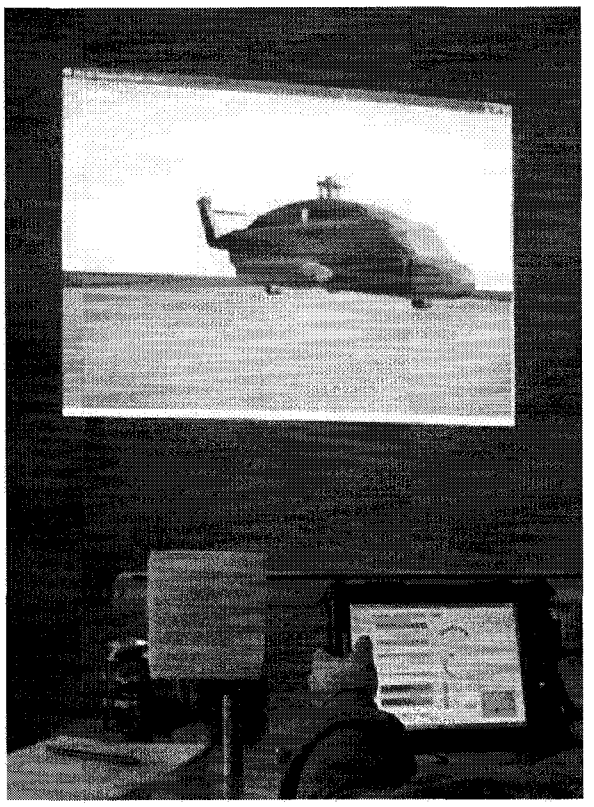

Figure 3.40: Photo of the FDMD user interface evaluation setup driven by the VFD-RT simulation.

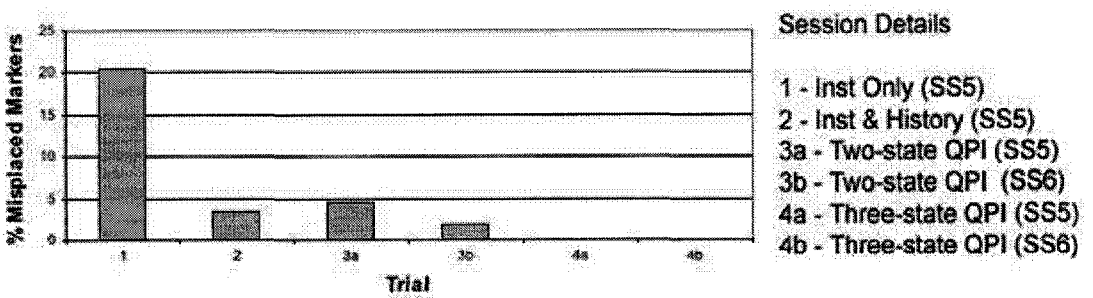

Figure 3.41: Sample result of the FDMD user interface evaluation; each of the six bars represent one specific configuration of the FDMD [8].

Some other important conclusions were drawn, including improvements to the existing screen settings with changing the scaling of certain features, reconsideration of the use of certain colours on the FDMD indicators, and recommended modifications of the button operations [8]. These results show that the VFD-RT simulation environment is applicable for the evaluation phase of product development.

Overall, the VFD-RT simulation environment was used to develop the FDMD from initial implementation to final evaluation. This case of the use of the simulation environment validates the capability of the VFD-RT to be applied to various stages of product development and for laboratory testing.

3.4 Summary

The VFD-RT simulation environment was:

- verified and improved through carefully organized unit testing of the source code;
- tested for the performance of its communication mechanism through the two-way message delay test, yielding a small delay value;
- tested for real-time performance under varying numbers of providers and clients and for characterization of its limitations;
- validated for a simple feed-forward data transfer simulation for research;
- validated for a feedback-type simulation that supports the interaction of virtual entities; and
- validated for simulations at different stages of product research and development, from software development to prototype evaluation.

From the performance tests, the limitations found led to some cautions required when using the simulation with higher numbers of connected providers and clients. In particular, the designs of the providers and clients should consider using *SendMessage()* for processing

copied data if tasks are time-critical. These providers and clients should also be connected to the VFD-RT in a sequence based on their importance to the simulation and their data processing time length. Due to the CPU capabilities, the simulation may not achieve close to real-time performance if the number of connected applications or the selected simulation frequency is too high. Within these limitations, the simulation environment was tested and found to be reliable.

These tests provide evidence that the simulation environment performs as intended, as a suite of applications that serves to control a variety of simulations. Throughout the development of the VFD-RT environment, the suite of applications – providers and clients – has grown to the following list:

- Data Display Client (DDC or VTC – VFD-RT Test Client)
- Data File Client (DFC)
- DynamicsViewer client (DVC)
- Experimental Ship Motion Provider (EMP)
- Motion Platform Client (MPC)
- RAS Provider-Client
- Ship energy client
- Ship Motion Provider (SMP)
- ShipMo3D Provider-Client (SM3D)
- TCP Client (TDR – TCP Data Router)
- VFD-RT Performance Test provider and client (VPTP and VPTC)

Based on the experience gained from creating these applications, templates for providers and clients were formed so that new applications can be readily added to the VFD-RT suite.

Chapter 4

Conclusion and Recommendations

This chapter provides a concluding discussion of the project as well as a list of recommendations for future development of the VFD-RT project.

4.1 Conclusion

The Virtual Flight Deck – Real-Time (VFD-RT) simulation environment is a versatile package designed for a variety of simulation needs including shipboard operation research, analysis, and development. The environment runs on the Windows XP operating system, widely used by the research community and industry. As a Windows application, the VFD-RT is integrated into the messaging system and takes advantage of the data copy message for data transfer between different components of the simulation, each of them being an independent Windows application.

The three-layer architecture of the VFD-RT divides the different component applications by their roles: the data input layer contains data providers that drive the simulation with mathematical ship motion models or with motion stored in files; the middle core layer holds the VFD-RT application responsible for time tracking and data structuring; and the clients in the data output layer produce the desired simulation results in various forms for various purposes. Certain components have the characteristics of both a client and a provider. These provider-client applications have facilities to handle each of their roles separately.

4.1.1 VFD-RT Applications

The core application is the essential component for simulation control of this centralized simulation model. Its purpose is to keep track of the passage of time, organize incoming simulation data from the input layer and periodically send data to the clients. The application operates two windows, one visible and another hidden, each tasked with the distinct role of managing the user and data interface, and of processing inter-application communication within the simulation, respectively. The user interface of the VFD-RT application allows the user to manage application connection through the list of providers and clients. Simulation runs can be controlled through buttons and parameter entry boxes.

The VFD-RT application employs specific time management, data transfer, and communication protocols to fulfill its purpose as a simulation core. The simulation time is managed through the Windows multimedia timer. A comparative investigation has concluded that, though the high performance timer has a higher resolution, the multimedia timer is more reliable in time keeping, particularly on multi-core processor systems. The multimedia timer has a resolution in the millisecond order of magnitude. The data transfer in the VFD-RT application is performed through a common data frame for data input from providers and output to clients. Through the use of direct addressing, data is efficiently transferred with a minimal amount of copying. The transfer to other applications is completed via the data copy message. A convention of connection establishment based on data flow was developed, along with the option in the provider or client application for an immediate or delayed data processing, depending on the timing requirements. The overall design of the core application that unites these features stems from use cases, which include simulation run controls and connection management.

The reliance on Windows for the core application has proved to have several advantages and drawbacks. Windows is currently available in a wide range of locations. Its intuitive user interface makes a simulation easy to run. The operating system also boasts an integrated messaging system with built-in infrastructure for data transfer. However, without direct control over every process, the operating system may produce noise that leads to slight

deterioration in performance in real-time simulations. Overall, the VFD-RT environment makes use of the benefits of Windows and offers provisions to minimize the shortcomings through the optimization and priority user options on the visible main window and through careful design of provider and client applications.

Throughout the development of the environment, providers and clients were constructed to complement the core as an application suite for a flight deck simulation environment. The ship motion provider is given as an example of a data generation module, responding to data requests from the core application. The data is passed on to clients, of which the 3-D visualization DynamicViewer client is presented as an example. These applications were tested for performance between the verification and validation cases.

4.1.2 Environment Evaluation

Extensive testing was performed at various levels of development to ensure reliability of data transfer between the data generation models in the providers and the client that uses this data. Starting from the code level, unit testing was completed to reveal improvements in the implementation. At the next level, the integrated system was tested through data copy message delay measurements and system load performance benchmarking. These evaluations are followed by validation at the operational level. The VFD-RT was put to the test first through feed-forward simulation for ship energy research, then through a feedback simulation supporting a 3-D virtual environment for the engineering analysis of the Replenishment-at-sea (RAS) operation. A final validation was conducted through use in the software-development, hardware-testing, and user-evaluation phases of a Flight-Deck Monitoring Display (FDMD) system.

The timer limitation is found to be the two-millisecond resolution of the multimedia timer, restricting the simulation loop frequency to below 500 cycles per second. Depending on the quantity and efficiency of the providers and clients to process data, this frequency maybe reduced. Efficiency of these applications can be improved through design, by delaying data processing until after the VFD-RT has serviced all providers and clients. However, some time-critical applications might be required to exchange this efficiency for on-time

data delivery by processing the data without delay. Selection of one of these two methods must be carefully considered with respect to the simulation conditions and objectives.

The VFD-RT development has provided a glimpse of its scope of application, namely in research, engineering analysis, and product development. Throughout the project, other possible uses for the simulation environment have been considered, including operator training, and general multi-body and vehicle simulations. Although the project development was guided by needs in flight deck simulation, the framework is applicable for a wide variety of fields. The communication infrastructure, the timing method, the data handling, and other features of the VFD-RT are directly transferable to many simulation cases, within the design limits of the environment. Furthermore, the VFD-RT is implemented in such a way that only minor code modification is required to be deployed as a generic multi-model centralized simulation. This extensibility makes the VFD-RT a versatile real-time simulation environment, in accordance with the project objective.

4.2 Recommendations

The VFD-RT project has the purpose of developing a suite of applications that serves as a versatile control centre for real-time simulation. As the project progressed, the addition of applications to the suite gradually extended the capabilities of the simulation environment to support future research and development work. These capabilities currently are:

- data monitoring (DDC);
- data recording (DFC);
- data visualization (DVC);
- motion platform control (MPC);
- network communication (TDR);
- ship motion generation (SMP, ShipMo3D);

- experimental and recorded time-series ship motion playback (EMP); and
- testing of new VFD-RT providers and clients (VPTP, VPTC).

Future development of the VFD-RT environment should concern the continual extension of the suite, to extend the applicability of the environment to simulation needs in research and development, in simulations of shipboard operations, and more. Some recommendations in the direction of future work are:

- to develop a method to create providers and clients with significantly less programming involved, such as through the use of application templates or a module creation program; users can thus focus on their research and development work rather than investing large amounts of time in programming and implementation;
- to create templates and wrapper applications for other programming languages that are more familiar to the simulation model developer, including facilities for integrating models developed in different languages into one provider or client;
- to streamline the performance testing of new providers and clients in order to rapidly identify the simulation timing limits that the application imposes; and
- to enclose the suite of VFD-RT environment providers, clients and core in a launcher application to simplify simulation runs.

References

[1] R. G. Langlois and C. A. Scribner. A Monte Carlo Simulation-based Approach for Helicopter/Ship Dynamic Interface Analysis. In *CASI AERO 2007 Conference*, Toronto, Ontario, April 2007.

[2] D. R. Linn. Development and Validation of a Planar On-Deck Helicopter Manoeuvring Simulation, Master's thesis, Carleton University, Ottawa, Canada, December 2003.

[3] A. R. Feldman. Development of an Experimental Aircraft/Ship Dynamic Interface Analysis Motion Facility for the Investigation of Helicopter Manoeuvring, Master's thesis, Carleton University, Ottawa, Canada, September 2004.

[4] J. P. Tremblay. Development and Validation of a Tire Model for a Real Time Simulation of a Helicopter Traversing and Manoeuvring on a Ship Flight Deck, Master's thesis, Carleton University, Ottawa, Canada, 2007.

[5] J. C. McKee. Simulating the Effects of Ship Motion on Postural Stability Using Articulated Dynamic Models, Master's thesis, Carleton University, Ottawa, Canada, March 2004.

[6] F. Khouli. Modelling and Attenuation Feasibility of the Aeroelastic Response of Active Helicopter Rotor Systems During the Engagement/Disengagement Phase of Maritime Operation, Ph. D. thesis, Carleton University, Ottawa, Canada, May 2009.

[7] A. Wall. A Discrete Approach to Modelling Helicopter Blade Sailing, Ph. D. thesis, Carleton University, Ottawa, Canada, January 2009.

[8] N. Bourgeois. Design and Development of a Flight Deck Motion Display System, Master's thesis, Carleton University, Ottawa, Canada, September 2008.

[9] B. York and S. Naylor. Real-time Simulation Under the Microsoft Windows Operating System, In *AIAA Modeling and Simulation Technologies Conference and Exhibit*, Monterey, California, August 2002.

[10] J. Nalepka, G. Williams, T. Dube, R. B. Bryant, and T. Danube. Real-time Simulation Using Linux. In *AIAA Modeling and Simulation Technologies Conference and Exhibit*, Montreal, Canada, August 2001.

[11] A. Buss and L. Jackson. Distributed Simulation Modeling: A Comparison of HLA, CORBA, and RMI. In *30th Winter Simulation Conference*, Los Alamitos, California, 1998.

[12] Institute of Electrical and Electronic Engineers (IEEE). IEEE Standard for Modeling and Simulation High Level Architecture (HLA) – Framework and Rules. IEEE 1516-2000, IEEE, 2000.

[13] Defense Modeling and Simulation Office (DMSO). Run-Time Infrastructure Programmer's Guide. US Department of Defense, 2000.

[14] The Object Management Group. Common Object Request Broker Architecture: Core Specification. 04-03-12, The Object Management Group, 2004.

[15] Sun Developer Network. Remote Method Invocation Home. *http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp*, May 2009.

[16] G. D. Padfield and M. D. White. Flight Simulation in Academia: HELIFLIGHT in its First Year of Operation. In *Challenge of Realistic Rotorcraft Simulation RAeS Conference*, London, 2001.

[17] A. A. Rodriguez, O. Cifdaloz, M. Phielipp, and J. Dickeson. Description of a Modeling, Simulation, Animation, and Real-Time Control (MoSART) Environment for a Broad Class of Dynamical Systems. In *45th IEEE Conference on Decision and Control*, Tempe, Arizona, 2006.

[18] A. A. Rodriguez, O. Cifdaloz, M. Phielipp, J. Dickeson, P. Koziol, D. Miles, M. Garcia, R. McCullen, J. Willis, and J. Benavides. Description of a Modeling, Simulation, Animation, and Real-Time Control (MoSART) Environment for a Class of 6-DOF Dynamical Systems. In American Control Conference, Tempe, Arizona, 2007.

[19] M. J. Corbin and G. F. Butler. *MulTiSIM: An Object-Based Distributed Framework for Mission Simulation*, Simulation Practice and Theory vol. 3. Elsevier Science, 1996.

[20] R. Tremblay. Low Cost, High Fidelity Flight Simulation Environment. In *AIAA Flight Simulation Technologies Conference*, San Diego, 1996.

[21] M. Roscoe and C. Wilkinson. DIMSS - JSHIP's M & S Process for Ship/Helicopter Testing & Training. In *AIAA Modeling and Simulation Technologies* Conference, Monterey, California, 2002.

[22] D. Lee, J. Horn, N. Sezer-Uzol, and L. Long. Simulation of Pilot Control Activity During Helicopter Shipboard Operations. In *AIAA Atmospheric Flight Mechanics Conference and Exhibit*, Austin, Texas, 2003.

[23] C. H. Wilkinson, M. F. Roscoe, and G. M. VanderVliet. Determining Fidelity Standards for the Shipboard Launch and Recovery Task. In *AIAA Modeling and Simulation Technologies Conference and Exhibit*, Montreal, Canada, 2001.

[24] J. W. Bunnell. An Integrated Time-Varying Airwake in a UH-60 Black Hawk Shipboard Landing Simulation. In *AIAA Modeling and Simulation Technologies Conference and Exhibit*, Montreal, Canada, 2001.

[25] NetApplication. Operating System Market Share. *http://marketshare.hitslink.com/operating-system-market-share.aspx?qprid=8*, April 2009.

[26] C. Petzold. *Programming Windows 95.* Redmond, Washington: Microsoft Press, 1996.

[27] C. Petzold. *Programming Windows 95.* Redmond, Washington: Microsoft Press, 1996.

[28] The Object Management Group. OMG UML 2.0 Specifications. *http://www.omg.org/uml/*, September 2007.

[29] J. Mischel. High Performance Timing Under Windows. *Dev Source*, November 2006.

[30] C. D. Chambers and M. Brown. Timing Accuracy Under Microsoft Windows Revealed through External Chronometry. *Behavior Research Methods, Instruments and Computers*, 35:96–108, 2003.

[31] A. R. J. M. Lloyd. *Seakeeping: Ship Behaviour in Rough Weather.* United Kingdom: A. R. J. M. Lloyd, 1998.

[32] K. A. McTaggart. *SHIPMO07: An Updated Strip Theory Program for Predicting Ship Motions and Sea Loads in Waves.* Defence Research Establishment Atlantic, Dartmouth, Nova Scotia, Canada, March 1996. DREA Technical Memorandum 96/243.

[33] K. W. Tsui. ADG DynamicsViewer Reference. Document No. ADL/07/KWT/1, Mechanical and Aerospace Engineering, Carleton University, Ottawa, Canada, 2009.

[34] F. Khouli, A. Wall, R. G. Langlois and F. F. Afagh. Proposed Hybrid Passive and Active Control Strategy for the Transient Aeroelastic Response of Helicopter Rotor Blades During Shipboard Engage and Disengage Operations. In *American Helicopter Society 64th Annual Forum and Technology Display*, Montreal, Canada, April 2008.

[35] A. Tucker. Adding Console I/O to a Win32 GUI App, *http://www.halcyon.com/ ast/dload/guicon.htm*, January 2009.

[36] K. A. McTaggart and R. Langlois. Physics-Based Modelling of Ship Replenishment at Sea Using Distributed Simulation. In *Society of Naval Architects and Marine Engineers Annual Meeting and Expo*, Providence, Rhode Island, October 2009.

[37] D. Bleichman, R. Langlois, M. Lichodzijewski, and D. Brennan. Development of a High Level Architecture Federation of Ship Replenishment at Sea. Technical Report, Feb. 2009.

[38] Canadian Department of National Defence. Seamanship Rigging and Procedures Manual, Report B-GN-181-105/FP-E00.

[39] K. McTaggart. *ShipMo3D Version 2.0 User Manual for Simulating Time Domain Motions of a Freely Maneuvering Ship in a Seaway.* Defence R&D Canada Atlantic, Dartmouth, Nova Scotia, Canada, February 2008.

[40] Canadian Department of National Defence. Flight Deck Motion Display Requirements Specification. September 2008.

[41] MOOG Motion Systems Division, MOOG 6 DOF 2000E Motion System Interface Definition Manual, Document No. LSF-0446, Rochester New York, November 1999.

Appendix A

Ship Motion Input File for the Ship Energy Validation Case (shpr.inp)

```
6_045_10.rsp   ship response input file name

79             number of frequency components
```

Appendix B

Ship Energy Client Input File (shipenergy.inp)

```
4.8d6   ship mass, M [kg or slugs]

4.5     roll radius of gyration, rxx

30.     pitch radius of gyration, ryy

30.     yaw radius of gyration, rzz

0.02    surge nondimensional added mass, A11/M

0.6     sway nondimensional added mass, A22/M

2.      heave nondimensional added mass, A33/M

0.15    roll nondimensional added mass, A44/(M rxx^ 2)

1.      pitch nondimensional added mass, A55/(M ryy^ 2)

0.3     yaw nondimensional added mass, A66/(M rzz^ 2)

5.d0    heave natural period, T3

10.d0   roll natural period, T4

5.d0    pitch natural period, T5
```

Appendix C

ShipMo3D Seaway Input File (hlaRasTestCase10BuildSeaway2.inp)

```
begin SM3DBuildSeaway2
label HLA RAS test case 10, Hs = 3.25 m, Tp = 9.7 s, Bretschneider
spectrum, phases from ShipMo3D 1.1
seawayFileName hlaRasTestCase10SeaState5.xml
waterDensity 1025.0
sampleParams 3600.0 0.1
seawayOption componentPhase
begin componentPhaseSeaway
    componentPhase  330.000  0.407  0.043  134.422
    componentPhase  330.000  0.445  0.099  77.577
    componentPhase  330.000  0.489  0.227  291.986
    componentPhase  330.000  0.562  0.342  96.537
    componentPhase  330.000  0.591  0.300  5.607
```

componentPhase	330.000	0.631	0.377	82.607
componentPhase	330.000	0.689	0.385	53.354
componentPhase	330.000	0.736	0.351	54.990
componentPhase	330.000	0.786	0.328	43.860
componentPhase	330.000	0.835	0.346	340.609
componentPhase	330.000	0.918	0.267	125.050
componentPhase	330.000	0.944	0.207	95.314
componentPhase	330.000	0.991	0.226	171.860
componentPhase	330.000	1.049	0.213	271.170
componentPhase	330.000	1.110	0.173	175.365
componentPhase	330.000	1.149	0.134	7.774
componentPhase	330.000	1.181	0.140	54.297
componentPhase	330.000	1.236	0.146	151.234
componentPhase	330.000	1.297	0.129	178.403
componentPhase	330.000	1.350	0.107	101.695
componentPhase	330.000	1.391	0.109	303.644
componentPhase	330.000	1.464	0.087	38.766
componentPhase	330.000	1.484	0.079	133.479
componentPhase	330.000	1.545	0.080	49.623
componentPhase	330.000	1.586	0.071	42.004

```
    componentPhase  330.000  1.635  0.070  77.974

    componentPhase  330.000  1.689  0.068  141.484

    componentPhase  330.000  1.746  0.060  96.847

    componentPhase  330.000  1.791  0.058  219.399

    componentPhase  330.000  1.854  0.050  72.818

    componentPhase  330.000  1.888  0.048  208.891

    componentPhase  330.000  1.953  0.047  349.151

    componentPhase  330.000  2.000  0.029  253.119
end componentPhaseSeaway
end SM3DBuildSeaway2
```

Appendix D

ShipMo3D Supply Ship Motion Input File (ProtecteurDeepFreeMo2.inp)

```
begin SM3DFreeMo2
label RAS Test Case Protecteur in sea state 5
saveFreeShipDBOption saveFreeShipDB
shipDBFileName ProtecteurDeepShipForMotion.bin
loadCondition 1025.0 10.09 -0.21 8.40 -0.063
timeSeriesFileName ProtecteurDeepTimeSeries.bin
seawayOption waves
seawayFileName hlaRasTestCase10SeaState5.xml
timeParameters 0.1 0.0 20.0 20.0
nonLinearOption linear
dispsFixedOMDeg 0.0 0.0 0.0 0.0 0.0 0.0
speedOKnots 10.0
rudderDeflectsODeg 0.0
rudderVelsODeg 0.0
rpmsPropellersO 368.0
begin maneuvers
setRpm All 368.0
```

```
setRudderCourse All 0.0
elapsedTime 200.0
end maneuvers
outTimeIntervals 0.4 10.0
outTimeSeries disp vel acc
outRudderProp noRudder noProp noAziProp
plotOption noPlots
end SM3DFreeMo2
```

Appendix E

ShipMo3D Receiving Ship Motion Input File (HalifaxDeepFreeMo2.inp)

```
begin SM3DFreeMo2
label RAS Test Case Halifax in sea state 5
saveFreeShipDBOption saveFreeShipDB
shipDBFileName HalifaxDeepShipForMotion.bin
loadCondition 1025.0 5.160 -0.10 6.49 -0.086
timeSeriesFileName HalifaxDeepTimeSeries.bin
seawayOption waves
seawayFileName hlaRasTestCase10SeaState5.xml
timeParameters 0.1 0.0 20.0 20.0
nonLinearOption linear
dispsFixed0MDeg 0.0 -52.25 0.0 0.0 0.0 0.0
speed0Knots 10.0
rudderDeflects0Deg 0.0
rudderVels0Deg 0.0
rpmsPropellers0 185.0 185.0
begin maneuvers
setRpm All 185.0
```

```
setRudderCourse All 0.0
elapsedTime 200.0
end maneuvers
outTimeIntervals 0.4 10.0
outTimeSeries disp vel acc
outRudderProp noRudder noProp noAziProp
plotOption noPlots
end SM3DFreeMo2
```