A Benchmarking Suite for Comparing
Middleware Performance

by

Allison Malloy, B.Sc.

A thesis submitted to the Faculty of Graduate Studies and Postdoctoral Affairs
in partial fulfillment of the requirements for the degree of

Master of Applied Science in Electrical and Computer Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering (OCIECE)
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada, K1S 5B6

January 2011

© Copyright 2011, Allison Malloy
NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.
Abstract

One of the main problems in critical mission system development is analyzing the performance of an application, which involves accurately measuring the underlying hardware and operating system or Run Time Infrastructure (RTI). Unfortunately, there is little guidance on how to best obtain these measurements, and the problem becomes exaggerated by the presence of communications middleware in distributed applications. MiddleMeasure was developed as a benchmarking suite to solve this problem. MiddleMeasure was tested to show that simple tests can be used to measure the throughput and latency performance between nodes in a network on various RTIs, optionally using different middleware solutions. Analysis of the results of the MiddleMeasure tests can be used to compare RTIs and middleware under different network and disk loading conditions and to assist in identifying bottlenecks in distributed systems. MiddleMeasure produces performance metrics that can be used to reduce risk in software development of distributed applications.
# Table of Contents

Chapter 1: Introduction ........................................................................................................ 2

Chapter 2: Conceptual Background ..................................................................................... 7

2.1 Software Development Methodology ............................................................................ 8

2.2 Importance of Benchmarking ....................................................................................... 9

2.3 Data Distribution Service (DDS) .................................................................................. 11

2.4 Common Object Request Broker Architecture (CORBA) ............................................. 14

2.5 TCP and UDP .............................................................................................................. 16

2.6 Data Loss Prediction ................................................................................................... 17

2.7 INtime Real-Time Operating System .......................................................................... 19

Chapter 3: State of the Art in Benchmarking and Performance Evaluation ...................... 21

3.1 Benchmarking Suites .................................................................................................. 21

3.1.1 Measuring Latency ................................................................................................. 24

3.1.2 Measuring Throughput ........................................................................................... 24

3.1.3 Existing Benchmarking Reports .............................................................................. 25

Chapter 4: Research Questions – Problem Statement ......................................................... 26

4.1 Problem Analysis ........................................................................................................ 26

4.2 Statement of the Thesis ............................................................................................... 28

4.3 MiddleMeasure ........................................................................................................... 29

4.4 Contributions of the Thesis ......................................................................................... 32

4.5 Scope .......................................................................................................................... 33

Chapter 5: Test Design, Theoretical Analysis and Predictions ........................................ 35

5.1 MiddleMeasure Test Suite Design .............................................................................. 35

5.2 Test Descriptions ....................................................................................................... 42

5.2.1 Single Receiver Latency Test: ................................................................................ 42

5.2.2 Multiple Receiver Latency Test .............................................................................. 44

5.2.3 Single Receiver Throughput Test ........................................................................... 45

5.2.4 Multiple Receiver Throughput Test ....................................................................... 47

5.2.5 DDS Tests .............................................................................................................. 48

5.2.6 CORBA Tests ....................................................................................................... 49
List of Figures

Figure 1: Embedded System Architecture Breakdown ........................................... 10
Figure 2: CORBA: Request from Client to Object Implementation .......................... 16
Figure 3: INtime RTOS Concept Diagram .............................................................. 19
Figure 4: Laboratory vs. Real Life Networking ....................................................... 23
Figure 5: MiddleMeasure Component Diagram ...................................................... 36
Figure 6: Control Process Flow Diagram ............................................................... 37
Figure 7: Slave Process Flow Diagram ................................................................. 38
Figure 8: Single Latency Test .............................................................................. 43
Figure 9: Multiple Latency Test .......................................................................... 45
Figure 10: Single Receiver Throughput Test ....................................................... 47
Figure 11: Configuration 1 ................................................................................. 54
Figure 12: Configuration 2 .................................................................................. 56
Figure 13: Middleware Testing: ACE TAO Single Latency Test Results ............... 61
Figure 14: Middleware Testing: ACE TAO Single Throughput Test Results .......... 62
Figure 15: Middleware Testing: Single Throughput Test Results ....................... 64
Figure 16: Middleware Testing: Single Participant Latency Results (Best Effort) ... 65
Figure 17: Middleware Testing: Single Participant Latency Test Results (Reliable) .. 66
Figure 18: Middleware Testing: Latency Test Results with Varying Payload Types .. 69
Figure 19: RTI Testing: InterCOM DDS Single Participant Throughput Results ...... 70
Figure 20: Middleware Testing: Multiple Publisher Throughput Test Results ........ 71
Figure 21: Middleware Testing: Throughput Test Results (Reliable) ..................... 72
Figure 22: Middleware Testing: Throughput Test Results by Number of Subscribers. 73
List of Tables

Table 1 Benefits of Publish/Subscribe Architecture ....................................................... 12
Table 2 Benefits of Client/Server Architecture ................................................................. 15
Table 3 MiddleMeasure Payload Options ........................................................................... 39
Table 4 MiddleMeasure DDS QoS Options ......................................................................... 40
Table 5 Configuration 1 Hardware ....................................................................................... 55
Table 6 Configuration 2 Hardware ....................................................................................... 58
Table 7 CORBA Tests ........................................................................................................... 60
Table 8 DDS Tests ................................................................................................................. 67
Table 9 TCP/UDP Tests ......................................................................................................... 76
Chapter 1: Introduction

In the realm of distributed systems application development today, there are many design constraints to consider; not the least of which is what platform(s) and what methods of communication will best suit a given project. System architects may have an idea of what platforms they would like to employ in the development of a system, but may not be able to prove the validity of their assumptions until well into the project.

In order to reduce the risk of a project system architects employ the tools at their disposal, from the use of development methodologies, prototyping, performance prediction tools and their past experience. The ability to reduce risk through performance benchmarking is the motivation behind this research.

Currently, it is possible to find benchmarking tools that evaluate specific communication methods, such as TCP, UDP, CORBA, and DDS, but not in a consistent, easily comparable manner. Users would have to run CORBA benchmarking, separate from DDS benchmarking and separate from TCP/UDP testing to get full spectrum results [1]. In many cases the benchmarking suites are inflexible for the end users, not allowing them to adapt the tests for their specific needs.

Ideally there should be a tool that allows the architects to validate that the proposed hardware, operating system and middleware will satisfy the system’s performance requirements. The tool should permit the system developers to measure the latency and throughput of communications between nodes in a network on various platforms and using different middleware tools. Latency is the time between the message being sent and the recipient beginning to process it, whereas throughput is the number of messages that
can be exchanged per second. The tool should be application independent but should also allow for some customization for application-specific needs if required.

The MiddleMeasure suite was developed and tested to demonstrate that it is feasible to create one benchmarking suite that will test communication latency and throughput across multiple platforms while outputting the results in a consistent format so that they can be analyzed and compared easily.

Using the output from the latency benchmarking tests run on real-time operating systems it is possible to accurately measure the minimum time to exchange data between nodes. Using the output from the throughput benchmarking tests in a network it is possible to predict the expected data loss percentage under varying network load conditions. Providing system architects with this information at the onset of the project development process will greatly reduce the amount of time tweaking the performance of the system, which can usually only be accomplished at the end of the project, the designers should be able to design the solution around the known bottlenecks or constraints of the chosen network.

The MiddleMeasure suite is an open source benchmarking suite comprised of reusable latency and throughout tests for Real Time Innovation’s NDDS, Gallium’s InterCOM DDS and the ACE TAO ORB middleware products as well as TCP and UPD throughput and latency tests. The tests have support for different operating systems, Windows XP, Linux and INtime. Also included in the suite is a test generation tool that allows users to construct instruction lists including all or a selected subset of the tests to be run with the included test harness. The test harness provides a client and server application that run
the tests in the desired configuration, control the execution (for example by killing any runaway processing) and recording the results. The configuration of the tests can include the payload type, payload size, quality of service settings, throughput amount, operating system and whether disk or network loading is being used. The disk and network loading applications are also included in the suite and can be used to simulate different real world loading conditions.

The results from the testing show that the throughput performance on Linux is slightly better than that on Windows. The results also showed that there are some anomalies around the 1024 byte payload size on Windows, due to the default buffer sizes employed by the operating system. The testing revealed that the DDS solutions have better throughput performance than the CORBA solution, and that for the small networks (that were under test) the InterCOM DDS performs better than the NDDS.

The testing performed on the INtime real time operating system resulted in metrics that were less deterministic than expected, but showed that the communication latency performance between two nodes running INtime is roughly 40 times better than when the sending node is running Windows XP and the receiving node is running INtime. In addition, the tests with the INtime operating system prove that the MiddleMeasure suite is extensible to include other RTIs. The UDP and TCP test results revealed that the TCP throughput performance can in fact be better the UDP throughput performance. The TCP protocol allows for messages to be bundled together and sent as one message thus improving performance. The operating system configurations can be modified to change
the message buffer size, or to disable this bundling altogether. Options that the system architects may find quite useful in the designing of their applications.

The conclusions drawn from the research are that the MiddleMeasure benchmarking suite is a useful, scalable and extensible tool that can be used to evaluate the latency and throughout performance of different RTIs and to measure the performance of different middleware solutions on one or more RTIs. The single suite can be extended to include latency and benchmark testing for additional hardware and operating systems combinations. Overall, the MiddleMeasure benchmarking suite is an improvement over other available benchmarking suites as it is an open source solution that allows user to evaluate different middleware options in addition to measuring the throughput and latency performance of different RTIs.

Chapter 2 provides conceptual background information about benchmarking techniques, Transport Protocols: DDS, CORBA, TCP and UDP, and data loss prediction.

Chapter 3 summarizes current DDS and CORBA benchmarking research.

Chapter 4 includes an analysis of the state of the art from Chapter 3 in order to present the limitations of the current benchmarking solutions and propose features that would make an ideal latency and throughput benchmarking suite. This chapter also includes the claim that it is possible to create a single benchmarking solution that can be configured to test the latency and throughput of different RTIs and middleware solutions and provide metrics that are immediately comparable. Also included in chapter 4 are the contributions and scope of the thesis.
Chapter 5 first analyses the processes involved in benchmarking RTI send and receive operations, using DDS, CORBA, and TCP. Secondly, it presents a detailed description of the MiddleMeasure suite that has been developed in this research followed by the test procedures used in this research.

Chapter 6 presents and analyzes the results of the development of the MiddleMeasure suite and results of the single participant latency test, multiple participant latency test, single subscriber and publisher throughput test, multiple subscriber throughput test, multiple publisher throughput test and TCP ping/pong test.

Conclusions and suggestions for future work are presented in Chapter 7.
Chapter 2: Conceptual Background

This chapter provides conceptual background information about software development methodology, the importance of benchmarking when designing a system, and describes the two middleware solutions under test DDS and CORBA; representing the publish and subscribe and client and server communication distribution mechanisms, respectively. Following those sections there is a section about TCP and UDP Transport Protocols and finally a section about data loss prediction.

The dominant publish and subscribe middleware solutions include DDS, Java Message Service (JMS), High Level Architecture (HLA) and the CORBA Notification Service (an extension of the CORBA standard) [2]. Joshi [3] concludes that DDS is the superior middleware solution for loosely-coupled systems, since RTI produces DDS it is unclear whether the whitepaper is impartial. An independent comparison of DDS, JMS and the CORBA Notification Service conducted by Vanderbilt University [4] shows that DDS solutions have better latency and throughput results than either of the other two solutions, as such DDS was chosen as the representative publish and subscribe middleware for this study.

CORBA was selected as the representative client and server middleware mechanism because it and DDS are managed by the OMG and share a common framework through the use of the Interface Definition Language.
2.1 Software Development Methodology

Before exploring the importance of benchmarking it is important to have an understanding of the software development process.

Software is traditionally developed by following one of several known development processes: the waterfall model [5], the spiral model [6] or iterative and incremental development [7]. Most commercial software producers use some variation or combination of these models. The waterfall method describes a process that starts with the requirements analysis, followed by software design, development and integration, then testing, deployment and maintenance. The spiral model expands on this process and allows the waterfall phases to be repeated in an incremental fashion to create larger applications with less risk by analyzing the requirements and producing a new design for each phase of the development, and an executable that can be tested at each phase. To further reduce the risk of large scale application development Iterative and Incremental Development divides its model into four phases; Inception, Elaboration, Construction and Transition. This method is a logical extension the Spiral Model, which should be used to complete the Construction phase. The Inception phase includes developing a project plan, determining customer requirements, producing an initial risk assessment and potentially creating a prototype. The Elaboration phase is used to further develop the project plan, design UML models and if not already done, create a prototype. The Construction phase follows at which point the solution is developed. Then the Transition phase wraps up the project by completing testing and adding in any enhancements to the solution.
2.2 Importance of Benchmarking

Benchmarking is a very vague term, benchmarking tests are simply comparative studies that use a set of tests to garner a different set of results for each entity under test. The scope of interest for this document is related to latency and throughput benchmarking.

There are many papers about the benchmarking of various transport protocols. Sachs, Kounev, Appel and Buchman [8] discuss the importance of publish and subscribe benchmarking as a tool to analyze the performance of a system; weighing quality of service with performance costs. Vanderbilt University has been a large producer of DDS research, the OMG and its members have published a number of papers in addition to holding several annual conferences to further the development of both DDS and CORBA. Tuma and Buble [1] concluded in their report “Technical Report on Open CORBA Benchmarking” that it is possible to create a small CORBA benchmarking suite whose test results can be used to deduce the performance of different systems. The difficulty with relying on published research is that all the metrics provided pertain to specific hardware and software combinations and the results quickly become out of date as both the software and hardware solutions become out dated themselves.

Benchmarks are often used by product vendors to make their tool seem better (faster, more reliable, smaller footprint and more resilient) than those of their competitors. This makes it difficult to evaluate different solutions for a particular problem without running an independent set of benchmark tests.
Ideally the performance analysis could be partially performed during the design or prototype development phases, to ensure that the preferred Run-Time Infrastructure (RTI) is appropriate for the system. The benchmark tests should quantitatively show the differences in latency and throughput with varying qualities of service for each of the RTI and middleware possibilities of interest.

In real time and time critical systems, the performance metrics supplied by the latency tests can be used to compute the timing constraints of the equipment so that the design can be evaluated against the requirements of the system prior to the project being complete therefore reducing the risk of the project [9].

A software application's architecture can be broken down into three logical components; the run-time infrastructure, the middleware and the application code [10][11].

| Application Code | Middleware | Run-time Infrastructure (RTI) |

**Figure 1: Embedded System Architecture Breakdown**

The application code is the software application that provides the main functionality of the system. For example, the application code of the GeoSurv II UAV project controls the UAV flight and the ground station components of the software [16].

The RTI is composed of the hardware, devices drivers and the operating system.
The middleware is in the middle and generally consists of a number of libraries that abstract commonly used functionality out of the application code. Third party libraries are often used in software development rather than implementing solutions from scratch. The benefits are two-fold: there is risk reduction from using a well-known tested solution and time savings from not having to develop some portions of the solution.

2.3 Data Distribution Service (DDS)

A Data Distribution Service (DDS) is networking middleware that simplifies network programming in an application [17] by facilitating the loose coupling between communicating nodes [4] and by shielding the application from the details of the message packing and distribution. In loosely-coupled systems, the data is the central component and applications communicate without knowledge of each other or the operations performed on the data. DDS is considered a data-centric service.

The Object Management Group (OMG) manages the definition and minimum set of capabilities of a DDS, and the standardized API that all DDS providers must use [18][19].

The purpose of DDS is to standardize the data-centric publish and subscribe model while hiding the details of the implementation from the application layer. DDS is well suited to distributed real-time and embedded (DRE) systems where large quantities of data must be shared in a timely manner to multiple hosts [16]. Nodes are defined as publishers or subscribers of data, or both, and use defined “topics” to share data samples. A publisher can publish data for a known topic and any subscriber who is interested in that topic will
receive the message, each of these messages is called a data sample. For example, in a situational awareness application one central data manager node can broadcast a message to many display nodes.

Topics are defined in Interface Definition Language (IDL)[17], a simple C like language that is then processed into other languages by an IDL compiler. The IDL separates object interfaces from their implementations; it provides abstraction, defines interfaces that applications can understand and can use easily, regardless of platform, operating system, programming language and network connection.

A publisher application can be in Java while the subscriber can be in C++ for example, but only one IDL file is required to describe the messages shared between the two nodes. It is through IDL that DDS handles inter-operability across various platforms and different coding languages, and when used in conjunction with the wire protocol facilitate the inter-operability between DDSs [18].

<table>
<thead>
<tr>
<th>Table 1 Benefits of Publish/Subscribe Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pros</strong></td>
</tr>
<tr>
<td>Many-to-many communication: ideally suited for this use</td>
</tr>
<tr>
<td>Time critical data distribution</td>
</tr>
</tbody>
</table>
DDS is ideally suited for data-centric distributed system, where there are limited or no directed communications between the nodes in the network. The pros and cons of using a publish and subscribe architecture are illustrated in Table 1.

The DDS architecture has a number of defined entities that allow it to be easily adapted into applications while maintaining a publish and subscribe support framework. Relevant entities for producing benchmark testing are include the DomainParticipantFactory, a singleton factory that is the main entry point to DDS and factory creator for the DomainParticipant; the entry point for the communication in a specific domain. The DomainParticipant acts as a factory for the creation of DDS Publishers, Subscribers and Topics. The Topic entity is the most basic description of the data that is to be published and subscribed to. The Publisher is the object responsible for the actual dissemination of publications, and the Subscriber is the object responsible for the actual reception of the data resulting from its subscriptions. The DataWriter is the entity that allows the application to set the value of the data to be published under a given Topic, and theDataReader is the entity that allows the application to declare the data it wishes to receive (by making a subscription using a Topic) and to access the data received by the attached Subscriber.

In addition, DDS supports setting various Quality of Service (QoS) definitions that allow application developers to manage the finer aspects of data distribution, such as reliability, history and lifespan [19].

The Reliability QoS setting options are Best Effort and Reliable. According to the OMG DDS Specification [20], using the Reliable setting will cause the DDS writer to wait until
the message was received by the all matched Data Readers before returning a success indication to the calling application. The DDS can be provided a max_blocking_time value after which the Data Writer will time out and return a failure indication to the application. Using the Best Effort setting will return a success indication to the application as soon as the data is sent.

The Liveliness QoS setting is used to avoid the amount of stale data delivered to the application. The options for the setting are Finite or Infinite. When using the Finite option, each data sample (or message exchanged) has an associated expiration time after which the data is removed from the cache of the Data Writer and Data Reader. Using the Infinite option sets the lifespan of the data to an infinite duration, removing the concept of an expiration time.

The History QoS setting is used to limit the amount of past history changes kept by the Data Writers. The options for the setting are Keep All, Keep Last and the OMG allows for an option depth setting to provide an in between value. The History setting is useful in cases where the same data changes value more than one. The Keep All setting will share the entire history of the sample from the first change to the latest change with the Data Readers.

2.4 Common Object Request Broker Architecture (CORBA)

The OMG group also oversees the CORBA specifications and like DDS, CORBA uses the Interface Definition Language to create a language independent message definition. OMG has standardized mappings between many languages: C, C++, Java and even older
languages such as Ada, Cobol and Python. CORBA is service-centric and represents the client and server model [12].

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point-to-point communication: ideally suited for this use</td>
<td>Many-to-many communication: not well suited as messages would have to be sent to each receiver separately</td>
</tr>
<tr>
<td>Well suited to naturally centralized data</td>
<td>Single point failure could lead to performance bottlenecks</td>
</tr>
<tr>
<td>Well suited for file systems and database</td>
<td></td>
</tr>
</tbody>
</table>

In contrast to DDS, CORBA is a client and server based middleware solution. It is ideal for use in distributed networks that require point to point communication, as illustrated in Table 2.

In creating CORBA applications, developers must create clients that pass the IDL message to the Object Request Broker (ORB), which is part of the middleware product, and servers in the receiving applications that receive the IDL messages from the Object Request Broker and process them in the Object Implementation [21].
In both DDS and CORBA, the middleware must provide IDL translation functions that output message definition files in a selected language. In CORBA, the translation functions also generate implementation file stubs that are to be filled in by the developers of the message receiving servers; this is the object implementation shown in Figure 2.

2.5 TCP and UDP

Underneath the middleware products like DDS and CORBA, lay the basic network protocols; Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). TCP is a reliable protocol. A TCP message is not considered sent until it is acknowledged as received by the recipient; the sender will continue to send the message until it receives an acknowledgement or until the number of retries has been exceeded. In general, the number of retries is configured for each application. In contrast UDP is unreliable, a message is considered sent even if no recipients receive it; no acknowledgement message is required so there is no mechanism for determining if the message was received by any other nodes.
To initiate TCP or UDP data transfers, application developers must use sockets to define the port and Internet Protocol (IP) address of a process to send and/or receive data. Sockets are basic coding building blocks that most programmers have used at one point or another. Operating systems provide an Application Programming Interface (API) to facilitate the use of network sockets. Without middleware, developers must handle opening and closing sockets along with the address and port management and the message transmission management [22].

### 2.6 Data Loss Prediction

When designing a system, it is important to understand the limitations of the hardware and software solutions chosen to develop on. In distributed systems a large part of the decision making should revolve around the receipt of data in a reliable and timely manner. It may be that some data is more important than others and must be received and handled as a priority above the rest of the system’s functions. To ensure that the solutions chosen are adequate for the application it is imperative that the data exchange requirements are well understood and that the limitations can be identified and mitigated.

In the context of distributed embedded systems, the potential for data loss during the transfer of data between two nodes is the most interesting point of failure. Transfer protocols differ in their approaches to handling potential data loss; TCP provides a reliable transport that ensures packets are received at their destination, while UDP assumes the application will provide packet loss support. In both CORBA and DDS the application developers can request certain quality of service from the middleware.
Data loss typically stems from an overloaded network that leads to some data not getting to the intended recipient(s). Depending on the quality of service settings and protocol used, temporary network congestion can result in even more congestion and worsening quality of service [23]. In TCP-based communications the loss of messages triggers the re-sending of messages worsening the congestion situation. It is important for application designers to understand which messages are critical and should have higher priority and potentially be retransmitted until received, and which ones can be lost without affecting the overall functionality of the solution. When too many messages are sent across a network, most routers will start to discard some, which ones are dropped can be controlled based on the priority and QoS settings [23].

The benchmark evaluation of the system is valuable so that the system designers can adequately weigh the importance of latency versus loss protection to suite their particular application requirements. In mission critical systems it is sometimes desirable to enter into a degraded state where some functionality is temporarily disabled until the network congestion ceases; this is called graceful degradation. For example, in their paper, Bolot and Turletti [24] discuss the graceful degradation of image processing on the internet. When packet loss is detected the receivers react and leave the current network layer, and join a different one.
2.7 INtime Real-Time Operating System

An operating systems used in the research is one Tenasys has developed; it is a real time operating system called INtime [25] that can run in parallel with non real-time versions of Windows on an x86 host.

Figure 3: INtime RTOS Concept Diagram [25]
The INtime software runs its own kernel that accesses the hardware interrupts and runs alongside the windows kernel as shown in Figure 3. Tenasys claim that the real-time INtime processes run at higher priority than any Windows processes.
Chapter 3: State of the Art in Benchmarking and Performance Evaluation

This chapter describes the utility of benchmarking suites, explanations of latency and throughput.

3.1 Benchmarking Suites

Benchmarking suites are very common in the middleware and networking world, they are a collection of benchmarking tests that measure the performance of a processor, or an application, or a system [26]. Typically, a suite is designed to prove that a particular solution is better than another, but they are rarely impartial.

Most suppliers of middleware products provide some benchmarking tests with their product. For example, RTI’s NDDS, Gallium’s InterCOM DDS, and RTI’s CORBA all provide such tools. In addition, there are examples of TCP or UDP communications latency and throughput tests readily available; through operating system forums from producers such as Microsoft and Linux and through specialized operating system developers such as Tenasys. It is possible to use the example tests to generate some test scenarios that may permit the user to evaluate DDS using different products on various platforms, or to compare DDS to CORBA on a given platform. Building a suite in such a way can be time consuming and requires effort to compare the output of the tests designed by different developers; as such it is preferable to use a suite that has already assembled the desired test and produces an output in a consistent format making analysis simpler.
DDS TouchStone [27] is a framework for benchmarking different DDS implementations that claims to be vendor neutral, but it sponsored by PrismTech the makers of OpenSpliceDDS.

DynBench [27] is a suite developed at the University of Texas that measures the quality of service and latency of data transfers using the DeSiDeRaTa middleware service. This suite is scalable, adaptable and configurable, but was not built to compare multiple middleware solutions.

There are many options for complexity and size of applications, there are cases where the real world environment and networks are large and complex, but there are also realistic target environments on closed simple networks without many hops of potential interference. In some scenarios, such as with the GeoSurv UAV II project, laboratory environments can adequately represent the real world context of how the application will run.

A network loader is a network traffic emulator or generator that simulates heavy network traffic so that stress tests can be performed. There are several existing tools that can accomplish this task: for instance the TrafficEmulator [28] or Paessler NetFlow Generator [29].
Figure 4 shows the difference between ideal networks in laboratory settings with real world situations. D. Bleichman [30] found that injecting a network load did not impact the sender’s ability to output messages but did limit the throughput of the message transfers and that the latency of message transfers was slowed for both UDP and TCP protocols but TCP was effected more for larger sized payloads.

Similarly, a disk loader is a tool that allows the processor on a given host to be loaded to simulate a heavy load or overloading situation on the host; Obrut [31] is an example of such a commercially available tool.
3.1.1 Measuring Latency

Latency between nodes in a network is the time between a request and the receipt of the request. The measurements used for latency vary greatly because it is not possible to determine exactly when the message is sent and when it is received. The DMSO developed a benchmark test for latency between Run Time Infrastructures: BmLatency [30][34], by their definition the value is half of the round trip message time. Round trip time is the time taken for the message-send request to be sent, until the receipt of the acknowledgment message in the sender. This approach however does not take into account the potential processing differences in the sending and receiving nodes. Burks et al [32] defined latency as the time between the sending of a message from the sender to the receipt of the message in the receiver, and use of this approach can be difficult because it requires that the sender and receiver clocks be synchronized.

3.1.2 Measuring Throughput

Throughput between nodes in a network is the measured value of messages sent from one node to another in a network in a given time period. The results are generally presented in bytes per second.

The DMSO has defined a throughput metric: BmThroughput that measures the time from the sender sending a batch of messages to the receiver having received them all [30][34]. In this case the DMSO does not use any round-trip calculations, thus there is a prerequisite requirement that all nodes have synchronized clocks.
3.1.3 Existing Benchmarking Reports

There are many benchmarking reports and published studies available, each with their own purpose and slant.

The leading companies that produce middleware solutions publish benchmarking results from the tests they perform internally. For example, the report by Real-Time Innovations on C++ DDS benchmarks on Linux [35] shows that, using reliable UDP messaging with ordered delivery, message latency is proportional to the message size. On the OpenSplice website, they present some benchmark results [36] comparing their solution to the OPRA DDS, but not to any other widely used DDS solutions. These test results can be useful when deciding what DDS to use, but they cannot be used to accurately determine or even estimate what the message latencies will be on other target solutions.

A study from Tuma and Buble [1] demonstrated that it is possible to create a generalized CORBA benchmarking suite to compare object request brokers without bias.

Vanderbilt University has produced many papers and comparisons of various middleware solutions; DDS is the main interest and is compared against CORBA and JMS. These studies can be considered impartial and may help system design select a middleware solution to meet their needs [37].
Chapter 4: Research Questions – Problem Statement

In this chapter, the state of the art research discussed in Chapter 3 is analyzed in terms of its limitations in section 4.1. The objective of the research is described in section 4.2. The new approach of the MiddleMeasure suite is discussed in section 4.3. Section 4.3 follows with a description of the scope of the thesis. Section 4.4 completes this chapter with the description of the contributions of the thesis.

4.1 Problem Analysis

Once an application has been developed it is possible to take performance measurements and to statistically determine the average response times and average amount of messages exchanged between the distributed components. This is used to determine whether or not an application will meet its performance requirements, but may leave the developers left with the challenge of reworking the solution if the requirements are not met; this is often difficult and time consuming, and therefore expensive. It would be more economical to determine if the end solution will satisfy the performance requirements earlier in the design and development phases of the application. Benchmarking during the design or prototyping phases helps to validate that the final solution has less risk in satisfying the performance requirements; or proving that the solution will not be satisfactory and should be re-evaluated before the development phase begins.

Application designers should not be saddled with the task of creating a benchmarking test suite when they begin a project, nor should they have to be subject matter experts capable of developing software to test different RTIs and middleware or communications
protocol. Using an open source solution that can be extended to include new operating systems and middleware solutions would speed up the RTI and middleware evaluation process. The metrics gathered from benchmark testing with such a solution could also be used to feed data into performance prediction software to further reduce the risk of software application development.

Benchmarking suites exist to measure software messaging latency and throughput but only produce results on specific hardware platforms. Users would have to run CORBA benchmarking, separate from DDS benchmarking and separate from TCP/UDP testing to get full spectrum results. Unless modified by the user, the results will be presented in a variety of formats and be hard to compare and difficult to analyze. Ideally, all the tests could be run using the same methods for measuring the latencies and throughputs. For example, all tests measuring the latency of DDS message transmission could either use a round trip or a non-round trip measurement approach, and all CORBA throughput tests could use the round-trip measurement approach with the same number of test iterations and message sizes. In an ideal solution, the results would be presented in a consistent format so that they are easily compared and analyzed.

There are currently no freely available tools that provide the ideal ability to use one application to generate benchmark tests and results across multiple platforms and using various middleware solutions. System designers are either saddled with the task of creating their own benchmarking tests or using data from comparable tests performed by other people. Currently, benchmarking data is readily available for very specific hardware and software versions, but it is often out of date or not relevant for the system being developed. It is therefore necessary to run benchmarking tests on any proposed RTI
to accurately compare the results. To accomplish this, it would be beneficial to have a set of repeatable tests that can be setup quickly and run on any RTI. The tests in the suite must be simple enough to be run on minimal footprint operating systems, but complete enough to produce repeatable and analyzable results. Furthermore, the tests should be extensible to be used on other platforms and operating systems.

Research into existing benchmarking suites did not reveal any comprehensive suites that test the latency and throughput of data transfers using DDS, CORBA and TCP/UDP tests. Also, the performance tests or test suites do not include a network and disk loader to simulate a variety of load situations.

In addition, there exists the problem of evaluating non-deterministic configurations. In order to produce reasonable results, the number of message transfers will have to be significantly high to produce a statistically significant result [38].

4.2 Statement of the Thesis

The ideal benchmarking solution described above can be approximated for latency and throughput measurements using a software suite made up of a set of tests that can be setup according to specified configuration parameters and that can be run across multiple platforms. The results from the tests are immediately comparable since the metrics are output in a common format.

The MiddleMeasure benchmarking suite has been developed to demonstrate the thesis and will be provided as an open source to be extended as required for future benchmarking needs.
4.3 MiddleMeasure

MiddleMeasure is a flexible application suite that will benchmark an RTI and quantify the latency and throughput for various data transfers. The MiddleMeasure benchmarking application suite was developed in order to provide small footprint tests that calculate the latency and throughput of message transfers between nodes in a network. In its current state, MiddleMeasure consists of simple tests written in C or C++ and a wrapper allowing the tests to be built for various operating systems. Furthermore, MiddleMeasure has the flexibility to switch between combinations of point to point data transfers using CORBA or TCP and multiple participant tests using DDS or UDP. Unlike the DynBench suite produced by the University of Texas the MiddleMeasure suite was designed to be used with various different middleware solutions. The details of the benchmarking suite are described in Chapter 5.

Experiments were performed on several operating systems to demonstrate the benchmarking suite in the context of the thesis. The experiments show the portability of the tests across several operating systems, including WinXP, Linux and INtime RTOS. The capabilities of the suite are shown through experiments involving two DDS solutions and one CORBA solution; demonstrating the portability of MiddleMeasure over different middleware solutions. The Real-Time Innovations’ NDDS is UDP-based whereas the Gallium InterCOM DDS is TCP-based; these two products serve as representative samples of DDSs. The CORBA solutions are all TCP-based. For the MiddleMeasure testing, the ACE TAO ORB [41] was used as the representative CORBA product. The details of the experiments are given in Chapter 6.
Since the development of the MiddleMeasure test suite, it was announced that the InterCOM DDS product would be moving to a UDP-based solution. This is a good example of how fluid the software development environments can be and how important it is for designers and developers to produce up to date and accurate benchmarking results instead of relying on testing performed in the past on other environments and with other tools.

Using the MiddleMeasure suite at the design and prototype phase will allow designers and developers to measure message latency accurately in terms of the time required for messages to be exchanged between nodes, and the throughput of messages that can be transferred per second. The tests included in the suite are repeatable and provide the ability to directly compare hardware and software combinations, thus allowing the designers to ensure they have selected the appropriate devices, operating system and middleware solutions to suit their needs prior to developing their application.

The MiddleMeasure benchmarking suite is intended to be used as a foundation for developing further performance tests for additional operating systems and middleware combinations. Reducing the risk in software development is consistently sought after and can be accomplished by determining early on which RTI is best suited for a particular solution. By providing concrete data inputs for performance modeling, the potential bottlenecks in the design can be found and mitigated early on in the development process; reducing the project’s risk.

In addition to reducing the risk of software development, the results can be used to provide inputs to simulation software such as for the GeoSurv II UAV Project. In his
thesis A Path Network Model for Predicting Performance of Loosely Couple DRE Systems in the GeoSurv II UAV Project, Zhihong Ke presents his simple Path Network Model that can be used to predict performance in a distributed real time embedded system (DRE) [12]. The accuracy of his model depends on knowledge of all parts of the overhead in the operating system kernel and of the middleware used. Ke notes that the environmental overhead from using a middleware solution are not measured easily and that in lieu of real data, estimates must be used. Using the simple latency tests in the MiddleMeasure suite, the best case times for transferring messages between network nodes can be measured and from there the middleware environmental overhead can be extrapolated.

Besides the time to exchange data, the other leading concern for distributed application developers is to determine when data loss may occur. Data loss can occur when messages do not reach the intended recipients to be processed. Intermittent loss of data can be due to network loading or disk loading. The MiddleMeasure suite includes two small applications to simulate these two scenarios. When running the latency and throughput tests the application developers are able to simulate the loading of the various network node disks and / or to simulate other traffic on the network. Analysis of the results of these tests is useful to determine at which points the messages may become dropped or slowed to a point that is no longer acceptable.

The MiddleMeasure suite permits systems’ engineers to accurately compare hardware and software combinations with minimal effort and will generate benchmark tests as required by the operator and produce qualitative results for the operator’s specific and current needs.
4.4 Contributions of the Thesis

The following contributions have been made through the research and analysis of the thesis:

- The research shows that it is possible to use the same small generic tests to compare the performance of UDP-based DDS solutions with the performance of TCP-based DDS solutions and to compare the latency and throughput performance of different RTIs using a single middleware solution. Furthermore, the tests can be readily adapted to compare new operating systems and middleware solutions.

- The research improves on the DynBench research from the University of Texas [26] by adding the ability to compare middleware solutions to a benchmarking suite.

The following practical contributions have been made by this thesis and the development of the MiddleMeasure test suite:

- The MiddleMeasure test suite is released as open source code to be extended and used as needed.

- The research provides a useful set of tests that can be used to measure the latency and throughput of message transfers on given hardware and software combinations.

- The research provides a baseline of test cases to build benchmarking tests for addition hardware and software combinations.

- The research provides methodologies for building benchmarking tests for multiple hardware and software combinations in a network.
• The research provides tools for generating test runs with parameters for operating system under test, data types and number of messages transferred.

4.5 Scope

The following constraints have been placed on the research with the goal of obtaining sufficient results in a reasonable timeframe:

All the tests are written in C / C++, the use of other programming languages is not considered at this time.

The networks under test are small 3 and 7 node networks, connected with Ethernet via a single switch. The tested network topology is representative of a class of smaller scale embedded real-time systems. For example, the nodes in the UAV aircraft are all connected to a single hub, physically located within a meter of each other, and there are no large "hops" onto unknown networks and/or unknown traffic competing for bandwidth. Large scale network testing or modeling is reserved for future research.

The test applications are single-threaded and run alone with the operating system on each node.

All of the test nodes used contain only one mono-code CPU. Multi-core and multi-threaded testing are reserved for future research.

The test suite is designed in such a way that further CORBA and DDS middleware solutions and/or different operating system support can be added in the future.
To validate the portability of the solution, the TCP latency test was produced and run on Tenasys' INtime RTOS extending the suite to include the other tests for INtime can be added in the future.
Chapter 5: Test Design, Theoretical Analysis and Predictions

MiddleMeasure was developed to show that it is possible to create a benchmarking suite that enables the architects of distributed system to obtain metrics about the latency and throughput of the selected RTIs and middleware. Ideally, the analysis of these metrics will reduce the risk of their project by providing a better understanding of the performance bottlenecks and restrictions in the design or prototype phase of the system’s development.

Section 5.1 describes the MiddleMeasure benchmarking suite design. The benchmarking suite is designed to run on multiple platforms and test different middleware technologies. The tests, the configuration of the network used to perform the tests, and the generation of tests are discussed in section 5.2. The expected results of the tests are discussed in section 5.3. The results of the testing are presented in Chapter 6.

5.1 MiddleMeasure Test Suite Design

The test suite is comprised of several components: the network loader application, the disk loader application, the automated test harness and the test executables that are launched on the target nodes by the launcher application.
As illustrated in Figure 5 the MiddleMeasure benchmarking suite is made up of several components: the tests to gather the desired performance metrics, the test generator tool, the test harness and the benchmark utilities. The tests are described in section 5.2.

The test generation tool is a command line application that outputs a file containing the list of tests to execute based on the input parameters.

The test harness is composed of a central control process that should be run on one host in the network and slave processes that should run on all other hosts in the network.
The control process directs the tests and issues instructions to various slave processes, as shown in Figure 6. The control process uses the instruction list, a text file output from the test generator, to command the slave process to run the desired tests, with specific parameters.
Figure 7: Slave Process Flow Diagram

The slave processes invoke the test applications in accordance with the instructions received and implement various test harness functions (run tests, terminate and skip test cases that take an excessive amount of time to execute), this is illustrated in Figure 7. Communication between the test harness processes is achieved through TCP/IP communication.

To setup network loading in the test environment, two nodes not involved in the test will be used to send streams of data to each other (i.e. continuously running the message latency test).

To setup disk loading in the test environment, the two slave node processes involved in the test will continuously perform I/O tasks on the hard disk while running the test.

The Generate Test Cases utility requires the following command line arguments:

1. Operating system used for the sender node(s)
2. Operating system used for the receiver node(s)
3. Number of Sender nodes
4. Number of Receiver nodes

The benchmarking test suite was designed to include tests for different operating systems. Initially the tests were created for Windows XP and Linux Red Hat operating systems; subsequently real-time tests were added for the INtime real-time operating system [25].

The test case generation software is flexible and can allow the developer to produce only a subset of the test cases required for their benchmark testing. The configuration header file also includes settings about which middleware solution is to be tested, what payload types to use and what quality of service settings to apply. Multiple tests and QoS options can be selected at once allowing all the tests to be generated and then run in one shot.

<table>
<thead>
<tr>
<th>Payload Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
</tr>
<tr>
<td>Octet</td>
</tr>
<tr>
<td>Char</td>
</tr>
<tr>
<td>Short</td>
</tr>
<tr>
<td>Float</td>
</tr>
<tr>
<td>Long</td>
</tr>
<tr>
<td>Struct</td>
</tr>
</tbody>
</table>

The configuration header file defines each of the payload options listed in Table 3 as a Boolean value that the user can set to true or false. The structure payload contains a long, a char array of size 8, and a float.
The available quality of service options for the DDS tests as presented in Table 4. The configuration header file contains Boolean definitions for each option and the user can set any combination of them to true in order to output the desired test instructions.

In order to produce an instruction list from the test generator tool, the user should first open the config.h file and select the middleware to test, test types to generate, loading conditions, data types, reliability, history and lifespan options. Only one middleware option can be selected at once, but all other options can be enabled without exclusion. For instance if disk loading and no disk loading are both enabled then tests will be generated for both scenarios. Next, the user must run the GenerateTestCases_win32.exe with the command line options that define the sender and receiver operating systems and the numbers of senders and receivers respectively.

A snippet of the output from the test generator tool is included in Appendix A.

The snippet of the instruction list represents one test that is to be performed using 4 slave nodes. In the Appendix A example, the Control Process sends the commands to the indicated slave nodes; ‘clean_all’ is sent to Slave 1 then the control process waits for an acknowledgement message, and sends the message to the other nodes. Once any
previously started tests have been halted on all the slave nodes the control node sends the commands to start the tests with the desired parameters. The names and location of the tests in the instruction list must match exactly the names and locations of the tests on the slave nodes. Prior to testing, all the tests were copied onto the slave nodes and placed in a subdirectory called ‘testApps’ in the directory where the Slave Process is running.

The commands for running the tests follow a format such as:

```
"run start /wait .\testApps\DDSBenchmark_PingMultiRTI.exe -o -b -h 1 -z 4 -i -n 3"
```

(from Appendix A).

This command directs the slave node to run the PingMultiRTI test using an octet payload (‘-o’), with best effort reliability (‘-b’), a history of one sample (‘-h 1’), a message size of 4 bytes (‘-z 4’), an infinite lifespan (‘-i’) and that the test is to be run with 3 pong receiver nodes (‘-n 3’). The format of the command line options are included in the constants.h file built into both the Ping and Pong applications; each of which are built to run with all possible permutations of the command line options; and it is the instruction list output from the test generator determine which tests are run in a given session. The ‘start /wait’ call in the command indicates that the slave process must wait until the process is finished prior to processing the next command; this is a Windows command line switch option.

Once the tests have completed the command line output is cleared from the control and slave nodes, then finally the logs are moved to a shared drive in a subdirectory named for the specific test case. The test names and locations as well as the move command are
defined in the Test Case Generator tool code and are operating system and middleware specific.

5.2 Test Descriptions

The follow subsections describe the tests included in the MiddleMeasure suite. The latency tests are run with one or multiple receivers, and the throughput tests are run with one or multiple senders. To ensure that the results are statistically relevant, the latency tests are run 20 times with 2000 iterations of each message sent during each run. After each of the 20 runs, the minimum, maximum and average latency of the 2000 message transfers are recorded. The number of iterations was chosen in an attempt to achieve statistically significant results; however no analysis has been done to date to determine what the correct number of iterations should be.

5.2.1 Single Receiver Latency Test:

This test uses a Ping and Pong mechanism to measure the time for a publisher (Ping) to send a message to an expectant subscriber (Pong) and receive the message acknowledgement back from the subscriber. The measured results correspond to the round trip latency described in section 3.1.

A single iteration of the latency scenario is described below:

1. Sender (i.e. Ping process) initiates test
2. Sender records current time in memory
3. Sender sends payload message
4. Receiver (i.e. Pong process) receives payload message

5. Receiver responds with acknowledgment message

6. Sender receives acknowledgment message

7. Sender records current time in memory

8. Steps 3 through 7 are repeated 2000 times

9. Sender calculates mean round-trip latency value and logs results

The test is run 20 times.

A corresponding sequence diagram is shown in Figure 8.

Figure 8: Single Latency Test
5.2.2 Multiple Receiver Latency Test

The Multiple Latency test is similar to the single version, but contains additional handshaking so that all the receivers can acknowledge receipt of the message.

A single iteration of the latency scenario is described below:

1. Receivers (i.e. Pong processes) initiate test
2. Receivers register with Sender (i.e. Ping process)
3. Sender records current time in memory
4. Sender sends payload messages to each Receiver
5. Receivers receive payload message
6. Receivers respond with acknowledgment message
7. Sender receives acknowledgment message from all Receivers
8. Steps 4 through 7 repeated 2000 times
9. Sender records current time in memory
10. Sender calculates mean round-trip latency value and logs results

The test is run 20 times.
5.2.3 Single Receiver Throughput Test

This test involves a single Sender application sending messages to a single Receiver application that measure the time to process a batch of messages.
A single iteration of the throughput scenario is described below:

1. Receiver initiates test
2. Receiver registers with Sender
3. Sender sends information message to Receiver containing number of messages to expect in the batch
4. Receiver responds with acknowledgment message
5. Sender sends start batch message
6. Receiver receives start batch message
7. Receiver records current time and expected payload information to log
8. Receiver responds with acknowledgment message
9. Sender received acknowledgment message
10. Sender sends payload messages to Receiver
11. Receiver receives payload message and increases message count
12. Receiver responds with acknowledgment message
13. Sender receives acknowledgment message from Receiver
14. Steps 10 through 11 repeated specified number of times
15. Sender sends end of batch message
16. Receiver received end of batch message
17. Receiver records timing and number of received messages to log
18. Receiver sends acknowledgement message
19. Sender receives acknowledgement message

The test is run 20 times.
5.2.4 Multiple Receiver Throughput Test

A single iteration of the multiple receiver throughput scenario is described below:

1. Sender starts with number of expected Receivers

2. Receivers register with Sender

3. Once the Sender has received message from each Receiver, Sender sends information message to Receivers containing number of messages to expect in the batch

4. Receivers respond with acknowledgment message
5. Once the Sender has received message from each Receiver, Sender sends start batch message

6. Receivers receive start batch message

7. Sender records current time and test parameters to log

8. Receivers responds with acknowledgment message

9. Sender receives acknowledgment message from all Receivers

10. Sender records current time and test parameters to log

11. Sender sends payload messages to Receivers

12. Receivers receive payload message

13. Receivers respond with acknowledgment message

14. Sender receives acknowledgment message from all Receivers, or times out waiting for an acknowledgement message

15. Steps 11 through 14 repeated specified number of times

16. Sender records timing information and number of acknowledged messages to log

17. Sender sends end of batch message

18. Receivers receive end of batch message

19. Receivers send acknowledgement message

20. Sender receives acknowledgement message from all Receivers

The test is run 20 times.

5.2.5 DDS Tests

Each of the DDS tests implements at least one publisher and at least one subscriber application. Both publisher and subscriber applications create an instance of a
DomainParticipantListener as a separate monitor class to view the connection status of the application. The monitor classes display all status changes.

Both publisher and subscriber applications create an instance of a DomainParticipantFactory and DomainParticipant to identify the application in the DDS Domain. The participants are created with the desired QoS settings from Table 4.

Both publisher and subscriber applications create Topics for the selected pre-defined message type from Table 3 (char, struct, long, etc.) and one for the utility message type. Utility messages are used to control the communications between the applications. Topics are created by calling the ‘create_topic’ function in the Participant, as defined by the OMG standard. The topic names are declared in constants files that are built into both the publisher and subscriber applications. The constant files also include the command line option definitions and message identifiers that are used to encode and decode the messages exchanged between applications.

Both publisher and subscriber applications create a Publisher, Subscriber, DataWriter andDataReader. For each application a DataReader and DataWriter instance is used to send and receive utility message by telling them to use the Utility Topic. A DataWriter is created for the selected Topic and the selected QoS settings by calling the ‘create_datawriter’ function in the Publisher, as defined by the OMG standard.

5.2.6 CORBA Tests

The latency and throughput tests for CORBA were adapted from the sample benchmarking tests included with the ACE TAO ORB.
The MiddleMeasure tests use the same IDL for the CORBA and DDS tests and outputs results in one format so that they can easily be compared in a spreadsheet application such as Excel.

The CORBA architecture is designed to support a client / server communications model. A client is developed to include the ORB libraries and the interface files that are produced from the processing of the IDL file(s). The client application triggers the functions in the interface files in order to communicate with the server.

A server is developed to include the ORB libraries and the implementation stubs that are produced from the processing of the IDL file(s). The stubs are completed in the server and include the code to process the messages from the client.

Establishing and maintaining the connection between the client and server is handled by the ORB instead of the application code.

5.3 Expected Results

The testing is intended to show that it is possible to create a useful benchmarking suite that can be used to validate the performance of RTIs with different middleware solutions. The envisaged usage of the MiddleMeasure tests can be broken down into several categories:

1. Tests that compare RTIs

2. Tests that compare different versions of the same middleware or of competing middleware products
3. Tests that compare different middleware solutions.

The expected results will be discussed in those same categories. In the research conducted with MiddleMeasure, the hardware used can be considered roughly equivalent, and does not play a large role in distinguishing the RTIs from one another. Also, the same network switch was used during the testing, so it is not considered to have an impact on the results.

5.3.1 Comparing Run Time Infrastructures

The Linux and Windows XP tests may reveal some small differences in latency and throughput performance but are expected to show roughly the same results.

When the same tests are run on INtime, it is expected that the deviation between runs will be much smaller and that the latency will be improved and more consistent and that the throughput will be more stable.

5.3.2 Comparing Middleware Products

The two DDS products under test differ significantly in their design; the Real-Time Innovations’ DDS is UDP based whereas the Gallium DDS is TCP based.

When comparing the two DDS solution it is anticipated that the Real-Time Innovations’ DDS would be faster in the multiple receiver tests with the reliability quality of service setting set to unreliable because it is UDP based as opposed to TCP based that inherently relies upon handshaking. On the other hand it is anticipated that the Gallium DDS should perform better when the reliability setting is set to reliable.
It is also anticipated that the difference between the metrics captured for Gallium DDS should be negligible between the reliable and unreliable QoS settings.

### 5.3.3 Comparing Middleware Solutions

The advantages of using a middleware messaging product are mostly in risk reduction and time savings when developing an application. They are not typically associated to improved performance over a simple TCP or UDP data transfers. It is anticipated that the latency results from the DDS and CORBA tests will show some overhead when compared with the TCP or UDP tests.

For the tests that compare DDS to CORBA it is expected that the DDS tests will perform better than the CORBA tests with multiple participants due to the use of the publish and subscribe methodology where data is shared via multicast transmissions to any number of subscribers. But it is anticipated that CORBA will perform better in the single sender and receiver tests where the client and server structure is better suited.

### 5.4 Run Time Infrastructure Extensions

The MiddleMeasure suite is designed to facilitate the addition of the RTIs on which to test. The test applications are written in C++ with most of the code being common across platforms, but dependent on a Timer class that is platform dependent.

When adding a new RTI, the developer would have to add a new Timer class to communicate with the operating systems timing calls. All the tests use the same Timer class, so the work would only have to be done once.
The developer would also have to create a new build configuration; either a makefile or a solution file for the new operating system.

The final step is to add the operating system option to the test generator tool options. The test generator tool was designed with an enumerated list of operating systems types, making it simple to add new operating system options. The test generator does have to be rebuilt when new operating systems are added.
Chapter 6: Testing with the MiddleMeasure suite

6.1 Test Environment

Two distinct network configurations were used in the development and testing of the benchmarking suite.

The first environment to support the benchmark testing consists of the isolated network with 3 nodes seen in Figure 11.

![Diagram of Configuration 1]

Figure 11: Configuration 1

The Control Node controls the execution of the tests.
The Sender Node sends payload messages and receives acknowledgements from the Receiver Nodes. The Sender Node calculates and logs the benchmark test results.

The Receiver Node receives payload messages from the Sender the responds with acknowledgement messages.

The hardware used to support the benchmark testing activity is described in Table 5.

<table>
<thead>
<tr>
<th>Node</th>
<th>Make</th>
<th>Model</th>
<th>CPU</th>
<th>Disk Capacity</th>
<th>RAM</th>
<th>Network Card</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control Node</td>
<td>Dell</td>
<td>Dimension 8400 Desktop</td>
<td>Intel Pentium 4 3.40 GHz</td>
<td>118 GB</td>
<td>1024 MB</td>
<td>Broadcom NetXtreme 57xx Gigabit Controller</td>
</tr>
<tr>
<td>Node 1</td>
<td>Dell</td>
<td>Optiplex 745 Desktop</td>
<td>Intel Pentium D 3.40 GHz</td>
<td>118 GB</td>
<td>1024 MB</td>
<td>Broadcom NetXtreme 57xx Gigabit Controller</td>
</tr>
<tr>
<td>Node 2</td>
<td>Dell</td>
<td>Optiplex 745 Desktop</td>
<td>Intel Pentium D 3.40 GHz</td>
<td>118 GB</td>
<td>1024 MB</td>
<td>Broadcom NetXtreme 57xx Gigabit Controller</td>
</tr>
</tbody>
</table>
The second environment to support the benchmark testing consists of an isolated Gigabit network with 7 nodes. A star topology was used as shown in Figure 12 with the nodes connected via a network switch. The Control Node was used to control the execution of automated tests via the automated test harness. Nodes 1 through 6 were used for the benchmark testing.

![Configuration 2](image)

The hardware used to support the benchmark testing activity is described in Table 6. Where possible, similar hardware was selected for the workstations to make the test
environment as homogeneous as possible. All workstations were equipped with at least two disk partitions to allow for them to be booted with Windows XP or Red Hat Enterprise Workstation 5.
### Table 6 Configuration 2 Hardware

<table>
<thead>
<tr>
<th>Node</th>
<th>Make</th>
<th>Model</th>
<th>CPU</th>
<th>Disk Capacity</th>
<th>RAM</th>
<th>Network Card</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control Node</td>
<td>Dell</td>
<td>Dimension 8400 Desktop</td>
<td>Intel Pentium 4 3.40 GHz</td>
<td>118 GB</td>
<td>1024 MB</td>
<td>Broadcom NetXtreme 57xx Gigabit Controller</td>
</tr>
<tr>
<td>Node 1 – 4</td>
<td>Dell</td>
<td>Optiplex 745 Desktop</td>
<td>Intel Pentium D 3.40 GHz</td>
<td>118 GB</td>
<td>1024 MB</td>
<td>Broadcom NetXtreme 57xx Gigabit Controller</td>
</tr>
<tr>
<td>Node 5</td>
<td>Dell</td>
<td>Inspiron 9100 Laptop</td>
<td>Intel Pentium 4 3.20 GHz</td>
<td>55 GB</td>
<td>1024 MB</td>
<td>Broadcom 440x 10/100 Integrated Controller</td>
</tr>
<tr>
<td>Node 6</td>
<td>Dell</td>
<td>Dimension 8250 Desktop</td>
<td>Intel Pentium 4 2.66 GHz</td>
<td>80 GB</td>
<td>1024 MB</td>
<td>Intel PRO/100M Network Connection</td>
</tr>
</tbody>
</table>
6.2 Tests performed

The following subsections provide details about the tests run from the MiddleMeasure test suite; the CORBA throughput test, DDS single and multiple participant latency tests, DDS single and multiple throughput tests, TCP latency and throughput tests for the INtime RTOS.

6.2.1 CORBA Tests

The CORBA tests were built for the ACE TAO ORB, the latency and throughput tests were adapted from the benchmarking tests included with the ORB.

The MiddleMeasure tests use the same IDL for the CORBA and DDS tests and output results in one format so that they can easily be compared in a spreadsheet application such as Excel. An example of the test output is included in Appendix B.
The ACE TAO tests showed results consistent with the expected results discussed in Chapter 5, as seen in Figure 13 and Figure 14.
Figure 13 shows the results of the ACE TAO CORBA solution latency testing when comparing reliable against best effort latency performance on the Windows platform. The graph shows that there is little difference between the reliable and best effort settings; this is the expected result since the ACE TAO ORB uses TCP/IP to transport the data.
Figure 14: Middleware Testing: ACE TAO Single Throughput Test Results

Figure 14 shows the ACE TAO throughput results on Windows. The throughput is mostly stable, with a little dip with packet sizes of 1024 bytes. The results presented are for a test performed with reliable settings, a history of 1 sample and transferring structure messages.

The tests were run twice, once with the DiskLoader running on the client and server and once without any disk loading. The graphs shows that for packets of size 512 and 4096 bytes the disk loading does slow negatively affect the throughput, but there is not much impact on other packet sizes. The expectation was that the messages sent with larger packet sizes would have all been negatively affected by the disk loading, so it is interesting that the results do not show a drop in throughput.
Other than running the DiskLoader no other changes were made to the tests or configuration of the network. The repeatable nature of the tests make this type of comparison possible.

It is also interesting to note that the throughput does drop on both graphs when the payload size reaches 1024 bytes, then begins to climb again. Bleichman [30] concluded that there are default buffer sizes and messages of 1024 bytes and larger may require data to be split into multiple packets and results in a performance hit. In fact the Windows operating system uses a registry entry \textit{FastSendDatagramThreshold} to overwrite the default datagram size of 1024 bytes, the nodes under test do not have this registry entry. The same behaviour is seen in all the following figures for the Windows operating system. The drop is throughput performance is mirrored by an increased latency at the 1024 bytes packet size, this is understandable as transferring more messages will take more time.
The graph in Figure 15 shows the difference in throughput performance between the DDS solutions under test, and the CORBA solution on the Windows platform, the quality of service settings were set to Best Effort, Keep Last and Infinite. As expected, the DDS solutions have better throughput performance for larger message sizes as they are designed to exchange larger amounts of data more effectively between nodes.
Figure 16: Middleware Testing: Single Participant Latency Results (Best Effort)

The graph in Figure 16 shows that the Real Time Innovations NDDS solution has the best latency performance with the QoS settings set to Best Effort, Keep Last and Infinite. The Gallium InterCOM DDS is second best since it maintains a lower latency for larger payload messages than the ACE TAO ORB solution. It can be noted that at the higher payload sizes the two DDS solutions have converging latencies. Repeated the test with larger size messages may show that the InterCOM DDS surpasses the NDDS latency performance.
The graph in Figure 17 shows that the Real Time Innovations NDDS solution has the best latency performance with the QoS settings set to Reliable, Keep Last and Infinite for payloads under 2048 bytes. The results show that the Gallium InterCOM DDS performs better for the largest payload size.

### 6.2.2 DDS Tests

The MiddleMeasure suite was used to run a variety of DDS tests. The combinations of settings are described in Table 8.
<table>
<thead>
<tr>
<th>Description</th>
<th>Platform Configurations (Publisher / Subscriber)</th>
<th>Payload Types</th>
<th>Qos Settings</th>
<th>Payload Sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Participant Round-trip Latency</td>
<td>Windows XP/ Windows XP, Linux / Linux, Linux / Windows XP</td>
<td>Octet, Character, Short, Long, Float, Structure</td>
<td>Reliability = BEST_EFFORT, Reliability = RELIABLE, History = KEEP_ALL, History = KEEP_LAST, Lifespan = INFINITE</td>
<td>$2^2$ to $2^{14}$</td>
</tr>
</tbody>
</table>

| Multiple Participant Round-trip Latency | Windows XP/ Windows XP, Linux / Linux, Linux / Windows XP | Octet, Character, Short, Long, Float, Structure | Reliability = BEST_EFFORT, Reliability = RELIABLE, History = KEEP_ALL, History = KEEP_LAST, Lifespan = FINITE, Lifespan = INFINITE | $2^2$ to $2^{14}$ |

1 A value of 10 sec. was used for the finite lifespan value.
<table>
<thead>
<tr>
<th>Single Participant Throughput</th>
<th>Windows XP/Windows XP, Linux / Linux, Linux / Windows XP</th>
<th>Lifespan = FINITE, Lifespan = INFINITE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Octet, Character, Short, Long, Float, Structure</td>
<td>Reliability = BEST_EFFORT, Reliability = RELIABLE, History = KEEP_ALL, History = KEEP_LAST, Lifespan = FINITE, Lifespan = INFINITE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2^2 to 2^{14}</td>
</tr>
</tbody>
</table>

The payload types did not have much of an effect on the latency performance.
Figure 18: Middleware Testing: Latency Test Results with Varying Payload Types

Figure 18 shows that none of the payload type options from Table 3 can be thought of as more or less performant than the others. For this reason all types are considered equivalent for the other throughput and latency tests performed.

The RTI comparison tests as seen in Figure 19 using Gallium’s InterCOM DDS produced the expected results; the Linux and Windows RTIs show very similar behaviour.
Figure 19 shows the expected consistent throughput performance of the InterCOM DDS when run in the Linux to Linux, Linux to Windows and Windows to Windows configurations. The graph shows the results of the test run without network or disk loading, with the Quality of Service settings set to Reliable, Keep All and Finite. The throughput results indicate that an RTI that includes Linux performs better than one with Windows.
A comparison of the Gallium’s InterCOM DDS to RTI’s NDDS on Windows shows that the InterCOM DDS performs better than the NDDS with larger payloads with fewer publishers, as shown in Figure 20. The tests were run with one subscriber and the QoS settings set to Reliable, Keep All and Infinite. This was the expected behaviour described in Chapter 5 for a test with Reliable QoS settings because of the use of TCP in InterCOM DDS and UDP in NDDS. In both the three-publisher and five-publisher test cases the InterCOM DDS has higher throughput than the NDDS; it would be interesting to test the same scenario on a network with additional nodes to see at which point the UDP solution becomes more effective than the TCP solution.
Figure 21: Middleware Testing: Throughput Test Results (Reliable)
Figure 21 compares the Real-Time Innovations’ DDS to Gallium’s InterCOM DDS in throughout tests where the number of participants ranges from 1 through to 5 and using increasing payloads for each test from 4 bytes to 4096 bytes. Figure 22 shows the same data presented in separate graphs, each with a different number of subscribers. In the figures, the tests were run with the QoS settings set to Reliable, Keep All and Finite.
lifespan. The results show that the InterCOM DDS produces higher throughput metrics than the RTI DDS when the number of participants is small, but the gap is reduced as the number increases. The results are as expected as the TCP-based solution should allow faster throughput in point to point communications, whereas the UDP-based solution uses multicasting as is better suited for sending data to many receivers.

Figure 23: Middleware Testing: Throughput Test Results (Best Effort)

Figure 23 shows the output of the same tests run with the QoS settings set to best effort, keep all and finite lifespan. The outputs from both tests are almost identical as seen in Figure 21. This is a deviation from the expected result; which expected the InterCOM DDS to perform better with Reliable QoS.
The graph in Figure 24 compares the latency performance of Gallium’s InterCOM DDS on different RTIs. The results in Figure 24 show that, as seen in the throughput tests (Figures 15 and 16) the RTIs running with Linux perform better than those running Windows XP, especially with smaller payload sizes. The tests were conducted with 3 subscribers and with the varying payloads from 4 bytes to 4096 bytes with QoS settings set to Best Effort, Keep All and Finite. The results were similar for the other QoS permutations.

6.2.3 TCP/UDP Tests:

The TCP and UDP tests are created as baselines for comparisons for the DDS and CORBA middleware options.
<table>
<thead>
<tr>
<th>Description</th>
<th>Platform Configurations (Publisher / Subscriber)</th>
<th>Payload Types</th>
<th>Payload Sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency test</td>
<td>Configuration 1: INTIME / INTIME (TCP only)</td>
<td>Octet, Character, Short, Long, Float, Structure</td>
<td>$2^2$ to $2^{14}$</td>
</tr>
<tr>
<td></td>
<td>Configuration 2: Windows XP/ Windows XP, Linux / Linux, Linux / Windows XP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Throughput test</td>
<td>Configuration 1: INTIME / INTIME (TCP only)</td>
<td>Octet, Character, Short, Long, Float, Structure</td>
<td>$2^2$ to $2^{14}$</td>
</tr>
<tr>
<td></td>
<td>Configuration 2: Windows XP/ Windows XP, Linux / Linux, Linux / Windows XP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The TCP and UDP test are basic ping and pong tests that use the same concepts as the CORBA tests described previously.
The data presented in the Figure 25 graph are the raw latency measurements in microseconds for 200 message transfers of 1024 bytes between two hosts; the sender running on WinXP and the receiver running on the INtime kernel. This presentation of the results was used in order to demonstrate the fluctuations in the metrics collected, as opposed to showing only the average of the results. The results were not as expected; the average and maximum latency values were significantly higher and the maximum values were less stable than anticipated. It would seem that the use of WinXP as the operating system for the sender is adding the jitter and additional overhead to the latency.
The graph in Figure 26 shows the raw latency information in microseconds for 200 message transfers of 1024 bytes between two hosts each running the INtime kernel. The results from the testing INtime kernel to another also display some spikes in the measurements, although not to the same extent as the lack of stability as in the previous figure, Figure 25. In addition, the results of this test show that the maximum and average latency values are significantly lower than in the cross-platform test, indicating that the high values may have in fact been due to the use of WinXP as the sender nodes operating system.

The TCP tests were adapted to run on the INtime system and were tested on the Configuration 1 environment. It is unclear whether the results are a product of incorrect testing or inherent from the solution.
Figure 27 shows that the UDP throughput testing produced unexpected results when run with single subscribers; the UDP throughput was expected to be greater than the TCP throughput. Other TCP and UDP tests have shown the same results. In the G. Xylomenos and G. Polyzos [39] hypothesize that transmit buffer overruns cause a reset in the interface controllers with UDP messages, and that due to the flow control mechanisms in the TCP protocol it is not subject to these problems. Another possible explanation for TCP performing better than UDP is that TCP uses the Nagle Algorithm [40] to bundle small TCP packets into one larger message. On Windows XP the maximum segment size (MSS) is 64KB, so as many messages as will fit will be bundled together. UDP on the other hand puts each message out on the bus as soon as one is received for transmission.
6.3 Challenges along the way

As with all research projects there were some challenges encountered and tangents taken along the way. Three main obstacles had to be overcome: the first was the lack of real-time DDS products to test against; the second was a tangent to investigate the use of the INtime OS and the third was the learning curve and lack of previous experience with WinCE.

6.3.1 Real-time DDS products

The focus at the beginning of the project was on DDS middleware solutions and evaluating their suitability for mission critical application development. Although there exist a number of commercial and several open source DDS products, they were not always available for real-time operating systems.

The tests to compare the RTI’s NDDS and Gallium’s InterCOM DDS had been developed and run on Linux and Windows and the next logical step seemed to be running the same comparison on WinCE. Unfortunately, InterCOM DDS did not have WinCE support. After spending some time adding in the support, it became clear that developing the real time version of the product was well outside of the scope of the project.

There exists a real time version of RTI’s NDDS for VxWorks and WinCE and in January 2009 PrismTech announced its open source version of a real time DDS solution for VxWorks that is now available. VxWorks was not included in the MiddleMeasure suite, but it could be added as part of future research to test the latency and throughput.
behaviour of these real-time DDS solutions. Unfortunately, InterCOM DDS does not yet have real time platform support.

6.3.2 WINCE development

Window CE is a commonly used real-time embedded operating system. But its use requires the development and building of the operating system before the application development can even begin. More effort than anticipated was required to build a version of the operating system that could be used to run the MiddleMeasure tests. The time required was extended due to the fact that the tests for the suite were originally written in C++ with the use of some advanced libraries; that are not necessarily available in the small footprint of WinCE.

In hindsight, it would have been more efficient to develop the tests in the reverse order. The WinCE and INtime tests had to be developed without the use of any advance C++ functions or libraries. These tests could have been used directly to produce the CORBA and DDS tests, whereas the reverse is not true.
Chapter 7: Conclusions and Summary of Contributions

The thesis is motivated by the ever present desire to reduce the risk in software development. In most cases it is only possible to judge the performance of a system once it is complete. Various methods are used to foresee the potential problems in an application; prototyping, performance modeling and of course leveraging past experience. The MiddleMeasure test suite was developed to show that it is possible to measure the latency and throughput performance of an RTI and middleware combination.

By using the test suite in the early phases of development, designers can determine what hardware, operating system and optional middleware will best suit the needs of the solution they are developing.

The conclusions of the research are discussed in Section 7.1 and suggestions for future research are highlighted in Section 7.2.

7.1 Conclusions

In this research, the MiddleMeasure test suite was developed as a means to reduce the risk of selecting a hardware and operating system combination and middleware for a software application. First and foremost, MiddleMeasure provides tests and an infrastructure for comparing the latency and throughput of RTIs. Secondly, the suite also provides tests and an infrastructure for comparing different versions of the same middleware products, and for comparing competing middleware products; such as Real-Time Innovation's NDDS with Gallium's InterCOM DDS. Third, the MiddleMeasure
suite currently provides tests for measuring the latency and throughput performance of
different middleware products; such as comparing the DDS middleware solutions with
the ACE TAO CORBA solution.

The test suite was used to compare latency and throughput of DDS middleware solutions
on Linux and Windows. The results in Figure 19 show that the throughput of message
transfers is slightly better on Linux than Windows XP. And the results from Figure 24
show that the InterCOM DDS latency is the same on Linux and Windows XP. The same
type of analysis can be done on any Linux and Windows XP RTIs using the same DDS or
CORBA middleware solutions by simply running the tests on the desired RTI.

The MiddleMeasure test suite was also used to evaluate those RTIs with two DDS
solutions and a CORBA middleware solution. The results in Figure 15 show that the
InterCOM DDS solution has the greatest throughput when the tests are run with few
subscribers, and that both DDS solutions perform better than the CORBA solution.
Figure 21 shows that when the number of participant nodes in the tests are increased to
just 5 nodes, the RTI NDDS solution has minimized the difference between its
throughput results and those from the Gallium InterCOM DDS. System designers should
be able to use these combinations of tests to determine which middleware solution will
best suit their needs.

The research produced a reusable test methodology and set of tools: tests, test generation,
network loader and disk loader required to implement the methodology. The experiments
were run on two different network configurations, proving that MiddleMeasure is
scalable to different sized networks. The tests using INtime demonstrate that the MiddleMeasure solution can be extended to include different operating systems.

Based on the results of the test cases, it is possible to use the tests and tools provided in the MiddleMeasure test suite to run tests on different RTIs and with different middleware solutions and to evaluate the performance of the basic message exchange in the network.

The conclusions drawn from the research are:

1. It is possible to use one benchmarking suite to evaluate the latency and throughout performance of different RTIs and to measure the performance of different middleware solutions on one or more RTIs.

2. One benchmarking suite can be extended to include latency and benchmark testing for additional hardware and operating systems combinations.

3. The MiddleMeasure benchmarking suite is an improved solution over the DynBench suite from the University of Texas as it allows the middleware options to be evaluated in addition the throughput and latency performance of the RTIs.

### 7.2 Suggestions for Future Research

The research completed thus far has proved that the concept of the MiddleMeasure suite allows performance comparisons to be completed simply and without bias.

The product of the research: the MiddleMeasure test suite, network loader and disk loader applications can be used to further develop other research projects that are on-going at Carleton University such as the GeoSurv II UAV project and the Path Network modeling prediction research.
The test suite can be extended to include tests for other operating systems such as vxWorks, LynxOS or QNX RTOS, and also extended to include tests for other DDS and CORBA solutions. Now that the real time versions of the RTI’s NDDS and PrismTech’s OpenSplice DDS are available for VxWorks would be interesting to evaluate their latency and throughput performance on that platform. To date the testing of Real Time operating systems has not included the middleware components. It is recommended that future work be done to incorporate the existing DDS solutions that were made to run on real time platforms such as the Real-Time Innovations’ DDS. It is recommended to begin with the simpler C-based tests created for WinCE and INtime RTOS testing as opposed to the C++ based tests that depend on some of the larger C++ libraries; that may not be required for small footprint applications.

Thorough testing with WinCE and INtime and potentially other real time operating systems could be performed. The results should show improved stability and speed when compared to the Linux and Windows XP tests.

Although the disk loader and network loader tools were created and are included in the MiddleMeasure test suite, the comparative RTI and middleware testing was not performed under different load conditions. Future research could include running RTI comparison tests with network and disk loading enabled to determine how the latency and throughput performance is affected. In addition the DDS comparison testing results could be enhanced by determining at what point the network loading and/or disk loading begin to affect the latency and throughput performance of the different solutions.
In addition to providing DDS versus CORBA middleware comparable results, it would be useful to produce some new tests that combine the use of both middleware concepts, since many large scale systems can benefit from having both a publish and subscribe communications model and a client and server communications model. Other research has shown that DDS is well suited to tactical mission systems for distributing large quantities of data [41] however personal experience indicates that there is often a need to include point-to-point communications as well, and CORBA is best suited for this type of communication. RTI promotes the complimentary use of the two middleware protocols [42].

The MiddleMeasure suite can be extended to include the IPC [42] and used to measure the latency and throughput performance of the GeoSurv II UAV RTI so that the resulting metrics can be analyzed and used to provide real values for the response times in the UAV System and be compared to the hypothesized values. In Ze's thesis he concludes that it is not possible to provide the actual worst case latency values, the MiddleMeasure suite network and disk loading tools can be used to provide a more accurate value.
References:


[36] Vanderbilt University DDS webpage, www.dre.vanderbilt.edu/DDS, last accessed on December 27th 2010

89


[42] ACE TAO ORB, http://www.theaceorb.com/, last accessed on December 27th 2010

[43] Inter-Process Communication, Carnegie Mellon Institute, http://www.ri.cmu.edu/research_project_detail.html?project_id=394&menu_id=261 , last accessed on December 27th 2010
Appendix A: Sample instruction list

Snippet of the instruction list output from the Test Generator tool.

/// Test Case RTI 493 ///
case_start RTI_493

slave 1 clean_all
await_completion 1
slave 2 clean_all
await_completion 2
slave 3 clean_all
await_completion 3
slave 4 clean_all
await_completion 4
slave 5 clean_all
await_completion 5
slave 6 clean_all
await_completion 6

slave 6 run start \testApps\DDSBenchmark_NetworkLoaderSender.exe -z 1024
await_completion 6

slave 5 run start \testApps\DDSBenchmark_NetworkLoaderResponder.exe


await_completion 5

slave 2 run start \testApps\DDSBenchmark_PongMultiRTI.exe -o -b -h 1
await_completion 2

slave 3 run start \testApps\DDSBenchmark_PongMultiRTI.exe -o -b -h 1
await_completion 3

slave 4 run start \testApps\DDSBenchmark_PongMultiRTI.exe -o -b -h 1
await_completion 4

slave 1 run start /wait \testApps\DDSBenchmark_PingMultiRTI.exe -o -b -h 1 -z 4 -i -n 3
await_completion 1

slave 2 clean_all
await_completion 2

slave 3 clean_all
await_completion 3

slave 4 clean_all
await_completion 4

slave 2 run start \testApps\DDSBenchmark_PongMultiRTI.exe -o -b -h 1
await_completion 2

slave 3 run start \testApps\DDSBenchmark_PongMultiRTI.exe -o -b -h 1
await_completion 3

slave 4 run start \testApps\DDSBenchmark_PongMultiRTI.exe -o -b -h 1
await_completion 4

92
slave 1 run start /wait \testApps\DDSBenchmark_PingMultiRTI.exe -o -b -h 1 -z 8 -i -n 3

await_completion 1

slave 2 clean_all

await_completion 2

slave 3 clean_all

await_completion 3

slave 4 clean_all

await_completion 4

slave 2 run start \testApps\DDSBenchmark_PongMultiRTI.exe -o -b -h 1

await_completion 2

slave 3 run start \testApps\DDSBenchmark_PongMultiRTI.exe -o -b -h 1

await_completion 3

slave 4 run start \testApps\DDSBenchmark_PongMultiRTI.exe -o -b -h 1

await_completion 4

slave 1 run start /wait \testApps\DDSBenchmark_PingMultiRTI.exe -o -b -h 1 -z 16 -i -n 3

await_completion 1

slave 2 clean_all

await_completion 2

slave 3 clean_all

await_completion 3

slave 4 clean_all
await_completion 4
slave 2 run start \testApps\DDSBenchmark_PongMultiRTI.exe -o -b -h 1
await_completion 2
slave 3 run start \testApps\DDSBenchmark_PongMultiRTI.exe -o -b -h 1
await_completion 3
slave 4 run start \testApps\DDSBenchmark_PongMultiRTI.exe -o -b -h 1
await_completion 4
slave 1 run start /wait \testApps\DDSBenchmark_PingMultiRTI.exe -o -b -h 1 -z 32 -i -n 3
await_completion 1
slave 2 clean_all
await_completion 2
slave 3 clean_all
await_completion 3
slave 4 clean_all
await_completion 4
slave 2 run start \testApps\DDSBenchmark_PongMultiRTI.exe -o -b -h 1
await_completion 2
slave 3 run start \testApps\DDSBenchmark_PongMultiRTI.exe -o -b -h 1
await_completion 3
slave 4 run start \testApps\DDSBenchmark_PongMultiRTI.exe -o -b -h 1
await_completion 4
slave 1 run start /wait .\testApps\DDSBenchmark_PingMultiRTI.exe -o -b -h 1 -z 64 -i -n 3
await_completion 1
slave 2 clean_all
await_completion 2
slave 3 clean_all
await_completion 3
slave 4 clean_all
await_completion 4
slave 2 run start .\testApps\DDSBenchmark_PongMultiRTI.exe -o -b -h 1
await_completion 2
slave 3 run start .\testApps\DDSBenchmark_PongMultiRTI.exe -o -b -h 1
await_completion 3
slave 4 run start .\testApps\DDSBenchmark_PongMultiRTI.exe -o -b -h 1
await_completion 4
slave 1 run start /wait .\testApps\DDSBenchmark_PingMultiRTI.exe -o -b -h 1 -z 128 -i -n 3
await_completion 1
slave 2 clean_all
await_completion 2
slave 3 clean_all
await_completion 3
slave 4 clean_all

95
await_completion 4
slave 2 run start \testApps\DDSBenchmark_PongMultiRTI.exe -o -b -h 1
await_completion 2
slave 3 run start \testApps\DDSBenchmark_PongMultiRTI.exe -o -b -h 1
await_completion 3
slave 4 run start \testApps\DDSBenchmark_PongMultiRTI.exe -o -b -h 1
await_completion 4
slave 1 run start \wait \testApps\DDSBenchmark_PingMultiRTI.exe -o -b -h 1 -z 256 -i -n 3
await_completion 1
slave 2 clean_all
await_completion 2
slave 3 clean_all
await_completion 3
slave 4 clean_all
await_completion 4
slave 2 run start \testApps\DDSBenchmark_PongMultiRTI.exe -o -b -h 1
await_completion 2
slave 3 run start \testApps\DDSBenchmark_PongMultiRTI.exe -o -b -h 1
await_completion 3
slave 4 run start \testApps\DDSBenchmark_PongMultiRTI.exe -o -b -h 1
await_completion 4
slave 1 run start /wait \testApps\DDSBenchmark_PingMultiRTI.exe -o -b -h 1 -z 512 -i -n 3
await_completion 1
slave 2 clean_all
await_completion 2
slave 3 clean_all
await_completion 3
slave 4 clean_all
await_completion 4
slave 2 run start \testApps\DDSBenchmark_PongMultiRTI.exe -o -b -h 1
await_completion 2
slave 3 run start \testApps\DDSBenchmark_PongMultiRTI.exe -o -b -h 1
await_completion 3
slave 4 run start \testApps\DDSBenchmark_PongMultiRTI.exe -o -b -h 1
await_completion 4
slave 1 run start /wait \testApps\DDSBenchmark_PingMultiRTI.exe -o -b -h 1 -z 1024 -i -n 3
await_completion 1
slave 2 clean_all
await_completion 2
slave 3 clean_all
await_completion 3
slave 4 clean_all

97
await_completion 4
slave 2 run start \testApps\DDSBenchmark_PongMultiRTI.exe -o -b -h 1
await_completion 2
slave 3 run start \testApps\DDSBenchmark_PongMultiRTI.exe -o -b -h 1
await_completion 3
slave 4 run start \testApps\DDSBenchmark_PongMultiRTI.exe -o -b -h 1
await_completion 4
slave 1 run start /wait \testApps\DDSBenchmark_PingMultiRTI.exe -o -b -h 1 -z 2048 -i -n 3
await_completion 1
slave 2 clean_all
await_completion 2
slave 3 clean_all
await_completion 3
slave 4 clean_all
await_completion 4
slave 2 run start \testApps\DDSBenchmark_PongMultiRTI.exe -o -b -h 1
await_completion 2
slave 3 run start \testApps\DDSBenchmark_PongMultiRTI.exe -o -b -h 1
await_completion 3
slave 4 run start \testApps\DDSBenchmark_PongMultiRTI.exe -o -b -h 1
await_completion 4
slave 1 run start /wait .\testApps\DDSBenchmark_PingMultiRTI.exe -o -b -h 1 -z 4096 -i -n 3
  await_completion 1
slave 2 clean_all
  await_completion 2
slave 3 clean_all
  await_completion 3
slave 4 clean_all
  await_completion 4

  clear_completion_messages
  case_end

  /// Move logs
slave 1 run mkdir RTI_493
  await_completion 1
slave 1 run move /Y Log* RTI_493
  await_completion 1

  clear_completion_messages
Appendix B: Sample test results:

MessageThroughputTestTAO no disk loading:
LogFile_Throughput_Type_u_Reliability_b_History_l_Size_64_LifeSpan_l.txt

<table>
<thead>
<tr>
<th>Throughput Timing - All times are in microseconds.</th>
<th>Variance (us^2)</th>
<th>Net Variance (us^2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>Min</td>
<td>Max</td>
</tr>
<tr>
<td>121 612</td>
<td>53 9574</td>
<td>150623</td>
</tr>
<tr>
<td>81 2726</td>
<td>53 9912</td>
<td>446</td>
</tr>
<tr>
<td>79 1547</td>
<td>53.9799</td>
<td>1719</td>
</tr>
<tr>
<td>84 1062</td>
<td>53 5562</td>
<td>8954</td>
</tr>
<tr>
<td>91 639</td>
<td>53 1099</td>
<td>58666</td>
</tr>
<tr>
<td>77 8218</td>
<td>54 0061</td>
<td>614</td>
</tr>
<tr>
<td>79.6693</td>
<td>53 9837</td>
<td>797</td>
</tr>
<tr>
<td>78 7351</td>
<td>53 8112</td>
<td>506</td>
</tr>
<tr>
<td>81 4009</td>
<td>53 9387</td>
<td>589</td>
</tr>
<tr>
<td>97 4417</td>
<td>53 8412</td>
<td>65707</td>
</tr>
<tr>
<td>79.2905</td>
<td>53 9499</td>
<td>468</td>
</tr>
<tr>
<td>110 808</td>
<td>53 8824</td>
<td>65147</td>
</tr>
<tr>
<td>88 2845</td>
<td>53 8712</td>
<td>31458</td>
</tr>
<tr>
<td>80 358</td>
<td>53.9799</td>
<td>657</td>
</tr>
<tr>
<td>89 4823</td>
<td>54.0361</td>
<td>45725</td>
</tr>
<tr>
<td>78 4085</td>
<td>54.0286</td>
<td>4845</td>
</tr>
<tr>
<td>78 4093</td>
<td>53 6387</td>
<td>829</td>
</tr>
<tr>
<td>83.7221</td>
<td>54 1299</td>
<td>15273</td>
</tr>
<tr>
<td>124 516</td>
<td>53 8224</td>
<td>116596</td>
</tr>
<tr>
<td>89 6623</td>
<td>53 9012</td>
<td>23398</td>
</tr>
</tbody>
</table>