COMPARATIVE EVALUATION OF COMMAND DISTRIBUTION VIA DDS AND CORBA IN A SOFTWARE REFERENCE ARCHITECTURE

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

MUSTAFA BERK DURAN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

APRIL 2014

Approval of the thesis:

COMPARATIVE EVALUATION OF COMMAND DISTRIBUTION VIA DDS AND CORBA IN A SOFTWARE REFERENCE ARCHITECTURE

submitted by **MUSTAFA BERK DURAN** in partial fulfillment of the requirements for the degree of **Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Gönül Turhan Sayan
Head of Department, **Electrical and Electronics Eng.** _____

Prof. Dr. Semih Bilgen
Supervisor, **Electrical and Electronics Eng. Dept.** _____

Examining Committee Members:

Assoc. Prof. Dr. Cüneyt F. Bazlamaçcı
Electrical and Electronics Engineering Dept., METU _____

Prof. Dr. Semih Bilgen
Electrical and Electronics Engineering Dept., METU _____

Prof. Dr. Ali H. Doğru
Computer Engineering Dept., METU _____

Assoc. Prof. Dr. Ece Güran Schmidt
Electrical and Electronics Engineering Dept., METU _____

Barış İyidir, M.Sc.
Software Engineering Department, ASELSAN _____

Date: 29.04.2014

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: MUSTAFA BERK DURAN

Signature :

ABSTRACT

COMPARATIVE EVALUATION OF COMMAND DISTRIBUTION VIA DDS AND CORBA IN A SOFTWARE REFERENCE ARCHITECTURE

Duran, Mustafa Berk

M.S., Department of Electrical and Electronics Engineering

Supervisor : Prof. Dr. Semih Bilgen

April 2014, 49 pages

Communication between modules in distributed system architectures plays a crucial role in proper system operation. Therefore, selection of the method for the communication of software running on different platforms becomes important. Two of the alternatives for data distribution are the Common Object Request Broker Architecture (CORBA) and Data-Distribution Service (DDS). In this study, effects of the selection on the overall software quality and performance are investigated for real-time embedded systems developed in conformance with a software reference architecture. For the purposes of this study, a benchmark project was prepared according to the application domain requirements and software reference architecture of the software engineering department. Four test cases were designed to animate possible scenarios that the system might come across. Test cases employ different numbers of user interfaces as peers, either as command sources or as display panels used in the project. Methods are evaluated in terms of software quality and performance metrics. Software quality metrics

are collected under coupling and complexity measurements whereas utilization and latency are measured for evaluation of software performance.

Keywords: CORBA, DDS, Middleware, Command distribution, Data distribution, Software reference architecture, Software performance, Software quality.

ÖZ

YAZILIM REFERANS MİMARİSİNDE KOMUT DAĞITIMI İÇİN DDS VE CORBA ALTYAPILARININ KARŞILAŞTIRMALI DEĞERLENDİRMESİ

Duran, Mustafa Berk
Yüksek Lisans, Elektrik Elektronik Mühendisliği Bölümü
Tez Yöneticisi : Prof. Dr. Semih Bilgen

Nisan 2014 , 49 sayfa

Dağıtık sistem mimarilerinde modüller arasındaki iletişim sistemin düzgün çalışmasında önemli rol oynar. Bu nedenle, iletişim yönteminin seçimi farklı platformlarda çalışan yazılımlar için önemlidir. Komut dağıtımı için iki alternatif; “Common Request Broker Architecture" (CORBA) ve “Data-Distribution Service" (DDS)tir. Bu çalışmada, yapılan seçimin yazılımın kalite ve performansına etkileri yazılım referans mimarisi kapsamında geliştirilen gerçek zamanlı gömülü sistemler için incelenmiştir. Bu çalışmanın amaçları doğrultusunda, uygulama alanı gereksinimleri ve bölümün yazılım referans mimarisine göre bir karşılaştırma projesi hazırlanmıştır. Sistemde rastlanması olası senaryoları canlandırmak için dört test senaryosu tasarlanmıştır. Test senaryolarında değişik sayılardaki kullanıcı arayüzlerine, komut kaynakları veya projede kullanılan ekran panelleri olarak yer verilmiştir. Yöntemler, yazılım kalite ve performans ölçümleri başlıkları altında değerlendirilmiştir. Yazılım kalite ölçütleri bağlaşım ve karma-

şıklık ölçümleri ile toplanırken, yazılım performansını ölçmek için kullanım ve gecikme süresi ölçütleri toplanmıştır.

Anahtar Kelimeler: CORBA, DDS, Özel yazılım, Komut dağıtımı, Veri dağıtımı, Yazılım referans mimarisi, Yazılım performansı, Yazılım kalitesi.

To my family

ACKNOWLEDGMENTS

I would like to express the deepest appreciation to Prof. Dr. Semih Bilgen for his support, understanding and guidance throughout the development of this thesis study. Without his supervision, patience and encouragement, this thesis would not have been completed.

I thank TÜBİTAK for its support throughout this thesis. I would also like to thank ASELSAN Inc. for the support and encouragement on academic studies.

I would like to thank Barış İyidir for his vision and ideas that he shared with me throughout this study. I am grateful to my team leader Evrim Kahraman and my colleague Sezgin Hayırlı for their tolerance during this study.

I really appreciate the support of my friends, who “hopefully" never resented me when I rejected their plans to hang out as I needed to “work on my thesis". With their love and understanding I managed to overcome the difficulties in the process.

Finally, I would like to express my special thanks to my parents Şeyda and Ufuk Duran and my fiancée Elçin Ergin for their love, trust, understanding and every kind of support not only throughout my thesis but also throughout my life.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

CBO	Coupling Between Objects
CBSE	Component Based Software Engineering
CCCC	C and C++ Code Counter
CORBA	Common Object Request Broker Architecture
COTS	Commercial-Off-The-Shelf Software
CPU	Central Processing Unit
DCPS	Data-Centric Publish Subscribe
DDS	Data Distribution Service
DLRL	Data Local Reconstruction Layer
EI	External Interface
FP	Function Point
GUI	Graphical User Interface
IDL	Interface Definition Language
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
MCN	McCabe Cyclomatic Number
MTTF	Mean Time to Failure
MTBF	Mean Time Between Failures
OMG	Object Management Group
ORB	Object Request Broker
OS	Operating System
QoS	Quality of Service
RAM	Random Access Memory
RTOS	Real-Time Operating System
RTPS	Real-Time Publish Subscribe
SCU	Software Control Unit
SLOC	Source Lines of Code
SST	Defense System Technologies (Tr. Savunma Sistem Teknolojileri)

WMC	Weighted Method Complexity
YMM	Software Engineering Department (Tr. Yazılım Mühendisliği Müdürlüğü)

CHAPTER 1

INTRODUCTION

As the projects in defense industry largely rely on distributed system architectures, communication between modules plays a crucial role in the proper operation of the system as a whole. Therefore, selection of the method for the communication of software running on different platforms becomes important. Two of the alternatives for data distribution are the Common Object Request Broker Architecture (CORBA) and Data-Distribution Service (DDS), both defined by Object Management Group (OMG).

CORBA is a distributed object computing middleware with quality of service (QoS) support which brings up a performance overhead while providing platform independence. DDS is a data distribution middleware focusing mainly on delivering the data within specified QoS requirements. In this study, CORBA and DDS are empirically compared as alternative communication methods to observe and assess their effects on software performance and quality.

The scope of this study includes the software projects both delivered and being developed by Software Engineering Department (YMM) of Defense System Technologies (SST) division in ASELSAN, Inc. The software reference architecture used in the projects is enacted by the department itself with motivation to increase mainly reuse and the overall software quality.

Software reference architecture is used to provide blueprints for the software projects developed in the same domain. In software engineering departments that adopt the component-based software development methodology such as

YMM, establishing software reference architecture proves useful mainly in the integration and reuse of the components. In order to do this, the reference architecture used in YMM differentiates the software operation between the control plane and data plane as well as the other reference architectures proposed in the literature for similar domains [24] [26] [37]. Introducing these two planes in the reference architecture brings up the necessity for classifying the messages shared by the components that reside in those planes; data and command, both having different requirements and focuses.

Having separated command distribution from data distribution, selection for the distribution method should be made according to the restrictions brought up by the requirements of the project. In order to achieve this, methods are evaluated under the metrics of software quality and performance. In this study, the evaluation metrics are selected and collected according to the application domain and requirements of YMM. Software quality metrics are collected under coupling and complexity measurements whereas utilization and latency are measured for evaluation of software performance.

The rest of this thesis document consists of the following sections:

In Chapter 2, background information on command and data concepts and CORBA and DDS middleware are introduced. Additionally, real-time embedded software performance and quality measurement metrics are presented.

Then in Chapter 3, experimental methodology is explained with metric selection, description for the prepared benchmark project that the measurements are taken on and the test scenarios that highlight the requirements of the application domain.

In Chapter 4, experimental results are demonstrated with the measurements on quality and performance related metrics.

Finally, in Chapter 5, the thesis is concluded with discussions on the measurements. The achievements and limitations of the study are reevaluated and summarized, also outlining perceived challenges and difficulties. Suggestions are presented for future work.

CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

In this chapter, differences between command and data concepts, real-time embedded software performance metrics and real-time embedded software quality metrics are reviewed, followed by discussions of the DDS and CORBA technologies.

2.2 Command and Data Concepts

Existence of domain-specific software architecture is a necessity for embedded systems. Software reference architecture aids to bring engineering discipline and predictability to software projects and prevents the engineers to develop new software for every project in the same domain. With the use of software reference architecture, product cycle time and development cost decreases whereas software quality and reusability increases [46]. As a result of this, software reference architecture becomes widely used in real-time embedded software projects. Primary goals for designing the software architecture are stated by [14] as the development of reusable architecture framework and development of reusable application components for the product line.

Design of the software reference architecture relies on the differentiation of control and computational aspects. Without this separation, difficulties in analyzing the application behavior occur. When the control plane is differentiated from

the computational plane, data flow and control flow could be separated and visualization and analysis of the system behavior becomes possible. Furthermore, it eases the addition and subtraction of components [24].

Data flow and control flow are the two important flow policies in software architecture [27]. Data flow represents the motion of computational data whereas control flow represents the motion of commands and indicators or flags (referred as “control data") that influence the sequence of operations [43]. Computational data carried in the data flow could be illustrated as large sized sensor data whose loss in the execution time causes disturbance in the operation rather than a failure. Loss of commands and indicators or flags in control flow on the other hand, may result in critical failure and damage. While “correctness" is of highest importance in control plane, data plane requires “efficient computation" in the first place [26].

Consequently, after differentiating the two data types mentioned in the architecture, it is necessary to take the necessary precautions, such as determining quality of service (QoS) specifications, for the sake of performance of the embedded system as both planes require attention on different aspects.

2.3 Real-time Embedded Software Performance

2.3.1 Introduction

An embedded system is a computerized system which in contrast to a general purpose computing environment, aims to achieve a dedicated set of tasks most often within specific hardware architecture and its restrictions. Embedded systems are used in almost every high-tech device for many different purposes. Real-time systems, on the other hand, are a subset of embedded systems [15]. The factor that differentiates real-time systems from other systems is their strict performance requirements [7]. Real-time software is expected to meet timeliness requirements in addition to the functional correctness [35]. Inability to meet its performance criteria means failure for a real-time system [19].

In a similar manner, physical constraints are the most important characteristics of an embedded system. Hardware used in the system on which the software runs is fixed and in most cases cannot be extended by the user. This brings up the necessity to decrease the hardware costs arising from memory, power, cooling and computing power. Consequently, developers are held responsible for optimizing the system to minimize the use of resources [15]. Usually, in order to decrease the costs of manufacturing the equipment, computers that are used in embedded systems are selected and built among the least powerful ones as long as they are capable of satisfying the requirements [4]. These restrictions necessitate optimizations in the system along with additional considerations on the system architecture, data distribution and algorithm selection.

Real-time systems are characterized by three important components; time, reliability and the environment. In real-time systems, time is considered to be the most valuable resource. Completion of a task should be scheduled within the stated time interval for that specific task. Inter-task communication should also be arranged to deliver the necessary messages between the tasks that interact with each other. Timing of these messages, in addition to the computational correctness, is important for the related tasks to operate properly [44]. In addition, in a real-time embedded system, some of the events received by the system that were triggered from the outside must be responded in a specified time interval. Exceeding this interval would result in system failure and a probable disaster [49]. Being a potential cause of a disaster in case of a failure, a real-time system has to be reliable. Failure of a real-time system could result in not only financial loss, but also loss of human lives, especially in the defense industry. Another active component of a real-time system is naturally its environment. It is necessary to consider the surroundings of the computer in the big picture as a whole [44].

2.3.2 Performance Metrics

A metric can be described as a characteristic of a system that can be measured. In software development point of view, a metric is generally related to software's

performance. Ideal metrics must contain some important qualities [31]. Metrics should be simple and quantitatively, precisely and objectively definable. The cost to obtain the metrics should be limited to a reasonable extent. The metric should be competent to fulfill its purpose of measure and they are also expected to be robust, so that they are resistant to small changes in the subject. Three fields are mainly covered when system performance is of consideration; response time, throughput and speed of the system.

Four important performance metrics are prioritized by [19]; performance profiling, A-B timing, response to external events and Real-Time Operating System (RTOS) task performance. [30] states processor utilization, memory consumption, memory available, memory utilization and disk utilization as fundamental performance metrics.

Capability of performance profiling brings up the information on the time spent for each function used in the system during runtime. Problematic areas are therefore more easily detectable, in other words, it is easier to observe on which functions the system spends more time and resources. Identifying those areas enables the developers to make optimizations on the system more efficiently [19]. Performance profiling metric could be considered under both speed-related metrics and throughput metrics according to the classification of [18] as it aids to obtain functional processing speed and processing rate of incoming messages.

According to [19], A-B timing is the most important timing measurement for real-time systems. It is a point to point timing measurement and enables the developers to check if a specific code segment meets its timing requirements. Similar to performance profiling metric, measurement of the time spent by the system to go from one point to another in the code, A-B timing metric could be considered under speed-related and throughput metrics of [18].

Response to external events metric is defined as measurement of the time spent by the software from receiving an external event to responding to that event after taking necessary action [19]. As it is related to the system response time, response to external event could be classified under speed-related metrics of [18].

RTOS task performance measurements are studied under two categories; task deadline performance measurements and task profiling performance measurements. Task deadline performance measurements cover the measurement of the time spent by a single task from being triggered by an event until meeting its deadline in a multi-tasking application. Task profiling performance measurements are similar to performance profiling measurements, except for taking the measurements on the task level instead of function level [19]. This metric could be considered under both speed-related and throughput metrics of [18] and processor utilization metrics of [30] as the utilization of the processor effects the response time of a task [7].

According to [18], existing software performance metrics prevent the evaluation of component performance as they mostly treat the system as a black box. For component based reference architecture as the one being used in YMM, six classes of metrics stated by [18] appear to be more suitable for evaluation of component performance in a real-time embedded system; utilization, speed-related metrics, availability, reliability, throughput and scalability metrics.

Utilization metrics concept refers to the extent to which a component's resources are being used when the system is under load. Types of the system resources may differ from one system to another. However, mainly the system resources under consideration of the engineers are; network bandwidth, CPU, memory, cache and storage. As a result, utilization metrics could be organized under three types; network utilization, server machine utilization and client machine utilization. Network utilization is typically the network traffic versus the number of users, whereas server and client machine utilization refers to the server or client utilization versus command rate for CPU, cache, memory and disk utilization [18]. Considering that a real-time embedded system must use its resources efficiently, measuring components' resource utilization under load is useful.

Speed-related metrics cover a variety of metrics that the engineers use to measure the processing speed of a system. System-user response time is one of the most preferred speed metrics which is used to obtain minimum, maximum and the average user response time for various user groups. If the system contains

different types of messages for communication, it is necessary to measure the processing time for the messages of different kind. Database connection and response times are necessary for the systems making use of database accesses with various types of database queries. Latency metrics, such as processing delay and communication delay, also form an important type of speed-related metrics as they are needed for taking different kinds of delay measurements in a system [18]. Additional speed-related metrics are introduced by [9].

Availability metrics are used for checking system availability which by definition is the system available time divided by the system evaluation time in total [48]. Component or system availability becomes important in measuring the performance [29]. According to [18], availability metrics could be investigated under three types; network, computer hardware and computer software availability.

Reliability of a system is usually expressed with its mean time to failure (MTTF) or mean time between failures (MTBF). [18] bases the component reliability on component uptime and downtime for services during the time interval of performance evaluation and divides reliability metrics under four categories; network reliability, computer hardware reliability, computer software reliability and application reliability.

Throughput metrics are used to measure the success of a component in message, event and transaction request processing aspects for the period of evaluation. Throughput metrics could be investigated under three types; transaction throughput, message throughput and task throughput. Typically, throughput metrics are total number of processed versus received transactions, processed messages versus received messages or completed tasks versus given tasks [18].

Scalability is defined as the deployment capability of the system for performing its functions and services within a range of various defined sizes. System and component capacity boundaries, data volume, number of concurrent access users, process speedup and throughput improvement are the areas that the system's scalability is measured on. Scalability metrics are examined in performance improvement and performance boundary and limit categories. Function speedup, transaction throughput improvement, boundary and limit are the examples of

scalability metrics [18].

In addition to these performance metrics, when the communication between software control units is in consideration, new performance dimensions occur. Delivery to simultaneous nodes, bandwidth utilization, media reliability, congestion and failure recovery are the performance dimensions stated by [2]. Within these dimensions, [2] emphasizes four metrics to be the key metrics for performance; latency, jitter, throughput and efficiency. All are in accordance with the component based metrics of [18].

Table 2.1 combines and summarizes the relationship between metrics of [18], [19] and [2] in accordance with the scope of this thesis.

2.4 Real-time Embedded Software Quality

2.4.1 Introduction

In this section, real-time embedded software quality is investigated under two parts. In the first part, widely accepted and used quality metrics in the literature, including the ISO/IEC 25010 standard, are discussed. In the second part, although not being described as a quality metric, reusability; whose effects on software quality is investigated in detail in [13] and is accepted as a major software quality factor in [17], is explained.

2.4.2 Quality Metrics

In order to measure software quality, it is important to use commonly accepted and well defined characteristics and metrics. In this manner, ISO/IEC 25010 is a widely used standard which describes the metrics to measure and evaluate software product quality. Before taking its final form as ISO/IEC 25010, ISO/IEC 9126 was published in 1991 and revised in 2001. [22] introduces two quality models, one is for internal and external quality and the other quality model is for quality in use. Six characteristics used to define internal and external qual-

ity are identified as; functionality, reliability, usability, efficiency, maintainability and portability. Quality in use characteristics are divided under four categories; effectiveness, productivity, safety and satisfaction.

Although the ISO/IEC 9126 is widely used and adopted as a software quality framework, many criticisms pointing out its shortcomings also exist. In [6], weaknesses of ISO/IEC 9126 are discussed along with the suggestions for resolving those issues. Terminology used in ISO/IEC 9126, absence of metric classification, problematic mapping between standards and representation of measurements for the metrics are the major subjects of criticisms in [6]. In their study, [5] pointed out the flaws in ISO/IEC 9126. Ambiguity in the definitions of metrics, insufficiency of counting measures, inappropriate phase assignments of attributes and insufficiency of attributes and measures are the major problems that usually surface at the end of a study.

As a consequence of these analyses, ISO/IEC 9126 was replaced by ISO/IEC 25010. ISO/IEC 25010 introduced new characteristics and sub-characteristics with an extended scope including computer systems as a whole rather than software only.

Software quality is investigated under two quality models in [23]. **Quality in use model** addresses the five characteristics observed as the product is used interactively whereas the **product quality model** contains eight characteristics pointing out static features of the software and dynamic features of the system. Five characteristics of quality in use model are; effectiveness, efficiency, satisfaction, freedom from risk and context coverage. Product quality model's eight characteristics are defined as; functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability and portability [23].

Functional suitability describes the extent of functions that the system is capable of providing as required by the user. Needs of the user is the main concern, as opposed to the functional specifications. Performance efficiency investigates the utilization of available system resources, such as hardware configuration, software configuration or other supplies used by the system, when the system is under conditions defined previously. Compatibility is the availability of the

software for interacting with other software in the same hardware or software environment. Usability describes the ease of usage of the product by defined users in order to fulfill pre-specified goals. Reliability is defined as the capability of a system to perform under stated conditions for defined amount of time. Since software does not wear out, design and implementation errors, non-conformance of requirements are targeted. Security covers data and information protection in the system, and also transmitted by the system, against other systems and actors. Limited authorization to data access is also a concern. Maintainability is the availability of the system for alterations that are applied by the maintainers in order to prolong its lifespan. Activities that intend correction, improvement or adaptation to changes in addition to software upgrades and updates are all included in maintenance efforts. Portability describes the ease and effectiveness of transferring the software from one environment, hardware or software, to another [23].

In their work, [39] analyze the relationship between quality metrics and physical metrics in embedded software systems. According to [39], embedded systems differ from the traditional software as the metrics used for embedded software are mostly physical metrics due to design constraints. In addition; reuse, time to market and price are also stated to be important for embedded systems. Works of [38] and [11] show parallelism with [39] in the selection of metrics. Coupling, cohesion, extensibility, population metrics and complexity are selected in all three studies among the metrics to be used for the quality measurement.

Coupling is defined as the measurement of the relationship between elements in software. Cohesion, on the other hand, is the measurement of the degree of relationship between those elements. Extensibility refers to the expansion capacity of the software. Population metrics measure the attribute, method and class counts. Complexity metrics are used to determine the difficulty levels in understanding the problem or algorithm [39].

Among the eight characteristics and their sub-characteristics defined in [23], ones that are considered applicable and beneficial to conduct the comparative analysis effectively are; time behavior, resource utilization and capacity which

reside under performance efficiency category. Furthermore; coupling, cohesion, population metrics and complexity are acknowledged as key metrics for software quality analysis.

In addition to these quality metrics, reusability under maintainability characteristic of ISO/IEC 25010 is investigated and explained separately in the next section.

2.4.3 Reusability

In [17], reusability is introduced as another major software quality factor as the purpose of software reuse is defined to be the improvements in both software quality and productivity. As stated in [17], definition of software reuse is; using the previously developed software or software knowledge to create new software. Both the software that is eligible for reuse and software knowledge can be counted as reusable assets. Software's probability of reuse is called as its reusability property.

Software reuse is investigated under two categories; opportunistic (ad-hoc) reuse, which is not a part of a repeatable process, and systematic (planned) reuse [33]. YMM aims to increase the systematic reuse in their projects.

The need and motivation for building larger scale systems that are more complex, reliable and less expensive while not exceeding the time of delivery directs the focus of developers on software reuse. In comparison to traditional methods of software engineering, software reuse is considered to be a better method. Since most of the organizations building software systems are operating on specific domains, the systems they build are mostly not new. They are generally the altered versions of existing systems the company has previously built. This condition, in other words; the concept of domain engineering became a key idea in software reuse and encouraged the application of reuse to improve quality and productivity [17].

Effects of reuse on quality are investigated thoroughly in [13]. Defect rate is observed to be decreased as a result of systematic reuse due to three major reasons;

the components that are planned to be reused are more carefully designed and tested [33], defects that occur with the components that are subject to reuse are approached with higher priority [34] and developers gets more experience on the component as it becomes reused [36].

According to the case study in [36], with the usage of a framework for development, increase in quality and productivity was achieved. Gross productivity also increased and implementation of a function point (FP) accelerated as the difficulties in the development of a component become resolved beforehand in the framework. Developers also become more experienced on the components due to the learning effect. Use of a design pattern in a software engineering environment is also encouraged in [12] as it enables the definition of a process and consequently opens the way for systematic reuse.

In [34], a study investigating the defect density and stability in accordance with reuse is explained. Reuse is claimed to be a factor improving software quality as components subject to reuse have lower defect density and higher priority in solving those defects than other components that are not reused. Less amount of modification effort is applied on the reused components, making them more stable than non-reused components which are more defect-prone.

The results and advantages that component based software engineering (CBSE) and reuse offers to developers and users are stated by [34] as; shorter development time and decreased total cost, architectures that are standardized and available for reuse, separation of skills by packaging difficulties into frameworks, easier acquisition of components and improved reliability in the shared software components. On the other hand, disadvantages are claimed to be the dependence on component providers and suspicion on the new technology in the case of not developing the components to be reused in-house in addition to the trade-offs for quality attributes and functional requirements [34].

Table2.1: Real-time embedded software performance metrics related to component based software architecture

Component Based Performance Metric	**Explanation**	**Covered Metrics in [19]**
Utilization [18]	Usage of resources when the system is under load	Task deadline and profiling performance measurements
System-user response time [18]	Minimum, maximum and average user response time for different users	Performance profiling, A-B timing, Response to external events, Task deadline and profiling performance measurements
Latency [18] [2]	Processing and communication delays	Performance profiling, A-B timing, Response to external events, Task deadline and profiling performance measurements
Availability [18]	Network, hardware and software availability	-
Reliability [18]	Network, hardware ,software and application reliability	-
Throughput [18] [2]	Success of a component in message, event and transaction request processing	Performance profiling, A-B timing, Task deadline and profiling performance measurements
Scalability [18]	Deployment capability of the system within a range of various sizes	-

Table2.2: Real-time embedded software quality metrics and their relationship with ISO/IEC 25010

Quality Metric	**Related Category in ISO/IEC 25010**
Time behavior [23]	Performance efficiency
Resource utilization [23]	Performance efficiency
Capacity [23]	Performance efficiency
Coupling [39] [38] [11]	Compatibility
Cohesion [39] [38] [11]	Compatibility
Population metrics [39] [38] [11]	-
Complexity [39] [38] [11]	Usability
Reusability [23]	Maintainability

2.5 Data Distribution Middleware: CORBA and DDS

Advanced real-time systems of today are mostly distributed and require co-operation and coordination of distinct computers. Major difficulty behind this is the fast delivery of data to many recipients. In order to resolve this issue, different data distribution middleware technologies were developed and those could be classified and investigated under three broad categories: client-server, message passing and publish-subscribe [42].

According to [42], for service oriented systems, client-server middleware is the suitable choice whereas message passing is preferable for free-form data sharing. Publish-subscribe covers both messaging and discovery while forming a data centric information distribution system.

Common Object Request Broker Architecture (CORBA) and Data Distribution Service (DDS) are widely used middleware technologies in real-time systems.

2.5.1 Common Object Request Broker Architecture

Common Object Request Broker Architecture (CORBA) is a framework used for object-oriented and distributed application development. As the technologies in hardware and software were diversified, real-time computing environments be-

came heterogeneous [32]. Increasing necessity for standardized communication between systems developed using different languages formed the basis for a middleware that aids establishing connection through different message formats and types [20]. As a result, a distributed object computing middleware that is also able to manage quality of service requirements in real-time systems emerged as an industry standard by Object Management Group (OMG) [41].

With foundation of OMG in 1989, development of CORBA began. The first version of the specification; CORBA 1.0 was introduced in 1991. In the following years, newer versions were released and in 1998, Minimum CORBA was introduced for embedded systems in order to provide an interoperable subset of CORBA for applications with resource constraints. Real-Time CORBA, containing the abilities for network, CPU and memory resource management [41], was released in 1999 and extended CORBA for building deterministic applications. It took two years for a team composed of several companies to prepare Real-Time CORBA which was planned to support Fixed Priority scheduling protocols [20]. Most recent major version (3.0) of CORBA specification was released in 2002 and updated twice with minor changes to 3.0.2 in the same year.

CORBA is a middleware under the control of operating system, carrying resemblance to a software bus. It is composed of two major components; Interface Definition Language (IDL) and Object Request Broker (ORB). IDL eliminates language dependence by specifying a common interface to distributed objects, whereas ORB facilitates the network connection by locating objects, managing communication and marshaling the data. ORB is to provide basic functionalities in order to enable heterogeneous objects operate together [32].

Figure 2.1 has been constructed to depict the mechanism for sending a request and receiving its result through the ORB. Client is the actor in need to invoke a method on the Object which is implemented by the code shown as Object Implementation. It is the responsibility of ORB to find and prepare the object implementation and establish data communication. It is necessary for the Client to know the type of the object and the operation it needs in order to make a

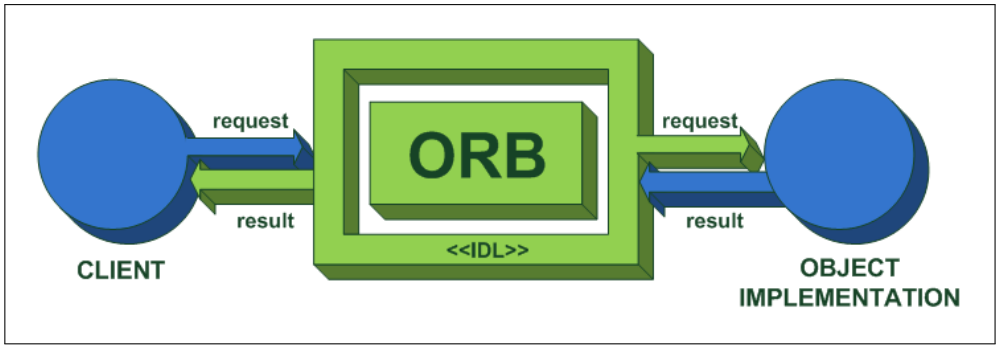

Figure 2.1: Request and Result Being Sent Through the ORB

request. The interface that the Client sees (IDL) is composed of operations and their parameters and it is language and location independent. The Client only knows about the logical structure. IDL specifies the interfaces of objects in order to define their types. ORB delivers the request to Object Implementation and retrieves the result [3]. ORB, therefore provides platform independence to CORBA as it is possible to use CORBA objects on any platform implementing the ORB.

Even though CORBA provides language and platform independence, its performance is a major drawback. There exist numerous studies in the literature on performance of CORBA and performance improvement methods for ORB implementation. [32][47][40][8]

According to [32], CORBA is not feasible for high performance communication service implementation. In their study, [32] compares CORBA to socket implementation and underlines platform independence, language independence and decreased complexity in number of interfaces (from NxN with a N node socket system to N with CORBA ORB) as the advantages of CORBA over socket. [32] states that, although socket implementation provides better performance, being OS specific makes it not suitable for heterogeneous and distributed computing environments.

Another study on improving CORBA performance is proposed in [47] demonstrating the efforts on caching. Similarly, transparency in location, OS and

language are stated to be strengths of CORBA compared to socket and remote procedure call (RPC) protocols. However, problems of the ORB are pointed out to be related with the message overhead and networking bandwidth.

Factors causing the performance overhead in CORBA are listed in [40] mainly as; excessive data copying and conversions, inefficient techniques used for demultiplexing in server, buffering methods and long function call chains inside the ORB.

2.5.2 Data Distribution Service

Data Distribution Service (DDS) is an open standard developed by OMG that caters publish-subscribe middleware for real-time systems [42]. The first version of DDS specification was released in 2003, updated to version 1.1 in 2005 and took its latest form as version 1.2 in 2007.

DDS Specification introduces two interfaces: Data-Centric Publish Subscribe (DCPS) and Data Local Reconstruction Layer (DLRL). DCPS interface is located at the lower level aiming to distribute data to related subscribers efficiently. DLRL is optional and resides atop DCPS level. DLRL aids to integrate data distribution service into the application layer easily. Typed interfaces, which translate the actual data types, are preferred to be used as they are simpler, safer and more efficient to use. In order to use typed interfaces, it is necessary to use a generation tool converting the type descriptions into fitting interfaces and implementations. This conversion fulfils the space between middleware and typed interfaces [1].

For the description of service behavior in DDS, Quality of Service (QoS) concept is used. QoS settings enable the developer to focus on what is required rather than how to implement that requirement. In a real-time application where the resources are limited, it is important to distribute the resource usage proportional to importance of the requirements through QoS properties [1]. As [10] states, main focus of DDS is on the distribution of data within guaranteed QoS restrictions.

DCPS model introduces the global data space concept available for use of any interested application. Accessing the data located in this data space is achieved by becoming a “Subscriber" and similarly, contribution of an application requires it to become a “Publisher". Whenever a Publisher sends data, the middleware is responsible for delivering the data posted on the global data space to interested Subscribers. DCPS model resides above a data model that describes the global data space through a set of data structures presented by a topic and a type. DLRL model reconstructs the data using updates enabling the application to use that data as local. The middleware is also responsible of updating the local copy of the data as well as distributing it to related Subscribers [1].

Real-time Publish Subscribe (RTPS) is a standard wire protocol defined by OMG for DDS. Method for transferring messages between nodes is not defined in RTPS specification and left for the providers to introduce their solutions. Three major DDS implementations are: RTI DDS, OpenSpliceDDS and OpenDDS [16]. A performance assessment study between these implementations could be found in [16].

According to [10], publisher-subscriber systems based on DDS offer narrow support for reliability in exchange for guaranteed QoS. Consequently, a routing method is proposed by [10] in their work. Similarly, various DDS implementations, solutions and analyses exist in the literature.

Considering the capabilities, advantages and shortcomings of CORBA and DDS, [25] states DDS and CORBA are compatible and complementary; DDS focuses on data whereas CORBA focuses on requests.

CHAPTER 3

EXPERIMENTAL WORK

The present study aims to compare and analyze the effects of using DDS and CORBA for command distribution on a real-time embedded system developed in conformance with a software reference architecture. In order to achieve this goal, it is necessary to define the priorities of the software development team and requirements of the software and underline the assessment criteria considering the YMM's application domain.

Selection of the metrics with reasoning and methods for obtaining those measurements are presented in this section. Following the metric selection, the project used for comparison is described and the experimental process is explained in detail.

3.1 Experimental Methodology

In order to analyze the performance and quality aspects of command distribution with DDS and CORBA, a benchmark project is prepared using the application domain requirements and software reference architecture of YMM. Four test cases are formed to implement the possible scenarios that may occur through the operation of the system. The benchmark project is described in detail in *3.1.2 Project Description* and the test cases are presented in the *3.2 Experimental Process* section.

Performance and quality metrics are selected carefully among the widely used metrics in literature reviewed in Chapter 2. Selection of the metrics that are

unbiased, applicable to the domain and distinctive for the comparison has been the primary concern.

3.1.1 Selection of Metrics

Metric selection was conducted in a way to highlight the difficulties in the selection procedure between usage of CORBA and DDS in time and mission-critical software systems. Both performance and quality concerns guided the selection of metrics for comparison of two methods in the reference architecture.

Among the metrics identified in Chapter 2, utilization and latency are selected as the performance metrics to be measured from the project. In terms of utilization, memory usage and CPU usage values are measured. For latency measurements, A-B timing is used to determine processing delays. These metrics are selected among the ones that build up the key restrictions in performance for the real-time systems developed by YMM.

For the comparison in software quality, coupling and complexity are selected as the quality metrics to be obtained from the project. Code size, code complexity and coupling measurements are taken from the benchmark project for software quality analysis.

3.1.2 Project Description

For the experimental work, a benchmark project was prepared using the software reference architecture of YMM. The project contains a System Simulator, that is; a narrowed-down version of a software control unit (SCU) implementing only the desired functionality used in the application domain. In order to send and receive messages via CORBA or DDS, System Simulator uses one of the external interfaces through its communication port. Figure 3.1 shows the system, excluding the internal SCU details, with System Simulator connected to CORBA external interface. It is possible to add CORBA or DDS functionality directly to the software units that are using the communication, in other words, inside the System Simulator. However, the reference architecture requires the

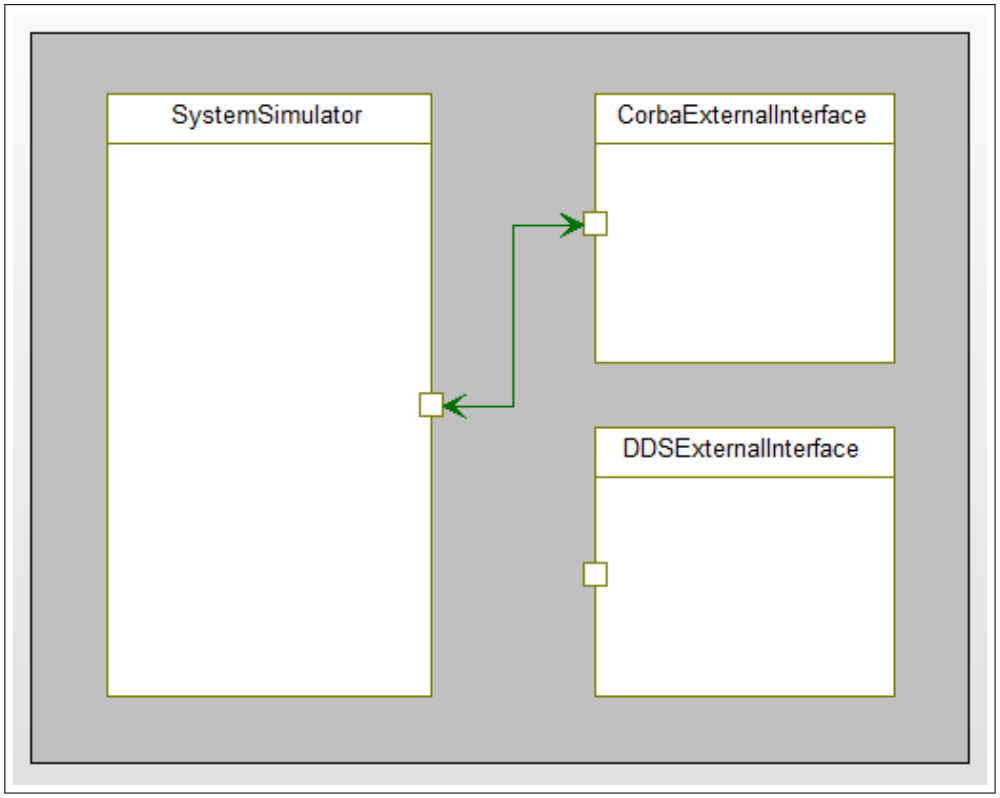

Figure 3.1: Representation of the System Using CORBA Interface

software units to be project independent. Reuse is of the essence and to prevent modification of a unit while using it in separate projects, external interfaces are used to implement that unit's provided and required interface functions and events. For this reason, CORBA and DDS external interfaces are prepared and it is possible to switch between the two by changing the link between ports.

One of the most common commands used in the domain (*"move to target"*) is selected and the functionality that is required to realize the command is implemented in the System Simulator. Moving turret to target scenario in the application domain is generally defined as follows;

- External application (i.e. user interface or target tracking software) sends the command containing target position information to SCU.
- SCU sends an acknowledgement message upon receiving the command (command is taken or rejected).

- SCU sends a status message regarding the change in state of the system (i.e. turret is moving).
- SCU sends the result (success or failure) of the operation.
- SCU sends a status message regarding the change in state of the system (i.e. turret is idle).

The message flow given above describes the command flow between SCU and an external application. In addition to command flow, data flow occurs during this operation as the SCU publishes angular position data of the turret periodically while moving.

Preparation of the System Simulator class and external interface classes is handled in a way to keep the provided and required interfaces between those classes unchanged and hence, isolate the external interfaces from the main software unit (System Simulator in this case). DDS Repository and CORBA IDLs are prepared by extracting the relevant functions and topics from existing YMM projects in order to fulfill the domain requirements and follow the software reference architecture of YMM. As explained in *2.2 Command and Data Concepts*, separation of control and data flow is an important part of the architecture. Consequently, the benchmark project is designed and implemented in a way to provide this separation. Related indication and status events are passed between the classes to indicate the preparedness of the data which is to be pulled separately by the class in need of that information.

3.2 Experimental Process

Four test cases that realize possible system architectures for distributed software in the application domain are developed for the investigation of performance and quality metrics.

Within the flow of the benchmark operation scenario, GUI software that is responsible for handling the commands from the user (GUI Command) sends the command through DDS or CORBA to SCU. External Interface module

captures the command and forwards it to the system. Considering the state of the system at the time of receiving the laying command, system responds to the command by sending a response and stating either the command is taken or rejected. If the command is taken, state of the system changes as the turret starts to move. An indication on the status change is also sent to the GUI software that is responsible for displaying the system status and turret position (GUI Display). In some cases, responsibilities for sending the commands and displaying the data are combined in single GUI software (GUI Command & Display).

The part after the status change contains a difference between CORBA and DDS cases since the remote function invocation methodology in CORBA puts the responsibility to retrieve the status information on the part that requires this information, in this case; GUI. In the DDS case, this responsibility is given to the external interface. Similarly, when the turret is in motion, knowing the system state, DDS External Interface periodically pulls the turret angles from the system and publishes whereas CORBA External Interface fetches the angle data from the system upon receiving a request from the GUI application. Upon completion, system sends the result indication to the external interface which informs the source of command (GUI Command) and changes its state.

3.2.1 Test Case 1: Single Source for Command and Display

In the first case, System Simulator using one of the external interfaces (CORBA or DDS) receives the command from the windows application which is the only source of the command in addition to being the only endpoint receiving the published data. Model diagram representation of this case could be observed in the Figure 3.2.

This case realizes a common scenario where there is a single user interface application to control and display the SCU behavior. An authorized user can start the laying operation using the user interface software that is running on Windows operating system and observe the behavior of the software using the same panel.

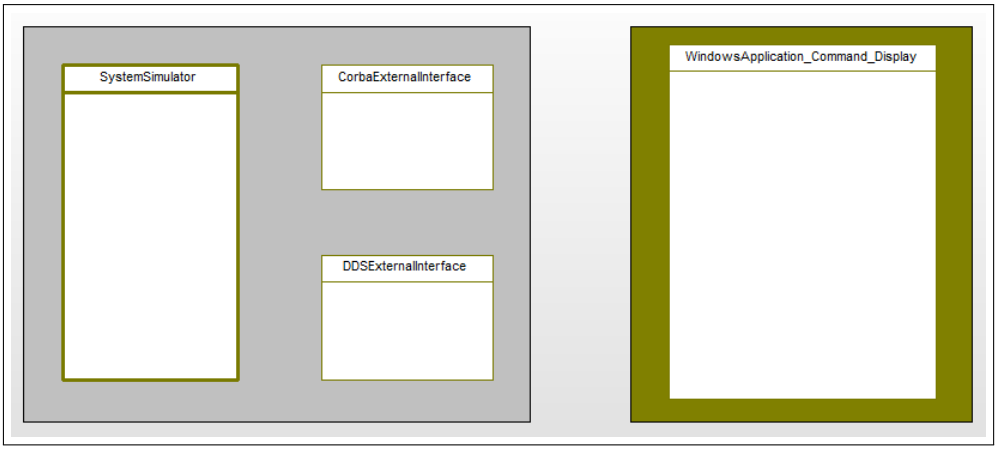

Figure 3.2: Representation of the First Test Case

For this test case, sequence diagrams demonstrating the flow of messages between the user interface, external interface (CORBA and DDS) and System Simulator are shown in figures 3.3 and 3.4.

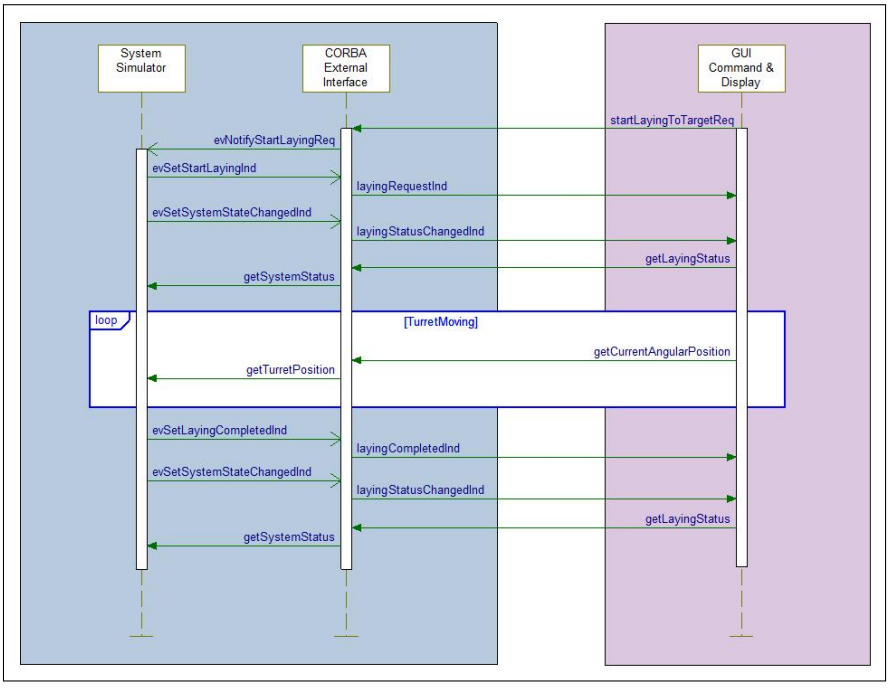

Figure 3.3: Sequence Diagram for the First Test Case with CORBA

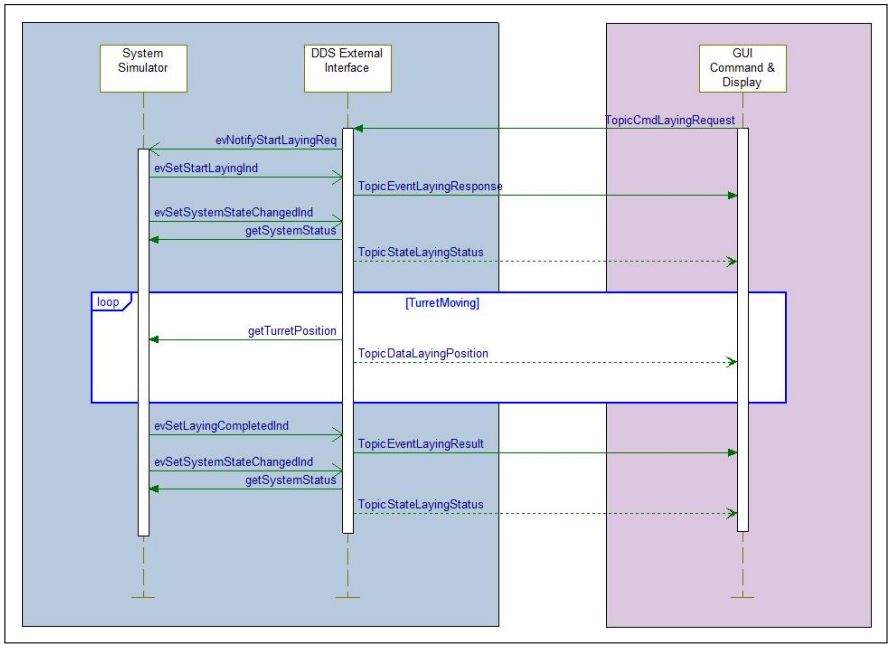

Figure 3.4: Sequence Diagram for the First Test Case with DDS

3.2.2 Test Case 2: Separate Command and Display Modules

For the second test case, source of the command and the application using the data published by SCU are separated. Most system configurations in YMM's application domain use this formation.

Figure 3.5 shows representation of the second test case. Similar to the first case, System Simulator only uses one of the external interfaces shown inside the grey box representing SCU in a test setup.

This case differs from the first test case via separation of command and display modules. In the application domain, there may be a command panel that does not display the results of some commands it sends or sometimes does not employ a display module at all. In that case, an additional interface that shows the system status information is used as a separate panel.

Sequence diagrams containing the scenarios with usages of DDS and CORBA external interfaces are given in figures 3.6 and 3.7.

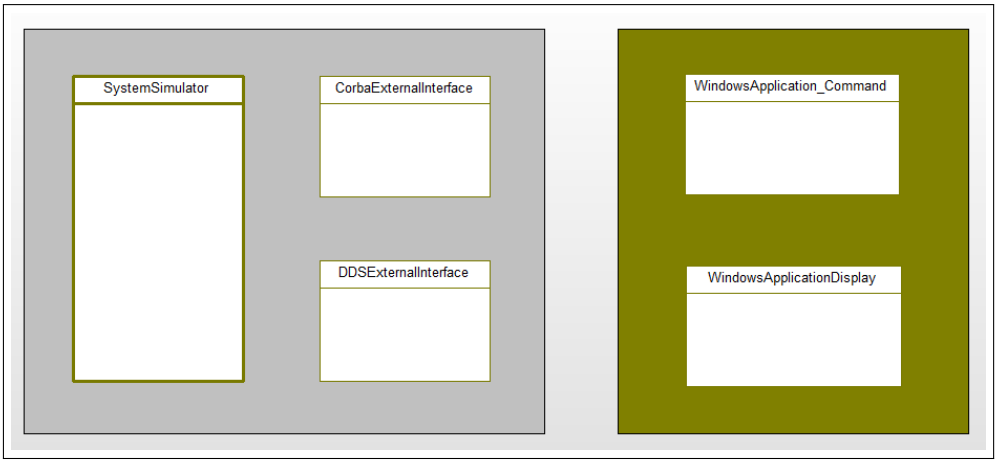

Figure 3.5: Representation of the Second Test Case

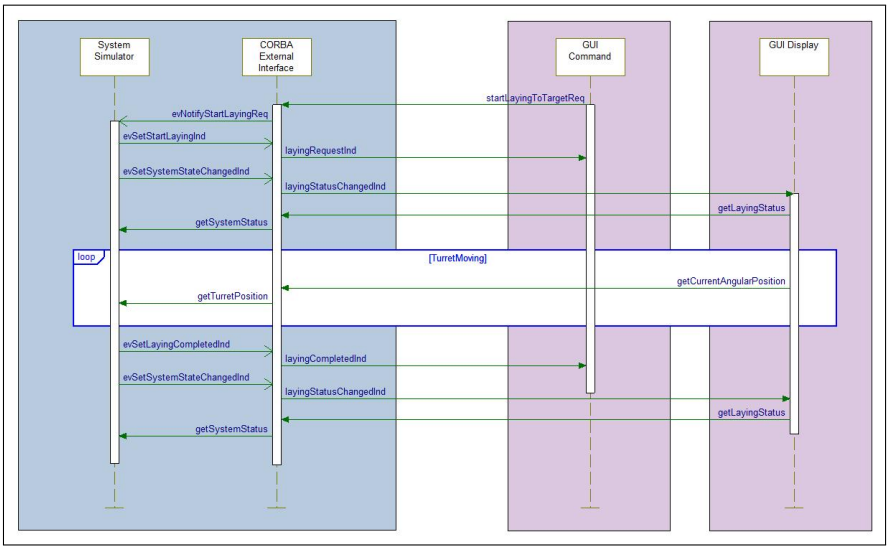

Figure 3.6: Sequence Diagram for the Second Test Case with CORBA

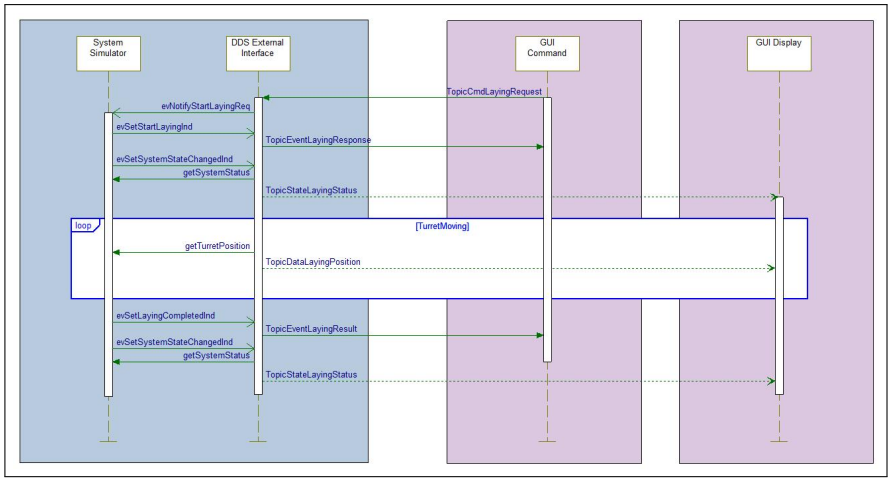

Figure 3.7: Sequence Diagram for the Second Test Case with DDS

3.2.3 Test Case 3: Multiple and Separate Command and Display Modules

The third case, whose model representation is given in Figure 3.8, additionally multiplies the number of receiving applications to provide basis for observation of multicast behavior of both methods.

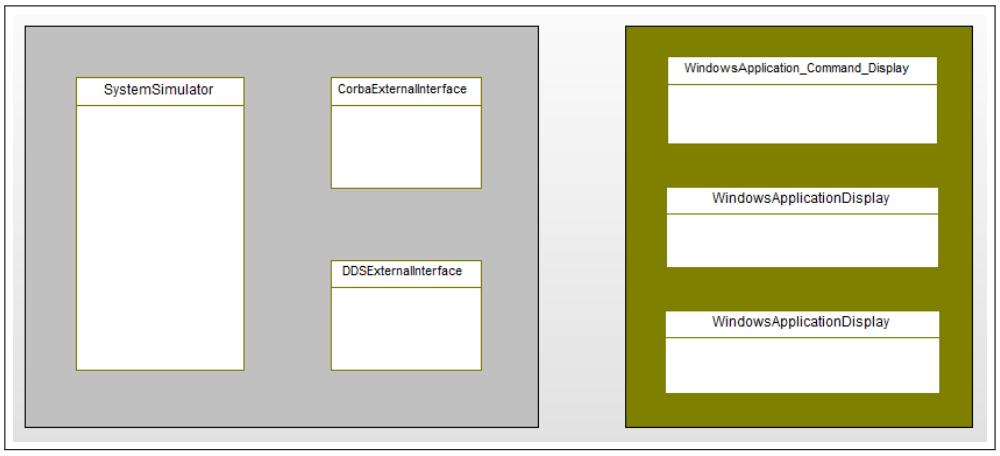

Figure 3.8: Representation of the Third Test Case

This test case introduces the conditions that may require more than one display modules for the user. Source of command is single and is also capable of dis-

playing the data. Additional display panels are sometimes required for different operators that work on different parts of the system and require the information on system status simultaneously. Furthermore, there can also be another software control unit that requires the data published by our SCU to operate properly.

Sequence diagrams for the third case could be observed in figures 3.9 and 3.10.

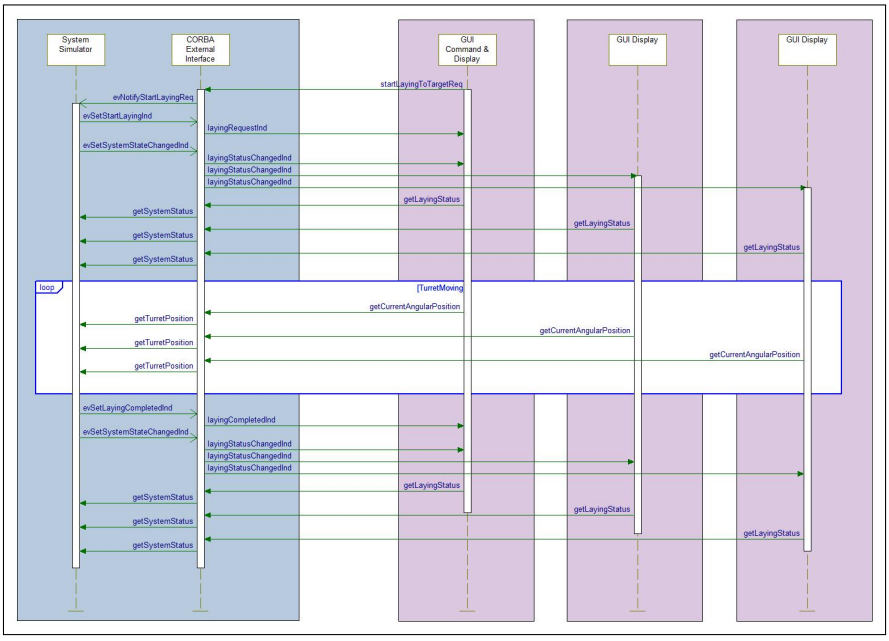

Figure 3.9: Sequence Diagram for the Third Test Case with CORBA

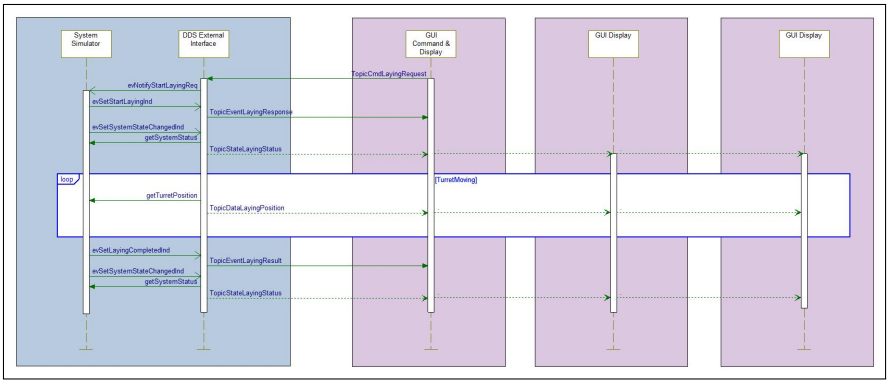

Figure 3.10: Sequence Diagram for the Third Test Case with DDS

3.2.4 Test Case 4: Multiple Sources of Command and Multiple Display Modules

The last test case realizes the scenario containing multiple publisher and subscriber applications existing in the same environment. It is not common to come across this case in the application domain, however, this case exists in theory and used for comparative analysis.

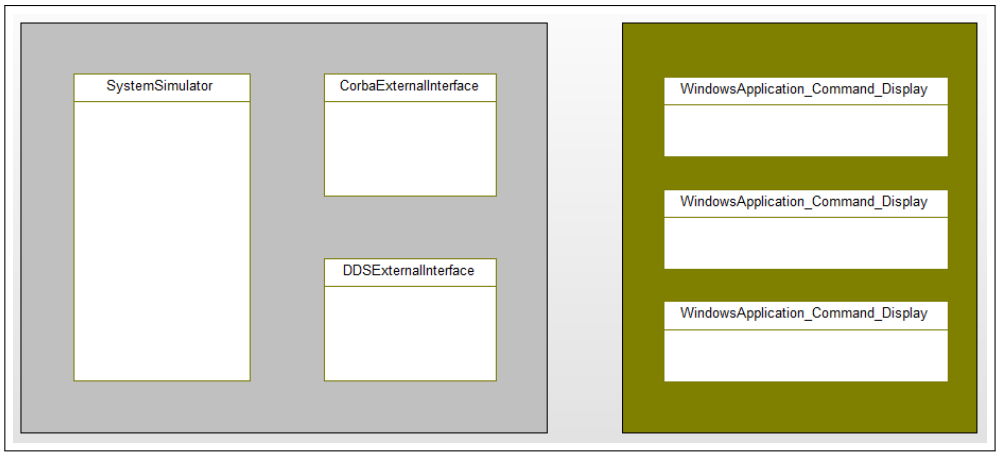

Figure 3.11: Representation of the Fourth Test Case

This scenario is similar to the third test case as it also contains multiple nodes for displaying the information provided by the SCU. However, there is an important difference between the two cases as the last test case distributes the responsibility of sending the command to three different software modules, which requires the SCU remembering the source of a command and send the responses only to source of the command. In the application domain, different SCUs require and provide services from/to each other and operate simultaneously.

For the last test scenario, figures 3.12 and 3.13 represent the sequence diagrams for systems using CORBA and DDS as external interfaces.

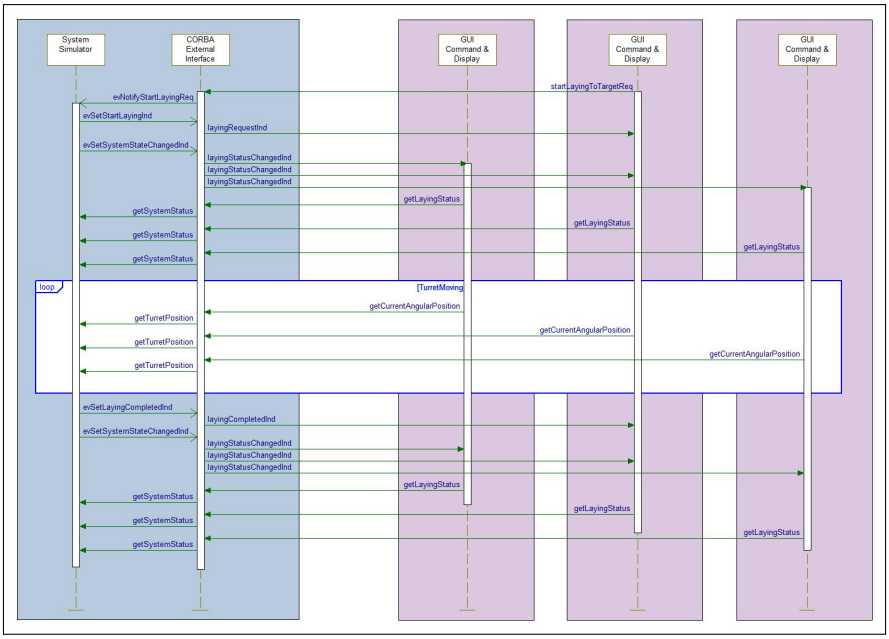

Figure 3.12: Sequence Diagram for the Fourth Test Case with CORBA

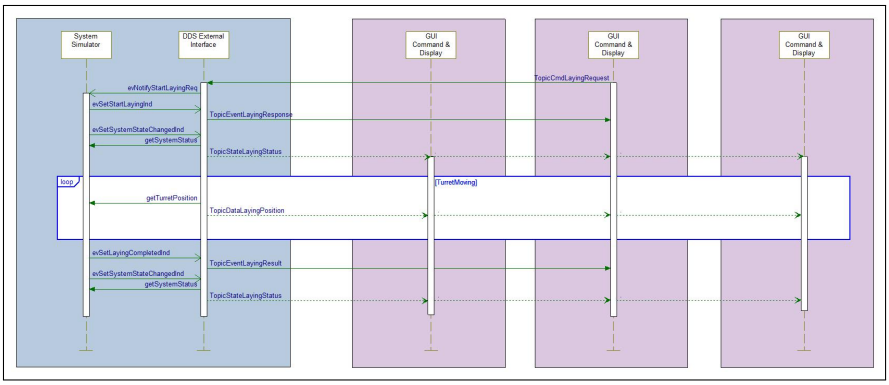

Figure 3.13: Sequence Diagram for the Fourth Test Case with DDS

3.2.5 Tools Used for the Experiments

For the development of benchmark project and windows applications in C++ programming language, IBM Rational Rhapsody 7.5 is used. DDS implementation used in mentioned software is COTS. Operating environment for the SCU is the VxWorks 6.8 running on MPC8640D processor.

CHAPTER 4

EXPERIMENTAL RESULTS AND EVALUATION

4.1 Measurement of Quality Related Metrics

Quality related metrics that are obtained and used in the evaluation are grouped in two categories; coupling and complexity. Code size and coupling between objects measurements are taken under coupling category, whereas cyclomatic complexity, weighted methods per class and percentage of branches are measured for complexity evaluation.

Code size is measured as source lines of code (SLOC) which gives the summation of non-commented lines of code.

Coupling between objects (CBO) is obtained via counting the non-inheritance classes that a specific class depends on.

Counting the number of flows in a module gives the McCabe's cyclomatic number per module (MCN) and this value being high refers to increased complexity in the code.

Weighted methods per class (WMC) is measured by counting the methods in a class and weighting them with a complexity coefficient, commonly cyclomatic complexity. In this study, WMC is measured with unity weight as the cyclomatic complexity is also given.

Another complexity measure is the percentage of branches in the code. This metric represents the count of branching points in the sequential execution of

the code, such as if-else and switch-case statements.

For the measurement of quality related metrics, upon completion of test case implementation, header and source files are generated for each test case scenario. Four different CORBA external interface classes (CORBA EI 1 to 4) are prepared for each test scenario as the scenarios get more complex from the first to last. On the other hand, the same DDS external interface implementation (DDS EI) is used in all four scenarios due to its multicast support.

Extraction of the metrics is handled via free static code analysis tools. First tool we used is SourceMonitor [45], which identifies the size and relative complexity of a module. Branch percentage measurements are obtained through SourceMonitor. Second tool we used is C and C++ Code Counter (CCCC) [28]. Measurements on metrics, such as MCN, WMC and CBO are taken from CCCC. The last tool is Understand [21], which is a source code analysis tool that we used to measure SLOC in our modules.

Measured quality related metrics for the external interface modules covering all four test cases are given in Table 4.1.

Table4.1: Extracted software quality metrics

Metric/Module	**DDS EI**	**CORBA EI1**	**CORBA EI2**	**CORBA EI3**	**CORBA EI4**
Source Lines of Code (SLOC)	914	1068	1150	1234	1318
Coupling Between Objects (CBO)	15	19	19	19	19
Weighted Methods per Class (WMC)	35	49	54	59	61
McCabe's Cyclomatic Number (MCN)	4.57	3.96	4.32	4.75	5.36
% Branches	12.2	12.5	13.5	14.2	16.1

As it can be observed from the Table 4.1, increasing number of peers lead to additional effort in the development of the external interface. Lines of code increases thus increasing coupling and the coding effort. Code size and percent-

age of branches for DDS external interface remains close to CORBA external interface only for the first test case.

Whereas the WMC count for DDS external interface stays lower than the CORBA external interface implementations; cyclomatic complexity (MCN) of DDS external interface resides between the values of CORBA EI2 and CORBA EI3 which correspond to the cases with 2 and 3 GUI applications. With the fourth test case, increasing the number of command sources further increase the complexity of external interface code.

Higher complexity increases the error proneness of the code. When the test scenarios are considered, for the CORBA external interface implementations, it can be observed that the design decisions (such as pulling the turret angles from the SCU periodically) are made to keep the complexity on the GUI application side. The reasoning behind this decision lies within the swiftly changing nature of the GUI application and the testing cost of GUI being lower.

4.2 Measurement of Performance Related Metrics

Performance related metrics that are measured for the evaluation are grouped in two categories; latency and utilization. A-B timing measurements are taken for latency evaluation, whereas CPU usage and memory usage values are measured for utilization.

Performance measurements were conducted for all four test scenarios with both DDS and CORBA implementations. SCU executables running on MPC8640D processor were analyzed in runtime with WindRiver Workbench System Viewer and A-B timing (Latency), CPU, RAM and stack usage (Utilization) measurements were taken for the test case scenarios explained in Chapter 3.

4.2.1 Latency Measurements

For latency metrics, A-B timing measurements were taken for each scenario from the beginning to the end of laying operation. In other words, we measured the

time spent by the SCU from the point where the first command is received from the command interface until the execution is completed and the result is sent. This method allowed us to observe the effect of using DDS and CORBA as external interface on rest of the software. Delay caused from the network was exempted.

Table 4.2 shows the A-B measurements (in milliseconds) for both external interfaces for all four test cases in tabular form.

Table4.2: Extracted latency metrics

Interface / Scenario	**Test Case 1**	**Test Case 2**	**Test Case 3**	**Test Case 4**
CORBA	5002.6	5003.5	5005.8	5005.3
DDS	4999.7	5000.7	4999.8	5000.4

The trend between test cases could be observed in the Figure 4.1. It can be

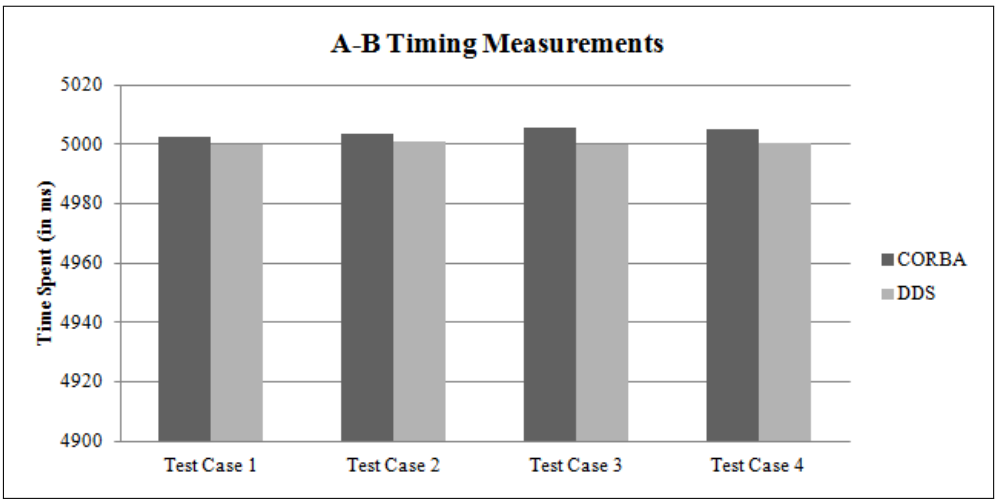

Figure 4.1: A-B Timing Chart for the Four Test Cases

observed from the measurements that the time spent by the software conducting laying operation is slightly higher for the cases where CORBA is used as the external interface module due to its performance overhead.

4.2.2 Utilization Measurements

Utilization measurements are taken under two metrics; CPU usage and memory usage. Memory usage is also divided into two categories, RAM usage and stack usage.

For all four test cases, average core usage values are taken as percentages from the beginning of the scenario to the end.

As the actual units such as servo systems, navigation systems or meteorological sensors that increase the workload of CPU are not included in the benchmark project, measured utilization of the CPU is not as high as it usually is during operation as it remains idle most of the time. However, for the purposes of this study, it is viable to compare the CPU utilization of SCU while using two different communication interfaces.

Table 4.3 shows the CPU utilization measurements (in percentages) for both external interfaces for all four test cases in tabular form. The trend between test cases could be observed in the Figure 4.2.

Table4.3: Extracted CPU utilization metrics

Interface / Scenario	**Test Case 1**	**Test Case 2**	**Test Case 3**	**Test Case 4**
CORBA	0.630	0.653	0.995	0.985
DDS	0.619	0.588	0.588	0.621

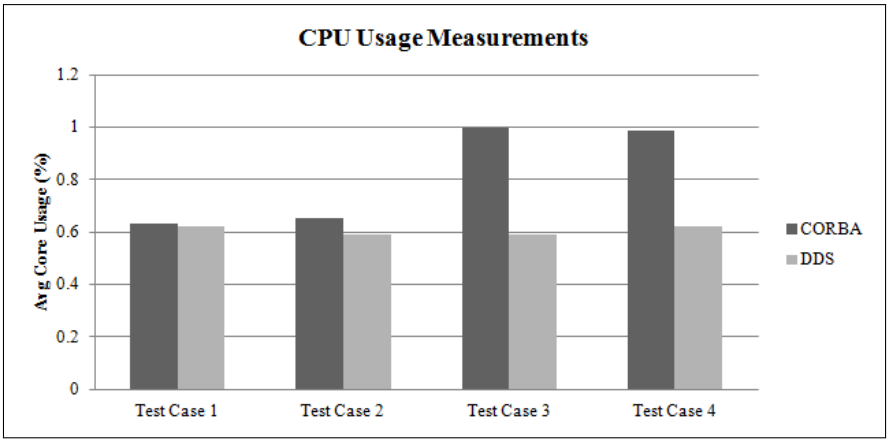

Figure 4.2: CPU Utilization Chart for the Four Test Cases

Considering the utilization aspect of the software, measurements show a consistency in numbers for the case where DDS is used as the external interface, whereas CPU usage of the software increases with the increasing number of peers in the cases where CORBA is used.

In addition to CPU utilization, RAM and stack usage metrics are collected for all scenarios with CORBA and DDS. Peak value of the allocated RAM size and the maximum size of the stack used during laying operation by the software are extracted to demonstrate the effect of using CORBA or DDS as the external interface in the project on memory utilization.

Measured maximum allocated RAM values (in kilobytes) for four test cases with both CORBA and DDS are given in Table 4.4. The trend between test cases in peak RAM usage could be observed in the Figure 4.3.

Table4.4: Extracted RAM usage metrics

Interface / Scenario	**Test Case 1**	**Test Case 2**	**Test Case 3**	**Test Case 4**
CORBA	34135	34142	34168	34171
DDS	29742	29721	29742	29721

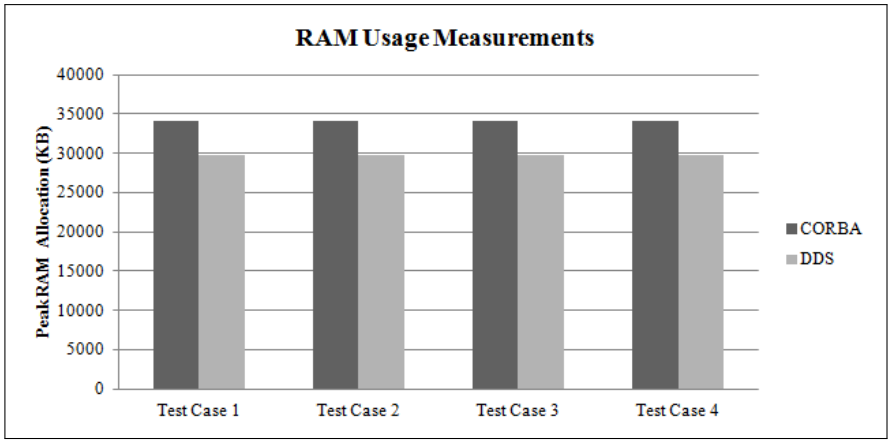

Figure 4.3: RAM Usage Chart for the Four Test Cases

Measured maximum stack usages for four test cases with both CORBA and DDS are given in Table 4.5. The trend between test cases in maximum stack usage could be observed in the Figure 4.4.

Table4.5: Extracted stack usage metrics

Interface / Scenario	**Test Case 1**	**Test Case 2**	**Test Case 3**	**Test Case 4**
CORBA	54352	61312	67984	67840
DDS	50288	50480	52352	51584

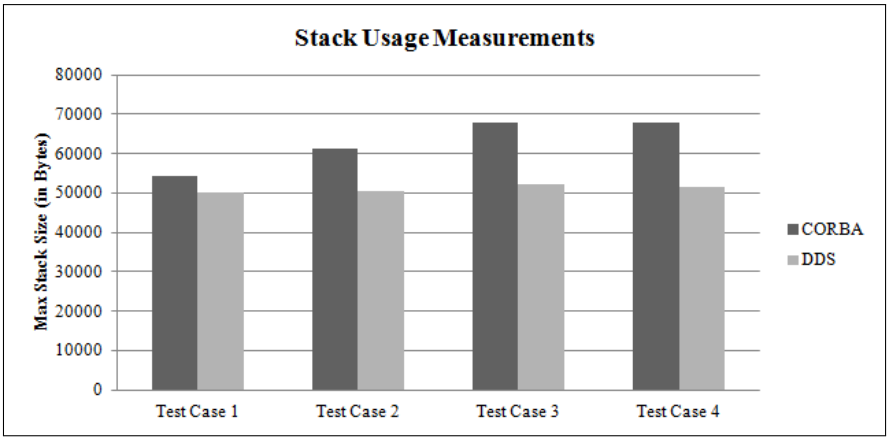

Figure 4.4: Stack Usage Chart for the Four Test Cases

Figures 4.3 and 4.4 show that the memory usage values for the cases with CORBA is higher than the ones with DDS as external interface. Both stack and RAM usages remain around the same values for the cases with DDS. However, stack usage of CORBA external interface increases with increasing number of peers as a new object is created by allocating memory from the stack for each new connection.

CHAPTER 5

DISCUSSION AND CONCLUSION

Distributed systems are widely used in defense industry, thus increasing the importance of the communication method selection. In this study, two alternatives, CORBA and DDS are compared within the application domain and software reference architecture of YMM. Studies concentrating on the performance and possible improvements for DDS and CORBA were conducted separately prior to this thesis. In our study, effects of the selection on the overall software quality and performance are investigated for real-time embedded systems developed in conformance with a software reference architecture.

For the purposes of this study, a benchmark project was prepared. In the development phase of this benchmark project, application domain requirements and software reference architecture of YMM were considered and laying operation was selected as it is a typical and commonly used operation containing both command and data distribution.

Four test cases were designed to animate possible scenarios that the system might come across. Test cases employ different numbers of user interfaces as peers, either as command sources or as display panels used in the project. The first test case represents the simple one to one connection. There exists a single source of command which is also used for displaying the data published from the simulator. Other three test cases represent one to n connection case where n is either two or three. The second case consists of two external applications, one command module and one display module, whereas an additional display module is added for the third case. The fourth test case is designed to increase

the number of command sources and again contains three command and display applications.

Measurements were taken under two main categories, quality and performance.

For quality evaluation, coupling and complexity aspects of the code were measured. Code size is observed to be increasing with the number of peers for CORBA, consequently increasing the effort spent for development and coupling. Cyclomatic complexity measurements showed that the complexity of CORBA for one to one and one to two cases remains lower than the complexity of DDS. However, increasing number of peers gradually increases the complexity of CORBA interface and it surpasses the complexity of DDS interface in the third test case. With higher complexity, code becomes more error prone and less maintainable. Reusability also decreases. Since YMM prioritizes reuse, DDS seems preferable.

For performance evaluation, latency and utilization measurements were taken. When the latency measurements are observed, a slight performance overhead could be noticed in the cases with CORBA external interface. As this time difference occurs at the ends of the scenarios, command distribution is not affected.

In terms of CPU and memory utilization, measurements show an increasing trend through the test cases. While the measured values for CORBA remain close to DDS in the first test case, they are worsened by the increasing number of command and data display modules.

Measurements show that, for the cases that require one to one communication, in terms of software quality, selection of CORBA results in lower code complexity and increased probability of reuse at cost of higher coupling and development effort. When the number of communication nodes increase, a reevaluation is necessary as the complexity of the code implementing CORBA also increases.

We can conclude from the performance measurements that the performance overhead brought up by CORBA creates a slight difference between the two alternatives in terms of latency. When the utilization measurements are evaluated, it can be observed that although the values are close for one to one communication case, the gap widens for increasing number of communications along with

increasing memory consumption of CORBA interface.

In future studies, this work could be significantly improved if more metrics could be obtained from a project with increased number of operational scenarios. Obtaining development effort measurements directly would play an important role in the design and decision-making process. Additionally, including network delays and bandwidth utilization would contribute to the performance evaluation.

In this study, we handled CORBA and DDS and the effects of their usage on the real-time embedded software projects in YMM's application domain separately. A future study could also include complementary usage of these two communication methods, in other words; a hybrid external interface could be built by combining the messages to be sent in CORBA and DDS.

Consequently, we have compared the effects of using CORBA or DDS on command distribution within a project developed in a software reference architecture concerning both performance and quality aspects of the software. It is important that concerns of a software engineering department in a leading defense industry company are highlighted in the aspects of comparison. Therefore this study employs significant data to provide the guidelines for making a design decision on the selection between CORBA and DDS in a real-time embedded software project.

REFERENCES

[1] Data distribution service for real-time systems version 1.2, January 2007. OMG Available Specification, Object Management Group.

[2] Meeting real-time requirements in integrated defense systems an rti whitepaper, 2007. Real-Time Innovations, Inc.

[3] Common object request broker architecture (corba) specification, version 3.3, November 2012. OMG Formal Document, Object Management Group.

[4] S. Agrawal and P. Bhatt. Real-time embedded software systems - an introduction. Technical report, TATA Consultancy Services, August 2001.

[5] H. Al-Kilidar, K. Cox, and B. Kitchenham. The use and usefulness of the iso/iec 9126 quality standard. In *Empirical Software Engineering, 2005. 2005 International Symposium on*, pages 7 pp.–, 2005.

[6] Rafa E. Al-Qutaish. An investigation of the weaknesses of the iso 9126 international standard. In *Proceedings of the 2009 Second International Conference on Computer and Electrical Engineering - Volume 01*, ICCEE '09, pages 275–279, Washington, DC, USA, 2009. IEEE Computer Society.

[7] Mesut Ayata. Effect of some software design patterns on real time software performance. Master's thesis, Middle East Technical University, June 2010.

[8] Kevin Butler, Mark Clement, and Quinn Snell. A performance broker for corba. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, HPDC '99, pages 3–, Washington, DC, USA, 1999. IEEE Computer Society.

[9] Brendon Cahoon, Kathryn S. McKinley, and Zhihong Lu. Evaluating the performance of distributed architectures for information retrieval using a variety of workloads. *ACM Trans. Inf. Syst.*, 18(1):1–43, January 2000.

[10] Antonio Corradi and Luca Foschini. A dds-compliant p2p infrastructure for reliable and qos-enabled data dissemination. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[11] Ulisses Brisolara Correa, Luis Lamb, Luigi Carro, Lisane Brisolara, and Julio Mattos. Towards estimating physical properties of embedded systems

using software quality metrics. In *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, CIT '10, pages 2381–2386, Washington, DC, USA, 2010. IEEE Computer Society.

[12] G. de Souza Pereira Moreira, D.A. Montini, D.A. da Silva, F.R.M. Cardoso, Luiz Alberto Vieira Dias, and A.M. da Cunha. Design patterns reuse for real time embedded software development. In *Information Technology: New Generations, 2009. ITNG '09. Sixth International Conference on*, pages 1421–1427, 2009.

[13] Berkhan Deniz. Investigation of the effects of reuse on software quality in an industrial setting. Master's thesis, Middle East Technical University, January 2013.

[14] Bryan S. Doerr and David C. Sharp. Freeing product line architectures from execution dependencies. In *Proceedings of the First Conference on Software Product Lines : Experience and Research Directions: Experience and Research Directions*, pages 313–329, Norwell, MA, USA, 2000. Kluwer Academic Publishers.

[15] Bruce Powel Douglass. *Design Patterns for Embedded Systems in C: An Embedded Software Engineering Toolkit.* Newnes, Newton, MA, USA, 1st edition, 2010.

[16] C. Esposito, S. Russo, and D. Di Crescenzo. Performance assessment of omg compliant data distribution middleware. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8, 2008.

[17] William B. Frakes and Kyo Kang. Software reuse research: Status and future. *IEEE Trans. Softw. Eng.*, 31(7):529–536, July 2005.

[18] Jerry Zayu Gao, Jacob Tsao, Ye Wu, and Taso H.-S. Jacob. *Testing and Quality Assurance for Component-Based Software.* Artech House, Inc., Norwood, MA, USA, 2003.

[19] Nat Hillary. Measuring performance for real-time systems. Technical report, Freescale Semiconductor, 11 2005.

[20] Cathy Hrustich. Corba for real-time, high performance and embedded systems. In *Proceedings of the Fourth International Symposium on Object-Oriented Real-Time Distributed Computing*, ISORC '01, pages 345–, Washington, DC, USA, 2001. IEEE Computer Society.

[21] Scientific Toolworks Inc. Understand source code analysis and metrics version 3.1. [Online] http://www.scitools.com/, February 2014.

[22] ISO. International standard iso/iec 9126, information technology software product quality part1: Quality model. Technical report, International Standard Organization, 2001.

[23] ISO/IEC. Iso/iec 25010 system and software quality models. Technical report, International Standard Organization, 2010.

[24] Kyo Chul Kang, Moonzoo Kim, Jaejoon Lee, and Byungkil Kim. Feature-oriented re-engineering of legacy systems into product line assets: A case study. In *Proceedings of the 9th International Conference on Software Product Lines*, SPLC'05, pages 45–56, Berlin, Heidelberg, 2005. Springer-Verlag.

[25] R. Karoui. Corba /dds, competing or complementing technologies? Technical report, PrismTech, 2005.

[26] Moonzoo Kim, Jaejoon Lee, Kyo Chul Kang, Youngjin Hong, and Seok-Won Bang. Re-engineering software architecture of home service robots: a case study. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 505–513, 2005.

[27] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[28] Tim Littlefair. Cccc - c and c++ code counter. [Online] http://cccc.sourceforge.net/, October 2012.

[29] V. Mainkar. Availability analysis of transaction processing systems based on user-perceived performance. In *Proceedings of the 16th Symposium on Reliable Distributed Systems*, SRDS '97, pages 10–, Washington, DC, USA, 1997. IEEE Computer Society.

[30] J. Meier, Carlos Farre, Prashant Bansode, Scott Barber, and Dennis Rea. *Performance Testing Guidance for Web Applications: Patterns & Practices*. Microsoft Press, Redmond, WA, USA, 2007.

[31] E.E. Mills, Carnegie-Mellon University. Software Engineering Institute, and United States. Dept. of Defense. *Software Metrics*. Technical report (Carnegie-Mellon University. Software Engineering Institute). Software Engineering Institute, 1988.

[32] Shivakant Mishra, Lan Fei, and Guming Xing. Design, implementation and performance evaluation of a corba group communication service. In *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, FTCS '99, pages 166–, Washington, DC, USA, 1999. IEEE Computer Society.

[33] Parastoo Mohagheghi and Reidar Conradi. Quality, productivity and economic benefits of software reuse: A review of industrial studies. *Empirical Softw. Engg.*, 12(5):471–516, October 2007.

[34] Parastoo Mohagheghi, Reidar Conradi, Ole M. Killi, and Henrik Schwarz. An empirical study of software reuse vs. defect-density and stability. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 282–292, Washington, DC, USA, 2004. IEEE Computer Society.

[35] A.K. Mok. Real-time software design-from theory to practice. In *Computer and Communication Systems, 1990. IEEE TENCON'90., 1990 IEEE Region 10 Conference on*, pages 394–398 vol.1, 1990.

[36] Maurizio Morisio, D. Romano, and I. Stamelos. Quality, productivity, and learning in framework-based development: an exploratory case study. *Software Engineering, IEEE Transactions on*, 28(9):876–888, 2002.

[37] Gerrit Muller. A reference architecture primer, March 2013. Buskerud University College.

[38] Marcio F. S. Oliveira, Ricardo Miotto Redin, Luigi Carro, Luís da Cunha Lamb, and Flávio Rech Wagner. Software quality metrics and their impact on embedded software. In *Proceedings of the 2008 5th International Workshop on Model-based Methodologies for Pervasive and Embedded Software*, MOMPES '08, pages 68–77, Washington, DC, USA, 2008. IEEE Computer Society.

[39] RicardoM. Redin, MarcioF.S. Oliveira, LisaneB. Brisolara, JulioC.B. Mattos, LuisC. Lamb, FlavioR. Wagner, and Luigi Carro. On the use of software quality metrics to improve physical properties of embedded systems. In Bernd Kleinjohann, Wayne Wolf, and Lisa Kleinjohann, editors, *Distributed Embedded Systems: Design, Middleware and Resources*, volume 271 of *IFIP - The International Federation for Information Processing*, pages 101–110. Springer US, 2008.

[40] D. C. Schmidt, A. S. Gokhale, T. H. Harrison, and G. Parulkar. A high-performance end system architecture for real-time corba. *Comm. Mag.*, 35(2):72–77, February 1997.

[41] Douglas C. Schmidt and Fred Kuhns. An overview of the real-time corba specification. *Computer*, 33(6):56–63, June 2000.

[42] S. Schneider and B. Farabaugh. Is dds for you? Technical report, Real-Time Innovations, Inc., 2009.

[43] D.C. Sharp. Avionics product line software architecture flow policies. In *Digital Avionics Systems Conference, 1999. Proceedings. 18th*, volume 2, pages 9.C.4–1–9.C.4–8 vol.2, 1999.

[44] K.G. Shin and P. Ramanathan. Real-time computing: a new discipline of computer science and engineering. *Proceedings of the IEEE*, 82(1):6–24, 1994.

[45] Campwood Software. Sourcemonitor version 3.4. [Online] http://www.campwoodsw.com/sourcemonitor.html, May 2013.

[46] Kelly L. Spicer. A successful example of a layered-architecture based embedded development with ada 83 for standard-missile control. *Ada Lett.*, XX(4):50–63, December 2000.

[47] Stephen Wagner and Zahir Tari. A caching protocol to improve corba performance. In *Proceedings of the Australasian Database Conference*, ADC '00, pages 140–, Washington, DC, USA, 2000. IEEE Computer Society.

[48] Alan Wood. Predicting client/server availability. *Computer*, 28(4):41–48, April 1995.

[49] Tao Zhou, Xiaobo (Sharon) Hu, and Edwin H.-M. Sha. A probabilistic performance metric for real-time system design. In *Proceedings of the Seventh International Workshop on Hardware/Software Codesign*, CODES '99, pages 90–94, New York, NY, USA, 1999. ACM.