Tracing Distributed Corba Applications

Date:	June 2, 2004
Thesis number:	528
Name:	ir. S.W van Swelm
Student number:	0023442
Education:	Informatics
Specialisation:	Information Systems
Institute:	University of Nijmegen
Department:	Software Technology
Internal Supervisors:	dr. M.C.J.D van Eekelen, ir. R.G de Vries
Referent:	dr.ir. G.J. Tretmans
Company:	Thales Naval Netherlands
ID Number:	di519
External Supervisor:	Piet Griffioen
Department:	Combat Systems, Innovation

Table of Contents

SUMMARISATION **5**

PREFACE **6**

1 THESIS INTRODUCTION **9**

1.1 BACKGROUND 9
1.1.1 Products & Technology *9*
1.1.2 Thales History *9*
1.1.3 Thales Middleware Technology *10*
1.2 PROBLEM DEFINITION 11
1.3 OBJECTIVES & RESEARCH QUESTIONS 12
1.4 THESIS ARRANGEMENT 13
1.5 RESEARCH RESULTS 14

2 INTRODUCTION TO COMPONENT TESTING **17**

2.1 COMPONENT BASED DEVELOPMENT TERMINOLOGY 17
2.1.1 DOC Middleware *17*
2.1.2 Components *17*
2.1.3 Component Interface *17*
2.1.4 Component Model *18*
2.2 COMPONENT TESTING DEFINITIONS 18
2.2.1 Component Testing *18*
2.2.2 Component Tracing *18*
2.2.3 Component Testability *18*
2.3 COMPONENT TESTING PROBLEMS 19
2.3.1 Distributed Programs Problems *20*
2.3.2 Debugging Problems *20*
2.4 COMPONENT TESTING ENVIRONMENT 21
2.4.1 Control Framework *21*
2.4.2 Tracing Framework *23*

3 A HIGH LEVEL DESIGN FOR A GENERIC CORBA FRAMEWORK **27**

3.1 TRACING LAYER 28
3.1.1 Trace Retrieval & Processing *28*
3.1.2 Deployment *29*
3.1.3 Management *29*
3.2 INFORMATION LAYER 30
3.3 MONITOR LAYER 30
3.3.1 A New Application of Network Diagrams *31*
3.4 EVALUATION. 32

4 INTRODUCTION TO CORBA MIDDLEWARE **35**

4.1 CORBA OBJECT MODEL 35
4.1.1 The Object Request Broker *35*
4.1.2 Stubs and Skeletons *35*
4.1.3 The Interposable Object Reference *36*
4.1.4 The Interface Definition Language *36*
4.1.5 Corba Services *37*
4.1.5.1 Interface Repository 37
4.1.5.2 Dynamic Invocation Interface & Dynamic Skeleton Interface 37
4.2 CORBA COMPONENT MODEL 37
4.2.1 Component Container *38*
4.2.2 CCM components *39*
4.2.3 Component Home *39*
4.2.4 Corba Communication *40*

5 AN EVALUATION OF TRACE DATA INTERCEPTION TECHNIQUES FOR CORBA **43**

5.1 MIDDLEWARE INTERCEPTION PROGRAMMING TECHNIQUES 43
5.1.1 Debug instrumentation *43*
5.1.2 Pre-processing Instrumentation *44*
5.1.3 Adapter Instrumentation *45*
5.1.4 Stub and Skeleton Instrumentation *45*

5.1.5 Middleware Instrumentation.....45
5.1.6 Interface Wrapping.....45
5.1.7 Network Message Tracing.....46
5.2 CORBA META PROGRAMMING MECHANISMS 46
5.2.1 Request Interceptors.....47
5.2.2 Smart Proxy.....51
5.2.3 Servant Manager.....53
5.2.4 Pluggable Protocol.....53
5.3 COMPARISON OF TRACE DATA INTERCEPTION TECHNIQUES 55

6 AN EVALUATION OF CORBA TESTING TOOLS.....59

6.1 CRITERIA FOR COMPARISON OF TEST FRAMEWORKS 59
6.1.1 Controllability Functionality.....59
6.1.2 Message Observability.....59
6.1.3 Tracing Framework Performance.....60
6.1.4 Interoperability Functionality.....61
6.2 CORBA TEST TOOLS UNDER EVALUATION 62
6.2.1 DSC toolkit.....62
6.2.2 Corba Trace.....66
6.2.3 The Silk Testing Toolkit.....70
6.2.3.1 Silk Observer.....70
6.2.3.2 SilkTest.....72
6.2.3.3 Silk Performer.....72
6.2.4 Ethereal.....73
6.2.5 MCITT.....74
6.3 AN EVALUATION OF CORBA TEST FRAMEWORKS 75
6.3.1 Controllability Functionality.....75
6.3.2 Trace Observability Functionality.....76
6.3.3 Tool Performance.....77
6.3.4 Tool Interoperability.....78
6.3.5 Final Comparison.....79

7 AN ANALYSIS FOR A TRANSPARENT CORBA TRACING MECHANISM.....83

7.1 TRACING DATA INTERCEPTION 83
7.1.1 Message Communication ID.....83
7.1.2 Timing Data.....84
7.1.3 Context Data.....86
7.1.4 Component Instance ID.....87
7.1.5 Message Contents Data.....87
7.1.5.1 A New Transparent Proxy Mechanism for tracing message contents 89
7.2 TRACING MECHANISM DEPLOYMENT.....92
7.3 TRACING MECHANISM MANAGEMENT.....93
7.4 TRACING MECHANISM PROCESSING 94
7.5 EVALUATION.....95
7.6 TOOL DEVELOPMENT RECOMMENDATIONS 96

8 CONCLUSION.....97

BIBLIOGRAPHY 101

APPENDIXES.....105

APPENDIX A: IOR.....106
APPENDIX B: GIOP/IIOP.....108
APPENDIX C: CORBA TELECOM LOGGING SERVICES 111
APPENDIX D: CORBA SCRIPTING LANGUAGE.....112
APPENDIX E: TCL SCRIPTING LANGUAGE.....113
APPENDIX F: LOGICAL CLOCK LIMITATIONS.....114
APPENDIX G: REQUEST & REPLY FLOW 115
APPENDIX H: REQUEST INFORMATION 116
APPENDIX I: SERVICE CONTEXT DATA 117
APPENDIX J: PICURRENT.....118
APPENDIX K: INSTALLING INTERCEPTORS 119
APPENDIX L: IOR INTERCEPTORS 121
APPENDIX M: HISTORY OF THE PORTABLE INTERCEPTOR 122

Summarisation

The software industry is realising the importance middleware for the construction of large complex distributed systems. Corba is a distributed object middleware technology, which allows the communication of distributed objects over a shared network and can be used for construction of distributed systems. Distributed Systems must just like regular system be tested to validate their functionality. However, unlike regular single process applications, testing a distributed system is a lot more difficult. Distributed computing environments usually consist of several physical machines with different hardware configurations, having installed different operating systems and middleware software, with different characteristics of the network connections between them. Unfortunately, traditional tracing mechanisms were usually developed for use in single-computer environment. Language level debuggers are therefore not very suitable for testing Corba applications because they are bound to a specific programming language, have no knowledge of components and manage distributions in proprietary ways.

In order to validate a distributed system, the tester must be able to observe the inside of a distributed system by tracing the internal communication events between all distributed objects. However, the internal communication in a distributed object middleware is normally hidden from the outside world. We therefore need tracing mechanisms, which can extend the middleware with tracing functionality. In order for a tracing mechanism to collect tracing data, we need to know which techniques are available to intercept communication from a distributed system. Our thesis will therefore evaluate several Corba techniques for interception tracing data. Besides conventional tracing programming techniques, we also looked at tracing potentials of existing Corba meta-programming mechanisms. All tracing techniques are eventually compared based on the characteristics, which are valuable for the construction of a tracing mechanism. For one of the meta-programming techniques, the portable request interceptor, we analysed how the technology could be used for the construction of a fully transparent tracing mechanism.

Because today's system consists of highly heterogeneous distributed computing environments, a trace of a distributed application can cross multiple technology boundaries. Our thesis addresses this problem by introducing a high level design for generic tracing framework. The generic tracing framework is designed to allow multiple middleware tracing mechanisms to collect tracing data and present it to the tester from a single monitor interface. Besides the design for a new testing tool, we also looked at several existing tools, which are designed to test Corba applications. Our thesis contains a detailed evaluation of several Corba testing tools, analyses their tracing mechanism and compares them based on a comprehensive list of criteria.

Preface

This report is the result of project performed at Thales Naval Netherlands in Hengelo. This report also fulfils the role of thesis for my graduation in Computer Engineering at the University of Nijmegen.

The reason I participated in this in project is that it was in line with my chosen graduation direction and because of my interest for middleware technology.

Many people contributed to my project for which I would like to pay my gratitude:

- First, I would like to thank my roommates, who never gave me a dull moment at Thales. The overall experience at Thales was enjoyable and undoubtedly a very valuable experience for my career.
- Second, I would like to thank my supervisors from the University, Marko van Eekelen, Rene de Vries, and Jan Tretmans, for their time, interest, and their wisdom. They provided valuable feedback and constructive criticism on my work, which helped improving it.
- Furthermore, I want to express my deep personal thanks to my supervisor at Thales, Piet Griffioen for his patience, and for being and motivation inspiration during time at Thales. It was very pleasant working with him and he was always helpful in discussing about my work.
- Special thanks to Harrold J. Batterham of Lucent Technologies for providing the DSC Toolkit and for answering my questions about tracing.
- Last, but not least I want to thank my family for their support and motivation during my work on my Thesis.

1 THESIS INTRODUCTION

In this chapter has the following structure:

- *Paragraph 1.1: Background*. In the first paragraph, we tell something about the background information of our thesis at Thales Netherlands.
- *Paragraph 1.2: Problem Definition*. In this paragraph, we explain the Corba testing problem and why additional testing techniques or test tools are required.
- *Paragraph 1.3: Objectives & Research questions*. In this paragraph, we explain on what main questions we have tried to find an answer during our internship.
- *Paragraph 1.4: Thesis Arrangement*. In this paragraph, we will give a short overview of how we arranged the chapters, what to expect and how the thesis can be read.
- *Paragraph 1.5: Research results.* In our final paragraph of this chapter, we discuss the results of thesis and its value for Thales.

1.1 Background

Because we conducted our research at Thales Netherlands, we will first tell sometime about Thales their products and technology, history and plans to use Corba middleware as part of their middleware architecture.

1.1.1 Products & Technology

Thales Netherlands creates high-tech defence solutions for naval and ground based environments. They combine their extensive and long experience with ongoing search for new techniques and technologies. This has resulted in a vast expertise in the field of radar, infrared, weapon control, display technology, communication equipment, and software support. The product range of Thales Naval Netherlands compromises system suitable for all classes and types of naval vessels, any weapon systems and any mission. Modern and highly capable sensor suites, together with the combat management system TACTICOS, equip new generation of frigates, corvettes and fast attack craft throughout the world. The naval capabilities include sophisticated anti-air warfare systems featuring APAR, GOALKEEPER, SMART-L and SIRUS. Thales Ground Based Systems provides solution for integrating air defence surveillance, track, and fire control purposes, as well as for border and battlefield surveillance purposes.

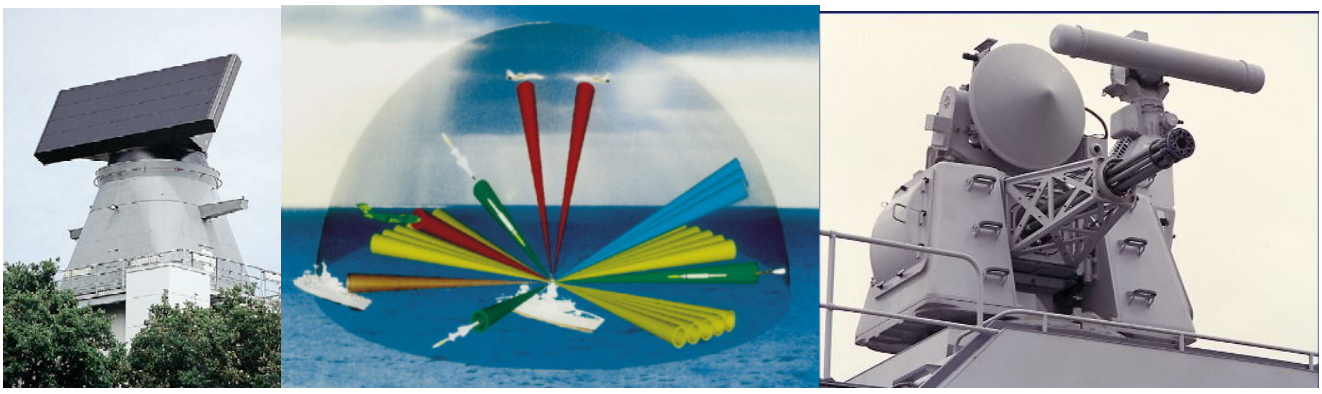


Figure 1.1: Some of Thales products: (from left to right): SMART-L, APAR, GOALKEEPER

1.1.2 Thales History

Thales Netherlands history goes back a long way:

- In 1922, Thales Netherlands was founded under the name “NV Hazameyer’s Fabriek van Signaalapparaten” to produce fire control equipment. The company grew rapidly and welcomed costumes from Sweden, Spain, and Greece.
- During World War II, the German army captured the factory virtually intact. Fortunately, a large number of the staff were able to escape to the United Kingdom and continued to work on there on fire control systems
- After the war, the Dutch government bought the factory and continued the company under the name “N.V. Hollandse Signaalapparaten”. In those years a lot of new techniques and technologies were developed, such as radar, fire control, computers, and air control equipment.

- In 1956, PHILIPS bought a large part of the shares from the government and became the main shareholder. The company flourished and opened plants in several cities across the country. Near the end of the eighties, Signaal employed over 5000 people and served customers in over 35 countries.
- After Cold War ended, Signaal was forced to reorganise and reduce staff due to the changed political theatre. Meanwhile PHILIPS decided Defence and Control Systems were not part of it core-business and therefore sold it in 1990 to Thomson-CSF. After the reorganisation and merger with Thomson-CSF, the company regained a new driving force. New systems were deigned, taking a leap in defence equipment and combat system.
- In 2000, Thomson-CSF changed its name to Thales. Being a member of this group, Thomson-CSF Signaal changed its name to Thales Netherlands.

1.1.3 Thales Middleware Technology

In the last twenty years, there has been a trend from custom build software and hardware systems to Commercial Off-The-Shelf (COTS) based systems. The practice of COTS is known in the engineering world to reduce cost, time, and risk. Building software systems based on COTS also offers several other advantages like the availability of tools and people specialised in standard software technologies. Because the advantages of COTS, the market is increasingly, demanding generic software solution based on COTS standards.

Customers of Thales are no exception to these market changes are demanding software solutions build with middleware standards. The role of the middleware is to ease the task of designing, programming and managing distributed applications by providing a simple, consistent and integrated distributed programming environment. Essentially the middleware is a distributed software layer/platform, which abstracts over the complexity and heterogeneity of the underlying distributed software with its multitude of network technologies, machine architectures, operating systems, and programming languages.

Thales originally developed all their middleware based on their own system requirements. Thales middleware solution, called SPLICE, is a data distribution software product designed specifically to let the operational software processes (TACTICOS) exchange data with each other and with Thales radar systems like APAR/SMART-L. SPLICE is a real-time distributed database management system (RDBMS) based on the publish-subscribe paradigm. In a publish-subscribe paradigm, client and server process are decoupled from each other and communication is one directional. A client process therefore does not communicate directly with a server process directly (point to point) but indirectly trough Topics. The SPLICE middleware takes care of delivering Topics published by servers to clients processes subscribed on a that Topic. Multiple server processes can publish the same Topics while a client can subscribe on multiple Topics. The advantage of this mechanism is that there is no single point of failure. For example, if a radar system on a ship fails to produce track data, redundant radar systems can transparently take over it functionality without having to reconfiguring the clients subscribed on this track data.

In the light of customer standardisation requirements, Thales wants to evolve their distributed system based on proprietary middleware to software systems base one generic middleware standard. However, some of Thales systems have very high performance and technical requirements, which could not be replaced by equivalent middleware technology. Thales Naval builds, very large complex systems with very diverse performance requirements. Therefore based on the system performance requirements (real-time, near real-time and non real-time), Thales divided their system architecture in three separate segments; Command Execution, Command & Control and Command Support. Combat Execution, is used for tactical operations like controlling weapon systems, Combat Control is used for radar systems and Combat Support for administrative systems. In figure 1.2 we can see that each segment consists of tree separate layers, the application layer, the business layer, and the data layer. This three-tier architecture, which makes an abstract distinction between presentation, business logic, and information storage, allows Thales the construction of large, scalable, reusable systems.

Although SPLICE is a highly effective versatile software product, it was never intended as a standard. However, the fast response times and fault tolerance requirement that are crucial for the effectiveness of Combat Execution, could not be replaced by any exiting COTS middleware solution. Thales therefore is currently in the process of standardising their SPLICE technology (SPLICE-2) as the new Data Distribution Standard (DDS) standard at the Object Management Group (OMG). The OMG is a consortium, which includes over 800+ companies represents the entire spectrum of the computer industry.

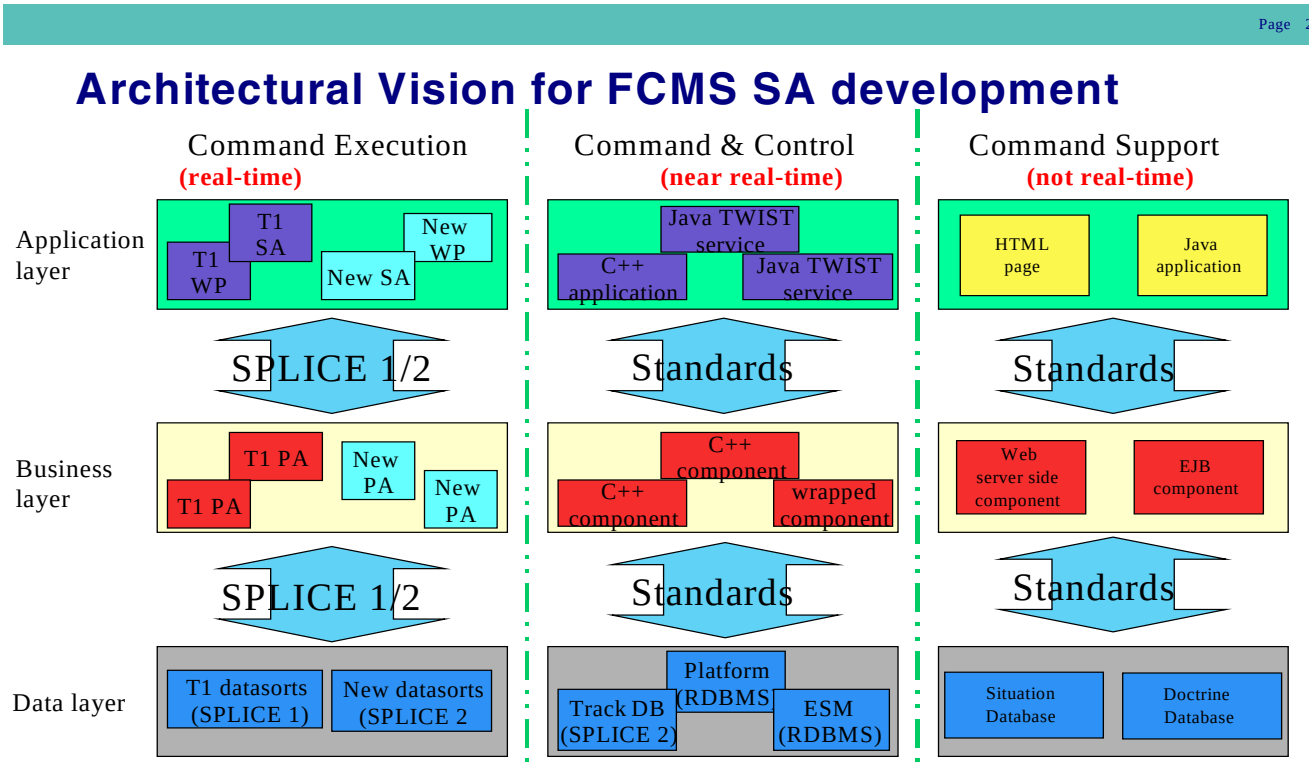


Figure 1.2: Thales Middleware Segment Architecture

The Command & Control, and Command Support segments one the other hand, have much lower performance requirements and can be replaced by existing COTS software solutions. One of these COTS middleware solutions found suitable to replace SPLICE middleware is the recently adopted OMG Corba Component Model (CCM). Because there was no suitable implemented version of CCM, Thales had to develop lightweight implementation of CCM themselves, which they called PERCO. In order to support the same functionary as SPLICE, PERCO also implements many other services like fault tolerance (FT-Corba) and Real-time functionality (RT-Corba). Thales plans to integrate PERCO in its development process and is currently looking at new tools, which can support the construction of high quality software.

1.2 Problem Definition

In the previous paragraph, we explained why Thales wants to use Corba middleware technology in their system architecture. In this paragraph, we explain the challenges Thales is faced with in respect to the integration of Corba technology. Although testing services are on the PERCO roadmap, other services currently have a higher priority. However, the construction of reliable Corba applications requires more than the availability of a CCM middleware implementation, they also require the tools to develop them.

Verifying Corba applications consisting of multiple individual software modules running in a distributed environment is not an easy task. Corba software modules collaborate with each other in complex ways to achieve the application goals. Although Thales SPLICE applications have to achieve similar goals, in contrast to Corba, SPLICE processes are a lot easier to diagnose. While it is sufficient for testers verify SPLICE application functionality by monitoring input and output of the individual Splice processes, a tester would quickly lose track of the overall distributed functionality when trying to diagnose a Corba application in a similar way. This is because Corba message communication is in contrast to Topic communication, a lot more diverse and context sensitive.

Testers therefore need to diagnose a distributed system at the abstraction level of a distributed system. To verify the correct behaviour of a distributed system, we need a tracing mechanism, which can trace an application ‘step by step’. Tracing a Corba application ‘step by step’ allows the tester not only to verify behaviour and integrity of each separate Corba process, but also the interactions between the processes. While SPLICE Topic messages that can are retrieved from the Splice RDBMS with a Splice tool, there is no similar tool available for Corba. Corba messages communication only travel directly or indirectly trough services between Corba processes and will be lost after a message reached its destination. Our tracing mechanism therefore needs to intercept the communication before it becomes lost.

In order for the tracing mechanism to intercept the Corba message communication flowing through a Distributed System, the system must be extended with tracing functionality. However, care should be taken not affect the functionality the existing distributed system or the development new distributed applications. The testing mechanism should therefore be as transparent as possible, which means that it should be able to test Corba application without altering its source code.

Although one would expect that tracing is hot topic in Corba, tracing currently does not receive the attention it deserves in the Corba community. Although the OMG has specified a Corba Scripting language [IDLScript] for testing Corba Applications, none of their specification documents mention the words monitoring or tracing anywhere and neither is there any OMG special interest group investigating Corba the testing problems. Instead, the OMG Corba community is mostly focused on adding additional middleware functionality like fault tolerance and security services.

1.3 Objectives & Research Questions

Our initial goal was to find a solution for “Thales Corba testing problem”. More specifically, we wanted to find the answer on the question: “What is required to test Thales Corba Applications? ”. Because Corba is a standard middleware product, the same research question is therefore also be applicable outside Thales. We can therefore rephrase the question to “What is required to test Corba system? ”. However the answer to this question soon turned out to be too diverse, therefore we narrowed the thesis question down to the part which proved to be most interesting aspect of test Corba application, which is “How to Trace Corba applications? ”. Because this report is mainly written as an advice for Thales Netherlands, we divided the thesis research question into two research questions, which is “how to test Corba applications with existing tools?” and “how to create a new Corba application tracing tool”. Each of these questions is further divided further in sub questions:

How to test Corba applications with existing tools?

- What are the problems when tracing Corba Applications? We answer this question by looking at the common problems when testing distributed applications.
- What should a good Corba tracing tool be able to do? We answer this question by listing the criteria for our ideal tracing tool.
- What tools exist for testing Corba applications? We will answer this question by reviewing several tools and compare them with each other based on the criteria we already set.

How to create a new transparent Corba tracing tool?

- What middleware techniques could be used to trace Corba Application? We answer this question by listing available techniques for tracing distributed applications and compare them with each other based on the characteristics, which allow the construction of a good tracing mechanism.
- What tracing facilities does Corba middleware facilitate? We answer this question by reviewing the facilities available in the Corba middleware possibly could be used for tracing.
- How can we build a transparently tracing mechanism? We answer this question by analysing the possible usage of interceptors for gathering tracing information transparently.

Our thesis is focused on finding a solution for the above-mentioned research questions, which should allow Thales Netherlands get a better understanding of what is needed to test their Corba Applications. The approach for our thesis was therefore based on defining research questions, answering them to the best of our ability and evaluates them of their usefulness.

1.4 Thesis Arrangement

Although our thesis is arranged in such a way that it be read from begin to end, a reader familiar with Corba middleware and Corba tracing techniques can skip directly to the chapter of interest. Because Corba applications are essentially distributed applications, we often generalise the Corba tracing problem to a generic tracing problem, where applicable. Our chapter and paragraphs are therefore arranged in such a way that we first describe the generic case before we discuss Corba specific characteristics.

In figure 1.3, we can see that our thesis is arranged in such a way that each chapter build further on the contents described of the previous chapters. To improve the reading experience of the reader we located technical details, which are not of immediate importance for the reader in the appendixes.

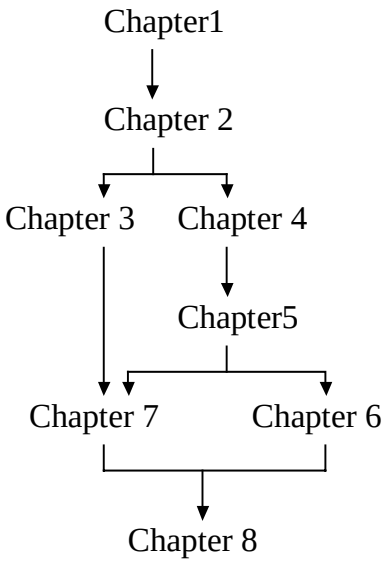


Figure 1.3: Chapter hierarchy

Chapter 1 to 8 contains the following information:

- *Chapter 1: Introduction*. The first chapter serves at the introduction to the rest of the thesis. The introduction includes a brief description of the Thales background story, problem definition, research questions we will answer during the length of this chapter and research results.
- *Chapter 2: Introduction to Component Testing*: This chapter serves as an introduction to components testing. We start by explaining the main concepts used in component based software and discuss the terminology used for component testing. We then explain the difficulties associated with testing Component Based Applications and why we need a component-testing framework. We end the chapter by explaining the main characteristics of a component testing framework
- *Chapter 3: A high Level Design for Generic Tracing Framework*. In this chapter, we introduce a generic tracing framework, which would Thales to trace their components in a heterogeneous distributed environment. We discuss the generic architecture and tasks that should be fulfilled by the framework.
- *Chapter 4: An Introduction to Corba Middleware.* In this chapter will introduce the Corba middleware technology. We first explain the main functionality and concepts used by Corba Object Model and then introduce the Corba Component Model, which build further on top of Corba Technology.
- *Chapter 5. An Evaluation of Trace Data Interception Technique for Corba*. After the introduction of Corba technology, we investigate the common middleware programming techniques and Corba meta-programming techniques to collect tracing data from a distributed Corba application.
- *Chapter 6. An Evaluation of Corba Testing Tools*. In this chapter, we will investigate several COTS solutions, which can be used to trace Corba systems.
- *Chapter 7. An Analysis for a Transparent Tracing Mechanism*. In this chapter, we investigate how the Portable Request Interceptors can be used to realise a transparent tracing mechanism for intercepting tracing data in a distributed Corba system.
- *Chapter 8. Conclusion & Recommendations*. In the final chapter of this thesis, we will summarise our main conclusions and give recommendations for the Corba testing problem.

1.5 Research Results

Overall, our thesis serves as an investigation of the problems and solutions for testing Corba systems.

In the context of our investigation, we have made the following research results:

- *In paragraph 2.4.2*, we give an overview of the most important tasks and problems of a tracing framework and we give an insight in potential application for a tracing framework in the development of distributed systems. This insight will be valuable for developers that want to get a wider perspective of the benefit of a tracing framework.
- *In chapter 3*, we introduce a high level design for generic tracing framework, which can serve a generic architecture for collecting tracing data in a heterogeneous environment. This design will allow the reader get a better insight in the problem and challenges a generic tracing framework is faced with.
- *In paragraph 5.1*, we made a list of conventional middleware programming techniques, which can be used to retrieve tracing data from a distributed system. This information will be useful for any middleware developer that wants to understand the advantages and disadvantage of programming techniques, which can be applied on any distributed object middleware technology.
- *In paragraph 5.2*, we reviewed several Corba meta-programming mechanism in the context of tracing. This insight will be valuable for any Corba developer that wants to understand the possibilities and limitation of meta-programming mechanism in the construction of Corba middleware services
- *In paragraph 5.3*, we made a comparison of all discussed trace data interception techniques and compare them with each other. This insight will allow the reader to get an overall picture of the advantages and disadvantages of the discussed techniques for the construction of a good tracing mechanism. The criteria used for our comparison can also serve as instrument for measuring the tracing value of programming techniques, which are not included in our evaluation.
- *In chapter 6*, we describe several COTS Corba testing tools, which we compared with each other based on criteria desired for a good testing framework. Special attention has been given to a new tracing tool, which has recently become available on the market. This information will be valuable for testers that a looking for a good Corba testing tool. The criteria used for the comparison can also be used as an instrument for evaluating other tools, which are not included in our evaluation.
- *In chapter 7*, we made an analysis of the application of the portable request interceptor for the development of a transparent tracing mechanism. This information allows the reader to understanding in the problems and possible solutions a developer is faced with when developing a tracing framework.

2 INTRODUCTION TO COMPONENT TESTING

In this chapter, we will explain why and how we need to test Corba applications.

We will discuss the following topics:

- *Paragraph 2.1: Component Based Development Terminology*. In this paragraph, we will introduce the reader to some important terminology used in component-based development.
- *Paragraph 2.2: Component testing Definitions*. In this paragraph we introducing the reader to component testing is and how we approach it.
- *Paragraph 2.3: Component testing Problems*. In order to get a better understanding of component based testing we explain what the problems of testing distributed applications and why traditional testing techniques are not effective.
- *Paragraph 2.4: Component testing Environment*. In this paragraph, we will explain what is required to create a component testing environment which can help the tester verify component based application.

2.1 Component Based Development Terminology

Because the goal of our thesis is to test component-based application made in Corba middleware, we will first discuss the main concepts used in Component Based Development (CBD).

2.1.1 DOC Middleware

One of the most popular middleware platforms is Distributed Object Component (DOC) Middleware. DOC Middleware uses an object-based programming model in which application applications are structured into potentially distributed objects (also called components) that interact via location transparent method invocation. Prime examples of this type of middleware are OMG's Corba, Microsoft's DCOM and .Net.

DOC Middleware solutions usually offer the following standard mechanisms:

- *An Interface Definition Language (IDL)*, which is used to abstracts over the fact that objects can be implemented an any suitable programming language.
- *An Object Request Broker (Orb)*, which is responsible for transparently directing method invocations to the appropriate target object (servant).
- *A set of common object services* (COS), which further enhance the distributed programming environment.

In chapter 4, we will explain these mechanism in more detail.

2.1.2 Components

In contrast to normal objects, components are self-sufficient pieces of software that can inter-operate via standard interfaces across networks, applications, languages, and platforms. However, in practice this interoperability has not always happened. Certain manufactures of object technology like Microsoft (.NET) and Sun (JAVA Beans) have their own standards and use it primarily to simplify their development for their own operating system. A component is similar to a class in the sense that it defines the behaviour and structure of a software component instance. Components are often implemented in the form of a module, which itself is an executable. A running instance of such an executable could therefore be called a component instance.

Components are also a way to encapsulate functionality available for reuse in other environments. They can offer the services of a traditional piece of code by wrapping around it and provide the interface for other services to use. Conceptually a software component is a software element that must conform to a component model, which can be independently deployed and composed without modification according to a composition standard. For software components to be independently deployable, they need to be clearly separated from their environment and from other components. To accomplish this, a software component encapsulates its implementation and interacts with its environment through well-defined interfaces, which we explain in the next paragraph.

2.1.3 Component Interface

A component interface provides methods to operate on and access to the public data within a component. To enable reuse and interconnection of components, component producers and consumers often agree on a set of component interfaces before the components are designed. A component interface therefore serves as a binding contract between a software component and its clients. The interface obligates the component to provide a certain set of services and tells its clients how it may use these services. Additionally, the interface may define certain constraints on the usage of these services, which both the component and its clients must adhere to. Although a component developer can decide to deploy the source code with the component,

developers of COTS components only want the customers to view the component by its interface definition language (IDL). A component interface could therefore also serves as data hiding mechanism that masks the component implementation from the component consumer.

2.1.4 Component Model

A component model defines how a component's behaviour is described by means of interfaces, other non-functional specifications, and appropriate documentation. Some components models, like Corba are also called open component standard. A component model is called an open standard if multiple vendors can implement the same component model. An advantage of open standard is that it can be used by any organisation that wishes to create a component model implementation. A disadvantage of an open standard is that components written for a specific component-model implementation might not always be compatible with other implementation component-model. The reason for this incompatibility between component model implementations is because of the competition between suppliers. Suppliers of component model implementation try to win customers by offering extra non-standardised features. If a Component Model is open standard, suppliers can also provide customers with the source code, which would allow enterprises to create a customised version of a component model implementation. The component model implementation must provide the necessary software to allow inter-component communication. The component model implementation is the dedicated set of executables software elements necessary to support the execution of components within a component model. A component model implementation can therefore act as a middleware technology in a three-tier architecture. In the chapter 4, we introduce Corba object model, which forms the basis for the Corba Component Model (CCM).

2.2 Component Testing Definitions

The history of software development has shown that there is insufficient attention for the testing. This while it should have been an integrate part of the software development process in the first place. Component Based Software is still a relatively young technology. While there are many books written about testing traditional software, there is currently not a single book about Component testing. In addition, relatively few papers address the problem of testing component-based software [Whitehead]. Because there is not a standardised terminology in component testing, the literature uses different terms for the same component testing concepts. To prevent further confusion we will first give some definitions of the most frequently used terms.

2.2.1 Component Testing

There are many ways of testing Component based systems. By Component testing, we specifically like to mention that we do not mean testing the DOC Middleware that facilitates the communication e.g. the Component Model, but rather the Component Based Applications. For example, the Term Corba Testing often abused by tools that claim to 'Test Corba'. However, they only test correct Corba Middleware behaviour, not functional Corba Component behaviour. What we do mean with testing Components is testing functional behaviour of individual Component in co-operation with each other in a component architecture environment. Testing component in co-operation with each other can be accomplished by tracing Component interaction, which will be discussed in the following paragraph.

2.2.2 Component Tracing

The word 'tracing' can often be a confusing term since it can be used for a wide range of meanings. A general definition [Mann] for tracing is "a step-by-step execution of a software system conducted in order to gain extra information or insight on how a system works, and which is not part of the normal execution output". The definition leaves a lot room for different interpretations since it's not specified what is meant by 'step by step execution'. This was left intentional under-specified to make the definition scalable for a wide range of purpose. For example, a 'step' might be represented by a very low-level step (like machine code instruction) or very high-level step (communication between computers). Further 'step-by-step execution' does not necessarily mean the system must halt between every step but it does require the ability to observe the trace. In our distributed context, we are mostly interested in the lowest grain available in a distributed system, which are the communication event in a distributed system. Thus by tracing a distributed system, we mean tracing component interaction, e.g. monitoring the communication between components. Monitoring is essentially a less strict form of tracing since it does not require a 'step-by-step execution'. Monitoring is often associated with following a process state online, over longer time period. Therefore, we prefer the term 'tracing' above the term 'monitoring' when observing the 'step-by-step' execution of a component-based application.

2.2.3 Component Testability

Testability is the ability to generate, evaluate, and apply tests to improve quality and minimise time-to-profit. Software systems with a high testability simplify the software testing process. It quantifies the extent to which a

design or a fielded system can be tested for the presence of manufacturing defects or failures. A testable system implies better fault coverage and fault isolation, shorter testing times, higher product quality, shorter time-to-market, and lower life-cycle costs. In other words, the higher the testability, the better the tester will be able to guarantee the overall quality of a software system.

Software system testability [Testability] can be increased from two viewpoints:

- *Controllability* of a System indicates how easy it is to control the software system under test, how easy is it to simulate a testing scenario, triggering the desired operation and collect the retrieved output. High controllability can be achieved by using a controlled test environment (see figure below), which can execute a test case without interference of the outside environment. Creating a controlled component testing environment will be further discussed in paragraph 2.4.
- *Observability* indicates how easy it is to observe a program in terms of traceability and how easy it is to relate invoked inputs with its observed outputs on a component interface. The traceability of a component-based application refers to the extent of tracing the application internal behaviour. In our thesis, we are mainly interested in observing internal communication between components (see figure 2.1), which allows us to diagnose distributed system. Therefore, techniques that allow the verification of testing input with observed output fall outside the scope of our thesis.

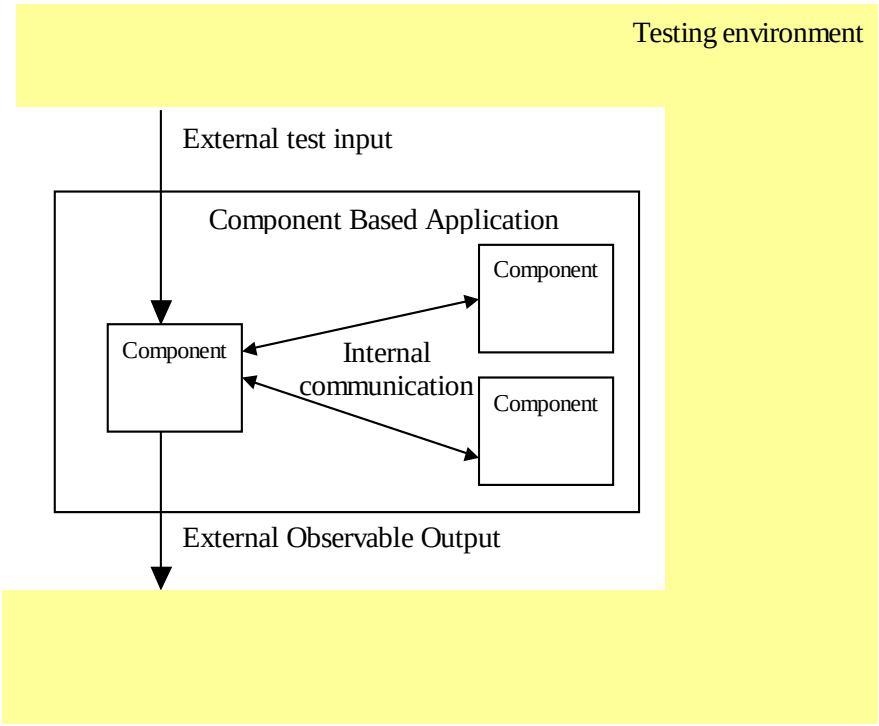


Figure 2.1: Observability & Controllability of Component Based Applications

2.3 Component Testing Problems

In this paragraph, we discuss some of the problems when testing in a distributed environment and why traditional software testing techniques do not work when testing component-based applications. Although Software components are supposed to be reusable in different environments, testing reused components is often an underestimated task. Due to lack of time, testers are often forced to select test cases based on the highest probability of finding faults in their targeted environment and are therefore based on the most commonly occurring inputs in a particular environment. Since these test cases can only verify a very small percentage of all possible test cases, a test case will only be valid in that particular environment. The assumption that that a tested component made in one particular component infrastructure is also valid in a new component infrastructure is therefore flawed. An example where this assumption had devastating consequences was in the infamous Ariane 5 rocket malfunction incident. The Ariane 5 rocket-guidance system reused software components from the Ariane 4 program, but their internal mechanism was not properly documented. Consequently, after a single uncaught exception caused by an integer overflow, resulting the entire control system shut down in mid-air.

Distributed systems constructed from COTS component are inherently even more complex than systems that are designed and written from scratch. One of the reasons for this is because of the enterprise heterogeneous nature of components and interfaces. The third party components often evolve in a way that is not under the control of the developers. This may include changes at the interface, requiring subsequent changes of the distributed system in order to stay compatible with newer components. Distributed systems composed of components from different vendors are particularly prone to problems, due to incompatibilities of versions, subtle differences in interfaces, and implicit requirements of the components, which are frequently undocumented. This is not surprising since there exists no standard way to describe the overall structure of such applications. Because structural information is captured only in source code, where it is not available for high-level analysis and documentation, the testability of a distributed system is further reduced.

2.3.1 Distributed Programs Problems

Distributed programs operate in a completely different environment than the single process environment. Besides all of the well-known problems, that makes tracing traditional single computer environment difficult to test, distributed software suffers from a whole range of extra problems which are part of the nature of distributed systems.

Some examples of the common problems [Scallan] in testing distributed systems are:

- *Deadlocks* can appear because of faults in synchronisation protocol between components, which prevents each other from completing their tasks. Deadlocks often only appear under special conditions and are therefore difficult to locate.
- *Control flow design errors,* especially exceptions and failures with multiple modules can be difficult to detect. In comparison to single process application, the control flow through a distributed application is usually much more complex, leading to a variety of design errors.
- *Race conditions* can occur when parallel working components of a distributed application are not properly synchronised to prevent different components from producing contradictory results. These synchronisation errors, usually the result of latencies caused by the network or by other processes, tend to occur sporadically and are not easy to reproduce.
- *Timeout failures* resulting from delays and bottlenecks in the network can cause distributed parts of an application to time out and produce failures. These failures may propagate through the rest of an application if not handled properly.
- *Performance bottlenecks* can appear in a distributed application when a complex operation is performed by time critical process, which can substantially slowdown application overall application performance resulting in race conditions.
- *Network failures* can often afflict a complex network. When network failures occur, it is important that the tester is to be able to detect and circumvent each point of failure.
- *Network resource limitations* can cause a distributed system to fail when the size of a system is ramped up. These scalability problems might not occur within single component testing configurations, but only appear at later stage deployment in the form of limited connections or insufficient bandwidth.

2.3.2 Debugging Problems

In traditional single process environments, tracing is often achieved by stepping through the application with a debugger tool. This tracing technique commonly referred a debugging, allows a programmer to trace an application at the level of abstraction of the programming language. Usually, compilers implement debugging by adding extra debugging information to the machine-level code so that every tracing step corresponds to a single instruction in the source code. When debugging is activated, the inserted debugging information raises an interrupt call at the instruction boundaries. After every interrupt call, control returns back to the debugger allowing the programmer to check the current state of the system before executing the next instruction. By repeating this debugging process, a programmer is able to walk through a single process-program “step-by-step”.

Unfortunately, this tracing mechanism will not work in component-based systems. In order to diagnose the problems mentioned in the previous paragraph, testers cannot simply debug a distributed system as if it was conventional single application. Distributed objects reside on separate systems, each maintaining their own memory and connections with other distributed objects. Moreover, distributed objects are implemented on different operating systems and written in different languages. Therefore in order to set up a step-by-step test through a distributed application with a debugger, testers would have fit every message entry and exit point among numerous processes in the distributed system with a separate code debugger. Although it is possible to set up debuggers at every process and step through a distributed application this way, it would be very time consuming and awkward. Detecting scalability problems with debuggers would be outright impossible because

of the large number of processes debuggers required. In addition, because every debug step effectively halts the distributed system, timeout problems can occur resulting in the total crash of distributed application.

Beside all the practical problems of debugging, debuggers would never be able to detect time critical failures such as bottlenecks, race conditions, deadlocks and timeout failures because that would require real execution time. Therefore, component-based applications require a completely different level of monitoring than traditional single-process applications. One effective way of diagnosing a distributed application would be to use component-testing environment, which will be explained in next paragraph.

2.4 Component Testing Environment

A DOC Middleware solution (like Corba) supports the construction of distributed systems containing many components. These components can interact in complex ways, not necessarily conforming to a client-server model. This is it is also the reason why testing these systems in isolation is so difficult because each component can have complex dependencies on any number of components.

We have explained that Language level debuggers are not very suitable for testing component based systems because they have no knowledge of components infrastructure and are often limited to boundaries of the same machine. Therefore, a tester needs techniques to verify component at the abstraction level of its design. To accomplish this, a tester needs a testing environment that can verify the behaviour and integrity of each separate component as well in co-operation with each other. In order to allow the tester to create a controlled test environment, the tester needs methods to trigger the components under test, monitor its interactions with other components, and verify the data flow through a distributed system with its system specification. We therefore need a test framework that is able to verify the correct behaviour of Corba Applications. By a Corba test framework, we mean a collection of components, which cooperate with each other to allow the tester, verify the behaviour of a Corba Applications with is intended behaviour.

We can divide required testing framework into two separate sub frameworks:

- *A Control Framework,* allows the construction of a controlled testing environment for the distributed system under test. A control framework usually consists of Active and Passive test components called Actors and Reactors. Actors act as client components, which can be controlled manually by the tester or automatically by some script. Reactors are prototyped server components that can call the services other Components when required.
- *A Tracing Framework,* allows the observation of traceable invents inside the system under test. A tracing framework consists at least of a monitor tool, which allows the user to trace the events in a system and some tracing mechanism which can gather tracing data from a distributed system.

In the next two sub paragraphs, we will further explain the characteristics and problems of each framework.

2.4.1 Control Framework

It is hard to test components in isolation and pairs because they can have multiple dependencies on other components divided over multiple operating systems, which have not been implemented yet (see figure 2.2). Because these components might not be available or trusted at the time of testing, they must be substituted by an emulator component that mimics the required component services. Beside unit testing, where one component must be tested in isolation, server emulation is also useful for system testing, integration testing, and conformance testing by provide a more controlled testing environment.

Although a tester could build a test environment for every test case, it would require considerable time and effort to create a simple emulation that can be used to replace an actual server in a testing scenario. This is a significant problem because it would require the redevelopment and rebuild of the test application whenever a change has to be made in the test case. To solve this problem, a tester could use an Actor/Reactor generation Testing tool, like MCITT (see also paragraph 6.2.5), which can automatically generates test clients/servers from a test scenario script. The main difference between Actor and Reactor is that Actor initiates request while Reactor reacts to request done by other components.

Most Corba vendors facilitate programmers with more or less integrated test Reactor generation tools, like Orbix Code Generation Toolkit (OCGT), that can generate client or server component from Corba scripting definition language. The problem of using these tools is that they often rely on propriety services from a specific Corba implementation, which might pose migration problems later on when switching between Corba middleware implementations.

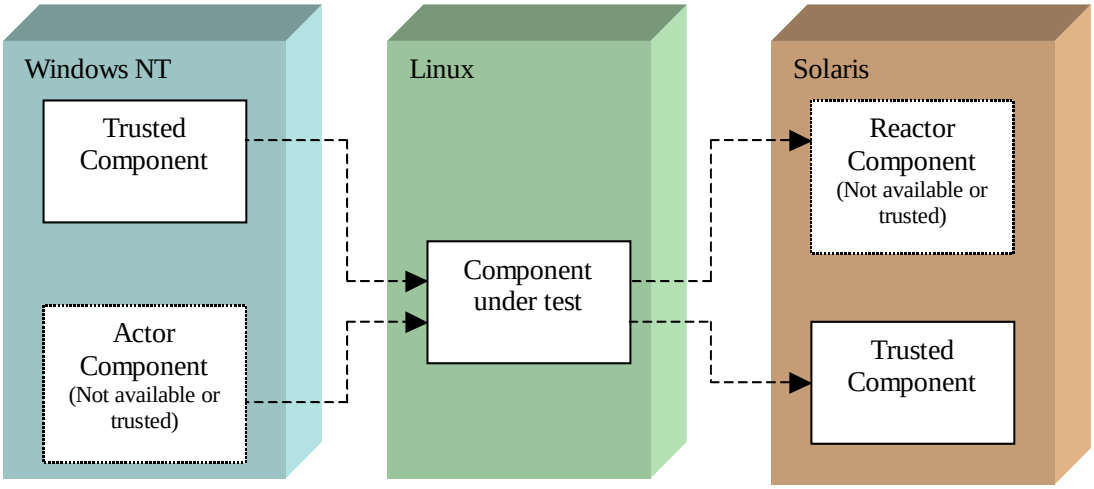


Figure 2.2: A Component requiring the services of other component, which are not available or trusted

Scripting languages are often also used to prototype a client or server component before it is implemented in the final language.

Scripting Languages offer the following generic advantages:

- *Simplicity of usage*, In comparison to static programming languages like C++, scripting languages are much easier to use for construction of simple test reactors. That is because scripting languages, offers a high-level abstraction mechanism that hides low level programming details like memory management and interface-calling conventions, which will help the user focus on the real problem.
- *Portability.* Because script is an interpreted language, script is portable to other operating systems different from the one it was developed on. This also encourages the exchange of scripts between users.
- *Reusability*. Script languages also offer the benefit to be used for performing regression testing performed after a modification the component.
- *Enhanced productivity*. Another advantage of scripting languages is that they do not require recompilation and re-linking after each change like statically compiled languages have to.
- *Easy to learn*. The ability to learn a scripting language is often more simple than a traditional language like C++, which requires all kinds of low level programming knowledge to use it effectively.
- *Reduced cost*: Simplicity and productivity respectively mean reduced training costs for users and reduced operating costs in conventional computer environments.

There are many languages, which can be used as a scripting language for testing Corba. We can identify several scripting language types:

- *Propriety Testing Scripting Languages*. These scripting languages are made specifically for one specific tool and are not portable to any other tool. An example of a proprietary language is the scripting language used by the MCITT tool, which we will describe in paragraph 6.2.5.
- *Generic Testing Scripting Languages.* These languages are designed as a standard scripting language, which are supposed to be portable between testing tools. A disadvantage of these languages is that they are often limited in their capabilities due to their drive to remain generic. In appendix D: OMG Corba scripting language, we will give a example of a Generic scripting language.
- *Extended Scripting Functionality.* Instead of creating entirely new scripting languages, some scripting languages extend an existing well known scripting language with extra testing functionality. An advantage of these scripting languages is that they can make use of existing tool support of the extended scripting language. In appendix E: Tcl Combat, we give an example of a scripting language, which is extended with testing functionality.

2.4.2 Tracing Framework

When the result of a component-based application, consisting of multiple components, is not what we were expecting, we want to know exactly what caused the problem. However, distributed applications can be regarded as black box that tries to hides its internal functionality. To solve this traceability problem, programmers need to know the exact component that behaved malfunctions and under what circumstances the error manifested. Because the smallest elements in a component-based system are components, and the smallest event is a communication between components, a tester will be mainly interested in tracing communication events between components.

In order to find a malfunction component, testers need to be able to observe the execution of a distributed application, 'step-by-step' without altering its behaviour. However, because distributed application can consist of multiple components, we cannot use a simple single threaded application and hope intercept all communication in a distributed system. We therefore need a framework, which can use several processes concurrently working together to collect tracing information and present it to a tester. We therefore call it a tracing framework because it allows testers to trace component communication events in a distributed system.

Although it can be argued that large scale distributed systems can be tested relying exclusively on component units test, a tracing framework allows testers to find problems faster. By finding problems at an early stage in the development, which would otherwise remain unnoticed until it manifests at a later stage in the development process, the development cost of distributed systems could be reduced dramatically.

A typical tracing framework achieves support for the following actions:

- *Trace Data Interception*. Because the communication in a distributed system is hidden and lost as soon an event has taken place, the trace framework must be able intercept communication events taking place between components at real time.
- *Trace Data Transportation*. Because distributed systems can consist of multiple systems, the tracing framework should support the user transport the tracing data to a central location where is can be analysed offline.
- *Trace Data Processing*. Because of the size and complexity of communication event, the tracing framework must be able to process into a readable format, which the tester can understand.
- *Framework Configuration*. Because internal communication events are invisible, the tracing framework must somehow alter the distributed system to collect this data. The framework should therefore support the user in the configuration of a tracing mechanism.

Collecting trace data collected by a tracing framework can be used for a wide range of utilisations in development of component-based applications:

- *Insight*. The ability to trace 'step-by-step' trough a distributed program is invaluable for gaining an understanding into the functional behaviour of a system. A Corba trace could help the viewer to obtain a picture of how existing components co-operate with each other. After gaining insight into the behaviour of a distributed application, testers might notice possible problems faster and programmers will see possible improvement sooner.
- *Educational*. Besides gaining insight for operational purposes, if adequately visualised, tracing is also useful for educational purposes. A trace of the execution of single application visualised as a sequence diagram is relatively easy to understand, yet powerful enough to visualise complex problems. Tracing can therefore be used in courses for trainees or students to learn how Corba Components communicate or more in general, how a distributed system functions.
- *Checking Correctness*. It is obvious that testers and programmers want to validate the implementation with the specification of a distributed application. By tracing a distributed application step-by-step, a tester will be able to observe every step and verify the observed behaviour with the specified correct behaviour.
- *Locating Errors*. Once detected that that the application does not behave as specified, tracing framework will allow the programmer to locate the first trace where erroneous behaviour was detected. By investigating the exact parameter values of the message, which triggered a component to produce the erroneous trace, the bug could be pinpointed to the source. The erroneous component can then be further inspected with a language debugger to find the exact line of code the problem started.
- *Analyse Scalability.* Component Based Applications must besides verified for correct behaviour of a single test case also be verified on correct behaviour of multiple invocations happening simultaneously or in short bursts. The tracing mechanism must there be able to discriminate individual test cases and allow the tester to analyse the scalability of the distributed application under test.

- *Stress Simulation*. In order to anticipate possible timeout problems, the tracing mechanism can be used to delay communication. A tracing mechanism, which allows the tester to add delays between trace events, allows the tester to simulate a system under stress.
- *Monitoring*. Even after a system has passed every test thrown at it, it might still fail when used over longer periods. Especially in mission critical systems, where constant uptime is of crucial importance, you want to be able to detect malfunction behaviour and if possible take appropriate action at real time. Because monitoring is over a longer period, the tracing framework data storage must be scalable enough to harbour huge quantity of tracing data. A tester would then be able to analyse the logged traces offline and use the results to prevent the malfunction from happening again.
- *Fault Tolerance Testing.* A tracing mechanism, which can intentionally sabotage component communication, for example by halting message communication, could be used to test the fault tolerance of a component-based application.
- *Fault Recovery*. Once a trace failure is detected, the system could use logged traces to recover the system back to a functional state. In the event of a system crash, the last recorded outgoing messages could be used to redirect all communication to another servant fulfilling the same functionality.
- *Timing Analysis*. If tracing does not significantly influence speed and a tracing mechanism can accurately measure the time traces occurred, tracing can be used to analyse the performance of components and communication performance of a distributed application. Assuming a message will generate at least two trace events, the event where a message is send from a component and the event where a message enters another component, we can calculate the communication latency. By measuring, the time elapsed between trace events succeeded after each other; the performance of individual Components can be calculated.
- *Detecting Security Breaches*. Another application for tracing is automatically detecting vulnerabilities of privileged programs by recognising program states that exhibit potential danger for the integrity of security problems. Once a possible security breach is detected, the system could employ counter measures to secure the system from harm.
- *Extracting Documentation*. If traces are logged into a persistent storage device, they can be used to automatically analyse and generate dynamic documentation about the dynamic behaviour of a software system. The dynamic documentation could be a great asset to static documentation techniques like inheritance graphs, which can be automatically extracted from component source code.
- *Message Replaying***.** Traced messages can be used to automatically generate a component, which mimics the exact same behaviour as the traced component. This could for example be accomplished by distracting a script from component instance interactions and use it as output for a Actor/Reactor component.

We have seen that tracing framework has many possible uses. Nevertheless, collecting tracing data is plagued with several problems, which has to be solved or minimised.

A Tracing framework must solve or minimise the following problems:

- *Framework Overhead*. Because distributed system normal does not allow the observation of its internal communication event, it must be extended with tracing functionality. However, any kind of modification to a distributed system requires additional action to be carried out, resulting in a delay of the original program. This delay causes problems especially for performance analysis, where the execution times have to be measured as accurately as possible. The tracing framework must therefore minimise the overhead cause by the tracing mechanism.
- *Transparency Problems*. The problem of modifying a distributed system, is that is lowers the distributed system portability to other environment requirements. For example, the instrumentation of a component source code for collecting tracing data, excludes the use of COTS component, from which the source code is not available. The tracing framework should therefore minimise the loss of transparency.
- *Time Synchronisation Problems*. In distributed systems, the time of the local clocks may differ from machine to machine. The problem is that this influences the ordering of communication events collect by the tracing framework, when using time as an ordering criteria. The tracing framework must therefore use some mechanism to make the event order back into its true order.
- *Identity Problems*. Traceability was not a design criterion when communication protocols were designed. Middleware communication protocols often do not contain the identity of a source. That because the middleware does not need it; they use the callback mechanism of network protocol for the delivery of a reply message. Although it is an efficient and effective design, this design feature makes tracing a lot more difficult. The tracing framework must there employ mechanism to derive the client identity.
- *Heterogeneity Problems*. Nowadays it is common that distributed applications consist of components using multiple programming languages, middleware types, and operating system systems. Thales Netherlands for example uses JAVA, C++, Corba, Java Beans, SPLICE, Windows, and several UNIX operating systems.

Although the value of tracing an application spanning multiple standards is commonly understood, the trend is that most tracing frameworks only focus on one specific middleware and programming language.

- *Open Standard Problems.* Although open component model allows a larger diversity in vendor solutions, they introduce a their own problems. Vendors often try to help developers by offering functionality, which is not part of the standard. Although this propriety functionality can be very useful in the construction of tracing framework, their application limits it usage with other implementation of the component model.

Now that we have introduced the basic characteristics, possibilities and problems of tracing frameworks, we introduce a generic design for a tracing framework, which allow high flexibility.

3 A HIGH LEVEL DESIGN FOR A GENERIC CORBA FRAMEWORK

In this chapter, we introduce the reader to high-level design for a generic tracing framework, which will allow tester to diagnose a heterogeneous component-based system. In contrast to other tracing frameworks, our tracing framework is mend to be context free from any technology dependencies like operating system, language, and middleware. That means we do not specify the technology that must be used to realize it. We only specify the architecture and functional requirements that should be fulfilled by the framework implementation.

Our strategy to achieve heterogeneously and platform independence is to use a tree tier architecture. In figure 3.1, we can see that our tracing framework is divided into tree abstract layers. Each layer conceptually fulfils a separate set of responsibility in the tracing framework:

- *Tracing Layer*. The responsibility of the tracing layer is to collect tracing data from a distributed system and supply it the Information layer. The tracing layer consist of one or more tracing mechanisms, each intercepting tracing data from communicating components, processing the tracing data into a universal tracing data, and storing it in the generic Information system located in the information layer.
- *Information Layer*. From an architectural point of view, the Information layer serves as the glue between the tracing layer and the monitor layer. The responsibility of information layer is to provide a generic information system that allows tracing mechanisms to store tracing data, and to allow monitoring tools to retrieve generic tracing data en present it to the tester.
- *Monitor Layer*. The responsibility of the monitor layer is to allow the tester to analyse collected tracing data. The Monitor layer should consists of one ore more Monitor Tools that allow a tester to diagnose a component-based application based on tracing information stored in the information layer.

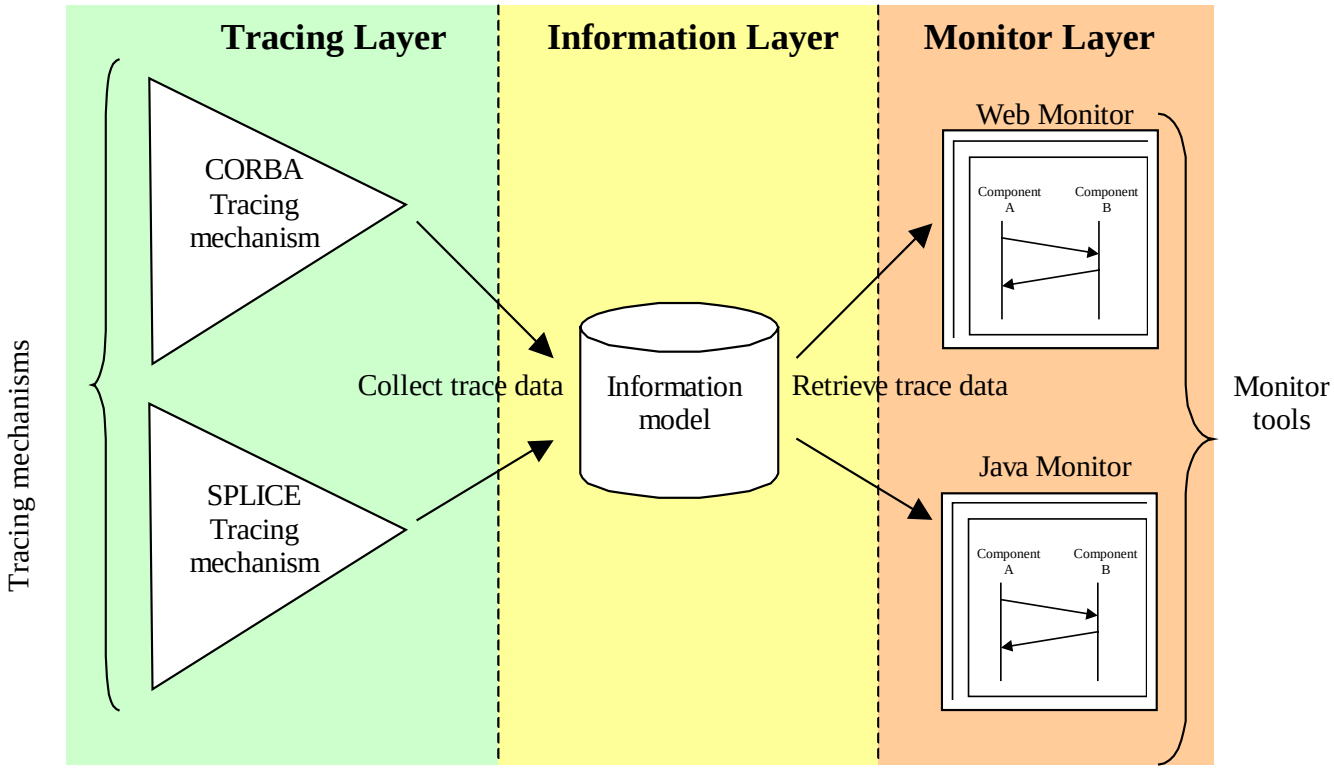


Figure 3.1: Tracing Framework Tree Layers Architecture

This tree layer architecture has the following advantaged:

- *Layer Transparency*. One of the big advantages of dividing tracing layer and monitor layer in a separate information layers is that both layers only have to deal with one intermediate communication protocol. Because if you would not separate them, either the tracing layer or the monitor layer would be forced to use each other communication protocol.
- Heterogeneous *Transparency*. Another advantage is that the used tracing mechanism is not forced to a single middleware technology and tracing technique. Tracing mechanisms are therefore free to utilise any combination of platform, middleware, programming language, and tracing techniques.

The next tree sub paragraphs will describe each layer in more detail

3.1 Tracing Layer

In this paragraph, we discuss the requirement for the tracing layer in our tracing framework. Figure 3.2 illustrates the Tracing Layer as a Data Flow Chart. Each arrow arriving at the tracing layer represents an incoming information flow and each arrow that is leaving represents an outgoing information flow. Each box represents a process that can be split further into a new Data Flow Chart. Depending on the direction an arrow enters or leaves a task box, a different type of information type is meant.

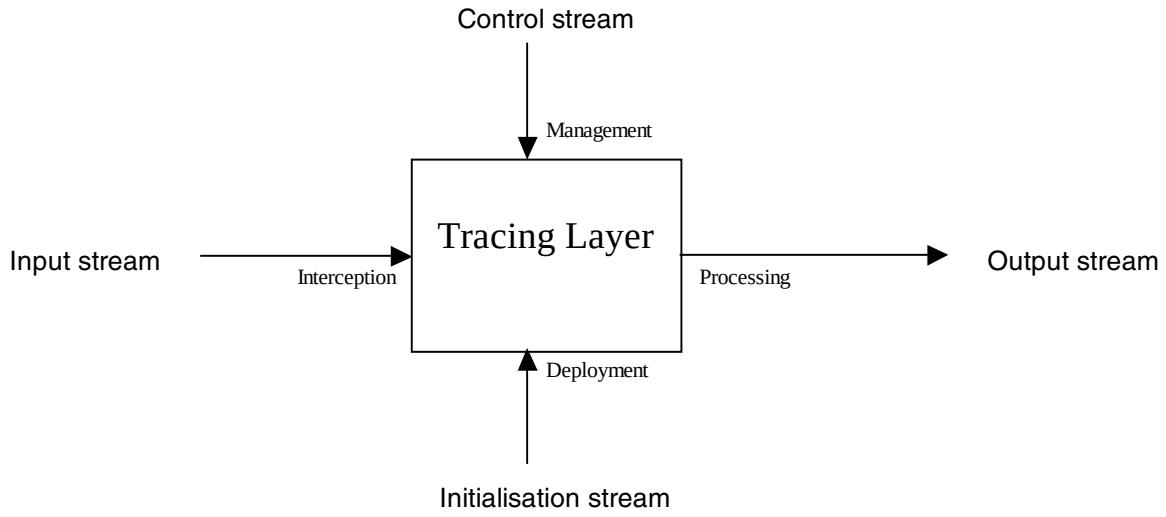


Figure 3.2: Tracing Mechanism Information Streams

For each arrow arriving or leaving at our Tracing Layer, we can identify a separate task that must be fulfilled by the tracing layer. Because there are four information streams, we can also define four separate tasks:

- *Interception*. The Interception task is the actual capturing of tracing data from distributed system. The information retrieved must gather sufficient information to allow the Processing tasks to populate the tracing information with generic tracing information.
- *Processing*. The Processing task is all actions that must be performed to fill the tracing layer with retrieved tracing data. This task includes at least at the transportation and processing of context sensitive tracing data into generic tracing format.
- *Deployment.* By deployment, we mean all actions that must be undertaken by the framework to install the tracing mechanism in a distributed system.
- *Management*. Once a tracing later is properly installed, the dynamic behaviour of the tracing layer should be actively manageable by some management mechanism.

For each task, we will now dedicate a separate paragraph.

3.1.1 Trace Retrieval & Processing

In this paragraph, we give an architecture that could fulfil the retrieval and processing tasks required by the tracing layer. The Tracing mechanism can be implemented by separating the retrieval task into tree separate components. Figure 3.3 illustrates how the components cooperate with each other to retrieve tracing data and transport it to the other layers. The Tracing layer can be achieved by the collaboration of the following three components, each playing a different role:

- *Tracer*: A Tracer represent the trace data interception mechanism in the middleware that is responsible for intercepting communication events in a distributed system and send it further to the locally hosted Collector component. The Tracer should strive to be as transparent to the distributed system as possible.

- *Collector*: Although Tracers could send intercepted communication directly from to the Processor. It offers several advantages to use an intermediate component. By locally buffering intercepted tracing data before its being sent to the Processor component, the communication overhead is minimised. Because internal communication causes no network traffic overhead, they can also function as gatekeepers, by blocking tracing events that are undesired by the user.
- *Processor*: The Processor is the component closest to the Information system. By locating the processor on a different host than systems that is being traced, the overhead of the Processor on the distributed system under test will be minimised. Its main task is to complete the final steps in the tracing mechanism, which is to supply the information layer with tracing data. Before the tracing data can be stored in the information layer, it must first be processed. Processing consists of filtering and combining the tracing events received from the Collectors into the generic tracing format and storing it in the information system.

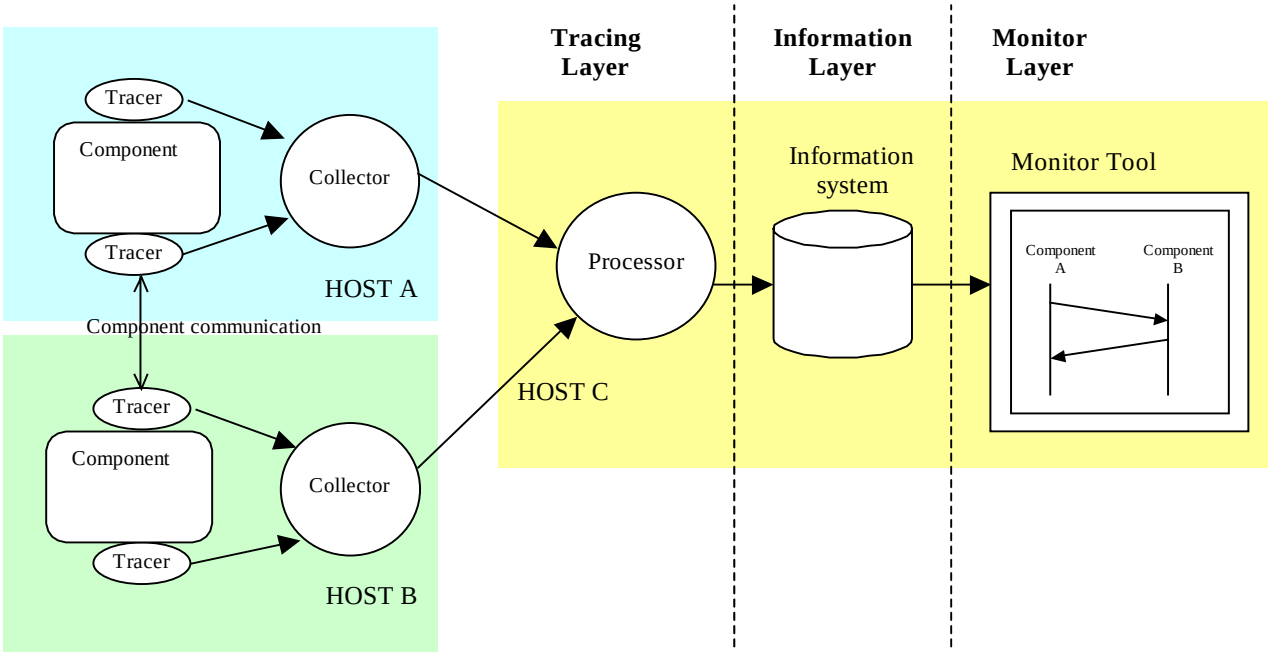


Figure 3.3: Tracing Mechanism Architecture

3.1.2 Deployment

A tracing layer often requires the modification of the distributed system before monitoring can commence. Deploying, registering, compiling, linking, starting and eventually uninstalling multiple Framework components for every time a component needs to be traced, is something the tester rather would like to see automated. However, deployment is often an overlooked problem in testing solutions. If a framework does not supported properly, the deployment of a tracing layer can easily delay a test process due to unforeseen deployment difficulties.

3.1.3 Management

Often, tracing all communication events is not always desired because most tracing events will be uninteresting and therefore only introduce unnecessary overhead. While trying to minimise the overhead in tracing mechanism is desirable in most cases, there are cases where an extra artificial delay is actually useful. By introducing artificial delays during message interception, the tracing mechanism could be used to simulate process or communication latency. Therefore, in order to prevent unnecessary tracing overhead or artificially introduce extra overhead, the tracing mechanism should be fitted with additional management functionality. The management functionality would have to locally maintain a profile, which would determine the tracing mechanism behaviour during tracing events. For example, a profile could contain a filter condition that specifies the communication events that should be discarded for tracing. In order to modify these tracing profiles, the tracing mechanism should therefore be fitted with some management control mechanism, which can alter the behaviour of the tracing mechanism.

3.2 Information layer

The Information layer must fulfil the following requirements:

- *Persistent Data Storage*. One of the characteristics for a good tracing framework is the ability analyse the intercepted tracing data offline, long after the trace of a distributed application through has taken place. In order to analyse tracing data offline, the information layer must therefore be able to store the data on a persistent data storage which can be retrieved at a future point of time.
- *Open Interface.* Another characteristic of a good tracing framework is that it offers an open framework with open well-defined interfaces. The tracing layer and the monitor layer can use this interface for exchanging tracing and management information between the layers. An open interface also allows easier development of the other tools that want to access or modify the information stored in the information layer..
- *Generic Tracing Information System.* The information system should try to store basic tracing information in an as much generic way as possible. This will allow the information system to trace system independently of the observed middleware technology.
- *Self-Describing Contents Data.* Contents data is the data exchanged between processes. The captured tracing contents data from a distributed system can vary widely in types and structures. In order to allow the other layers have certain flexibility in the storage of complex data types, the contents data should be self-describing with multiple levels of abstraction. Self-describing data specifies the data structures and types used to represent the data in the correct way.

Storing information in a generic format can be achieved by restricting tracing data to atomic tracing information supplemented with self-describing meta-data. The smallest atomic tracing information in a in any network-oriented middleware is a message event between two processes. The generic tracing information model should therefore contain at least the following required tracing data (see table below).

Trace data	Description
Message ID	Uniquely identifies a message
Source Process ID	Uniquely identifies the source process of a message
Designation Process ID	Uniquely identifies the destination process message
Source Process Time	The exact global time at which the communication started at the source
Destination Process Time	The exact global time at which the communication ended at the destination

Note that a process is an abstract concept, which depending on the level of abstraction can have different meaning. Although a Process ID uniquely identifies a process in the tracing framework, there should also be some context information linked with every process ID, which describes it location of a process in its context. Because we often can differentiate multiple levels of abstraction in a distributed system, a process can be scaled to a higher context level. For example, a process ID in a Corba/CCM environment can be scaled up and down through following levels of abstractions (see table below). This context information of a process can be used by the monitor layer to group processes together.

Abstraction level	Context data	Description
7	Building	The building at which the computer is located
6	Network	The network at which on the computer is located
5	Computer	The computer which contains the process
4	Processor	The processor which runs the process
3	Container / Orb	The container/ orb associated with the Component/Object
2	Component / Object	The component which uses the Object Request Broker
1	Interface	The facet or receptacle which was uses for communication
0	Thread	The thread from which the communication event was intercepted

3.3 Monitor Layer

In Monitor Layer, the tester will be able to analyse the tracing data stored in the information layer. The monitor layer could make use of several visualisation techniques to visualise tracing data. Although we will not describe a design for the monitor layer, we will discuss how the tracing data could be visualised by a monitor tool.

Tracing data can be analysed at different levels of abstraction:

- *Message level,* is the lowest level of abstraction. It should allow a user to inspect a message contents, the source and destination identity of message. A standard visualisation technique which is can be used to display the message level is a hierarchically structured parameter tree.
- *Process level,* allows user to analyse how unique processes exchange messages with each other. A standard visualisation technique, which is often used to analyse message communication at the process level are sequence diagrams.
- *System level,* allows a user to analyse an overall distributed system. This is often achieved by performing automated statistic analysis over all intercepted communication. Although this technique can quickly identify communication anomalies, it might miss important problems. In the next paragraph, we introduce new visualisation technique, which can help the tester analyse messages at the system level.

3.3.1 A New Application of Network Diagrams

Testers that need to diagnose multiple invocations made on the component based application need some effective way of visualisation. A technique, which is often used to visualise invocations between processes, is the sequence diagram. A sequence diagram is a standard visualisation technique where the interaction between processes can be analysed. Processes are visualised as vertical lines while invocations a visualised as arrows. Because western cultures read from top to bottom, from left to right, it is custom to visualise a sequence diagram in the same format. The advantage of this visualisation technique is that it allows the viewer to observe all communication actions of a process, chronological ordered by time. Sequence diagrams can also be used to visualise the interaction between component instances. Although a real component instance may in fact use multiple processes, we can model a component instance by a single process. Because all communication between network-oriented middleware is based on directed messages, we can model all directed message communication as a sequence invocation between processes. In the figure 3.4, we can see how each arrow in the collaboration diagram can be represented as a arrow in the sequence diagram.

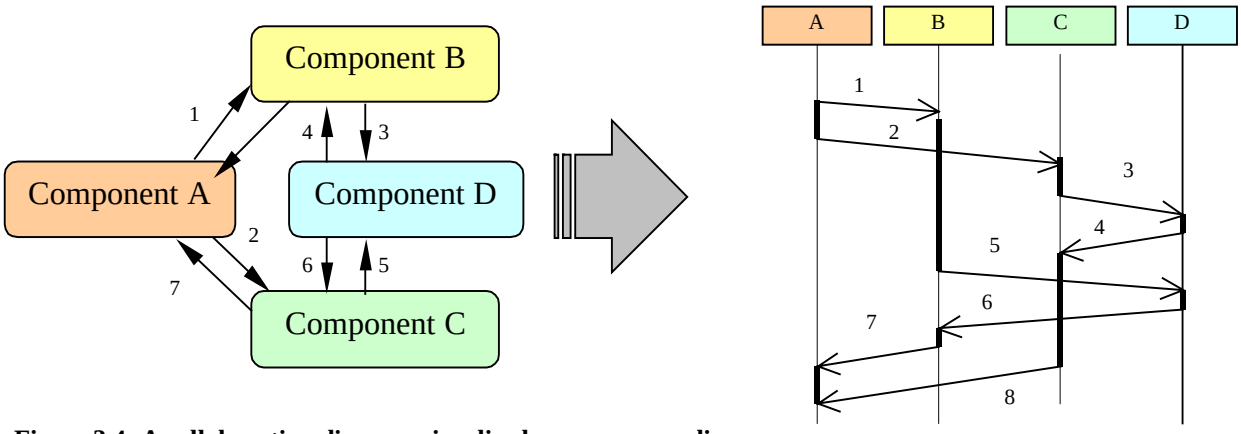


Figure 3.4: A collaboration diagram visualised as a sequence diagram

Although a sequence diagram allow the viewer to observe parallelism, crossing sequence lines can easily clutter a sequence diagrams (see figure 3.4). Using sequence diagram for system level analysis is therefore insufficient for this task because the system overview can easily be lost. Instead, we need a visualisation technique that emphasises the phenomenon's, which that are most interest for system level analysis. When analysing the parallel behaviour of a distributed system, the phenomenon's, which are most interesting is the latency between the trace events and casual relationship between tracing events. A visualisation technique, which could satisfy these demands, is to usage network diagrams. Network diagrams are used to visualise the time critical relation ship between parallel events. Network diagram consists out two main building blocks, nodes and directed arrows. The nodes in a network diagram represent unique events or disjoint group of events. A directed arrow that connects two nodes represents a transition between the two nodes. We can transform every message diagram isomorph into a labelled network diagram. In figure 3.5, we can see how a sequence diagram can be transformed into a network diagram in which every node represents a trace event and the connection lines can either represent an invocation of a process.

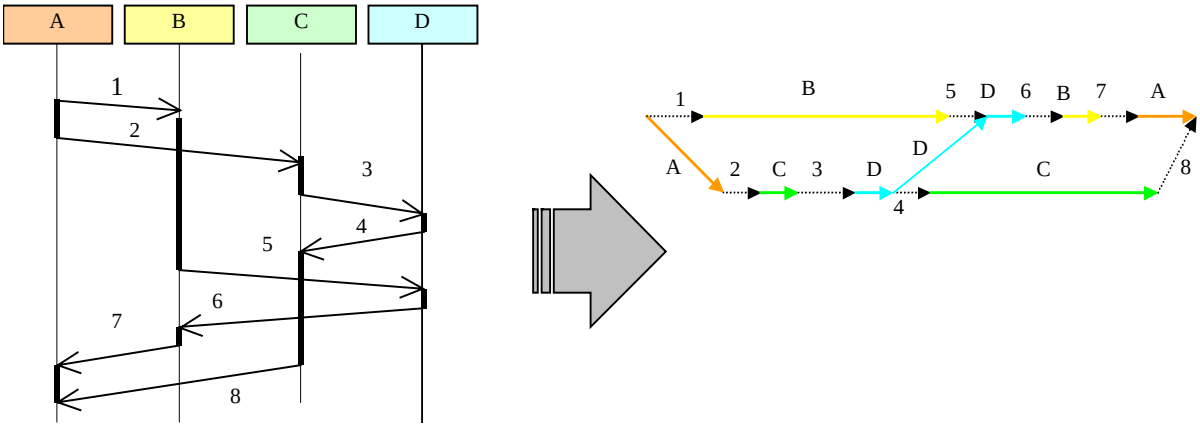


Figure 3.5: Sequence diagram visualised as network diagram

Depending on the number of incoming and outgoing arrows, different parallel events can be observed in a network diagram:

- Nodes that are only connected by two arrows (one incoming and one outgoing) visualise an event communication chain. Notice in figure 3.5 that we can now clearly see how these communication chains visualise two parallel processes in sequential order.
- When nodes contain two outgoing arrows, it means an event is followed by two simultaneously communication chains. These nodes, which mark the start a new chain of sequential events, effectively visualising to creation of a parallel process.
- When nodes contain two incoming arrows, it means that beforehand of the event, two simultaneously transitions happen. Notice in figure 3.5 that the nodes that mark the end a chain of sequential events effectively visualise a possible synchronisation between two communication chains.

3.4 Evaluation.

We have presented high level architecture and functional requirements for the construction of a generic tracing framework. By separating the framework into three separate layers, we achieve a higher level of flexibility able to function in a heterogeneous environment. For the tracing layer we have shown a universally applicable architecture for collecting and transferring tracing data from a distributed system to the information system. The generic information model allows the tracing mechanism to collect tracing data from any network-oriented middleware. In chapter 7, we will show how a tracing layer functional requirements can be fulfilled by a transparent Corba tracing mechanism.

4 INTRODUCTION TO CORBA MIDDLEWARE

Now that we introduced the reader to the main concepts and problems behind Component Testing, and introduced a generic tracing framework for diagnosing middleware we will look at the middleware we want to monitor, which is Corba Middleware. This chapter is by no means a complete description of the Corba middleware but is merely mend as an introduction to Corba middleware technology.

This chapter will have has the following structure:

- *Paragraph 4.1: Corba Object Model.* In the first paragraph of this chapter, we introduce the reader to Corba object model and explain some important concept and mechanism. The basic Corba concepts introduced in this paragraph will be essential for understanding the following chapters.
- *Paragraph 4.2: Corba Component Model*. In the last paragraph we tell something about the Corba Component Model, which builds further on top of the Corba Object Model.

4.1 Corba Object Model

Corba Object Model is the product of a consortium called the Object Management Group (OMG). The OMG objectives are to foster the growth of technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA Reference Architecture defines an underlying infrastructure of services and mechanisms that allow objects to intemperate. Corba allows object to be executed in any Operating System such as Microsoft Windows or UNIX using an Operating System specific Object Request Brokers (Orb). Unlike other Object Models like Microsoft DCOM, Corba is an open standard, which means that the OMG promotes the standard, but is not a middleware producer that provides the middleware implementation.

The basic structure of this paragraph will be as follows:

- Paragraph 4.1.1: We introduce the basic Architecture of the Corba Object Model
- Paragraph 4.1.2: We explain how a basic connection between Corba Object can be established
- Paragraph 4.1.3: We describe how client server applications can be constructed.
- Paragraph 4.1.4: We describe some important Corba services used for the construction of Corba applications.

4.1.1 The Object Request Broker

In the Corba Object Model, the Object Request Broker (Orb) plays a central role in all Corba communication. It allows clients to connect and call a remote object without knowing where the distributed object resides on, in what operating system it executes on, or in what programming language the object is implemented in. Because neither the client nor the server has to worry about the location of each other, we say the Orb functions as a transparent communication layer between client and server. The Orb will automatically take care of all message routing between client and server. The next paragraph describes how client and servant are able to communicate with each other.

4.1.2 Stubs and Skeletons

A Corba client request is not handled directly by the Orb but by a stub instance. Stubs implement the proxy pattern [Gamma] that marshals operation information and typed parameters into a standardised request. Likewise, a Corba object does not respond to an Orb directly but through a skeleton (see figure 4.1). The skeleton implements the Adapter pattern [Gamma], which *un-marshals* the operation information and typed parameters information located in the standardised request format. The stub and skeleton therefore serve as the “glue” between the servant and Orb, allowing an invocation on a client object to pass through a stub instance on exit and through a skeleton on entry.

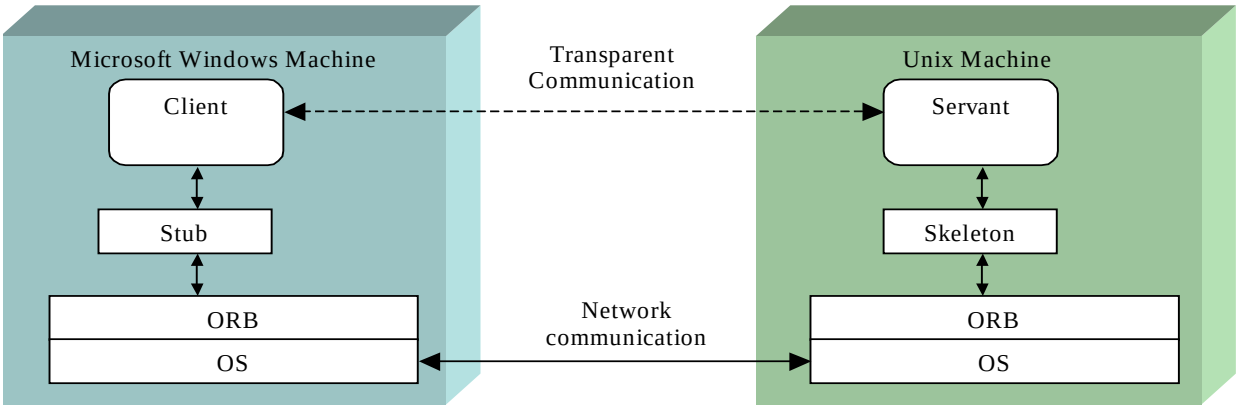


Figure 4.1: Distributed Communication through ORB

4.1.3 The Interposable Object Reference

The only thing a client needs to do is to call upon the services of a distributed object is to acquire the interface object reference. Because an interface object reference has no meaning outside the context of the Orb, a client must first generate an interface object reference from an Interoperable Object Reference (IOR). An IOR is a generic format for identifying Corba objects generated by the Portable Object Adapter (for more details see appendix A). In order to acquire an IOR, a client could retrieve the address from a file made by the server or use a specialised common object service (COS) like Corba Naming or Corba Trading service. The Corba Naming service functions like a telephone 'White pages' for objects in the sense that allows a client to find object reference by name. The Trader Service is like a telephone 'Yellow Pages', which allows object services. Once an application has acquired IOR, it has all the information it needs to connect to the object and make remote invocations on the remote objects.

4.1.4 The Interface Definition Language

The stub and skeleton implementation source code does not have to be written manually but is automatically generated from schemas defined using the Interface Definition Language (IDL). Figure 4.2 show how an IDL compiler generates a stub and skeleton file from an IDL file. The stub and skeleton is then included by client and server source file, which is eventually generated into a client and server application by a normal language compiler.

The stub and skeleton generated by an IDL compiler are fixed, e.g. the code emitted by the IDL compiler is determined at translation time. An advantage of statically generated stubs and skeletons is that it eliminates common sources of network programming errors and provides a mechanism for providing language and platform transparency. A disadvantage of using statically generated stubs and skeletons [Nonbour] is that the fixed behaviour is not always be sufficient to perform middleware specific functionality, such as tracing or security. To overcome these problems, developers could instrument the stub or skeleton or employ meta-programming techniques explained in the next chapter.

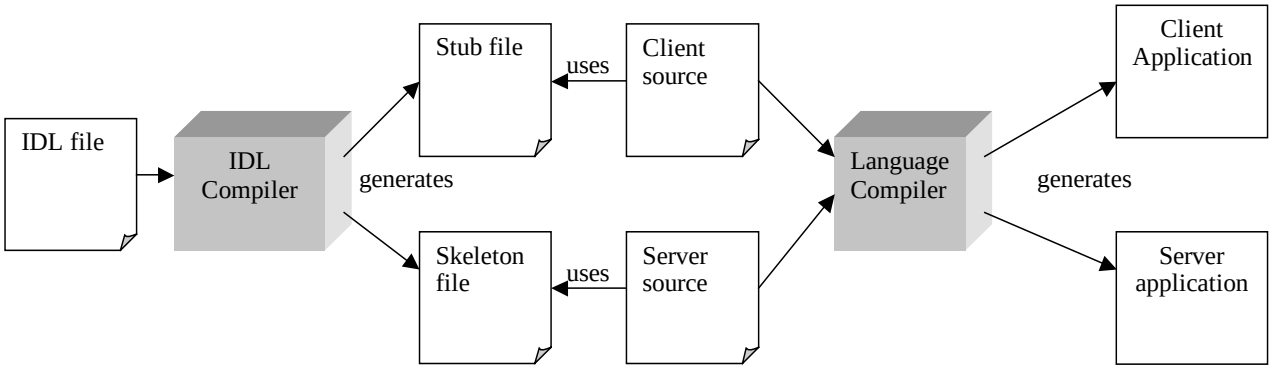


Figure: 4.2 IDL stub and skeleton generation

4.1.5 Corba Services

Services are extremely generic and are independent of any application domain. Although we already introduced some COS Corba services like the naming and trading service, in this paragraph will now explain some other Corba services, which can play an important role in the construction of a testing framework.

Corba offers the following generic services, which are useful for creating a Corba testing framework:

- *Interface Repository*. This interface allows the developer to retrieve metadata information on the available interfaces and components. This will be useful for the actor application to list available interfaces and help to innovating them it the right way. Another very useful application of the interface repository is to translate a intercepted IOR address back to a object name
- *Dynamic Invocation Interface (DII).* This interface allows a client application to establish a connection to a Corba component at run time. This will be especially useful in actor applications, which has to connect to a component, on the fly.
- *Dynamic skeleton interface (DSI).* The DSI is the server variant of the DII. This interface allows an Object Request Broker (Orb) to deliver requests to object implementations that have no compile-time knowledge of the interfaces they implement defined using Corba IDL.

The next sub paragraphs will give a short description of the mentioned Corba services.

4.1.5.1 Interface Repository

Corba is a self-describing system. In fact, every component, system level object, every service that lives on a Corba bus (Orb), even the bus itself is self-describing. Self-describing data, which also referred as metadata, allows independent developed components to dynamically discover each other existence and collaboration. Tools such as Actor/Reactor tools can use this metadata to obtain inheritance structures and class definition at run time.

In Corba, metadata is derived from the metadata located in IDL files. All Corba available metadata is stored into retrievable metadata repository called the Interface Repository. The Interface Repository is hierarchy ordered set of classes whose object instances represent the information that is in the repository. The Interface Repository classes provide operations that allow a Corba object to read, write, or destroy the metadata that is stored in a metadata repository. The Interface Repository is automatically populated with metadata by the statically or dynamically generated stubs and skeletons, or manually through the Interface Repository write functions.

4.1.5.2 Dynamic Invocation Interface & Dynamic Skeleton Interface

Normal Corba client to servant communication use stubs and skeletons, which has to be statically generated by a IDL compiler (see also paragraph 4.1.4). A disadvantage of static generated stubs and skeletons is that the communication link between client and servant must be known before compilation. The Dynamic Invocation Interface (DII) and Dynamic Skeleton Interface (DSI) enable a client and servant to establish a connection at run-time. The DII and DSI is essential the mechanism which allows an Actor and Reactor tools to connect to a component under test without recompilation. Without it, an Actor tool would have to be recompiled for every time it switches connections. The DII and DSI also enable deferred synchronous interaction clients client and server object where a client can decide whether to wait for a response. This asynchronous form of communication allows monitor mechanisms to send tracing information to a central location that can further process the tracing data. While the DII provides more flexibility over a compiler-generated stub/skeleton, it is also more resource costlier because with the DII, a remote request cannot be made in a single Remote Procedure Call (RPC).

4.2 Corba Component Model

Because Thales Netherlands will use PERCO, a Corba Component Model (CCM) implementation for the development of Corba applications, we now give a brief overview of CCM. CCM is designed for large-scale, distributed enterprises and Internet applications that need to run with transactional assurance, security, and high throughput. Although Corba offers a tremendous amount of flexibility in creating distributed systems, it still requires a fair amount of technical knowledge to program with it. This is because POA policies, transactions, security, and other resources combine in thousand of different ways and it would require considerable skill to select the best combination out of all the alternatives. Of these combinations, a few patterns have proven to be very successful, have demonstrated that it is usually not necessary to consider the others.

By introducing CCM components and the container-programming model, the Object Management Group (OMG) realises a true Component Model standard that enables developers to assemble application from reusable

Corba components instead of developing them from traditional Corba objects which have to implement lots of low level middleware functionality by themselves.

With the introduction of the new Corba standard (version 3.0), OMG also aims to reduce the complexity of constructing Corba applications and lowering the steep learning curve for starting Corba developers. The CCM Container Programming Model wraps complex Corba services in a layer that exposes a simpler, higher-level interface with fewer choices.

The basic structure of this paragraph will as following:

- *Paragraph 4.2.1*. In this paragraph, we explain what containers are how they affect tracing
- *Paragraph 4.2.2*. In this paragraph, we explain CCM components.
- *Paragraph 4.2.3*. In this paragraph, we explain how component are managed.
- Paragraph 4.2.4. In this paragraph, we explain more about Corba communication techniques.

4.2.1 Component Container

In this paragraph, we will give the readers a brief overview of the component container-programming model and how it affects tracing. Containers were designed to support scalable servers which allows an application to set policies that control among other things activation/ deactivation patterns for the executing code and data that constitutes a Corba Component instance. In figure 4.3, we can see that the CCM container is an essentiality specialised Corba Portable Object Adapter (POA), which hides all low-level middleware functionality from components it contains. In contrast to Corba servants, that have to establish a connection to the Orb and maintain their own life cycle with the POA, CCM components do not have to perform any low level Corba operations. In fact, they do not even have access to them. This causes a problem with any Corba object programming technique, which have to perform any low level Corba operations like Orb initialisation, and object reference operations.

A component container offers the following main functionality:

- Internal interfaces that enable the component to connect with the container
- Manage the life cycle of components by activating them at the request of a client and deactivate them to preserve system resources.
- Automatically Forwards client request to Corba Services
- Provide Call-back mechanisms to inform component about interesting events, such as new messages from the transaction or notification services

Although the OMG attempts to make the development of distributed systems al tot easier with their introduction of the new CCM container, they neglect to specify any services that can help testers to trace components. However, outside OMG group, there are currently some groups working on services, which are potentially useful for tracing Cobra. As part of PERCO Container Programming Model, Thales France is in currently in the process of developing logging services which can log communication at the component level [Philips].

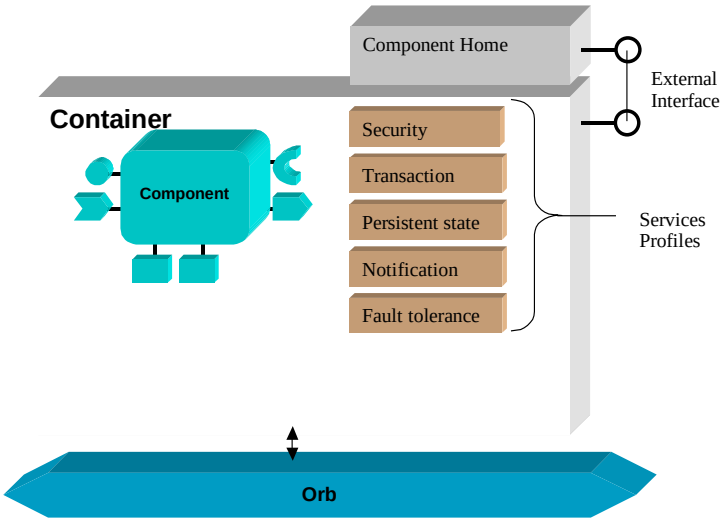


Figure 4.3: CCM Container Model

4.2.2 CCM components

With CCM components, OMG realises the promise of creating self-contained units of software code consisting of its own data and logic, with well-defined connections or interfaces exposed for communication. To allow CCM component to be compatible with other component object models, like Java beans, CCM makes a clear distinction between Basic components and Extended components.

Although basic component has to conform to the much stricter Corba Component Model, they are almost identical traditional Corba Objects and are therefore compatible. Basic components only expose an equivalent interface which consists of operations and attributes inherited from supported interfaces, attributes in the component body, and a number of standard base interfaces that a component are required to have.

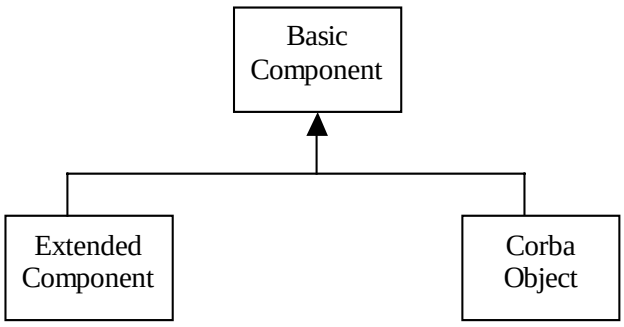


Figure 4.4: Component Type Inheritance

In figure 4.4, we can see that an extended component is a specialisation of a basic component because it inherits all requirements of a basic component and adds additional features to them. Although the extra features enhance component reusability when compared to traditional Corba objects, the drawback is that they make extended components incompatible with traditional objects. The extra features of extended components are made available through ports (see figure 4.5). Ports are additional interfaces that allow a component to have event driven communication or provide extra interfaces other than those it inherits. The extra interfaces, called facets, allow components to expose different views to it clients. Clients that want to establish a synchronous connect to one of these facets can use Receptacles obtain the object reference to the component instance. This object reference can than be used to delegate operations on server component.

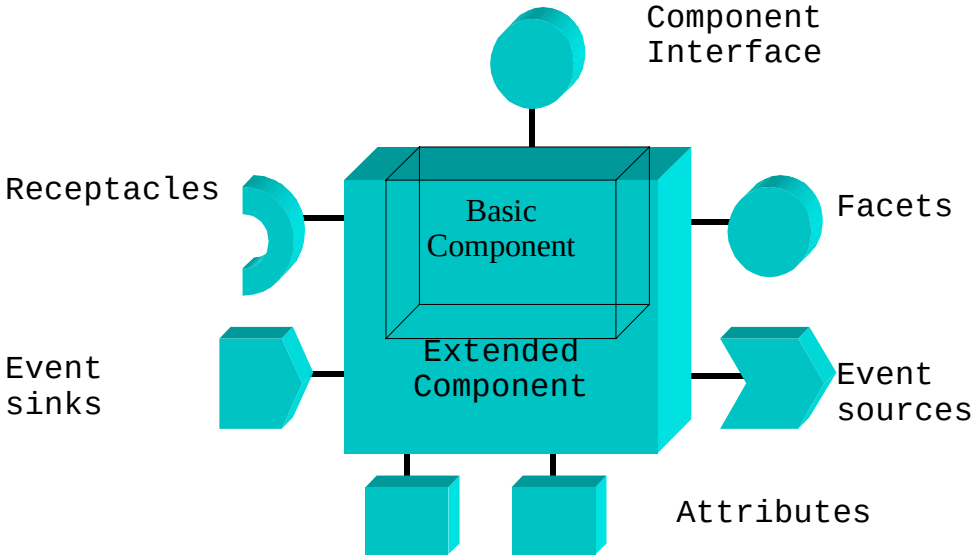


Figure 4.5: Extended Component

4.2.3 Component Home

In contrast to Corba, where a client has to establish a connection with a Corba object itself, CCM clients cannot create component instances directly but they have to use a component home to find an existing component instance or create a new component instance for them. A CCM client therefore has no access to low level objects like interface object reference and Orb object. The interface of a component home provides operations

to manage component life cycles, and optionally manage associations between component instances. In order to allow the client to find a component home in the first place, there is a special finder interface available, which functions much like the Corba Naming System.

4.2.4 Corba Communication

Because communication plays an important role in the construction of Corba tracing framework, we will explain some of the communication available in Corba/CCM:

- *Two- way message communication*. Synchronous Two-way message communication is the basic way of communicating between Corba objects. Although this method is relatively simple to use, the biggest disadvantage is the low performance caused by the communication. That because a client is locked during send and requirement of a remote procedure call.
- *One-way message communication*. This method of communication allows clients to send a message to a target object directly without expecting any return. This technique it relatively complicated to implement properly because the client must establish and maintain reliable a communication link with the destination object. That is because there is no guarantee that a one-way Corba message will arrive at it destination. However properly implemented it could potentially achieve the best performance possible by a Corba middleware solution.
- *Indirect message communication*. Instead of communicating the information directly between Client and Servant, a message is sent indirectly trough standard Corba Event/Notification service. Although the Event/Notification service uses standard one-way communication internally, this event-based communication mechanism allows the communication between client and server to be de-coupled. Corba Event/Notification service therefore allows clients to send information to multiple servants without knowing of their existence. The notification service also offers a Quality of Assurance functionality, which guarantees the delivery of all send messages, and optional offers the usage of filters, which can block undesired events. Although the usage of the Event/Notification service makes event based communication a lot more simple, a Corba Object still has to perform many actions to implement it.
- *Publish-Subscribe message communication.* The CCM standard also offers a publish-subscribe communication technique in the form of event sources and event sinks. Although component use standard Corba Event/Notification services under the hood, the developer won't be bothered with any of communication issues and can therefore focus more on the functionality of the component.

5 AN EVALUATION OF TRACE DATA INTERCEPTION TECHNIQUES FOR CORBA

To verify correct behaviour of Corba applications, and pinpoint a misbehaving component, it is necessary to trace the exact propagation of messages as they flow through a distributed system, entering and leaving components. In chapter 3, we described a generic tracing framework design containing a tracing layer architecture, which has to collect tracing data from a distributed system. However, except for the messages a client sends and receives from a server component, the internal Corba application data flow is hidden from the client by the middleware and must be intercepted somehow. Therefore, we will now investigate the possible techniques to intercept this trace data from a Corba application. In chapter 7, we further analyse one of the techniques for the construction of a transparent tracing layer.

In this chapter, we will discuss the following topics:

- *Paragraph 5.1 Middleware Interception Programming Techniques.* In this paragraph, we look at several programming technique, which enable us to collect tracing data independently of the used middleware technology.
- *Paragraph 5.2: Corba Meta programming Interception techniques.* Corba offers several Meta programming mechanisms and facilities, which can be used intercept Corba communication events. Programmers can use Meta-programming mechanisms to transparently change the Corba middleware behaviour to collect tracing information whiteout modifying the middleware or components.
- *Paragraph 5.3: Comparison of Trace Data Interception Techniques*. In the final paragraph of this chapter, we compare all tracing techniques we discussed in this chapter based on the characteristics that are important for developing an effective tracing mechanism.

5.1 Middleware Interception Programming Techniques

In this paragraph, we will explain several middleware-programming techniques, which can be used to intercept trace data and look at their strong and weak points. Each middleware interception technique is described in a separate sub-paragraph.

5.1.1 Debug instrumentation

Debugging instrumentation is probably one of most used programming technique by developers to trace distributed systems. This programming technique is similar to adding debugging code to the source code, could for example be implemented by calling a logger object just before and after a procedure call. Typically, a logger object provides some functions for this, so it can receive a client parameters or result as in arguments. For example, in the client source code, every remote procedure call to a distributed object could be preceded and followed by a call to the Logger object (see figure 5.1).

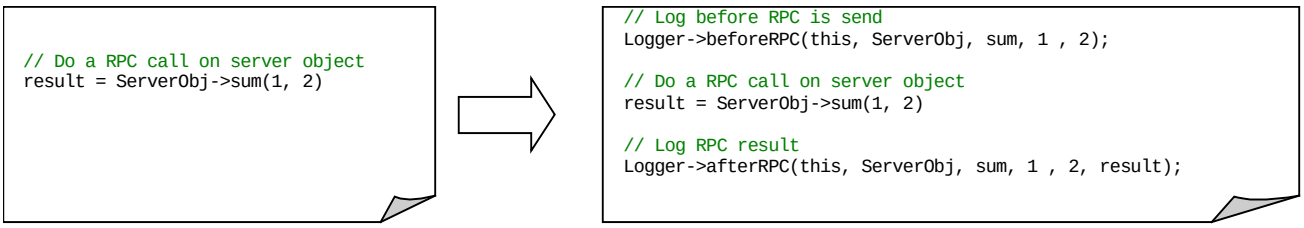


Figure 5.1: Instrumentation client source (before and after)

On the server object, the Logger object could be called just after the start and just before the end of a method (see figure 5.2).

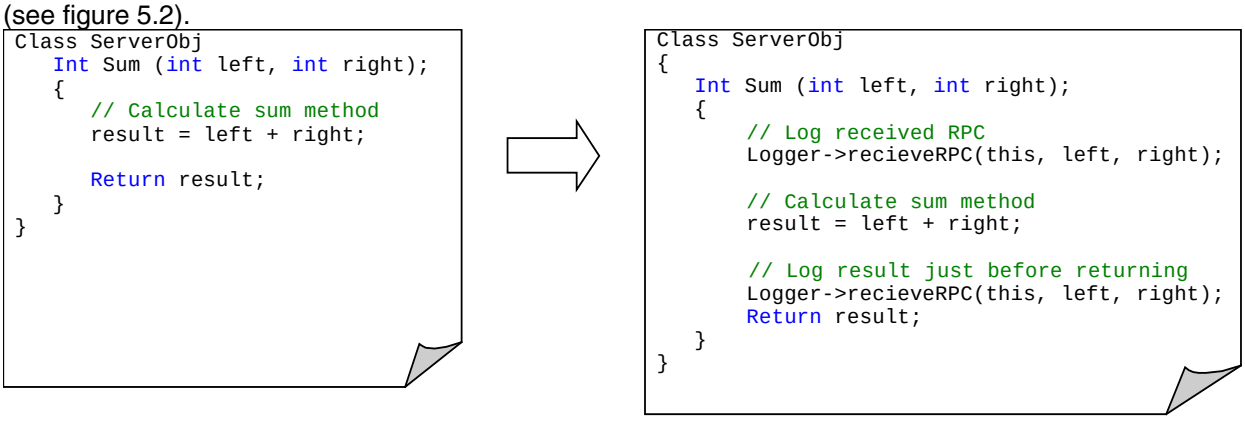


Figure 5.2: Instrumentation of server source code (before and after)

Another advantage of this technique is relatively simply to implement because no complex programming techniques are required. Another advantages of instrumentation of the component source code is that it allows access to all information in a component. This includes the all contents data exchanged between components which allows tester to verify the input parameters with the output parameters.

However, this technique is not very elegant because all the modifications blow up the source code, reducing the readability, and increasing the chances on mistakes. Although this technique is relatively simply to implement on the short term, in the long term it will cost time because manual instrumentation is labour intensive and error prone. Another disadvantage however, is that component instrumentation cannot intercept message communication like exceptions and other middleware communication events. Component instrumentation is therefore often used in combination with other tracing techniques, which do not have access to this information. Although this instrumentation technique can only be applied when the component source-code is available, this technique is middleware independent because it can be applied completely separate from any middleware architecture.

5.1.2 Pre-processing Instrumentation

By automating all instrumentation tasks, Pre-processing solves the some of problems, which plagued Debug instrumentation. Automated instrumentation can be achieved by using techniques like lexical and syntactic analysis to find the locations of interest and automatically instrument them with tracing code. The rules actually describe the location where a RPC enters or leaves an object, can be defined by describing the tracing points and program, script, etc that must be inserts the instrumentation into component code. Using a parser with access to a middleware interface repository could also automatically derive this information. An example of where this principle is applied is in the TMT Monitor framework that allows automated tracing for the Middleware standards DCOM, Corba, and Java RMI [TMT]. However, adding extra lines of tracing code in the original source code is not a very elegant solution. The source code becomes longer and becomes tied with a specific tracing technique. To un-tie a component implementation from the instrumentation technique, we have to separate the original source code from the instrumented source code. The separation between original file and instrumentation is achieved by using a pre-compiler that makes a copy from the original source file, which it instrument with tracing code. Figure 5.3 visualises how the instrumented file produced by pre processor would have to be further compiled into an object file by a traditional language compiler. This technique would separate the original clean source code from the instrumented source code creating a transparent instrumentation technique, which does not alter the original source code. However, because the source code must still be recompiled, it is not fully transparent.

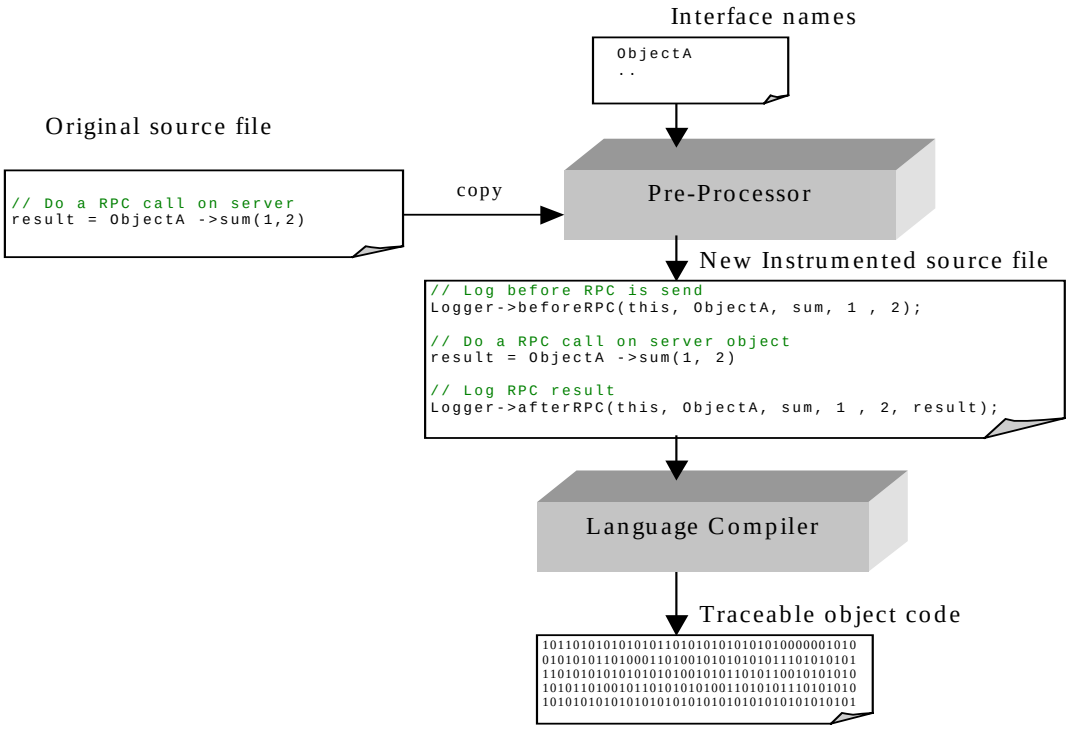


Figure 5.3: separation between original source and instrumented source

5.1.3 Adapter Instrumentation

An Adapter is an object that does not implement target interface itself but only performs some tracing actions and wraps to interface similar with the original interface. By offering the same interface as the target object, a Corba object can use the adapter exactly as the original. Because an adapter conceptually located between the Corba object and the target object, it can be used to intercept all communication flowing between an application and target object. An example of how the adapter programming technique is applied is during the connection between client and server. Adapter wraps Corba Interface object reference and creates a proxy object between the application and the stub/skeleton interface object. Because the proxy object will be invoked every time information flows between application and stub/skeleton, it can be used to log all communalisation flowing between client and servant interface. Besides intercepting message contents, the proxy object can also generate context information by linking intercepted parameter values with the component name and unique identifier. In chapter 7 we will see how a tools uses this technique is to collect tracing data (see also paragraph 6.2.2).

5.1.4 Stub and Skeleton Instrumentation

The standard Corba object model uses a fixed stub generation mechanism. An IDL post processor can instrument these statically generated proxies with tracing code, which intercepts and logs all communication events. The advantage of this technique is that it requires neither modification of the component, nor modification of the middleware libraries. However, changing the stub code is not easy because it requires specific knowledge about the stub communication mechanism and costs a lot of time to modify the code every time the IDL file is compiled. Although stubs and skeletons generated by a Java IDL compiler are equal, a separate instrumentation is required for stubs and skeletons generated by IDL compiler generated in different languages, like C++. Another problem is that this instrumentation technique can only be applied on stubs or skeletons generated by an IDL compiler. Dynamically created stubs, generated by the DII and DSI (see paragraph 4.1.5.2) cannot be instrumented and therefore not extended with tracing functionality. In chapter 6 we will see how the DSC Toolkit uses this instrumentation technique to collect tracing data (see also paragraph 6.2.1).

5.1.5 Middleware Instrumentation.

Because Corba middleware is responsible for establishing and maintaining a communication links Corba objects, they have access to all Corba communication events. Some Corba distributions like ACE TAO or JacOrb, provide the complete middleware library source code. With enough understanding of the Corba implementation, developers can instrument these libraries to extent a middleware with additional tracing functionality. Once the instrumented libraries are recompiled, all components using the instrumented libraries can be transparently traced without any changes the components source code. Note however, that depending whether or not these libraries are static or dynamically loaded, component need to be recompiled (static) or not (dynamic) for first time usage. The main problem of middleware instrumentation is that it effectively creates a proprietary middleware solution, which will have to be maintained. Moreover, the effort and cost of changing to another middleware implementation greatly increased or becomes impossible by the lack of middleware source code. Because Corba is still under development, it is expected that vendors will continue create new versions of their Corba implementation, which increases performance, repair bugs or add new Corba features. Beside that, modifying the middleware is often simply not possible because the vendor does not supply the middleware source code. Therefore, in order to remain middleware transparent, we have to stay away from any proprietary middleware instrumentation solution.

5.1.6 Interface Wrapping.

One simple technique that is commonly used to trace the communication between components is to use a Wrapper. A Wrapper is a component, which function a proxy between client and server and which can log all invocation communication between client and server. The basic strategy is to rename the original component to a different name and to introduce a wrapper, which identifies itself as the original component. In Corba, this is accomplished by changing the server registration in the naming service in the following way. First, the original server, which interface has to be monitored, is deleted from the naming service and reregistered under different name. Then, a new component, called the wrapper, is registered with the name of the original server. The result is that any client that wishes accesses the monitored server will automatically call the wrapper, which logs all invocation before it finally calls the original server interface. However, because all clients that use the same server will call the wrapper, and the wrapper has no method of knowing where invocation originated from, the intercepted invocation cannot be linked with a single component instance, and is therefore not uniquely identifiable. Note that in order to make the interception mechanism work, the Wrapper must take special care to perform its renaming actions after the original server has been registered but also before the client can retrieve the original object reference, otherwise a client will never call the Wrapper. Although, a Wrapper will alter neither

the client nor the server source code, the message arriving at the server is not exactly the same as the message that was send by client. This is a problem because the message send between wrapper and server is a new message, and therefore not message transparent.

5.1.7 Network Message Tracing

A completely different technique of intercepting trace data from a distributed system is to use network sniffers. Network sniffers can collect and analyse Corba network messages by listening directly to all communication on Local Area Network (LAN). By intercepting these network messages and decode them based on their IIOP message protocol (see appendix B for details), it will be possible to deduce the method invocations between components. The advantage of this technique is that it is completely transparent because it does not alter a system in any way. However, the interception will be restricted to the information contained in the IIOP. The problem is that except from opaque message identifiers IP address and port number, a message cannot pinpoint the exact source of the invocation. Another problem is that network sniffers can only capture broadcasted IIOP message travelling between components hosted by different computers. In the next chapter, we will give an example of a network sniffer.

5.2 Corba Meta Programming Mechanisms

Now that we have shown some common trace data interception technique, we will discuss how the Corba middleware itself can facilitate tracing. We will explain what Corba meta-programming mechanism is available and how they can potentially be utilised for constructing a tracing mechanism. We assume the reader has sufficient knowledge about Corba Middleware we described in the previous chapter.

Although the original Corba core provided many self-describing features like the interface repository (see paragraph 4.1.5.1), support for tracing, was simply not considered. Fortunately, the Corba community is starting to realise that we need (meta-object) additions to Corba core that will make tracing possible. The result is that the Corba community has introduced several new meta-programming mechanisms. Corba can use these meta-programming mechanisms to improve the adaptability and flexibility of distributed application, while avoiding obstructive changes to existing applications and middleware.

In order, to modify the behaviour of Corba middleware in a transparent manner we need ways to change the meta-objects that lay on the invocation path between client and server [Nanbour]. Meta object are programmable objects, that refine the capability of base-level objects, which are the objects comprising the bulk of application programs [Zimmerman]. As all operation invocations pass through meta-objects, certain aspects of application and middleware behaviour can be adapted transparently when system requirements and environmental conditions change by simply modifying the meta-objects.

In a distributed objects middleware environment, stubs, skeletons, and certain points in the end–to-end invocation path can be treaded as meta-objects because they can be modified. When a client invokes an operation on a server, a stub implemented as a meta-object can act in conjunction with transport-protocol meta-objects to access and/or transform a client operation invocation into a message and transmit it to a server. Corresponding meta-objects on the server apply request transformations on the operation invocation message and dispatch the message to its servant. An invocation result is derived in a similar fashion in reverse direction.

The following meta-programming mechanism can be used for the interception of trace data:

- *Request Interceptors*, are embedded hooks into the Orb that can intercept all messages passing through them. They allow the forwarding and inspection of messages, and can be used to transparently transfer contents information through the Corba application.
- *Smart Proxy*, are customised meta-object which function as proxy object between a client and specific servant interface
- *Servant Managers. Is a meta-object*, which can be used to forward or inspect the message contents just before and after it replies. Servant managers can also detect the first and last time the servant object instance is used and use it for the initialisation or destruction of other processes.
- *Pluggable Protocol Framework* is a new Meta programming mechanism, which can modify the communication protocol between Orbs.

Figure 5.4 gives a good overview of the several meta-programming mechanisms described above. Note that the figure also visualised the location of the CCM container. That is because Corba Component Container Framework (see also paragraph 4.2.1) can be used as meta-programming mechanism to trace the communication between the components. A Container extended with build-in logging service, which collect tracing data automatically, can implement by using request interceptors or servant manages.

Another mechanism that can be used to trace Corba communication is Orbix Message Filters mechanism. Message Filters is a propriety Orb mechanism supported only by a small group of Orb vendors (Orbix / VisiOrb). Similar to a request interceptor, a Message Filter can intercept communication events at the Orb. However in contrast of Request Interceptor, it functionality is limited to its interface. The advantage of Message Filters is that it does not require the registration or initialisation of any meta-object, but can be activated directly in the Orb. Because this mechanism is effectively a modification of standard Orb behaviour, and not a meta- programming mechanism, we classify it a middleware instrumentation technique.

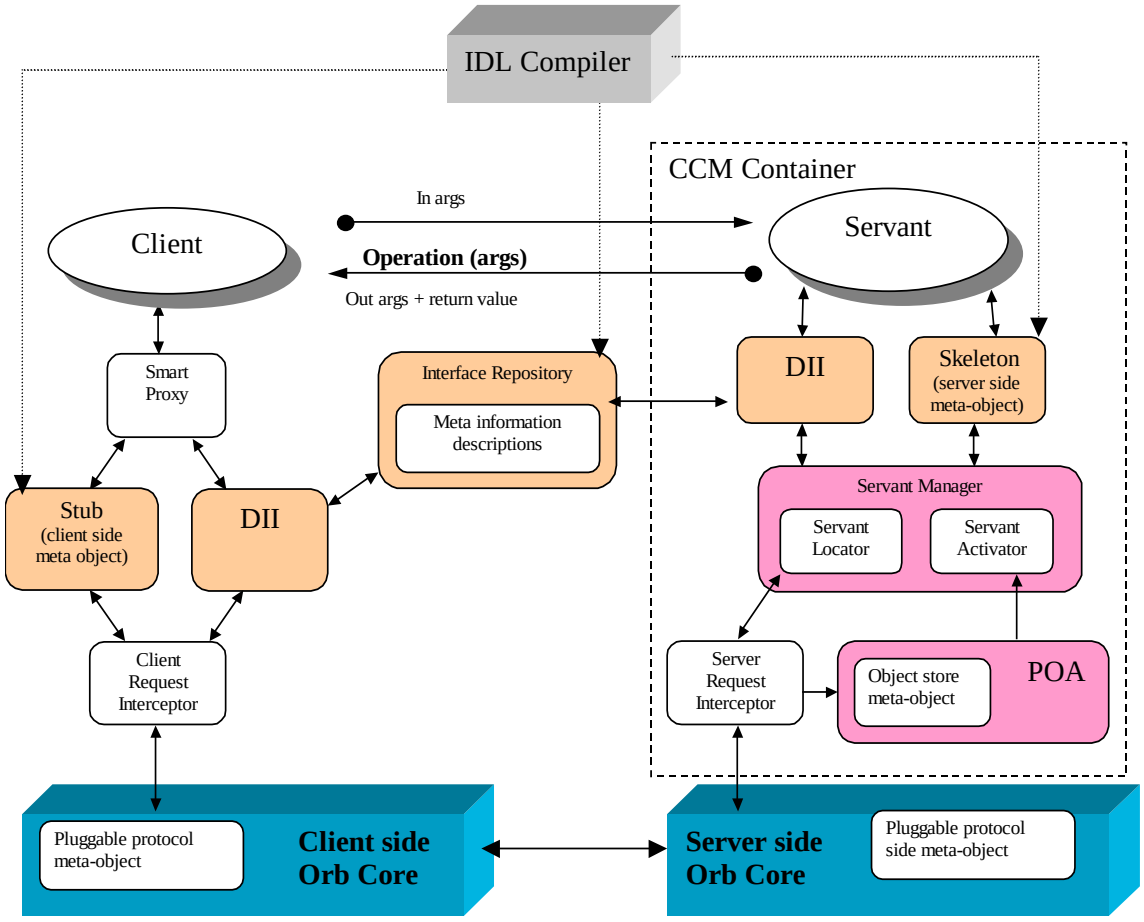


Figure 5.4: Meta programming mechanism in the Corba Middleware

Each meta-programming mechanism will be further detailed in separate subparagraph.

5.2.1 Request Interceptors

The Request Interceptor is part of Corba standardised Portable Interceptor specification. Beside the Request Interceptor, the Portable Interceptor also defines a second type of interceptor, named the IOR interceptor (see appendix L). Request Interceptors are a meta-programming mechanism used in Corba to increase the flexibility of both client and server applications [Nanbor]. This mechanism is useful for making invocation decisions, checking invocation rights, or performing other operations that apply to the entire request as a unit. Request Interceptors were introduced as an answer to add specific network-oriented capabilities such as authentication and flow control to distributed applications.

Portable Interceptors are not usually part of a typical Corba environment, implementing them is considered an advanced programming task. Interceptors must be registered before the Orb in the client or server application becomes initialised. This is because in contrast to the intuition, Portable Interceptors are not linked to a Corba object, but to the Orb object used by the Corba object. After the Orb is initialised, all Request Interceptors installed on an Orb will affect all GIOP messages (see appendix B) passing through that Orb.

The Orb that is used by both client and server to establish a communication can be fitted with multiple independent Portable Interceptors. Just like triggers in a DBMS, Request Interceptors are triggered after certain communication events take place in the Orb. From abstract point of view, these interaction points are located at the boundaries of a component. Figure 5.5, shows how An interaction point can be an interface instance at entry or reference to a remote object at exit. Request Interceptors allow the programmer to execute its code, just after a message leaves an interaction point, or just before it enters an interaction point.

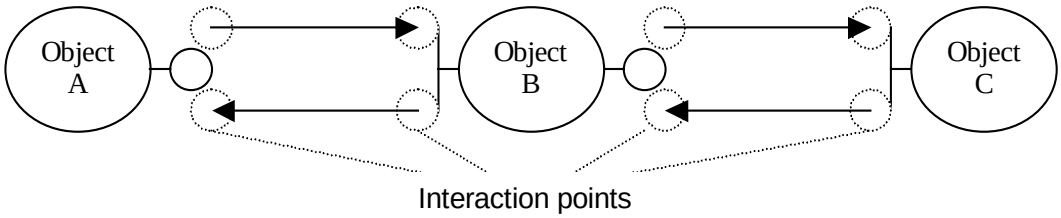


Figure 5.5: Object interceptor interaction points

There are two types of request Interceptors; the Client Request Interceptor and the Server Request Interceptor. The both Request Interceptors are activated at different stages in the request reply invocation. Figure 5.6 visualizes the different paths of invocation points a request reply message can activate. Request Interceptors function like hooks into the Orb through which Orb services can use to intercept the normal flow of execution of the Orb. There are ten different hooks from which six are used for very special purposes. We can see that each Request Interceptor is called at least twice during the request-response cycle of an invocation: once when a request is going from the client towards the servant, and once when a response returns to the client.

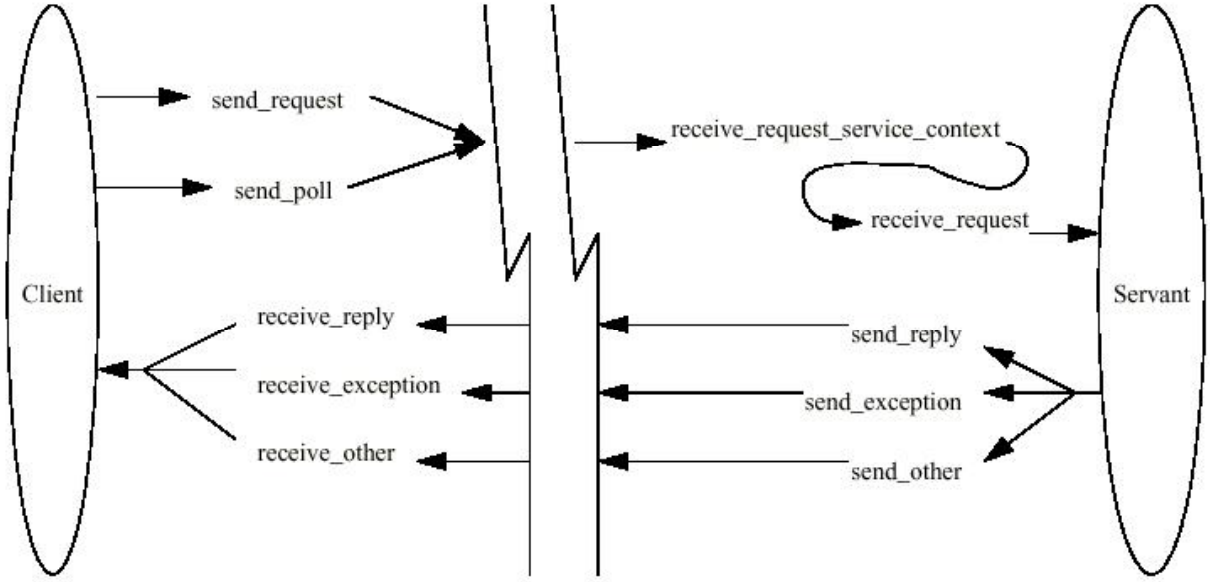


Figure 5.6: Possible flow through Request Interceptor Hooks

Depending on the message event, different methods (hooks) are called by the Orb:

- Before sending a request message at the client:
  - When the message is a Time Independent Invocation, the Orb calls `send_poll`
  - Otherwise, the Orb calls `send_request`
- Before a request message reaches the server, but before calling the servant manager:
  - The Orb calls `receive_request_sequence_contexts` (which allows access to service context fields)
- Before a request message reaches the servant:
  - The Orb calls `receive_request` (which allows access to all information available in the message)
- Before sending a reply message at the server:
  - If the servant throws an exception, the Orb calls `send_exception`
  - If the servant replies normally, the Orb calls `send_reply`
  - Otherwise (e.g. when the call is redirected), the Orb calls `sent_other`

- Before the reply message reaches the client:
  - If the client throws an exception, the Orb calls `recieve_exception`
  - If the client receives a normal reply, the Orb calls `recieve_reply`
  - Otherwise, the orb calls `recieve_other`

Beside the standard interception functionality, Request Interceptors also offer following advantages and possibilities:

- *Message Redirection*. A Request Interceptor can temporary or permanently redirect a request to a different location at any interception point other than a successful reply. This is useful for transparently redirecting a request to an exact copy of the initial target object when the initial object has gone offline. This error recovery behaviour is especially useful for creating fault tolerance services.
- *Message Blocking*. A Request Interceptor can stop a request from reaching the target by raising a system exception in the inbound path. A system exception function like an interrupt call in the sense that forces the Orb to stop the current transmission and take alternative actions. This functionality is especially useful for authorization by security services or for the construction of proxies.
- *Message Inspection*. A Request Interceptor allows read access to the request or reply fields located in intercepted GIOP message. Depending on the intercepted message this information includes but is not restricted to, target object, operation name, input parameters, output parameter, return value, and other fields (for details see appendix H).
- *External Invocation*. During its activation, the Request Interceptor can call other Corba Components or use standard Corba Object Services like the naming service or Interface Repository. This feature allows the Request Interceptor to exchange information with external services.
- *Exception Handling.* By inspecting message exception fields, Request Interceptors can be used for exception handling. This is both useful for Fault Tolerance services trying to recover form a communication failure, as well for tracing services, which want to log it as a tracing event.
- *Middleware Transparency*. In contrast to other instrumentation techniques, which require the modification of middleware, interceptors installed by parameters, do not change any line in the middleware source code. This is an advantage because if the middleware remains unmodified, the middleware also remains upward compatible with future updates from the Corba middleware vendor. Note however that the installation of interceptors by parameters is different for every Corba implementation.
- *OMG Standard*. The Request Interceptor has been part of OMG official Portable Interceptor standard for quite some time now. Although the initial request interceptor had a bad start (for details see appendix M), most members of the OMG now recognise the current Portable Interceptor specification as a Standard Corba facility. Most Orb vendors now support the OMG Portable Interceptor in their current Corba middleware implementation or plan to support in their future version. The advantage of using a standardised Corba meta- programming mechanism is that in contrast to proprietary vendor facilities, customers can switch Orb vendors, without too much technical difficulties.
- *Point to Point tracing.* Request Interceptors can be used to collect tracing data from both sides of a communication and in both directions. Other meta-programming mechanism, which can also be used for tracing, only allow tracing at one side. For example, Smart Proxies only allow tracing at the client side while Servant Managers can only trace at the server side. In addition, it is important that intercepted includes all Orb-to-Orb information flows, not just communication between systems. This might seem trivial but a tracing technique like sniffing are only able to detect the communication between Component hosted on different systems, not between component hosted on the same system.
- *Communication Diagnosing.* The ability to detect all incoming and outgoing messages allows Request Interceptors to detect exceptions, communication failures, and performance problems. Communication failures can be detected by messages that are sent but never reach their destination. Performance problems can be detected by determining the latency of the message communication or the processing latency between receiving and sending messages. All this diagnosing information allows the testers to verify the correct communication behaviour of both components as well as the used middleware services.
- *Piggybacking*. The Request Interceptor offers two facilities which tracing framework can use to transparently propagate context data through a distributed Corba Application. Request Interceptors can send service context data embedded in GIOP messages without modifying the client, server, or Orb (for details see appendix I). In order to transfer the tracing data received from incoming messages to outgoing messages, request interceptors can use the PICurrent. This shared memory mechanism allows services to transfer context data between the interceptor context and application context (for details see appendix J). Propagating tracing data through a system is invaluable for tracking the causality relationship between individual messages travelling between Corba Components. By transparently propagating tracing context information containing a logical clock value, a tracing mechanism is able to determine the exact sequence of intercepted communication events.

Although the functionalities described above make Request Interceptor a powerful meta-programming mechanism, we have found many limitation and problems. The foremost reason why interceptors have these build-in limitations is that the integrity of the intercepted messages has to be secured. Another reason of its shortcomings is that they were originally developed only to facility the minimum functionality required to allow security services to operate. Although the literature often mentioned them as potential mechanism for tracing Corba, they were never designed for this purpose.

We can summarize the limitations and problems of Request Interceptors as follows:

- *Portability problem*. Although the Portable Interceptor specification has been around for quite some time now, there are still Orb vendors, which do not comply with the official OMG standard. Non-portable interceptors often were the result of the first interceptor specification (for details see appendix M). One of the main reasons why these propriety non-portable Interceptors still exist is because these pioneer Orb vendors have obligation to their existing customers, which rely on their proprietary functionality. Although most proprietary Request Interceptor implementations seem only different in name, they also apply different functionality rules during message interception making migration to OMG compatible Corba implementations complicated. Another problem is that Portable Interceptors must be registered in the namespace of an Orb instance. This means the registration of Portable Interceptors is strongly connected to that of the Orb. This is a problem because this mechanism is not the same for all Orb implementations and even the specification standard is not programming language independent.
- *Overhead problem*. Although a Request Interceptor (which does not throw any exception) does not alter the behaviour of the Corba middleware, depending on the implementation of the installed interceptors, it imposes a small or larger degree of increased overhead to all Corba communication causing the overall system to degrade. In comparison to other Corba meta-programming mechanisms, Request Interceptors cause a relative large amount of latency. This is because the large number of interfaces that are called during an invocation and because the information related to the request is bundled into `anys`, which have a higher parameter conversion overhead for their insertion and extraction operations.
- *Limited installation.* Interceptors can only be installed by registration an interceptor into the Orb before initialisation, not after or during initialisation. Therefore, in order to register an interceptor afterwards, a client or server would first have to halt and reinitialise their Orb. Although most Orb implementations allow the registration of interceptors through parameters, the official PI specification only defines how to register on interceptor by method calls. This is however, a direct violation of a transparency because a component source code must be altered. In addition, after altering the source code of the component, it must be recompiled, and linked before usage. Online Corba Component must therefore always be shut down and restarted before request interceptors can do their work.
- *Limited configuration*. A request interceptor must always implement all interface methods defined by the Portable Interceptor specification. Although an method implementation of a Request Interceptor can be empty, upon the arrival of any communication event, the will Orb still invoke all methods before returning to its original tasks. Request interceptors have also no facility for uninstalling themselves at runtime. Therefore, once a Request Interceptor is installed on an Orb, it will always intercept all request and replies exchanged between client and server until the Orb is brought offline. Because an interceptor cannot be uninstalled at real-time, even if interceptor does nothing, it still causes 10% latency hit.
- *Limited ordering.* While interceptors may be ordered administratively, there is no concept of order with respect to the registration of interceptors. Because the invocation order is Orb implementation dependent, the order in which the interceptor interfaces are called, cannot either be inspected or modified. Although there most Orb implementations that actual allow to change the order of request interceptors in a customisable fashion, assumptions on the order of invocation of multiple interceptors result in non-portable interceptor code.
- *Limited internal communication.* Corba objects are not limited in the number of Orbs. Although, request interceptors located on a same Orb can make use of the PICurrent mechanism (see also appendix I) to exchange context information among them selves, Corba does not supply any communication mechanism, which can share information between Orbs that have no established connection. This means Corba objects using multiple Orbs must use proprietary techniques to share context information.
- *Limited external communication.* Although Request Interceptors can communicate with the outside world by doing invocations themselves, there is no standardised facility for external processes to exchange information with an active Request Interceptor.
- *Limited parameter modification.* Although Portable Interceptor can add new service context and access service context data inserted by other Request Interceptors, it cannot alter this service context. In addition, request interceptors can read input parameters, but they cannot affect a request by changing a parameter or affect the outcome of an invocation by supplying the response itself.

- *Limited blocking*. Although Request Interceptors can block message from reaching its destination, it is limited to using the standard exception technique. However sometimes alternative blocking techniques are desired. For example, when a service wants to temporary block a message because the host is currently busy with a high priority task.
- *Limited exemption handling.* When an Orb, fitted with multiple request interceptors receives a message, and one of the interceptors throws an exception, all remaining interceptors, which are not called already will not be able to intercept a message or notified of the exception. Therefore, caution must be taken for any Orb, which is fitted with a Request Interceptor that can throw exceptions.
- *Limited forwarding addressing.* A forward exception can only redirect message to a single alternative address. In addition, although Request Interceptor can forward a message to another servant object, which can answer to the same interface call, it cannot forward a message back to the client. The reason is that a client is not addressable and can therefore not receive rerouted messages.
- *Java contents access problems*. The main problem of request Interceptors on a Java Orb is the impossibility of accessing contents information in the message like input, output, and return value. Java request interceptors can only access the operation name, concerning the signature of an operation. Another potential problem is the fact that JAVA interceptors can only be installed with parameters.
- *Limited identification*. Although most Corba objects will use a single unique Orb, the Corba specification does not forbid a Corba object from using multiple Orbs. Worse, multiple Corba objects can share the same Orb. Because a client request interceptor cannot be associated with a single component instance, they cannot be used reliably for the identification of component instance.

In the last paragraph of this chapter, we will evaluate the Request Intecept with other trace-data interception techniques.

5.2.2 Smart Proxy

A common problem in Middleware technology is that an application needs to add extra client specific tracing functionality, which should be logically part of the middleware architecture, but the client implementation code is not available. Although this functionality could be achieved by extending the stub source code with tracing functionality, we explained that there are several disadvantages to this instrumentation technique (see paragraph 5.1.4). What developers really need is a transparent meta-programming solution, which allows an application developer to change the functionality of the stub selectively without actually modifying the generated stub, middleware, or client source code [Nabour]. The solution to this problem is provided by ACE TAO, is the Smart Proxy meta-object [Koster].

The Smart Proxy is similar to the Adapter instrumentation technique (see paragraph 5.1.3) in the sense that it allows developers to modify the behaviour of interfaces. Similar to the Adapter instrumentation, a smart proxy object has a one-to-one connection with a Corba object, which allows a tracing mechanism to identify a client object by it Smart Proxy identity. In figure 5.7, we can see how this meta-programming mechanism, which allows the developer to transparently override the default stub implementation, is located between the client and the default client proxy (the stub). A Smart Proxy will not replace the functionality of the default stub (marshalling parameters and de-marshalling the return value), but will delegate standard stub tasks to the default-generated stub. Although a smart proxy use a similar proxy technique use by the Adapter instrumentation, in contrast to adapter instrumentation, smart proxies can achieve this functionality without modifying the client or target objects.

The Smart Proxy is also comparable with the Request Interceptor (see paragraph 5.2.1) in the sense that both can participate in the communication between a particular client and a particular servant. However, in contrast to the Request Interceptor, a Smart Proxy is solely a client mechanism. That is because the only invocation point occurs whenever an operation is invoked through a stub and is therefore called only once during each invocation and once during a reply. Note however, that in contrast to the Request Interceptor, which can intercept Orb message communication, a smart proxy can only intercept method invocations

Although the Smart Proxy offers relatively little freedom in modifying any details of the call, they could perform action like collecting tracing data, throwing user exceptions or use other Corba services like access the Current. The Current allows the Smart Proxy to communication with installed Portable Interceptor Current (PICurrent).
In comparison to request interceptor, which causes a relative large overhead, Smart Proxy imposes a minimal overhead on a distributed application. However in contrast to request interceptors, a client can only have a single smart proxy for each interface whereas multiple interceptors can be registered with an Orb.

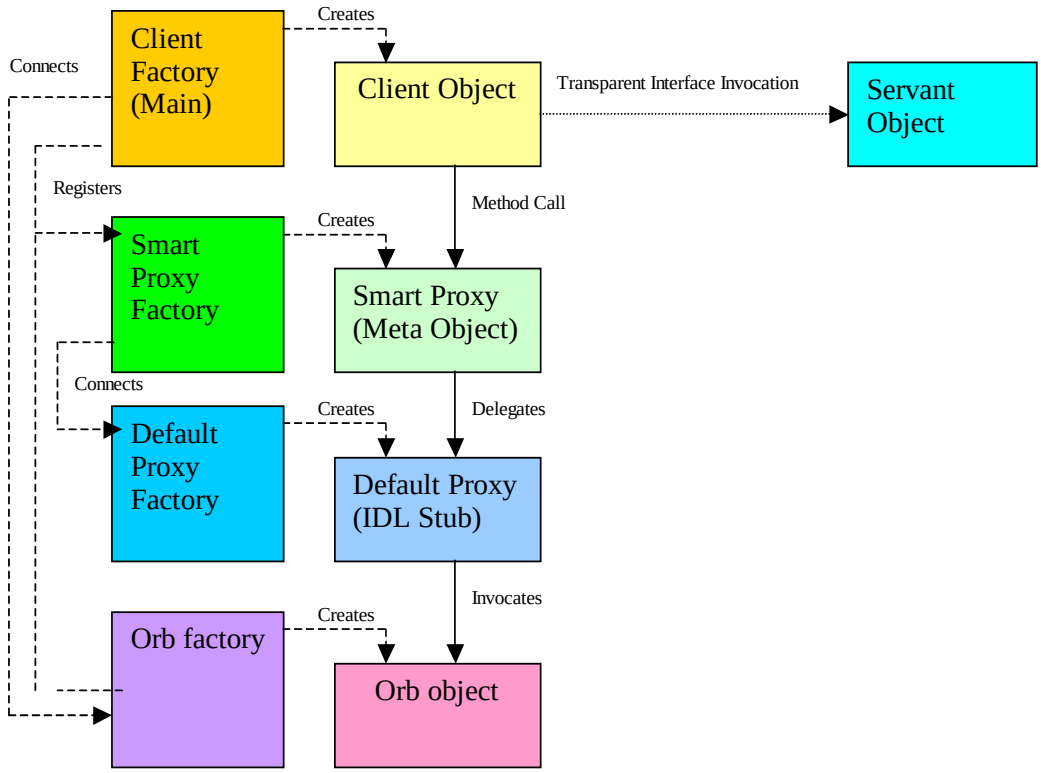


Figure 5.7: Smart Proxy Architecture

In figure 5.7 we can also see that the Smart Proxy consists of two parts, a Smart Proxy factory class, and a Smart Proxy meta-object. Similar to the request interceptor, which is registered to the Orb by a special factory class, a Smart Proxy is registered to the object reference by a smart factory class. The Smart Proxy factory does not have to be written manually but can be automatically generated by the TAO IDL compiler for every interface in the IDL file.

The activation of smart proxies can be achieved the following methods:

- *Automatically,* by the Orb. During initialisation, the Orb will activate the Smart Proxy factory that creates the Smart Proxy meta-object whenever a client application connects to a target interface with the `_narrow` operation. This can be achieved using TAO Component Configuration pattern [Schmidt] and adding an entry to `svc.conf` configuration script. Although this method is fully transparent for the client, it only allows a per-interface policy. In the per interface policy, the Smart Proxy is used for all targets objects associated with a particular IDL interface.
- *Manually,* by first adding a smart proxy factory initialisation-call after the Orb is initialised, and then add a separate initialisation call for each object reference that needs to be traced. Thus, if smart proxies are installed before a client accesses these interfaces, the client can transparently use the new behaviour of the proxy returned by the factory. Although manual instrumentation is less transparent, it allows a per-object policy. In the per-object policy, each object can have a separate smart proxy, which is less transparent but more flexible. This fined grained control will be especially useful for trace configuration because it allow a trace framework to exclude any object, which are not of interest to the test case.

Although the Smart Proxy is currently not an OMG standard, the ACE consortium (the developers of the TOA) has made a request for proposal at OMG, called Smart Proxies. TAO, is the currently not the only Orb vendor which implements a Smart Proxies model. Many other Orb vendors support this feature as a proprietary solution or are planning to implement in future releases of their middleware. We expect to see a similar process that took place after the introduction of Portable Interceptors; initially many proprietary smart proxy solutions will emerge but eventually most Orb vendors will conform to the new standard once the official OMG specification becomes accepted

5.2.3 Servant Manager

The Servant Manager is just as the name suggests, a mechanism that manages the servant object located at the server. This meta-programming mechanism allows server application developers to strategize the selection, loading, unloading, and activation of object implementations. The Orb will invoke methods on servant managers to activate or deactivate servants on demand.

A servant manager is similar to a server-side Request Interceptor in several respects. They both can intercept requests and affect the outcome of request invocations before they are dispatched to servants. Like the Request Interceptor, an invoked servant manager method could raise forward exception to forward a request message to another servant. Invoked methods can also create a new servant instance on the fly (see figure 5.5). Thus, the servant manager may choose or even create a servant for processing the request at real time. This mechanism is especially useful for load balancing or fault tolerance schemes, since it enables the simultaneous existence of many parallel servants from which the servant manager may choose.

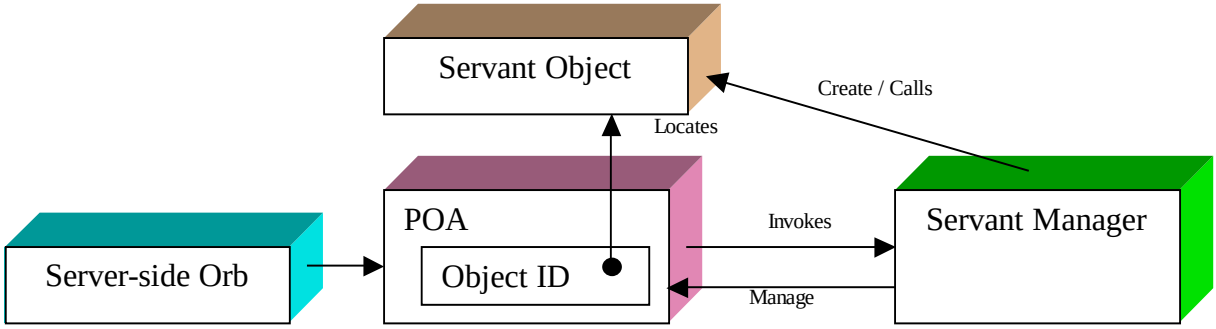


Figure 5.8: Managing Resources with a Servant Locator

However Unlike Request Interceptors, Servant Managers only affect the Portable Object Adapters (POA) that install them and can therefore only provide access to a limited subset of request information [Wang]. As a result, they are more tightly coupled with POA's and servant implementations than Portable Interceptors. Rather than modifying the internal Orb behavior, Servant Managers are used primarily to coordinate the resources necessary to activate and deactivate servants. Because of its integrated relation with the POA, the Servant Manager is part of Corba official POA specification standard [Corba 2.4].

A servant manager is registered with a POA as a callback object, to be invoked by the POA when necessary. An application server that activates all its needed objects at the beginning of execution does not need to use a servant manager; it is used only in case an object must be activated during request processing. Servant Managers are therefore responsible for managing the association of Object Id value (a unique number in the POA's Active Object Map) with a particular servant and for determining whether an object exists or not (see figure 5.8).

Depending on the active server policies, the POA uses two kinds of servant managers, the *Servant Activator* or the *Servant Locator*:

- *The Servant Activator* is activated the first time a servant is activated. This Servant Activator provides two methods, a constructor and destructor method called `incarnate` and `etherealizes`. The Orb calls the `incarnate` method when a client wants to access the servant for first time usage. The `etherealise` method is the opposite of the `incarnate` method, in that it is called when a servant object is deactivated or no longer needed by the POA.
- *The Servant Locator* is invoked every time a request is made to the servant. The Servant Locator provides two hook methods; a pre-condition method and post-condition method called `preinvoke` and `postinvoke`. The `preinvoke` method is called by the POA to locate a servant before every request on an object. The `postinvoke` method is as expected called after every request when a servant is no longer in use.

5.2.4 Pluggable Protocol

The Pluggable protocol framework is another meta-programming mechanism recently introduced by ACE TAO Orb [Dougles]. Just like smart proxy framework, Pluggable protocol framework is in the process of becoming a standard by the OMG in the Extensible Transport Framework specification [ExtTrans]. The pluggable protocol framework makes it possible to separate the component architecture from the communication protocol of the

Orb [Kuhns]. This allows developers to add new protocols without requiring changes to Corba components. In order to achieve this, the component architecture and high level Corba services access the communication protocol using the facade design pattern, which should make the replacement Corba existing communication protocol relatively easy [Gamma].

Pluggable Protocols are intended to overcome the shortcomings of the standard IIOP and GIOP middleware communication protocol (see also appendix B). However, the standard CORBA GIOP/IIOP interoperability protocols are not well suited for applications that cannot tolerate the message footprint size, latency, and jitter associated with general-purpose messaging and transport protocols. Fortunately, the CORBA specification defines the notion of "Environmentally-Specific Inter-ORB Protocols" (ESIOPs) that can be used to integrate non-GIOP/IIOP protocols beneath an ORB. To allow end-users and developers to take advantage of ESIOP capabilities, it is useful for an ORB to support a Pluggable Protocols framework that allows custom messaging and transport protocols to be configured flexibly and used transparently by applications. Figure 5,9 illustrates how the high-level communication protocols rely on the low-level communication protocols in the Pluggable Protocol architecture.

Pluggable Protocols could potentially be used in the construction of tracing framework. Similar to request interceptors (see paragraph 5.2.1), Pluggable Protocols offers a set of interfaces, which are triggered by the Orb. Similar to smart proxies, higher-level application components and Corba services use the Component Configurator pattern [Schmidt] to dynamically configure custom protocols into TAO's Pluggable Protocols framework without requiring obstructive changes to themselves or the Orb. In order for a pluggable protocol to be usable by TAO without making any modifications to TAO itself, its interfaces must be fully implemented to provide the functionality dictated by TAO's pluggable protocols framework interface. This functionality is implemented within several components, namely the Acceptor, Connector, Connection Handler, Protocol Factory, Profile, and Transport.

Especially the Transport Components is useful for tracing purposes because it is activated during Orb-to-Orb or Orb-to-Object communication. During these events, the Pluggable protocol is able to log a communication message source, destination, and parameters. Unfortunately though, creating a new protocol is quite tedious and error prone [Mann]. In contrast to portable interceptors and smart proxies that offer high level interfaces, Pluggable protocols are much harder to implement because they deal directly with the Corba communication infrastructure. Because of the low level the communication protocol operates on, it is not easy to recognise part of the original high-level message. Therefore implementing a pluggable protocol involves exactly the kind of low-level error-prone programming aspects that middleware is supposed to shield the developer from. However, in contrast to portable interceptors and smart proxies, which alter the semantics of objects, pluggable protocols framework can only alter the underlying Orb transport mechanism. Thus, they do not permit fine-grained control over objects since they affect all objects in an Orb and it is very hard to vary the transport mechanism at the level of object references [Nanbour].

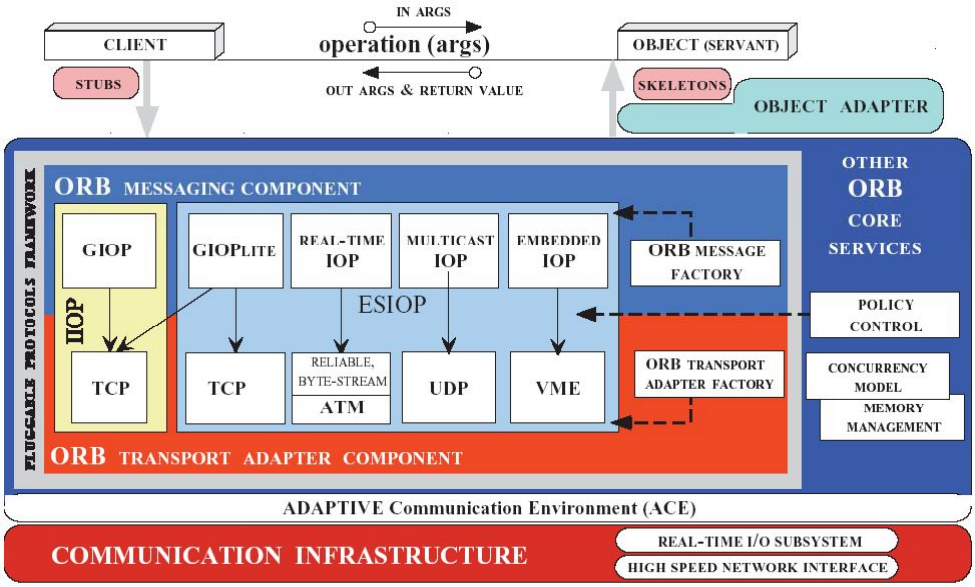


Figure 5.9: Pluggable Protocol Framework Architecture

5.3 Comparison of Trace Data Interception Techniques

We will now make a comparison of all trace data interception techniques we discussed in the previous paragraphs.

We will compare the based on the following characteristics with scales:

- Transparency. Determines the level of transparency and effect on the distributed system:
  - -- Client/Server source code is instrumented with many modifications
  - - Client/Server/Middleware source code is instrumented with minimal modifications
  - +/- Orb/stub/skeleton source code is instrumented but the client/server source code isn't
  - + Not any source code is instrumented but the Orb must be proprietary initialised
  - ++ The distributed system does not require any alteration
- Middleware independence. Determines the level of independence on the Orb vendor:
  - -- The techniques is proprietary and there are no attempts for standardisation
  - - The techniques is current proprietary but is being standardised.
  - +/- The techniques can be implemented in any open source middleware
  - + The techniques is standardised but not implemented by all Orb vendors
  - ++ The techniques can be applied on any distributed system
- Language independence. Determines the level of programming language independence of a technique.
  - -- The technique is only possible in one specific language
  - - The technique is bind to one specific language, but future binding are expected
  - +/- The technique is bind to on specific language but can be adaptable to other languages
  - + The technique is available in multiple commonly supported languages
  - ++ The technique is possible for any language
- Tracing Overhead. Determines how much will the tracing technique affect the system under test.
  - -- The technique will halt or alter the distributed system resulting in different system behaviour
  - - The technique will slow down the distributed system altering some system behaviour
  - +/- The technique will slow down a distributed system but not significantly
  - + The technique will lower the distributed system performance but almost non detectable.
  - ++ The technique will not alter distributed system at all
- Context Piggybacking. Determines the technique ability to transmit service context along messages:
  - -- The technique cannot be used for transmitting service context
  - - The technique can transmit context with parameters by wrapping the interface externally
  - +/- The technique can transmit context with parameters by wrapping the interface internally
  - + The technique can transmit context by populating the service context field
  - ++ The technique can transmit context by modifying the service context field
- Message community. Determines the level of observability of communication events in a system:
  - -- The techniques is only able to detect communication event between computers
  - - The techniques can only detect message entering or leaving at only side
  - +/- The technique can only detect message entering or leaving at both client and server side
  - + The technique can detect All communication events between Orbs
  - ++ The technique can detect All communication events in the Orb
- Identity Tracing. Determines the ability to detect the source and destination of a message.
  - -- The technique can only detect the interface of the destination.
  - - The technique can detect the server IOR and client IP / port address.
  - +/- The technique can detect the server IOR and client Orb ID
  - + The technique can detect the client or server object
  - ++ The technique can detect both client and server object
- Technique Complexity. Determines the difficulty of implementing the trace technique in practice.
  - -- The technique is difficult to implement implementation and different for every case
  - - The technique is difficult to implement but is repeatable for different cases
  - +/- The technique requires some technical expertise to implement but not too difficult
  - + The technique is relatively easy to implement but still requires some time to create
  - ++ The technique does not requires technical expertise except from its deployment and usage
- Cost Effectiveness. Determines the long-term cost effectiveness of trace technique.
  - -- The technique is labour intensive, very complicated and awkward to implement
  - - The technique is labour intensive, error prone but manageable.
  - +/- The technique is relatively easy to implement but still requires manual work
  - + The technique is semi automated and only requires some initialisation
  - ++ The technique is fully automated and does not require any deployment activities

In the table below, we compare all discussed tracing techniques. Note that for comparison reasons, we also included the language level debugger.

Trace Technique	Interoperability				Observability				Deployment	
	Transparency	Middleware Independence	Language Independence	Tracing Overhead	Context Piggybacking	Message communication	Identity Tracing	Contents Tracing	Technique Complexity	Cost Effectiveness
Language Level Debugger	++	+	+	--	--	+/-	++	++	+	--
Debug Instrumentation	--	++	++	+	-	+/-	++	+	+	+
Pre processing	-	++	+/-	+	-	+/-	++	+	--	+
Adapter Instrumentation	-	+	+/-	+	-	+/-	+	+	-	+
Stub/Skeleton Instrumentation	+/-	+	+/-	+	+/-	+/-	+	+	-	+
Middleware Instrumentation	+/-	+/-	+	+	++	++	+	+	--	++
Interface Wrapping	+	+	+	--	-	+/-	--	+	+/-	+/-
Network Sniffing	++	++	++	++	--	--	-	+	++	-
Request Interceptor	+	+	+/-	-	+	+	+/-	+/-	+/-	+
Smart Proxy	+	-	-	+	--	-	+	+	+/-	+
Pluggable Protocols	+	-	-	+/-	++	++	+	+	-	+
Servant Manager	-	++	+	+	--	-	-	-	+/-	+
Message Filters	+	--	+/-	+	--	+	-	+	+	+

Concluding we can say there is not a silver bullet which allows use to develop the ideal tracing mechanism. Although we can combine some techniques to increase the system observability, it will also inherit the lowest level interoperability and deployment of the combined techniques. For example, a combination of the Request Interceptor with the current Smart Proxy would create a tracing mechanism with excellent observability but with mediocre interoperability. One particular combination that is interesting to notice is the combination of Stub/Skeleton instrumentation with Request Interceptor. They complement each other on observability and both offer reasonable good interoperability and therefore are in our view the best combination to create tracing mechanism. Note that if smart proxies or Pluggable Protocols become an official OMG standard, their value as a trace-data interception technique would be increased dramatically. They would allow the construction of a transparent, interoperable tracing mechanism. Unfortunately, history has shown that the standardisation of a meta-programming mechanism does not make instantly interoperable between Orbs. After the standardisation of the Request interceptors, it tool several years before Orb vendors implemented it properly in the middleware solution. Because we cannot wait until that to happen, we need to look for other solutions.

Because there already exists a tracing mechanism, which uses a combination of stub/skeletons instrumentation and Request interceptor, we have looked how to create a tracing mechanism based exclusively on the Request Interceptor. In spite of all its problems, we still think Request Interceptor is still one of the best trace-data interception techniques currently available for Corba. In chapter 7, we show how far we can push the request interceptor to create a transparent tracing mechanism with maximum deployment and interoperability performance. However, before we do, we first investigate which tools can be used for testing Corba systems and analyse how useful they are as a testing solution for Thales in the next chapter.

6 AN EVALUATION OF CORBA TESTING TOOLS

In this chapter, we make an evaluation of Corba testing tools and compare them with each other.

The structure of this chapter will be as following:

- *Paragraph 6.1: Criteria for comparison of Test Tools*. In the first paragraph, we define the criteria we will compare the tools under evaluation.
- *Paragraph 6.2: Corba Test Tools Under Evaluation*. In this paragraph, we review five Corba tools, which can be used to test Corba applications.
- *Paragraph 6.3: A Comparison Of Corba Test Tools*. In the last paragraph we make a final comparison of the Corba test tools under evaluation based on the criteria set by paragraph 6.1.

6.1 Criteria For Comparison of Test Frameworks

In paragraph 2.3, we explained that traditional testing techniques would not work in a distributed system. Then what kind of tooling would we need in order to test Corba Applications sufficiently? In this chapter, we specify what functionality is desired in a good Corba testing tool.

We can divide the criteria's of the required testing tool into four criteria categories:

- *Message Controllability;* specifies the message control the testing framework should aim for offer to create an effective controllable testing environment.
- *Message Traceability*; specifies the tracing functionality the tool should aim for to allow tester to observe Corba message communication more effectively.
- *Tracing Framework Performance;* allows testers to use tracing mechanism more efficiently. An efficient tracing tool allows a tester to trace a system in shorter time with fewer problems.
- *Tracing Framework Interoperability*; specifies how the testing framework should be able to support testing functionality in different distributed environments.

The following four sub paragraphs will give a list of each requirement category

6.1.1 Controllability Functionality

A testing control framework with high Controllability contains the following control requirements:

- *Dynamic deployment.* In order to perform simple test case without changing the distributed system, the utility should be able dynamically connect and invoke a component interface without recompiling any source files or restarting any applications. To assist the tester in the invocation process, the testing utility interface should allow the tester to browse through exiting components collection and review available interfaces.
- *Manual invocation*. Testers require a quick user-friendly Actor/Reactor utility, which allows them to manually create a request or reply message. Once an interface is selected, the tool could further assist the tester by restricting the parameter input.
- *Automation invocation*. In order perform regression tests or perform complex test cases, the control framework should be able to execute a test script by actor and reactors utilities. The test script could be created by the tester by manually, generated from some static test definition language, or recorded from manual invocation.
- *Workload invocation*. Simple test scenarios often do not represent the real world. Real world distributed application must cope with multiple invocation at the same time. The testing framework should therefore be able to create a similar environment by generating an invocation workload. This would allow testers to verify both functional behaviour as well as the performance of the system under stress.

6.1.2 Message Observability

The tracing framework should aim for the following message observability:

- *Message tracing*: The tracing framework should allow testers to trace a component-based application on the lowest granularity in a distributed system, which is to trace all communication between the components. In order to give testers a better insight in distributed system, communication should not be restricted to request and reply messages but also all other communication event used by the middleware to connect and maintain a connection.

- *Message visualisation.* To allow tester analyse a distributed system effectively, the intercepted tracing data should be visualised in some understandable format. To visualise the communication between components, the tracing framework interface could for example use sequence diagram.
- *Message context tracing*: Tracing the communication between unique processes is not enough. A tester needs to be able to link the source and destination of a message with existing components. The tracing mechanism should therefore collect context information about the component that send or received a message.
- *Message contents tracing*: All DOC Middleware standards use some sort of communication protocol that enables components to send messages too each other. The parameter and result contents of these messages are of special interest to the tester because they enable him to verify the input with expected output of individual component instance. Components that are found to return unexpected results can then be traced further traced with traditional debuggers to pinpoint the malfunction behaviour inside the component.
- *Message causality tracing.* In complex distributed systems, multiple invocations can happen at the same time. Diagnosing a distributed system demands that the tester is able to understand the casual relationship between invocations. This requires that testers need to be able to trace the propagation of a single request through a component application. A tester should therefore be able to select a logical subset from the total collection of tracing data in which each event is the result of a previous event.
- *Message timing analysis.* Although tracing a single propagation through a network is a good way to test the basic functionality of a component based application, it is not a very realistic test. In a real distributed environment, a component-based application must be able to handle multiple requests at random intervals without choking. To diagnose and find time critical dependencies, testers require means to effectively analyse the send and arrival times of messages, the latency of messages travel and response times of components.
- *Message Filtering*: Distributed applications that run for a long time can produce a lot of tracing events. Not all trace events are interesting to the tester. What a tester is interested in is to find traces with special properties. For example, an exception with specific message contents. Such a framework would allow testers to observe and record method invocation and exceptions selectively, helping the tester avoid potential failures that would otherwise plague the performance of a distributed application.

6.1.3 Tracing Framework Performance

The tracing framework should aim for the following tracing framework performance:

- *Distributed Tracing*: Because a component-based application can reside distributed of multiple systems, a message sequence trough a distributed application can therefore visit multiple systems. Although a tester could follow a trace sequence by tracing every system a sequence flows trough, it would be awkward and very time-consuming. A far more effective method of tracing a distributed system would be to allow testers to trace distributed system from a central location. This requires that framework takes care of gathering tracing data from a distributed system and present it to the tester trough a single user interface.
- *Runtime tracing*: In order to diagnose distributed application real behaviour, the monitor framework must be able to trace a component-based application at run time. This requires that the tracing process does not halt during execution (e.g. tracing must be in real execution time). The interception technique should therefore allow the tracing framework to collect tracing data without interfering with the normal flow of execution.
- *Offline tracing.* Testers need to be able to analyse large amounts complex tracing information long after the tracing data is recorded. Large systems that are traced over a long period can produce huge amount of tracing events. Every tracing event in traced data sequence can consist of multitude of complex typed fields that must be individually analysed by the tester. This requires that the data is stored in a persistent data-storage device that is powerful enough to store large amounts complex typed data.
- *Online tracing.* In contrast to offline tracing, where data is analysed after tracing data is collect, online tracing means that the tracing data can be analysed as soon as it becomes available at runtime. In a distributed environment, this requires the tracing framework collects tracing data, transmitting it to a central location, process the data, and presenting it to the tester without too much delay.
- *Minimal Overhead*. One of the main problems of online tools is their interference with the observed environment, as the observation itself influences the system under test and potentially affects its functional behaviour. Thus, keeping the overhead of the tracing mechanism as small as possible by is an important testing requirement. A tracing framework can prevent overhead by controlling the influence of the tracing mechanism on the observed systems
- *Trace Configuration:* Tracing every component in a system in not always desired. Testers that are only interested in tracing a specific list of components should be able to do so with minimal effort. Testers generally do not like to get into technical details. The tracing framework should therefore strive to enable testers to activate and deactivate the tracing mechanism with a simple interface.

6.1.4 Interoperability Functionality

The tracing framework should aim for the following requirements to the interoperability:

- *Open Framework:* A tracing framework is never finished. New advances in testing techniques will become available which if applied could further improve the testing process. By using an open framework constructed from replaceable, extendable standardised modules with well-defined generic interfaces, the framework would have a longer life expectancy.
- *Implementation Independence*. The implementation source code of Corba component might be unavailable and only exist in binary format. Components that hide their implement details must still be tested. the toolkit must therefore be able to test components without modifying any components implementation code.
- *Language Independence.* Distributed Applications made in a Language Independent Middleware solution can consist of components written in multiple languages. The framework tracing mechanism should therefore not be tied to a single programming language but instead provide an architecture that supports language independent tracing. This will enable testers to trace communication between components written in different programming languages.
- *Platform Independence.* Some middleware solutions like Corba are Platform independent, which means the middleware, is not tied to one specific platform. In a Platform independent middleware solution, components based application can be scattered over hosts running different operating systems. Therefore, a tracing framework must be able to collect tracing data from component located on heterogeneous operating systems.
- *Middleware Independence.* Distributed applications are not always limited to the usage of a single middleware solution. Distributed applications can consist of component communicating with components made in a different middleware. Thales architecture for example uses multiple middleware solutions for their distributed applications. Components made in different middleware products communicate through special constructed bridges that translate middleware communication from one middleware specific protocol to another specific protocol. In order for a tester to observe a complete trace through heterogeneous middleware application, the framework must be able to collect tracing information from different middleware products simultaneous and present it to the tester in a generic way.
- *Monitor Environment Independence.* Even if the framework is able to collect all tracing data independent of the operating system, middleware and programming language a component is developed in, the tester does still need some tool to analyse the collected tracing data in. However, testers might not always have access to all platforms. The framework should therefore allow the tester to analyse the tracing data on any platform.
- *Standard Based.* Open middleware standards like Corba are not restricted by the implementation of a single vendor. Although an open standard allows customers more choice, they do not make testing any easier. Vendors often try to distinguish them self from competing vendors by providing extra but also proprietary solutions. Although these additions can be very useful for developing new applications, these applications quickly become incompatible with the Middleware implementations from other vendors. Because Companies wish to switch vendors without difficulties, the tracing framework must be vendor independent. Thales for example, had to migrate from Orbacus to JacOrb because Orbacus support stopped. In order to be vendor independent, the tracing framework must avoid using propriety middleware interfaces and exclusively use standardised middleware interfaces.
- *Vendor Adaptability*. The toolkit should strive to be compatible with most used Orb distributions. However, every Orb is slightly different from each other due to slightly different naming conventions and different implementation features. If you consider that there are over a hundred different Orb implementations, each consisting of multiple versions, it is not suppressing that full compatibility is close to impossible to achieve. The test framework should therefore support facilities that allow the administrator to make unsupported Orbs compatible with the testing toolkit.

6.2 Corba Test Tools under Evaluation

Which tools are available on the market for tracing Corba? This chapter will answers this question by giving the reader an overview of tools that were designed to test Corba application.

We have examined and evaluated the following Corba tools:

- *Lucent DSC Framework Testing Toolkit*
- *Segue Silk Testing Toolkit*
- *Corba Management Component Interface Testing Toolkit (MCITT)*
- *Ethereal Network Sniffer*
- *Corba Tracer developed by the University of Nantes*

Note that we were unable to give every tool an equal amount of attention due to lack of time and information. One tool specific, the Lucent DSC toolkit will proportionally have more attention because there was the a lot more information available for it and we were actually able to test it firsthand.

6.2.1 DSC toolkit

Until recently, the DSC Toolkit was an IDE toolkit developed by Lucent Technologies used exclusively by Lucent Research Groups in the development of distributed systems. Although their initial plan was to make the source and binaries available for non-commercial purposes, the tool is currently available under license conditions. Lucent Technology in Hilversum was so helpful to send us an old version of the DSC Toolkit for evaluation purposes.

The DSC Testing Toolkit also called Monitor framework, is part of the DSC Framework. The DSC framework originated in the MESH project. It was created to support the design and development of a large distributed component-based architecture for a multimedia, multiparty services platform, named the MESH platform. The architecture of the MESH platform was based on the TINA architecture, which in turn is based on the ODP model. The DSC framework was further enhanced in the FRIENDS project. During the FRIENDS project, they included many elements of the Corba Component Model (CCM) in the DSC framework.

Lucent original intention was to replace the DSC framework with a commercial CCM implementation when available. At the time, only a few freeware and commercial implementations had been announced, and no existing product was found which matched their needs. Therefore, they decided to make the DSC framework CCM compliant. However, the progress of the CCM standard was very slow and did not seem to acquire the necessary industry support and momentum. During the time they tried to adopt elements of the CCM specification in the DSC framework, they discovered many inconsistencies and open questions in the specification. The CCM standard also felt like to large and too complex for Lucent needs (an opinion that is also shared by Thales). Although they admit too have adopted some good elements from the CCM, they have no intentions to work any longer towards a CCM compliant implementation of DSC [DSCDoc].

Although the Testing toolkit was originally designed as a verification technique for the Distributed Software Component (DSC) Framework, it had many similarities with the CCM standard. The presumption was that if the DSC framework would be made compatible with the OMG CCM standard, it could theoretically be applied to any Corba based architecture. DSC uses a thin, abstract Orb layer that unifies differences between different Orb vendors. Although DSC can make use several different Orbs, the tool only pre configured to work with Orbacus and JacOrb. (The evaluation version only contained support for constructing Java Components under JacOrb or Orbacus), Lucent has demonstrated that they also have a prototype C++ version support the freely available TAO Orb.

The DSC Toolkit offers a user-friendly testing utility that can be configured as an actor, reactor, or both. DSC Test utility allows testers to create a controlled environment (see figure 6.1) were components can be prototyped by a test components. Test components (Actor/Reactor) can connect to multiple test subjects at runtime, by using Corba DII or DSI facilities (see paragraph 4.1.5.2). The test component allows the user to select a test subject, an interface, operation, and parameters through a selection menu.

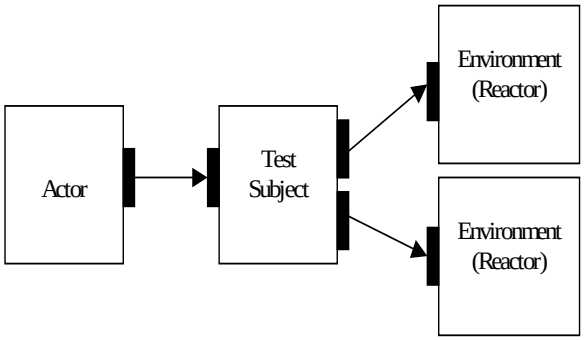


Figure 6.1: Actor Reactor Framework

The test utility helps the tester to construct a valid message request or response by constraining the input values in the parameter tree (see picture 6.2). The test utility can derive this restriction data directly from Corba Interface Repository. The object reference of a target component can be obtained through the naming service, or can be provided directly as a Corba IOR string, which can be resolved automatically into a valid object reference. Note that each reactor can be controlled from a centralised test console, regardless of the location of the reactors in a distributed environment.

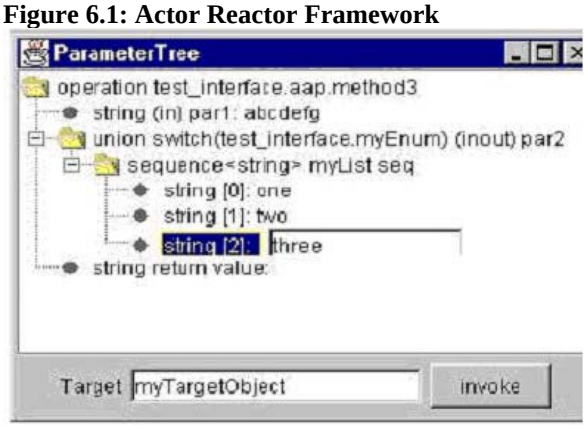


Figure 6.2: Actor Parameter tree interface

The DSC toolkit allows the Actor to run Actor script which can be used automatically invoke target interfaces. The actor script is written in a Java enabled version of the popular Tcl script and is powerful enough to simulate complex test cases. Besides using the Tcl script for the automation of the Actor, the scripting language can also be used for the automation of the Reactor. A Reactor script can be programmed to respond to any invocation on a Reactor component. A Reactor can therefore be used as a substitute for components that have not yet been developed but which external interfaces are known. Reactor components are automatically generated from component specification files by the DSC Generation tool. They provide an empty implementation component, which can be used in a testing situation where the components being tested need to interact with other components, which are not yet available.

The DSC tracing mechanism is customisable which allows better control over the overhead it invests in the observed distributed application. The graphical user interface (called Viewers) only requires the tester to select the component interaction points for examination. The framework management mechanism
The tracing mechanism generates all necessary information needed to identify the component, its interface, the operation, the parameter values and its source and destination.

Before any component interaction can be traced, the components in the System Under Test (SUT) have to be prepared for usage by the tracing framework. The DSC framework must both add tracing code to the stub and skeleton source code generated by the IDL compiler, as well install interceptors on the Container Orb. By default, DSC will automatically augment all component stubs and skeletons) created by the IDL compiler. Although the extra overhead caused by the tracing mechanism is low, changing some variables in a property file can manually turn off the augmentation. Once the modified stub and skeleton code is recompiled, the interception mechanisms that is now located between components, can send intercepted communication events to a predefined location (see figure 6.4). Whether or not this tracing code is executed depends if the 'monitor code' was activated before the component was instantiated.

To activate the component tracing mechanism, the container that instantiated the component, can manually activate tracing behaviour in a component property file. This file also includes property settings for the initialisation of the Orb. Because the Java version of the DSC Framework can use the Orb property setting for the activation of the interceptors with parameters, it requires no additional changes to the Component implementation. However, the C++ version is a more complicated because not all C++ Orb support initialisation of the Request Interceptor by parameters. In these cases, the component source code must be slightly modified to register the interceptors.

The monitoring framework in DSC is part of the component runtime that observes invocations at the component interface level. The tracing framework generates all the necessary information needed to identify the component, the involved interface, the operation, the parameter values, etc. The collected tracing data can be used for generating graphical representations in the form of a message sequence diagram. Because the events produced by the tracing framework are self-contained and ordered, a specially designed visual tool can dynamically generate a message sequence diagram showing the interactions between the DSC components in a distributed application.

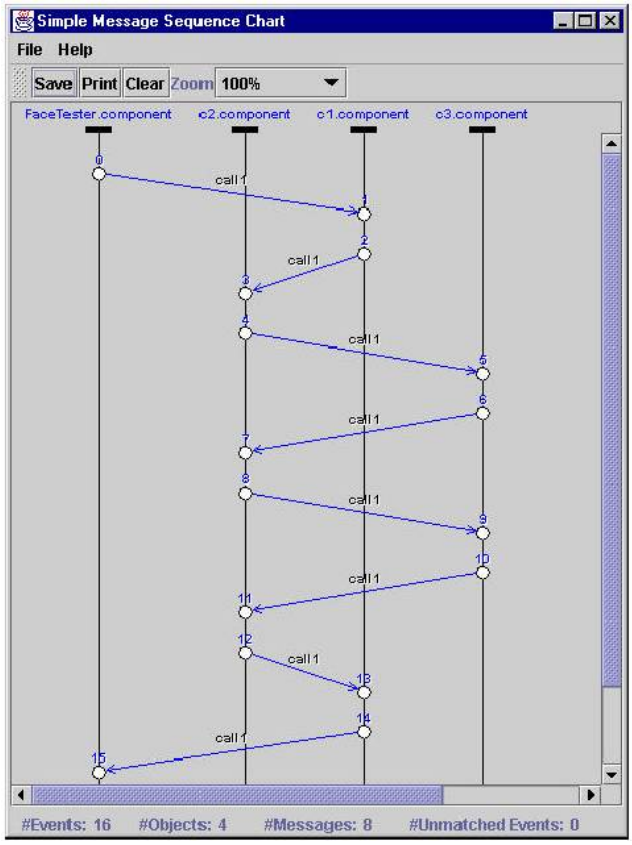


Figure 6.3: Message Sequence Diagram

After an invocation, the tool will immediately generate and display a complete message sequence diagram of the last invocation (see figure 6.3). The complete diagram can also be saved to a database and analysed off-line. The DSC tracing framework can accomplish this by intercepting the interactions and sends the intercepted contents data along with tracing context data to a central Trace Server Component. Figure 6.4 shows how the Trace Server component forwards stores recorded interaction in a Local Database, (see figure).

The DSC tracing framework is extendable because it offers an offers an API for development of custom event consumers and an API for the extension of the event model by adding support for new type of events.

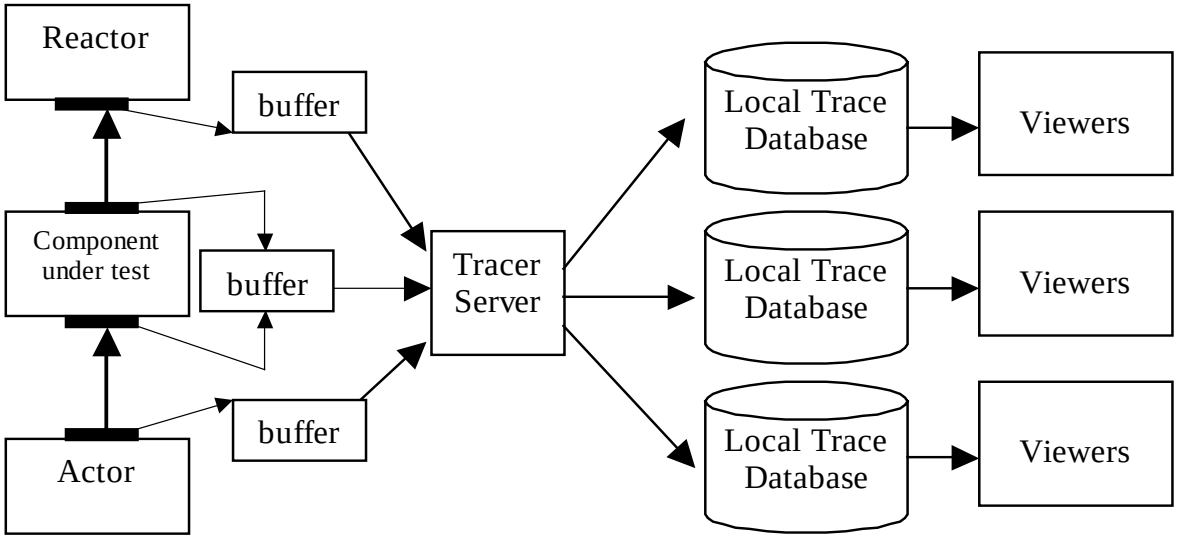


Figure 6.4: DSC Framework Architecture

Lucent Technologies developed their test tool even before Corba Request Interceptors were available. Their tracing mechanism was initially based exclusively on instrumented the stub and skeleton with tracing functionality. Besides reading the contents information of an invocation, they added context information into the message by wrapping the interface with additional parameters. The context information, contains a logical clock value, has to be transmitted along with every message in the SUT. The logical clock value is a technique used to construct message sequence diagram by the viewer utility. The drawback of this interface wrapper technique is that it requires both stub and skeleton to be instrumented with tracing functionality. This is because an instrumented skeleton can only read the extra parameter data send by instrumented stub. Communication which

used Corba object services that cannot be instrument like Corba transaction service and event service would become impossible.

Fortunately, the OMG Portable Interceptor solution arrived just time to solve their tracing mechanism problems. Instead of using their proprietary wrapper technique, they could replace it with the Service Context mechanism of Corba Portable Interceptors (see appendix I). Portable Interceptors were in contrast to their original wrapper technique, message transparent, which means that it will be compatible with all Corba communication, including transaction and event services.

However, they could not abandon their instrumentation technique completely because of two reasons.

- The first reason is that JAVA interceptors cannot read contents information exchanged between components. This contents information is necessary in order to verify the functional behaviour of the Components Based System.
- The second reason is that beside contents information, the instrumented stub/skeleton can also supply component context information like component ID. Interceptors cannot generate this information because interceptors can only be installed on an Orb that is shared by multiple components in the container. A request interceptor installed on a Container Orb therefore has no way of knowing which component did an invocation.

The DSC framework therefore uses a combination of Request Interceptors to transmit context data and stub and skeletons instrumentation for accessing the message contents data and client/server identity. The context information (logical clock and identification data), which has to be exchanged between stub/skeleton and interceptors, is achieved by using a slot in the PICurrent (for details see appendix J). Figure 6.5 illustrates how multiple components use the PICurrent to transfer context information to the interceptor

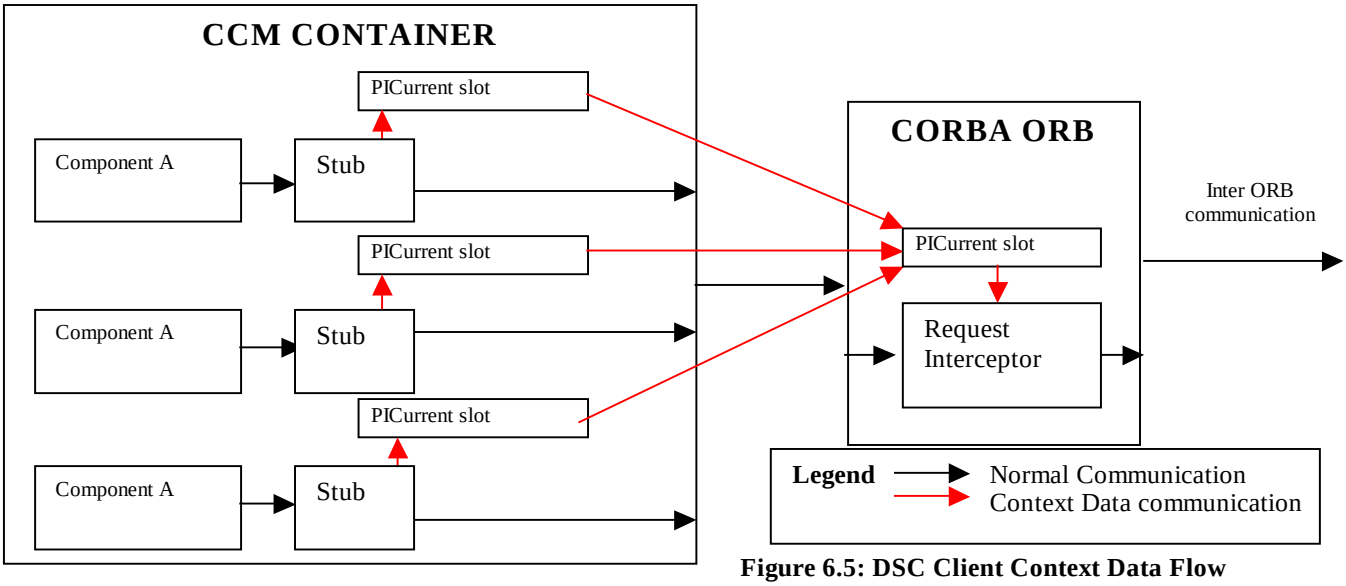


Figure 6.5: DSC Client Context Data Flow

The initial implementation of the DSC test tool used a post IDL processor to instrument the stub and skeleton generated by the IDL compiler. The post IDL processor used standard syntactical analysis techniques to instrument the stubs and skeletons with additional tracing code. Because this instrumentation technique proved to be more complex than anticipated, they are now using an instrumented IDL compiler for the generation of Java stub and skeletons. Although their Java IDL compiler depends heavily on the JacOrb IDL compiler, it is only marginally modified. The stubs and skeletons generated by this JAVA IDL compiler are exchangeable by all possible Java Orb implementations. This is because the stub and skeleton classes in Java have to conform to a specification. However, this will not work for C++ stub and skeletons, for which no standard stub or skeleton class specification exists. The prototype C++ IDL compiler must therefore be manually instrumented for usage other that the currently supported Corba implementations. Fortunately, the instrumentation is relatively simple to perform by the developers at Lucent Technology. Summarising we can conclude that the DSC Toolkit offers both a good tracing framework for the verification of most Corba middleware applications.

6.2.2 Corba Trace

Corba Trace [Corba Trace] is an open source Corba Java tracing tool, which can be downloaded and used under the free license of the Free Software Foundation. The tool is developed over a period of two years (2000-2002) by a team of students at the University of Nantes in France. Although Corba trace is written in SunJava 1.4, the blue prints are available to rewrite it in any other Corba compatible language. However, the downloadable Corba trace version only supports Sun JDK 1.3.1 / 1.4 with Orbacus Orb 1.4.0 / 1.3.4 on the Windows 2000 or Linux operating system. Although they provide the full design documentation of Corba Trace, it is only available in French.

Corba Trace tracing mechanism is based on combination of Request Interceptors (see paragraph 5.2.1) and Adapter Instrumentation (see paragraph 5.1.3). In the figure 6.6, we can see the Adapter in active state after deployment. The Adapter, which is located between the Corba object and Stub/Skeleton, collects contents data (in and out parameters) and context information (component name). The Request Interceptor is used for transmitting tracing context data between the Orbs.

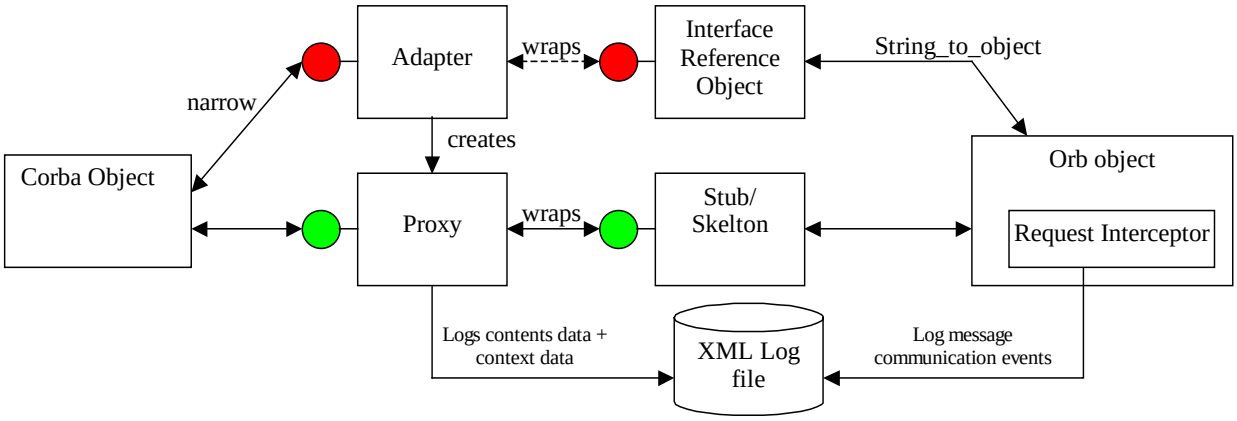


Figure 6.6: Adapter wraps Orb interface and logs contents information

During the execution of the Corba Application, the tracing mechanism intercepts Corba communication and stores it locally in logs. After the tracing process is finished, the produced logs (which have been created in XML format) are processed with the Log2SequenceDiagram utility into message sequence diagram. In figure 6.7, we can see how Log2XMG processes the XML produced by an interceptor into XMI, which can be viewed by UML sequence diagram viewer.

Before any tracing data can be visualised, the log files created by tracing mechanism must first be collected, filtered, and processed. However, Corba Trace does not contain any integrated data distribution framework like the DSC Toolkit (see paragraph 6.2.1). The user therefore must transmit the intercepted tracing data manually. Fortunately, Corba Trace makes life a little easier by offering an integrated FTP client tool, which allows the user to download Log files from traced systems.

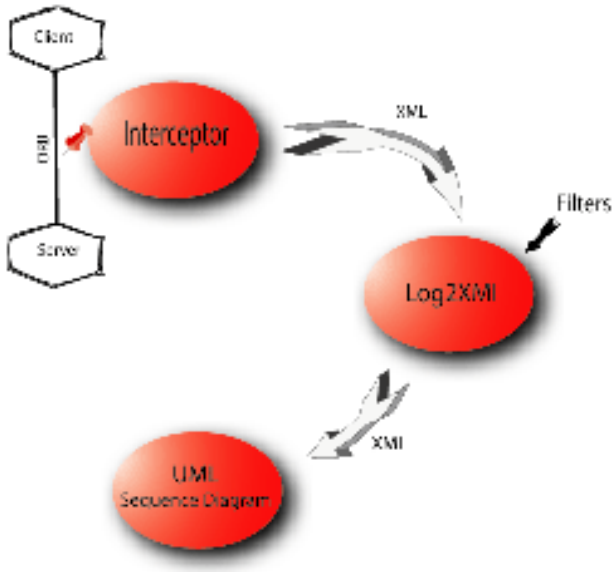


Figure 6.7: Processing of intercepted tracing data into

Before the intercepted tracing data can be analysed by a viewer, the collected Log files must first be processed into a readable format with the Log2SequenceDiagram utility.

The Log2SequenceDiagram utility performs the following processing tasks:

- Parses log files, and merge all partial-messages to get full message.
- Synchronises all local objects clocks to a common clock.
- Applies some used defined filters to get to filter out undesired information.
- Generates a sequence diagram in some common file format, which can be directly display by a viewer.

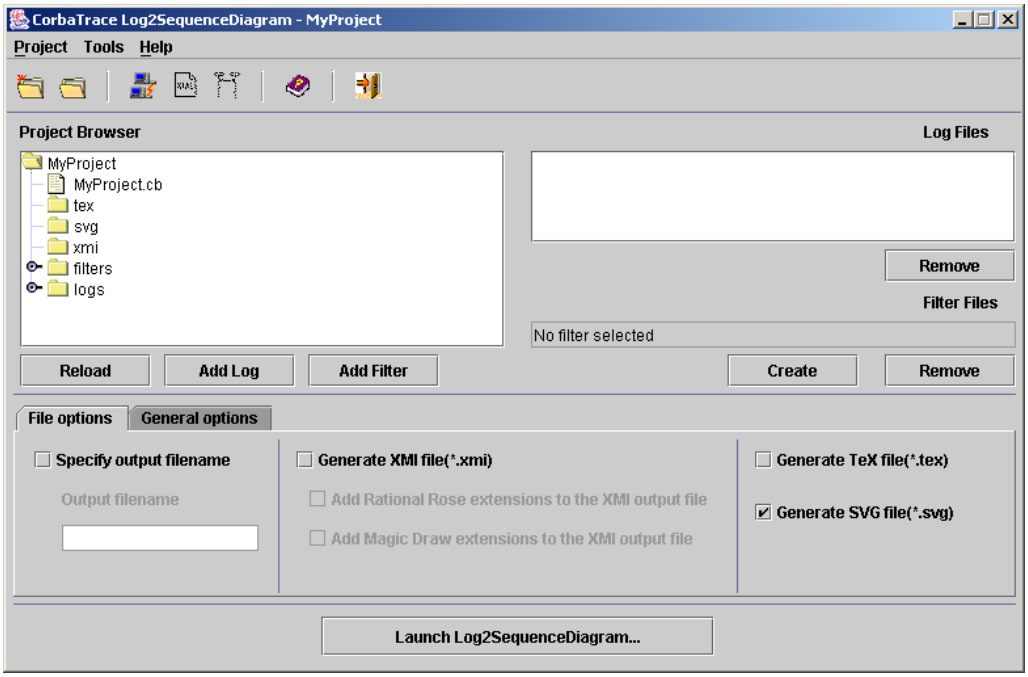


Figure 6.8: Corba Trace Filter GUI

Because intercepted tracing data can contain irrelevant tracing data, the Corba Trace Log2SequenceDiagram utility can apply filters on the collected log files before the final sequence diagram is generated (see figure 6.8).

The following types of filters (see figures 6.9, 6,10, 6,11) can be applied on the log files:

- *Message type* filter allows the filtering of intercepted messages based of the type of communication event.
- *Date Time* filter allows the filtering on the interception timestamps generated by the tracing mechanism. There are three types of date filters: after a date, before a date, or between two dates.
- Method Contents filter. For each operation, the interface name and operation arguments' values can be specified. Defined argument types can be one specific type and value, or an exact argument position and specific value.

Although the filter mechanism does not allow the automatic clustering of objects with a casual relation ship, the user narrow a sequence diagram to a set of interaction objects by applying filters on either object level global level.

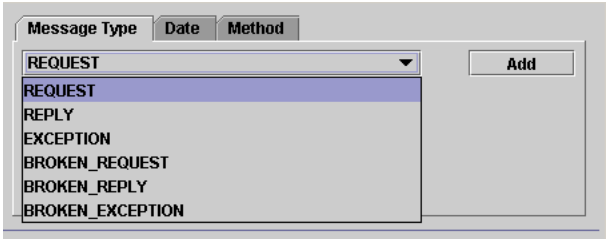


Figure 6.9: Message Event selection

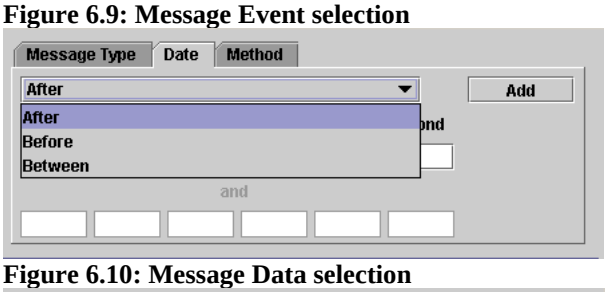


Figure 6.10: Message Data selection

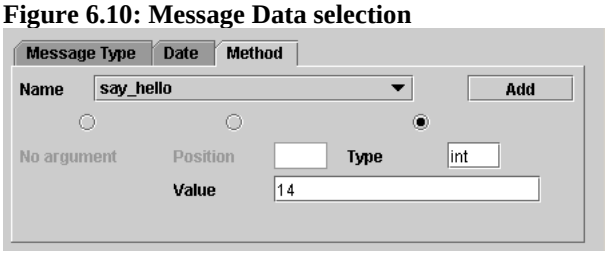


Figure 6.11: Message Method selection

Although intercepted Corba communication cannot be monitored online, the message contents data and message interaction data can be analysed offline by inspecting the generated files with a compatible viewer.

The Log2sequencediagram utility can generate the following common formats:

- SVG is the format recommended by Corba Trace because it provides the best graphical quality and is provided with the tool
- XMI is the most common formats and can be viewed by any tool that supports the XMI format like common UML tools such as MagicDraw (see figures 6.12 and figure 6.13) and Rational Rose.
- Latex is also a common format, which can be viewed by any compatible Latex viewer.

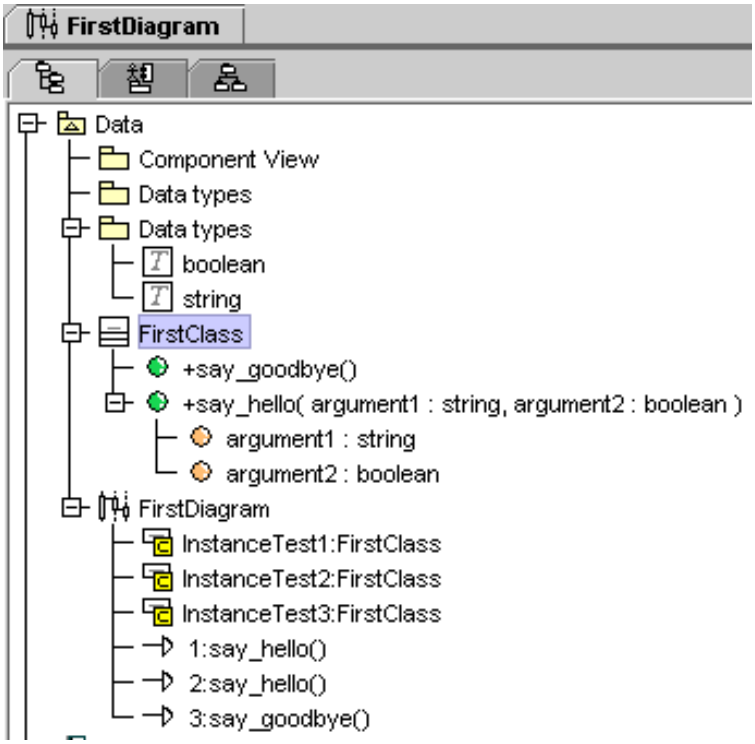


Figure 6.12: MagicDraw Message Data Tree Viewer

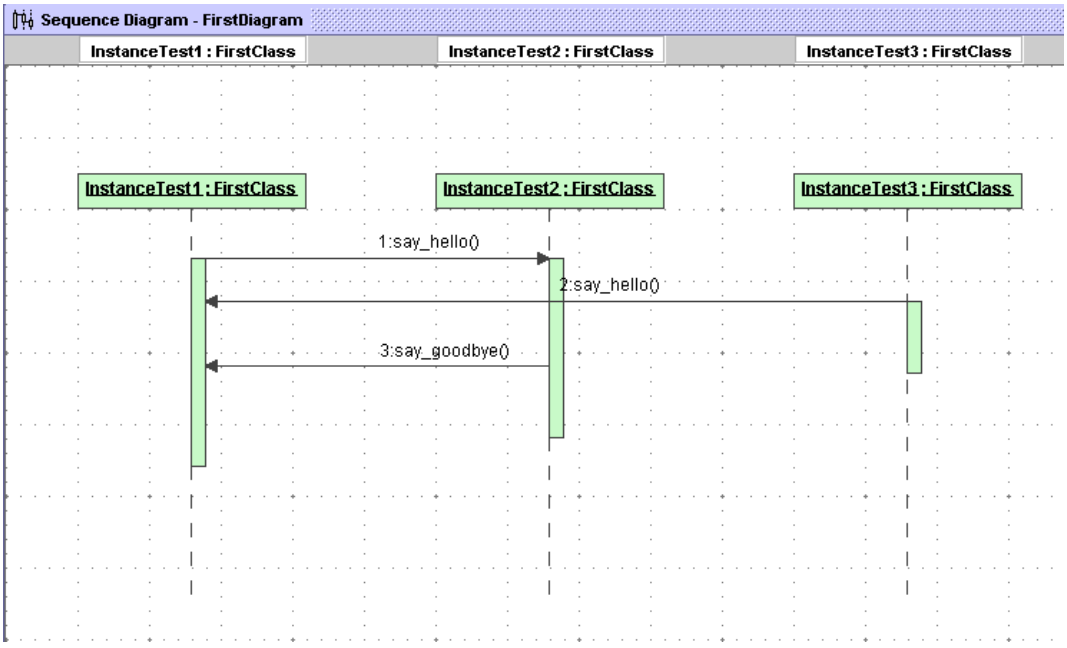


Figure 6.13: MagicDraw Message Sequence Diagram Viewer

Although the 'Corba Trace' tracing mechanism cannot be configured at real time, it can be set manually by setting the active logging policy level. Before any Corba messages can be traced, both client and server source code must be instrumented with additional tracing code. In the two source code examples below we can see how existing component are instrumented with a proxy object, which only need to initialise the Adapter with an existing object reference. While this is an insignificant modification in cases where the source code of components is available, it precludes the use of this tracing mechanism in incases where commercial off-the-shelf (COTS) components are used.

Once initialised, the proxy object behaves exactly the same original object reference object. The instrumentation will trick the Corba object thinking that it is communicating with the stub or skeleton while in fact it communicates with an intermediate proxy object. Although this programming technique minimises the instrumentation required in a source code, an adapter is very Orb implementation dependant. This is because every Orb implementation behaves slightly different and offers different functionality. Fortunately, the design documentation can be used to make the Adapter compatible with other Orb distributions.

A bigger problem however, is that this instrumentation technique will not work in CCM. That is because Corba Trace initialisation procedure requires an object reference and Orb object. Both are not available in a CCM Component. Although underneath, CCM still uses Orb object and object reference, you will not find them in the client or server source code. Instead, they located in the component home, which maintains all object reference and the container, which initialises the Orb. Nevertheless, Corba trace is still a useful tool for testing Simple Corba applications. Further study should therefore be done to determine whether Corba Trace mechanism could be adapted for CCM applications.

```
import corbaTrace.InterceptorServer;
...
class MyClass {
    ...
    interceptorServer= new InterceptorServer();
    ...
    Orb orb = org.omg.Corba.Orb.init(args,props);
    obj = orb.resolve_initial_references("RootPOA");
    POA rootPOA = org.omg.PortableServer.POAHelper.narrow(obj);
    poa_interceptor = interceptorServer.create_poa(orb, rootPOA, "myHelloPOA");
    ...
    obj = interceptorServer.activate_object(poa, helloImpl,"My Component");
    Hello hello = HelloHelper.narrow(obj);
        ...
    orb.run();
    orb.destroy();
}
```

Instrumented server source example

```
import corbatrace.InterceptorClient;
...
class MyClass {
    ...
    interceptorClient = new InterceptorClient();
    ...
    Orb orb = Orb.init(args, props);
    ...
    obj = orb.string_to_object(ref);
    obj = interceptorClient.active_interception(obj, orb);
    interceptorClient.activate_log(orb, "My Component");
    ...
    Hello hello = HelloHelper.narrow(obj);
    ...
}
```

Instrumented client source example

6.2.3 The Silk Testing Toolkit

Segue Software has one the largest e-Business testing customer bases and describes their Silk product family as 'e-business reliability solutions'. They have developed a large range of commercial available tools for testing distributed systems, which can operate standalone or in combination with each other. They divided their testing framework into separate tools, each aiming to cover different aspects of testing.

Because one of these tools, called Silk Observer was specifically made to trace Corba applications, we were very anxious to take a closer look at this tool. Unfortunately, the tool is no longer commercially available. Due to lack of interest from the industry, and the introduction of other component technologies like .NET and Java Beans which have a much larger market, Segue quit all support for their tracing tool. They claimed to have integrated some of tracing technology of Silk Observer into a special edition of Silk Performer, called Silk Performer Component Test Edition (see figure 6.15). Although, the Silk Performer Component Test Edition currently does not support Corba at all, Segue Software has revealed that there are plans to integrate Corba in their Component-testing product in future version. Their tool would then be able to trace Component based application consisting of multiple component models including Corba/CCM.

Because Segue Software offers many testing tools, we will restrict ourselves to the most interesting tools for creating a testing framework. In the next sub paragraphs we describe following segue testing tools:

- Silk Observer: a message tracing and monitor tool
- Silk Test: a actor/reactor test tool with script automation
- Silk Performer: a message record and workload generation tool

6.2.3.1 Silk Observer

Silk Observer facilitates monitoring and diagnostics in a distributed application environment by capturing and presenting communication details between Corba objects. Silk Observer was on of the first testing tools designed specifically for tracing Corba applications. Black and White Studios developed the Silk Observer, which was originally called Object Observer. In 2000, Black and White Studios was taken over by Segue Software, which made the tracing tool part of their testing tool family and re-branded the tracing tool to Silk Observer.

Silk Observer provides the following testing tool support:

- A *Profile Editor* allows the user to specify where to monitor traffic ‘on-the-fly’. By selecting which parameter, return values, methods, object types should be monitored, the user can define filter sharp profiles to scope the traffic to only the interfaces of interest. The Profile Editor is able support these features thanks to the Corba interface repository. Moreover, the user can specify the monitoring points i.e. client sending, server receiving, server sending or client receiving. The configuration is saved in files and can be assigned to an instrumented Corba object at runtime.
- A Monitor for graphical display of the data. Using the viewer, a tester can selectively monitor and record method invocations and exceptions to avoid or eliminate bottlenecks, race conditions and other potential failures that can impact the performance of an application. Information presented includes which application components are running, which objects are being accessed, and the methods being invoked (including parameter values and thrown exceptions).
- An *Analysis Tool* in which data recorder combined with a parser for off-line request-reply timing analysis of tracing data stored in log files. The parser provides some communication timing analysis functionality by computing request latency measures from the timestamps of the request reports. Object/Observer has to parse the message buffers completely in order to retrieve and reconstruct the parameter and return values of the requests.

Although Silk Observer supports monitoring of both Java and C++ applications, it can do so only with the Orbix and Visibroker Orbs on the Windows or Unix operating systems. Silk Observer tracing functionality is based on a combination of instrumentation of the component source code and a proprietary message logging facility. The instrumentation, which is similar to Corba Trace (6.2.2), is achieved by linking an additional class library to the source and by initializing the tracing mechanism with the interface object reference of the servant.

Each CORBA object that is to be monitored must instantiate a so-called Probe object. The Probe object collects data about method invocations and stores that data in local log files. The Probe object also takes care of retrieval of context data, process identification data, and retrieves profile data from Profile editor. The Probe can achieve message tracing by using a proprietary filtering message logging functionality offered by the Orbix or Visibroker Orb. This logging facility which similar to request interceptors, allows the interception of

communication passing from or to a Corba object. The advantage of this technique is that in contrast to request interceptors, the filters do not have to be registered. This means Silk Observer can modify the tracing overhead without restarting the Corba object Orb. However Silk Observer tracing mechanism reliability on proprietary filtering, it make in impossible to use the tool with Orbs, which do not support this proprietary functionality.

Figure 6.14 illustrates the architecture of Silk Observer in which data flows from tracing mechanism to monitor tools and from monitor tools back to tracing mechanism. On the one hand, it gathers request traces from the Probe objects and forwards them either to the viewer GUI or to a data recorder that stores the tracing data in log files. On the other hand, it forwards configuration requests from the viewer GUI to the corresponding instrumented Corba objects. Silk Observer data distribution uses standard Corba objects, which open to user specific changes because the OMG IDL interfaces from all Corba objects are available and its usage is well documented.

For the communication between tracing mechanism and graphical user interfaces, Silk Observer tracing mechanism uses a separate Collector object on each monitored host. In order for the Collector component to be aware of deployed Probe objects, the Probe objects must register itself to the Collector when they are created. The Collector object, forward the recorded information to a central component called the Observer. The Observer keeps all information about the system in a central database. This database may be queried for new data continuously, resulting in actual time processing. The data may also be analyzed at some later point in time.

The amount of information recorded by the Probe objects is controlled by so-called profiles. A profile defines which interfaces, messages and parameters should be recorded. When Probe objects are created, they request a profile from their Collector. The Collector in turn contacts the Observer to obtain the requested profile. In order to influence the amount of information collected at runtime, new profiles can be uploaded from a Monitor component to deployed Probe objects in the system. The Probe objects combined with the ability to change the filtering conditions, allows user to reduce of the amount of information that needs to be processed to a minimum.

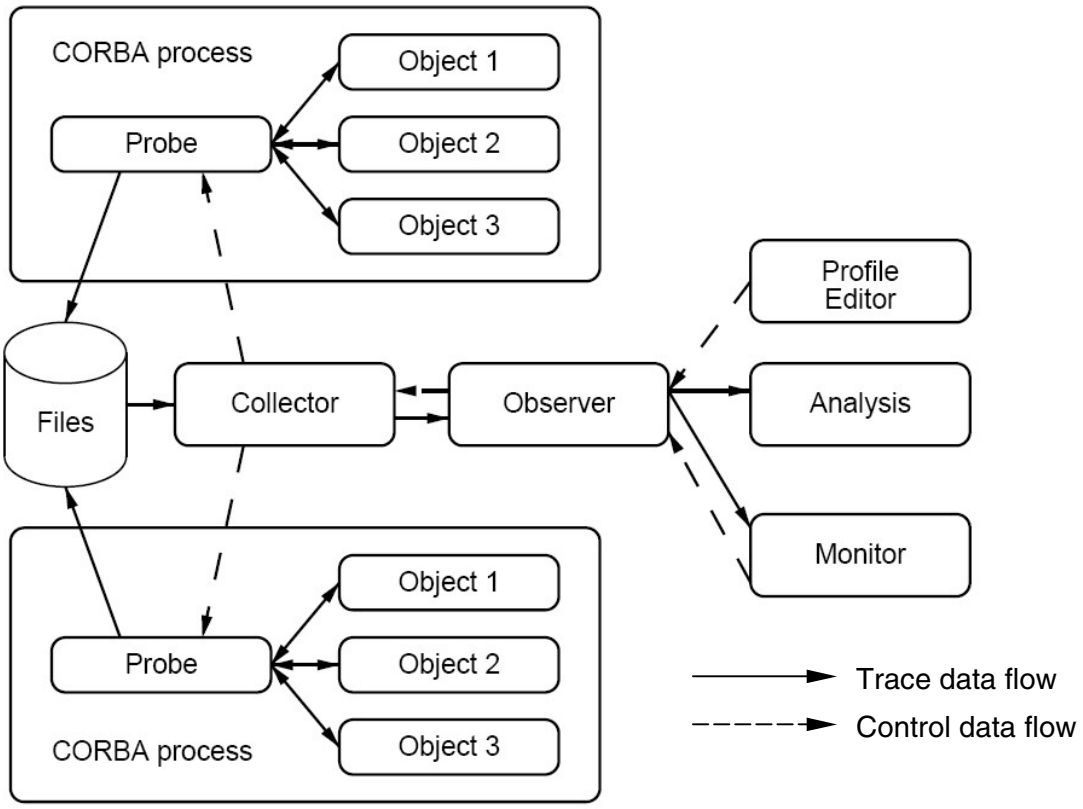


Figure 6.14: Silk Observer Architecture

6.2.3.2 SilkTest

Silk Test, originally known as Silk Pilot, is an object-level solution for validating the behaviour of Corba-based servers. The GUI offers an easy point-and-click mechanism that allows testers to connect to Corba servers and to issue requests based on the interface descriptions that are automatically retrieved from a Corba Interface Repository (see also paragraph 4.1.5.1). The sequence of method calls issued during a session can be converted into script code for later replay.

Silk Test is available in a Standard and Professional Editions. The Standard Edition is an entry-level product featuring interactive testing of objects. The Professional Edition ads test automation and code generation capabilities. Silk Test is aimed to allow the tester to test the behaviour of distributed objects within application's server components. The Silk Test supports regression testing by an integrated testing scripting language called 4Test. The 4Test scripting language is an object-oriented language specially designed for regression testing. Because the Silk Test tool interprets the 4Test scripting language, it is also platform independent. Silk Test can be used to test Corba servers implemented in any programming language, as well as pure Java servers through local public interfaces and RMI. Silk Test also explicitly supports the Enterprise JavaBeans (EJB) component model.

6.2.3.3 Silk Performer

Silk Performer provides a load testing facility for CORBA servers. Load testing is based on observation of IIOP packages with the Internet Recorder. IIOP is Corba message format, which uses standard TCP/IP network protocol for transmission (see also appendix B). The Internet Recorder is integrated into the socket library, and records IIOP traffic before transmitting it to the actual clients. The IIOP packages can be replayed exactly as recorded or configured to generate different traffic profiles. Reconfiguring and changing invocations parameters types cannot be determined by observing IIOP only. Silk Performer therefore is able to extract parameter information from interface definitions, which enables parameters types to be associated with IIOP messages. With a workload wizard it is then possible to generate traffic seemingly coming from different machines with a variable number of concurrent users with variable transaction frequency. During the load tests, results can be presented with a GUI tool showing virtual users, transaction status, response times, data throughput, and occurring errors.

Segue Software now also offers a special version of Silk Performer called Silk Performer Component Test Edition which can verify the performance and interoperability of remote components. Unlike unit testing tools which only evaluate the functionality of a remote component being accessed by one user, Silk Performer Component Test Edition can tests components under concurrent access by up to five virtual users to emulating realistic server conditions (see picture below). However, the new tool supports almost all popular middleware products (including Microsoft new middleware product .NET) except for Corba.

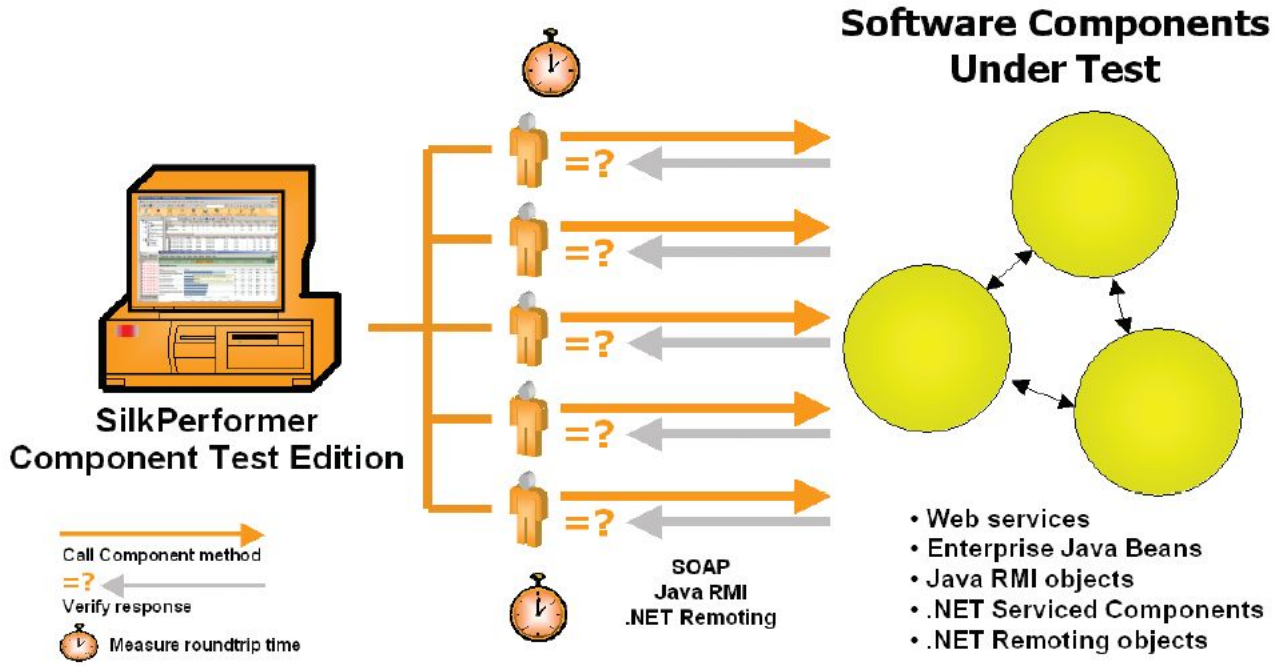


Figure 6.15: Silk Performer Component Edition Performance Test

6.2.4 Ethereal

Ethereal is an open source, freeware network sniffer that is able to capture network traffic. Data can be captured “off the wire” from a live network connection, or read from a captured file. Live data can be read from many types of network including Ethernet used by the Internet. The main advantage of this approach is that it is completely transparent because neither the middleware nor the component needs any change. However, tracing Network communication involves very low level and tedious work. Besides logging network traffic, the sniffing tool also allows observer to analyse communication statistics. These statistics are useful for detecting network traffic problems. The main feature which makes Ethereal a possible candidate for Corba tracing tool is its ability to filter it captured traffic. A filter creation GUI allows users to create filters on any protocol or field that Ethereal knows about. Ethereal Version 0.9.14 currently supports over 407 protocols, including GIOP, Corba communication protocol (see appendix B). By applying a special GIOP filters, Corba messages could be separated from the network traffic noise (see figure 6.15). Note however that communication between components hosted on the same machine cannot be inspected because these messages do not travel over the network. Although it theoretically possible to use Ethereal sniffing tool to trace Corba Application on a Host to Host level, it will be very impractical to pull it off. Each captured raw Corba message must manually dissected into useful tracing information and large distributed can only be performed offline from captured files. On the other hand, it is fully transparent and can be used with virtually no latency overhead to the distributed application.

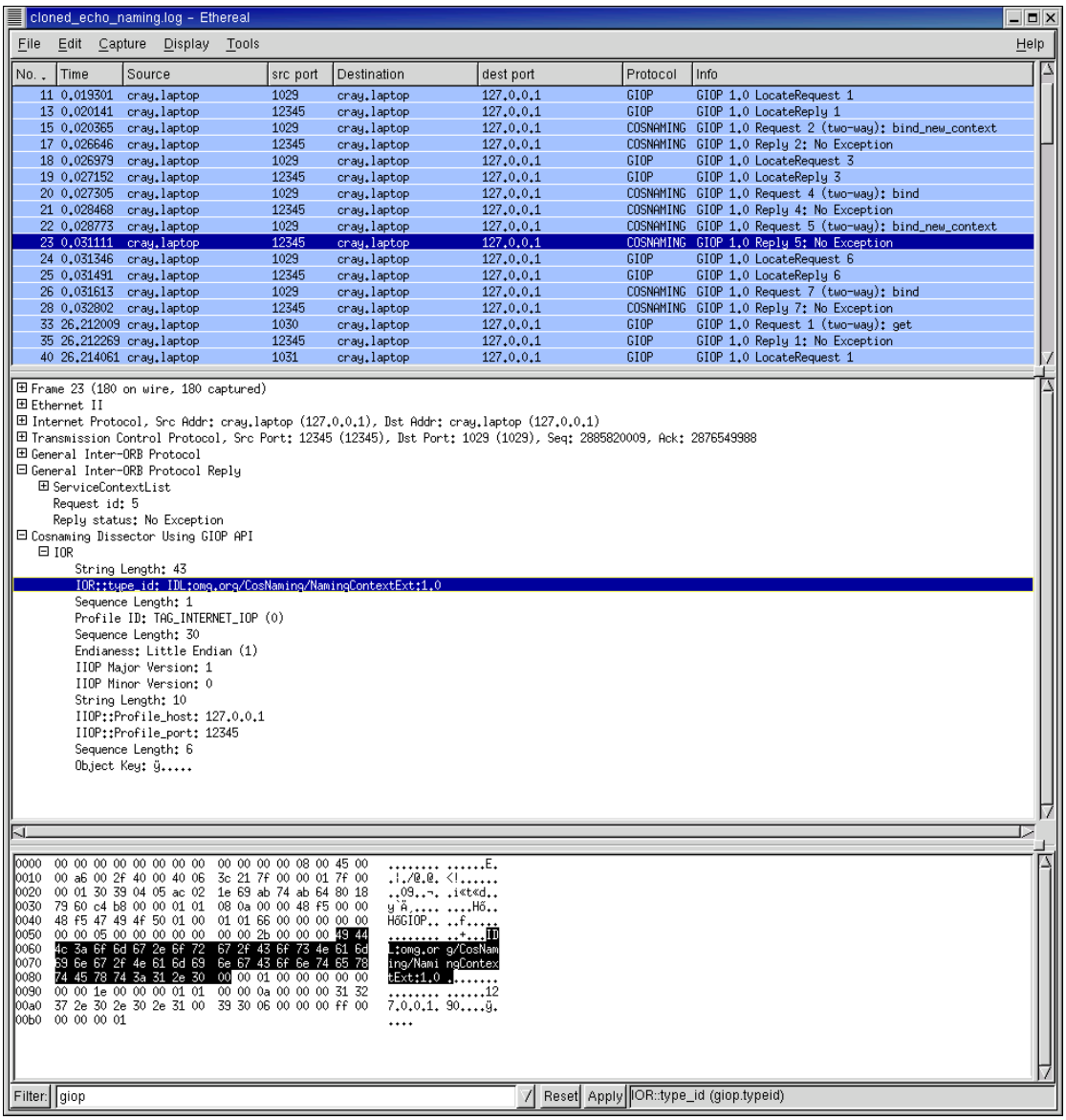


Figure 6.16: Ethereal Message viewer

6.2.5 MCITT

The Manufacturer's CORBA Interface Testing Toolkit (MCITT, pronounced "M-kit") reduces the amount of labour required to verify the behaviour of components in a complex distributed system. It was developed in support of a project co-funded by the NIST Advanced Technology Program. Although the project has ended and no future releases are expected, It is now a freeware stand-alone tool which can be freely download directly from the OMG website.

The MCITT toolkit minimises the amount of effort needed to produce simple emulations (dummy components) that can be used to replace servers in a testing scenario. The MCITT accomplishes this by allowing the user to generate source code files in a higher programming language like C++ or Java. The generated test emulation application consist of CORBA boilerplate code, memory management, and stubs for unused operations. The generated emulation code also provides conformance test assertions, automatic inclusion of conformance test boilerplate, and timed loops for performance evaluation. Test assertions can even be derived automatically from Component Interaction Specifications. Although MCITT can create automated controlled environment with minimal afford, the definition language only supports static testing scenarios and does not supports any interactive manual invocation modes.

There are two complementary ways of defining behaviours for dummy components:

- *The Procedural way,* by using Interface Testing Language (ITL). ITL is a super-simplified procedural language for specifying and testing the behaviour of Corba clients and servers. Besides the obvious advantage of script languages, ITL also offers specialised support for conformance testing, performance testing and server emulation. The ITL compiler uses a set of files called a binding to translate ITL into the implementation language. Different bindings can be created to absorb the differences between one platform and the language, thus achieving a higher degree of platform independence.
- *The Declarative way*, by using Component Interaction Specifications (CISs). Unlike an ITL file, which specifies the behaviour for one component only, a CIS can describe all of the interactions between all of the components in the system. A CIS can be used to generate multiple servers that will execute the scenario that it describes. An interaction scenario consists of a tree of Corba request having specified input, outputs, and/or returns values. The tree is rooted at the test client that initiates the entire chain of events. In order to capture the tree structure of the interceptions in a flat ASCII script, an outline numbering convention similar to that UML Collaboration Diagrams is used.

The figure right illustrates the process by which a server is built using MCITT. The process is similar to that used to build a CORBA server from IDL (see also paragraph 4.1.4) and a C++ implementation, but in this case the C++ implementation is generated by MCITT from the ITL and/or CIS files that the user provides. The advantage is that it is less labor-intensive and less error-prone to specify the behaviours needed for a testing scenario in ITL or CIS, letting MCITT generate all of the boilerplate code and stubbed-out operations, than it is to implement the entire scenario directly in C++. A "smaller win" is that equivalent servers can then be generated for different platforms with relative ease by selecting different *bindings* when MCITT is invoked. Another advantage of static generated test servers is that it can be used in combination with any tracing instrumentation technique.

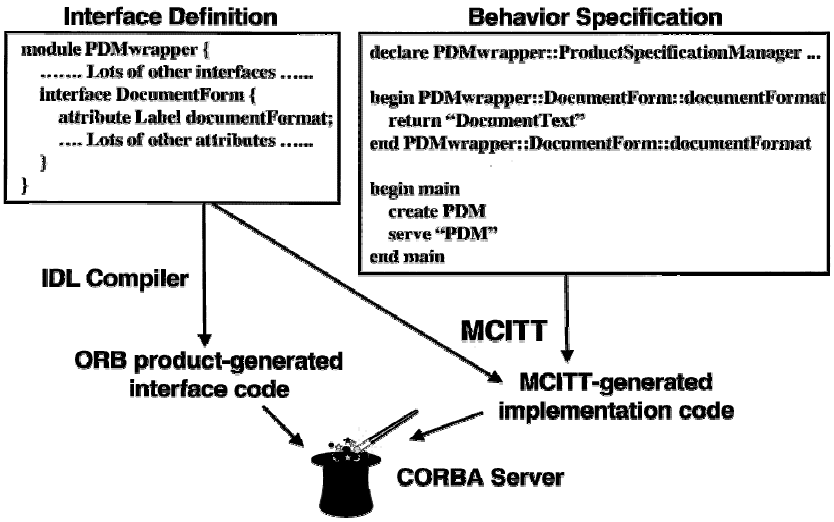


Figure 6.17: MCITT Test Server generation

Currently, the MCITT distribution comes with bindings for Orbix 2 with the Sun C++ compiler (Solaris) and the Visual C++ compiler (Windows NT 4.0), and a partial binding for OrbixWeb 3. By copying and modifying the existing bindings with a text editor, developers can easily create bindings for other platforms. The platform changes can then be applied to all ITL compilations that select the new bindings.

Beside ITL and CIS functionality, the MCITT offers also following supporting tools:

- Idlmkmf is a specialised IDL make makefile utility. It generates makefile rules for all the IDL files in the working directory, plus a few useful variable definitions. Then it is only necessary to add rules specific to the clients and server being build to produce a complete makefile for the Corba application.
- TEd is the NITS Test Editor. It is a tool for non-interactive editing of test scripts and programs. Ted is used for example to automate the migration of ITL files from a C++ binding to JAVA. Written for the NITS SQL conformance-testing program, it provides much of the functionality of the standard Unix tool sed, but is more portable and has some operational differences that make it better suited for editing test cases. Because of its usefulness in this regard, it has been included in MCITT.

Although MCITT cannot be used to trace the interaction between components directly, it can be used for the construction of a controlled testing environment, which logs all Corba communication between interfaces.

6.3 An Evaluation Of Corba Test Frameworks

In this paragraph, we will try to compare the different Corba testing tool based on the criteria specified by paragraph 6.1. For every criteria group, we made a separate subparagraph containing a table, in which we rated every individual criterion for every evaluated tool. Note that criteria that could not be compared validly are rated as not applicable (n.a.). In the last subparagraphs, we will make our final comparison based on the information of the previous subparagraphs and give our recommendation of the best tracing for Thales.

6.3.1 Controllability Functionality

Based on the information available to us we evaluated the performance of the controllability functionality of the Corba testing tools with the following conclusions:

Test Tool	Functionality	Comment	Rating
DSC Testing Toolkit	Dynamic Invocation	By using Corba Interface Repository, DII and DSI, the DSC tool is able to offers a user-friendly GUI, which supports the dynamic selection and connection with components interface and helps the user create a correct message by constraining message input.	++
	Manual Invocation	The DSC actor/reactor tool can be used in manual mode, interactive mode (interruption mode) and fully automated mode.	++
	Automatic Invocation	For automation the actor/reactor utility can use the popular Tcl scripting language, which allows the simulation complex component behaviour.	+
	Workload Invocation	Although DSC does not offer active workload support, testers can use multiple scripted actor/reactor utilities to create a realistic workload.	+/-
Corba Trace		Corba trace does not support any controllability functionality itself and relies on the usage of external test tools.	n.a.
Silk Observer/Test/ Performer	Dynamic Invocation	The Silk Test GUI can connect and issue request based on the interface descriptions that are automatically retrieve from the Corba Interface Repository.	+
	Manual Invocation	Although Silk Observer self cannot invocate any component, Silk Test offers good manual invocation functionality.	+
	Automatic Invocation	The Silk Test supports regression testing by a proprietary integrated testing scripting language called 4Test. However, the 4Test scripting language is an object-oriented language specially designed for regression testing.	+
	Workload Invocation	Silk Performer is able to record IIOP traffic, modify its parameters and use it to generate a large workload	++
Ethereal Sniffer		Ethereal network sniffer does not support any controllability functionality and relies on the usage of external test tools.	n.a.
MCITT	Dynamic Invocation	The MCITT will use Corba DSI and DII services for the connection to other components.	+
	Manual Invocation	Although the MCITT tool cannot send a simple invocation directly, it is able to generate and launch a small scripted component performing the same action.	+/-
	Automatic Invocation	MCITT can generate multiple automated test components from a single definition file, which will save the tester time and prevents.	++
	Workload Invocation	Although MCITT was not originally indented for this purpose, the ability to generate multiple test component from a single file can be exploited to create a large group of test component producing a large invocation workload	+

6.3.2 Trace Observability Functionality

Test Tool	Functionality	Comment	Rating
DSC Testing Toolkit	Message tracing	The DSC toolkit is able to trace all message communication flowing through distributed system including those generated by DSC control utilities	++
	Message visualisation	The DSC toolkit generates a sequence diagram from all traced communication events. The sequence diagram contains invocation information and can be zoomed.	+
	Message context	An invocation message contains both a client and server name, which is automatically derived from the IDL file. In order to make instance uniquely identifiable, the component name is concatenated with a generated postfix	++
	Message contents	The DSC toolkit is able to read all message contents information with it instrumented stubs and skeletons, which is exchangeable with all Corba implementations.	++
	Message casualty	Message casually plays a central role in the DSC trace framework. The DSC tracing mechanism collects causality information at all interception points, which can be separately visualised in the sequence diagram.	++
	Message timing	Although DSC tracing mechanism collect tracing information, it must be manually analysed by the tester.	+
	Message filtering	The DSC of some basic message filtering behaviour, which can be set by modifying the property files.	+
Corba Trace	Message tracing	Although Corba Trace is able trace most Corba message communication, it will not be able to trace message produced by external component testing tools. This is because the tracing mechanism requires the modification of it control tool source code which not always available. Corba trace therefore requires the tester to create his own instrument	+
	Message visualisation	Corba trace can generate a UML sequence diagram based on log files, message types, message contents and time stamps a view in multiple formats	++
	Message context	Although an intercepted invocation can contain both client and server name, it name must be manually instrumented in the Corba objects source code.	+
	Message contents	Although The Corba trace is able to read all information inside a message with adapter mechanism, it must manually instrumented in every Corba object source code.	+
	Message casualty	Although the sequence diagrams produced by Corba Trace can be used to derive some casual relation between message, it will does not actively support causality tracing.	+/-
	Message timing	Although Corba tracing mechanism can collect timing information, it must be manually analysed by the tester.	+
	Message filtering	Although Corba trace mechanism offers some filtering capability, it must be manually set in the Corba object source code.	+
Silk Observer	Message tracing	Silk Observer is able to trace all message communication generated by component using the instrumented libraries.	+
	Message visualisation	Although we know Silk Observer can visualised intercepted message with a viewer, we know little of the characteristics of the viewer.	+
	Message context	With the classes instrumented in the source code, Silk Observer is able to collect context data	+
	Message contents	Because the tracing code is instrumented in Orb itself, it is able to access all message contents information.	++
	Message casualty	Although Silk Observer does not actively supports causality tracing, Silk observer can be used to determine the relation between messages by comparing timestamps.	+/-
	Message timing analyses	Silk Observer support the ability to perform automatic timing analysis of intercepted message	++
	Message filtering	Silk Observer allows tester to define filter profiles to scope the traffic to only the interfaces, methods, or parameters of interest.	++
Ethereal network sniffer	Message tracing	Although The Ethereal network sniffer is able to intercept all message communication between computers, message communication between components located on the same computer cannot be detected.	+/-
	Message visualisation	Ethereal sniffer does not support any visualisation of messages.	--
	Message context	Messages only contain the name of the interface being invoked, not the component name of either client or server.	-
	Message contents	Although Ethereal sniffer is able to capture, filter, and transform the message contents into some readable format, it will be very hard to determine which component produced it.	+
	Message causality	By comparing timestamps, a tester could reconstruct the causality between messages created by simple test cases. However, it will quickly become very hard for complex test cases.	-
	Message timing analyse	Although Etherial offers some good statistical time analysis function, the fact that it can only analyse intercepted network message limit its use	+/-
	Message filtering	Etherial filter mechanism allow the exclusion of undesired messages	+
MCITT		Except for the invocation results returned by its non-interactive test components are not able to observe any internal Corba communication.	n.a.

6.3.3 Tool Performance

Test Tool	Functionality	Comment	Rating
DSC Testing Toolkit	Distributed tracing	Locally intercepted messages data is automatically distributed to global trace server, which can distribute the tracing data further to multiple viewers.	++
	Runtime tracing	Because DSC tracing framework uses asynchronous communication for the distribution of intercepted trace data, Corba applications under test will not be interrupted. However, there exist a small chance the extra communication overhead will overload the Orb causing brief interruption to the Corba application.	+/-
	Online tracing	The DSC tracing distribution framework allows tester to analyse intercepted tracing messages online during the interception process.	+
	Offline tracing	Every viewer will locally store received tracing data, which can be viewed offline anytime. However, the locally stored tracing data cannot be modified.	+
	Tracing overhead	In order to minimise overhead, intercepted trace data is locally buffered before it distributed to the viewers. However, due to the extra overhead caused by the trace data streams, context data and interceptors, both the application under test as the overall system will slightly be reduced.	+
	Trace configuration	Although DSC does support an integrated online configuration interface which allows a user to select the desired interface directly in the monitor. In order to activate the DSC tracing mechanism, the user must set some local configuration files, call the corresponding makefile and redistribute the components files to the distributed location.	+
Corba Trace	Distributed tracing	Although the user must manually collect tracing data located in local stored log files, the GUI has a build-in FTP client that can help the user download all log files to one central location.	-
	Runtime tracing	By storing intercepted tracing data in local log files, the tracing framework will not interrupt Corba applications	+
	Online tracing	Tracing information can only be analysed effectively after the test completed, the log files has been collected, filtered, and processing into a viewable format.	--
	Offline tracing	Collected tracing data can be analysed offline any time afterward by standard viewers, even by users who do not have installed the Corba trace tools. Normal text editors can modify both raw log files and trace export files.	++
	Tracing overhead	By storing tracing locally in log files, the trace mechanism does not generate additional communication overhead. However, the usage op Corba Portable Interceptor will always introduce a small delay.	+
	Trace configuration	Each Corba application must be manually instrumented with some specific library calls.	-
Silk Observer	Distributed tracing	The Silk Observer tracing mechanism is able to collect tracing data from multiple hosts and send it to a central viewer	+
	Runtime tracing	All instrumented object can be traced at runtime	+
	Online tracing	The viewer is able to display intercepted tracing while the tracing mechanism is still in progress.	++
	Offline tracing	Tracing data is stored in log files, which can be analysed offline.	+
	Tracing overhead	The trace data distribution steams generate most trace overhead.	+
	Trace configuration	Although all components under test must be manually instrumented with a single library call, once instrumented the tracing mechanism can be configured from a central graphical user interface.	+
Ethereal network sniffer	Distributed tracing	Although Ethereal sniffer can capture broadcasted network message communication from multiple systems, it does not support distributed tracing from multiple sniffing locations.	+/-
	Runtime tracing	Because Etherial sniffer only listens to existing network traffic, it will not change the runtime behaviour at all.	++
	Online tracing	Captured network messages can be inspected on arrival.	++
	Offline tracing	Captured network messages can be manually stored and can be analysed afterwards.	+
	Tracing overhead	When Etherial sniffer tool is deployed on a computer system, which does not contain any components, the Corba application will not be interfered in any way.	++
	Trace configuration	Etherial network sniffer does not need to be configured.	++
MCITT		MCITT does not support any tracing functionality	n.a.

6.3.4 Tool Interoperability

Test Tool	Functionality	Comment	Rating
DSC Testing Toolkit	Component Transparency	Although the component source code does not have to be instrumented, the stub and skeleton do. Fortunately, a modified IDL compiler automatically performs all instrumentation.	+
	Language Independence	Originally, the DSC trace framework could only be applied on component written in Java. Fortunately, Lucent has also worked on a C++ extension, which allows DSC Toolkit to trace Corba applications consisting of components written in JAVA or C++.	+
	Platform Independence	The Java version of DSC Toolkit is applicable in any JAVA enabled operating system. The C++ version of DSC Toolkit is available for Windows and Unix.	+
	Middleware Independence	Because of the proxy instrumentation tracing technique applied by DSC Testing toolkit is shared by all static Corba applications, the tool is applicable in most stand alone Corba implementation as well as CCM solutions.	+
	Monitor Independence	The DSC viewer, which is written in JAVA, is able to monitor both C++ components as well as JAVA components.	++
	Open Framework	Although DSC testing toolkit an open framework, it does not supply the source code. Fortunately, Lucent is willing to extend their tool with additional functionality.	+
	Standard Based	DSC tracing framework uses Portable Interceptors, which is well known Corba standard supported by most modern Orb implementations.	+
	Vendor Adaptability	Using other Java Orb implementation than those already supported can easily be accomplished by modifying specific initialisation files. However using other C++ Orb is more difficult but can be done by calling help from Lucent.	+
Corba Trace	Component Transparency	Corba Trace requires the manual instrumentation of all components implementation source code before any Corba application can be traced.	--
	Language Independence	Although Corba Trace only supports JAVA, because of the lack of support it is not unlikely new version of Java will not be compatible.	--
	Platform Independence	Corba Trace support the Microsoft Windows and Unix operating system	+
	Middleware Independence	Corba Trace is designed for standard Corba objects, which initiate their own pivate Orb. Corba Trace contents tracing functionality is based on replacing the private Orb object with an adapter object. However, Corba components do not have a private Orb. Corba trace therefore cannot work in conjunction with CCM.	-
	Monitor Independence	Corba trace can generate a sequence diagram in multiple common formats, which can be analysed by any compatible viewer.	++
	Open Framework	Corba Trace includes the tool source code and design documentation. Note however that the design documentation is only available in French.	+
	Standard Based	The interception of message is based op Corba portable interceptor.	+
	Vendor Adaptability	Although Corba Trace only contains library support for Sun JDK and Orbacus Orb, Corba Trace supplies the design documentation which can be used to make the framework compatible with Orbs from other vendors that support Corba portable interceptors	+
Silk Observer	Component Transparency	Although the tracing mechanism requires the instrumentation of the component source code with a single library call, once instrumented and the tracing mechanism can be activated/deactivated after a single mouse click.	-
	Language Independence	Silk Observer supports both Java and C++ programming language.	+
	Platform Independence	Silk Observer supports both Windows and Unix operating system	+
	Middleware Independence	Silk Observer is only able to trace the Visbroker and Orbix Middleware solutions.	-
	Monitor Independence	The tool can be used in Microsoft windows or UNIX operating system.	+
	Open Framework	Silk Observer is in the first place a commercial tool, which is not intended as an open framework. However, the server component contains an open DLL interface definition, which could potentially be used to extend Silk Observer tracing framework with additional functionality.	+/-
	Standard Based	Except from the supported programming languages, operating system and middleware, the Silk Observer tool does not support any other standards.	--
	Vendor Adaptability	Because of it dependence on proprietary message filter functionality, the tracing framework is only compatible with the Visibroker and Orbix Orb.	--
Ethereal network sniffer	Component Transparency	Ethereal network sniffer is fully transparent	++
	Language Independence	Ethereal network sniffer is fully language dependant	++
	Platform Independence	Ethereal network sniffer is supports Mircrosoft Windows and Unix OS	+
	Middleware Independence	Ethereal network sniffer can trace any non encrypted network message	+
	Monitor Independence	Ethereal network sniffer is supports Mircrosoft Windows and Unix OS	+
	Open Framework	Ethereal network sniffer is open source and contains development documentation	++
	Standard Based	Ethereal network sniffer support almost all known network message protocols.	++
	Vendor Adaptability	Although Ethereal network sniffer already contains over 400 build-in filters, the user can manually construct new filters.	++

MCITT		MCITT Interoperability cannot be validly compared with the other tracing tools	n.a.

6.3.5 Final Comparison

We will now make our final comparison. For overview, we collected all ratings from the previous tables into a single comparison table. Note that we added an extra criterion to the table, which we did not rate in any of the previous tables. Although 'cost' was not a criterion, we added in into the final comparison table because it can play a relevant role in our final caparison.

Group	Functionality	Lucent DSC Testing toolkit	Corba Trace	Silk Observer	Ethereal sniffer	MCITT
Message Controllability	Dynamic Invocation	++	n.a.	+	n.a.	+
	Manual Invocation	++	n.a.	+	n.a.	+/-
	Automatic Invocation	+	n.a.	+	n.a.	++
	Workload Invocation	+/-	n.a.	++	n.a.	+
Message Observability	Message tracing	++	+	+	+/-	n.a.
	Message visualisation	+	++	+	--	n.a.
	Message context	++	+/-	+	-	n.a.
	Message contents	++	+	++	+	n.a.
	Message casualty	++	+/-	+/-	-	n.a.
	Message timing	+	+	++	+/-	n.a.
	Message filtering	+	+	++	+	n.a.
Tracing Framework Performance	Distributed tracing	++	-	+	+/-	n.a.
	Runtime tracing	+/-	+	+	++	n.a.
	Online tracing	+	--	+	++	n.a.
	Offline tracing	+	++	+	+	n.a.
	Tracing overhead	+	+	+	++	n.a.
	Trace configuration	+	-	+	++	n.a.
Tracing Framework Interoperability	Component Transparency	+	--	+	++	n.a.
	Language Independence	+	--	+	++	n.a.
	Platform Independence	+	+	+	+	n.a.
	Middleware Independence	+	-	+	+	n.a.
	Monitor Independence	++	++	+	+	n.a.
	Open Framework	+	+	+/-	++	n.a.
	Standard Based	+	+	+	++	n.a.
	Vendor Adaptability	+	+	+/-	++	n.a.
extra	Cost	--	++	?	++	+

For each criteria group we first discuss the best tool:

- *Message Controllability*. For controllability, we can identify a clear winner. While the DSC Control tools are very useful for quick testing, the MCITT tool is very good creating an automated test environment and Segue tool family shines in generating large workloads.
- *Message Observability*. The DSC offers clearly offers the best message observability. Especially the causality tracing functionality will be very useful when tracing multiple Corba application at the same. Second best is Silk observer because of it of it excellent timing analysis functionality. Third comes Corba Trace, which offer some Observability but limited. Ethereal network sniffer has the worst communication observability. The IIOP network messages captured by the Ethereal sniffer is lacking context information, is limited to communication between hosts and cannot be visualise in a sequence diagram.
- *Tracing Framework Performance*. Although Ethereal sniffers has the best framework performance, its comparison to other trace tools is rather unfair. While the other tracing tools must deploy complex tracing mechanisms, Ethereal sniffer cheats performance by not deploying any tracing mechanism. Instead, it intercepts tracing data by simply listening to all broadcasted network message traffic. The honour of best tracing framework performance is shared between Silk Observer and DSC Toolkit. Both tools are able to configure the tracing framework from a distributed graphic user interface and allow users to view a sequence diagram being build up at real time. Corba Trace ends on the last place because of its inability to trace applications online and its lack of configuration support.
- *Tracing Framework Interoperability*. Not surprisingly, the Ethereal network sniffer also shows the best interoperability because except from encrypted messages, it can read any network message and thanks to it availability of the source code it is portable to almost any operating system. Second best is without a doubt, Lucent DSC tracing tool. Thanks to its stub/skeleton instrumentation and usage of Corba Portable

Interceptors standard, Lucent DSC Toolkit, is able to trace any statically linked Corba/CCM application. Combined with their support for multiple programming languages, operation systems, Orbs, and adaptability to other orb implementations, makes DSC the second best interoperable tool in our test. Third comes Corba Trace. Their tracing framework should be applicable to pure Corba application build on any Orb that supports Corba Portable Interceptor specification. Its interoperability is mainly limited to by instrumentation technique applied. It requires the instrumentation of both client and server source code and requires the usage of the object reference, which is not directly accessible in CCM client/server source. The tool with the worst interoperability in our comparison is the Silk Observer. It requires both the instrumentation of the source code similar to Corba Trace but beside that only function with Orb implementations supporting a specific proprietary filtered logging facility.

Our overall tool recommendation is a mixed bag. All tools offer some advantages and disadvantage compared to each other. We therefore cannot point is single best tracing tool. We therefore must decide on criteria, which are most important for Thales.

Ranging from most important criteria to less important criteria, we can make the following notes:

- Most importantly, the tool must be available to now, which excludes the Silk Observer tool because the official distributor no longer sells its tracing tool. Note however, that Silk Observer successor, Silk Performer Component Tester, will support Corba tracing in the future.
- Secondly, the tool must be able to trace the systems real behaviour, which excludes the MCITT tool because its proxy technique alters the architecture of a component-based software system and therefore its runtime behaviour.
- Thirdly, it must be able to trace all Corba communication, not just the network messages or invocation results. This requirement excludes the Ethereal network sniffer because it can only intercept message travelling between hosts.
- Thales requires the tool must be able to trace communication online and at runtime, which excludes Corba trace because can only trace offline after the test is completed.

The only tool that remains, is the DSC Testing toolkit created by Lucent Technologies. Their tool can be applied to statically linked Corba environment. Because PERCO CCM implementation links all component statically, all CCM Corba application developed with PERCO should therefore be traceable with Lucent tracing framework.

Although DSC will currently only supports component written in Java, they have a prototype for tracing C++ components and are willing to help Thales integrate their tracing technology into Thales Middleware technology. An alternative to the Lucent DSC Toolkit would be to use Corba Trace. Although it does not support online tracing and configuration, it can produce good sequence diagrams and it extendable. The source code and French design documentation would allows Thales to ingrate Corba Trace into their CCM implementation and extent the tool with missing tracing framework functionality like configuration and online tracing.

7 AN ANALYSIS FOR A TRANSPARENT CORBA TRACING MECHANISM

In chapter 3 we introduced a high level design for the construction of a tracing framework that could habour multiple tracing mechanims. In chapters 5, we descused several trace data inteception techniques that might be used for the construction of a Corba tracing framework. In this chapter, we analyse the possibilities for the construction of transparent tracing mechanism, based only on the request interceptor (see paragraph 5.2.1). Although there are other tools that have done something similar, our solution does not require alteration the component source or middleware and can trace Java Corba objects.

The reason we use Request Inteceptors is because it currently the only trace data inteception technique that it allows the transparent inteception of all message communication events at both client and server side, can transparently transer context data between inteceptors and can be transparently deployed. The only two requirement on the testing requirement is are that that every Corba object initiates its own unique Orb object and that the Orb supports the registration itercepors by parameters.

Based on the tasks required by tracing mechanism, we will analyse the following tasks:

- *Paragraph 7.1 Tracing Data Interception*: the interception of message data in from the middleware
- *Paragraph 7.2 Tracing Data Processing*: the retrieval and processing of trace data
- *Paragraph 7.3 Tracing mechanism Management*: the management of the mechanism tracing behaviour
- *Paragraph 7.4 Tracing mechanism Deployment*: the deployment of the trace data interception mechanism

7.1 Tracing Data Interception

In this paragraph, we explain how we want to retrieve tracing data from Corba applications. As already explained, we want to use request interceptor to retrieve tracing data from the system under test. In order to trace Corba applications, Request Interceptors must collect the following tracing data types:

- *Communication Message ID*. Because a Corba message can be intercepted multiple times during a message reply sequence, and our tracing framework is only interested in atomic information flows, the tracing mechanism needs some way to discard duplicate message information. A possible way to achieve this is uniquely identify messages within our system. When processing intercepted messages, the tracing layer can combine the intercepted information with the same communication message ID into an atomic information flow entrance.
- *Timing Data*. The goal of Timing Data is two folds. It allows testers to measure the performance of a system and to analyse the casual relationship between tracing events. The performance of a system can be measured by calculating the time difference between tracing events. Casual relationship between tracing events can be derived from sorting communication events in chronological order.
- *Context Data*. Although a unique component instance ID allows a tester to identify a Component instance, it will be of little meaning if he does not know what component he is dealing with. The collected tracing data should therefore also include meaning component name. The Component name does necessary have to be unique, it should however allow the tester a indication what to look for
- *Instance ID.* Our high level tracing mechanism requires that that the smallest unit of operation, a component instance, is uniquely identifiable within our framework. The ability to uniquely identify a component instance is very important in tracing because a tester needs to be able to verify the in and output of a unique identifiable process.
- *Contents Data*. Message Content Data is all information exchanged between client and server. This information includes input parameters, output parameters, return value and function name called by the client. By comparing the input parameters with out parameters and return value, a tester is able to verify the correct functional behaviour of a component

In the following sub paragraphs, we explain how each tracing data types can be retrieved using request interceptors.

7.1.1 Message Communication ID

A unique communication ID allows the tracing mechanism to group communication events belonging to the same remote message invocation. In this paragraph, we will look at how intercepted messages can be made uniquely identifiable within our tracing framework. Similar to the generation of a component instance ID, a Central ID Broker component could be use retrieve a unique communication message ID. However, because the tracing interceptor would have to wait on the Broker to return a Message ID, a huge non-negligible overhead to all traced communication would be created. Therefore, we should look for techniques, which can create a communication message ID locally.

A natural course of action would be use the attributes contained in the intercepted message (see Interceptor Request Info). We cannot use the `request_id` as an identifier because it only identifies an active request reply sequence between two Orbs. Once a request reply message is concluded, the same `request_id` might be used again. It is therefore theoretically possible that two non-related messages are send with exactly the same attribute values. Although we cannot uniquely identify a message by any combination of the attributes retrieved from the message, there are several techniques to uniquely identify a message:

- Physical Timestamp. We can make the message uniquely identifiable by combining the `request_id`, `target_id` with a physical stamp. We can create a physical timestamp by simply taking the current system time. Timestamp. Because our tracing frameworks required a physical timestamp anyway, this technique is the easiest way to achieve message uniqueness. Note however, that special care must be taking in the granularity of the physical time stamp. The time stamp must be precise enough to prevent duplicate message from occurring ever occurring, which might not always be the case.
- Logical Timestamp. We can make a message also uniquely identifiable by combining interceptor ID with a logical timestamp. A logical timestamp is a locally maintained counter that is incremented after every usage. Care should be taken to prevent overflow by using a sufficient large counter. The interceptor ID must be able to uniquely identify the counter instance. The advantage of a logical timestamp is that the uniqueness is always guaranteed.

The Message ID should be created for every unique message, the first time it is intercepted by any of the deployed tracing interceptors. The message ID should be transmitted along with other tracing data to the next interception point. The picture below illustrates the message ID flows between interception points during a request reply sequence. Between the outbound client to server and inbound server to client interceptions points, the information transmitted by context data, and between server outbound and inbound interception point, the information travels with the Orbs shared memory mechanism, PICurent (see appendix J).

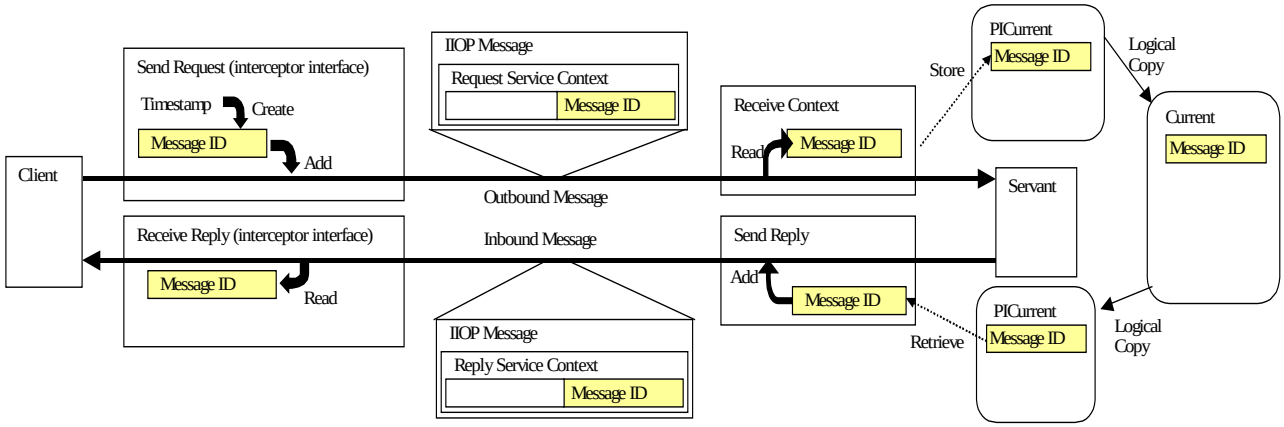


Figure 7.1: Message ID travelling by service context mechanism and PICurrent

7.1.2 Timing Data

Timing data is any information that tells us something about the ordering of events through time. A single timed event therefore has no meaning itself. A timed event only has meaning if it can be validly compared with another timed events.

We can identify two main purposes of timing data:

- *Time Measuring*. By measuring the time between two related events, we can measure the commutative time it took for processes happening between the first and last event to complete. The events that happen during a process can vary widely. A process could only be a simple component instance returning a reply but it could also be complex process that negotiated with hundred other processes before returning a result. A measured time can tell interesting context sensitive information. For example, the time difference between a request message arriving at a component instance and a reply message leaving the same component instance can tell us something about the response time of the component instance.
- *Time Sorting*. Eventually, a tester would like to display communication events in chronological order. When all intercepted communication events are accompanied by a timestamp, the tracing framework is able to reconstruct a sequence diagram visualising the communication between component instances. Tracing Interceptors can create a timestamp by acquiring the internal clock value directly after the Orb triggers an interception point. By chronologically sorting the communication events, the tracing framework can also discover the casual relationship between messages. For example, when a message is send by a component

instance after receiving a message from another component instance, our tracing framework could conclude that there is a casual relationship between the incoming and outgoing message. When sufficient messages can be found with a casual relationship, the tracing framework would allow the tester to trace a message flow through a system from beginning to end.

It may seem simple to sort communication events based on the timestamp taken from the interception point internal clock. However, acquiring reliable timing data in a distributed system is not a trivial matter. That is because distributed system systems lack a globally maintained and directly accessible clock value. Instead, every computer in the distributed system maintains its own physical internal clock. When trying to compare timed events taken from different systems with non synchronised physical clock values, the derived information can be unreliable or result in tachyons. Tachyons are contradicting measurement, which logically cannot be true. In figure 7.2, we see an example of a tachyon. When the clock value of the client is higher than the clock value of the server, an intercepted request at the server might appear to be received earlier than the same intercepted request send from the client. Unless either component travelled trough time, this could of course not be possible. The next paragraph will investigate the possible solutions the this problem

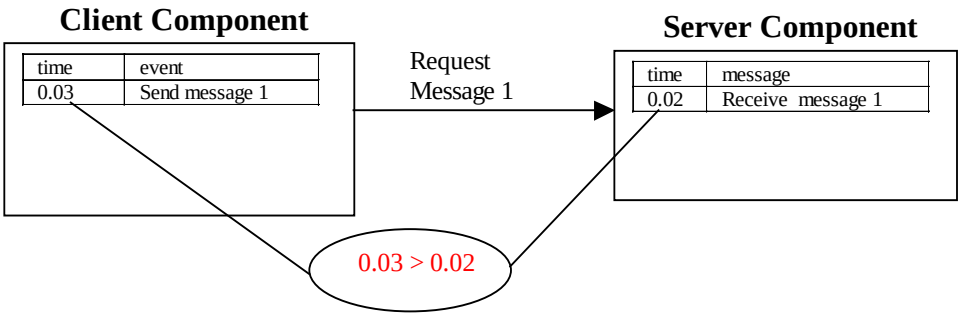


Figure 7.2: Tachyon Example

Fortunately, distributed clock problem is a well-known problem for which several solutions are available. We will sketch some of them an discuss how they could be applied by our tracing mechanism:

- *External Synchronised Clocks*. If the all clocks in the distributed system are synchronised with some external synchronisation mechanism, our tracing mechanism could simply use the system internal clock value for timestamping all intercepted communication events. This could for example be achieved using the Network Time Protocol (NTP) or by embedded atomic clock radio receiver synchronising the clock after receiving a pulse. Since every host in our tracing framework is fitted with a Collector, the Collector would also be the logical place to synchronise the host local clocks value.
- *Time Service*. Since the problem of inconsistent local physical clocks arises in many distributed applications, the OMG defined a Corba Time Service specification [26]. This service can be used to obtain consistent timestamps along with error estimates. Unfortunately, the Corba Time Service is only implemented in only fraction of all Orb implementations.
- *Logical Clock*. When the precision of the available clock is lacking, or even if there is no physical clock available at all, the tracing mechanism should use an alternative time mechanism. Instead of using a real clock value, the tracing mechanism could also employ a logical clock for chronologically sorting events on time. That is because a logical clock reflects the 'happened-before' relation or causality relationship. By guaranteeing to be consistent with the partial order defined by the relation of logical precedence, a logical clock can prevent tachyons from ever occurring. In order to prevent tackyons between components, the local counter value is sent along with every message. This can be easily implemented by sending service context between request interceptors. The basic idea is that the Tracer installed on a Corba object maintains a counter, which represent the current logical clock state of a Corba instance. After every communication event, the Tracer sends its logical clock value along with other intercepted information to the locally hosted Collector for further processing. Depending on the logical clock value received, the Tracer takes different actions. When the tracer receives a logical clock value lower or equal to its own logical value, it will simple increment it logical value. When a component receives a logical clock higher than its current logical clock, it will simply set it local logical clock one value higher than received value. If the component received no logical clock from the other component, it will only increment it current logical clock value. By synchronising the logical clock value after every communication event, it is ensured that the logical clock value of a server receiving a message is always larger than the logic clock of the sender. However, there are some problems associated with the logical clock, which will be further explained appendix C.

- *Physical Clock with Logical Clock Correction*. Another possibility to creating a more reliable autonomous clock is to use physical clocks with logical clock correction. This approach tries to combine the benefits of a physical with a logical clock. While the physical clock is mainly used timestamping, the physical clock value is just as a logical clock sent along with every message. Just as with logical clocks, the receiver can check if a tachyon would be generated, and it can be avoided by setting the receiver's physical clock accordingly. If however the receiver does not have the permissions to change the system clock, the tracing mechanism could simply maintain a displacement, which starts at 0, and updates the displacements in such cases. Although the time-stamps do not reflect the physical time precisely, today's computer are usually equipped with clocks of high precision so that in practice, the time stamps generated this way provide a good approximation of the physical time.

7.1.3 Context Data

Gaining access to a meaningful object/component name is one of the most difficult problems when tracing Corba Applications. Although at the server side, we can easily identify a server by the `target_id`, on the client side this simply not possible because Corba message do not contain any `source_id` identifying the client object/component. The tracing mechanism can use the request interceptor to gain access to a meaningful name, by one of the following techniques:

- *Dynamically by using Interface Repository*. Corba Interface Repository (also see paragraph 4.1.5.1) can be used to derive a meaningful name dynamically. Although IOR itself does not uniquely identify a Corba object, by retrieving component contents information linked to an IOR, the Interface repository can be used to reconstruct a meaningful Component name. The request interceptor can gain access to his own IOR after the first time a request interceptor receives a message from another component, it can read his own IOR contained `target_id` attribute (for more details see appendix H). In figure 7.3, we can see that the IOR is captured by the request interceptors, transmitted through the collector to the processor, and used to retrieve a meaningful name from the interface repository before it if finally stored in the Trace information model.
- *Statically by constant declaration*. A meaningful name could be statically implemented as a constant string declaration located in the Interceptor source code. This technique however requires extra deployment preparation because every Orb must be initialised with a unique interceptor implementation containing a unique string. To solve this complexity, simple source code generation tools could be used which use a template of a standard interceptor and only add a unique component name.

The Dynamic strategy is preferable above the static one because we can use a single request interceptor implementation for all Tracers.

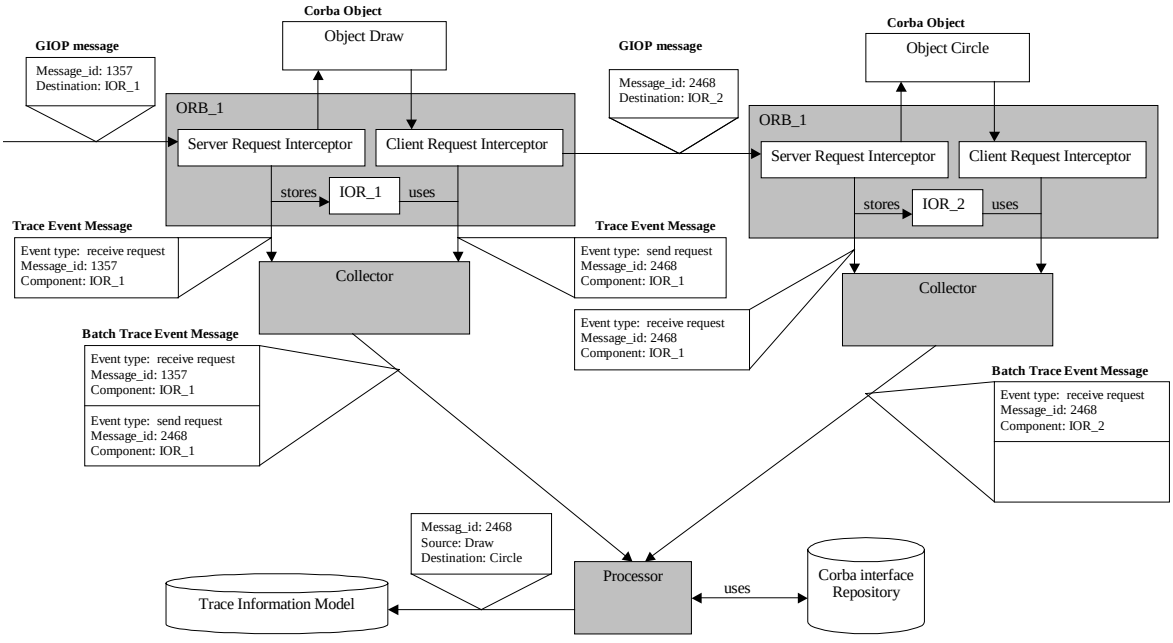


Figure 7.3: retrieval of source and destination context from IOR

7.1.4 Component Instance ID

As explained before, Corba uses IOR to connect Corba objects with each other (see also appendix A). Although IOR uniquely identifies a Corba Object, they cannot be used as an instance ID because IOR identify Corba object implementations, not an instance. Client objects that do not implement any Corba interface simply do not even have an IOR. Moreover, a Corba Object may contain several Orb instances, and may consequently be associated with multiple IORs. Therefore, component instance IDs have to be generated somehow.

Many methods could be used to create a unique component instance ID. Because we cannot list all possibilities, we restrict our selves to three methods:

- *Centralized ID Generation*. In the centralized method a special ID broker component is used for the distribution of Component Instance ID. This ID Broker component would maintain a global counter, which would be incremented after every time a new ID was requested. At startup, or the first time that a communication event occurs, every component registers itself with the ID Broker, whereupon it gets a unique ID, together with some policy information. In order guarantee uniqueness of every ID, The counter would have to large enough to prevent overflow and semaphores could be used to force every ID distribution to become single atomic step. A disadvantage of Centralized ID generation is that create extra overhead, but this is negligible because it only occurs at the start.
- *Decentralized ID Generation*. In the decentralized method, components themselves generate a their own unique id. By combining a unique object ID with a globally maintained counter, a unique component instance ID can be constructed. An Instance ID can be constructed by generating a pseudo random number. Although absolute guarantee for uniqueness can never be achieved, we can however make it extremely unlikely duplicate ID occurring by seeding the random generator with context sensitive data like IP number, time stamp and process ID.
- *Prepared ID Generation.* Prepared ID Generation is a combination of the two former techniques. Just like Decentralized ID Generation method, a new Component Instance identifier is constructed by combining the Component ID with an instance counter. The difference lies in the way the Component ID a created. Instead of generating a Component ID at run time, a Component ID generated centralized fashion before compilation. For every new component a unique ID is generated a registered at a central location for future usage. The serial number is simply implemented as a constant declaration in the source code. By deploying every component with a unique tracing interceptor containing a unique ID, interceptors can use the Component serial key to create a unique component instance ID. An advantage of this technique is that Components will always bear the same component ID, even after recompilation. A disadvantage is that every trace interceptor must contain a different component serial number.

7.1.5 Message Contents Data

By Message content data, we mean the information exchanged by method invocation calls, which contains in and output parameters and function name. The reply message contains beside all outbound information, output variables, and result value. The Tracer can collect the Message contents data together with other information contained in the Corba message by accessing the attributes in the request info object, which the request interceptor received from the Orb (see appendix H for more details).

Not every interface of the request interceptor can access this contents information. Request and reply message Contents data can only be read at four interception points (`send_request`, `receive_request`, `send_reply`, `receive_reply`). In figure 7.4, we can see how the four tracing points can be used to retrieve context data from request and reply message from client and server and send it to the Collector.

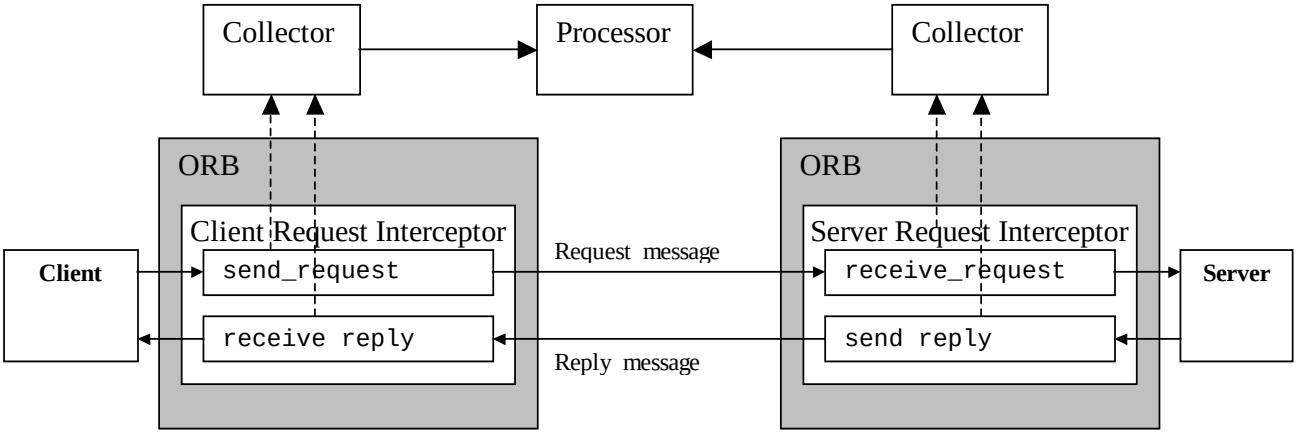


Figure 7.4: Intercepting Content information

Although the reply contains all available message content data, we could theoretically intercept all contents data by implementing only a single interception interface. However, we risk missing important tracing data because something might go wrong during communication. For example, a client method invocation might never reach its destination or a server throws an exception causing our interception point never to be invoked. Therefore, the safest strategy would be to use all 4 interception interfaces for collecting contents data. To increase performance, the interceptor behaviour could use a trace profile to determine which interception points will send trace events to the Collector. However, even if we use four interception points, we still have some circumstances that could jeopardise tracing process.

As already mentioned, request interceptors on a Java Orb are not able to access the message contents (see paragraph 5.2.1). As long as either client or server is not made in Java, we will be able to access the message contents information on the non Java Orb. However, when both client and servant component are implemented in Java, our tracing interceptors will not be able to access all required fields in the intercepted message. To overcome this Java shortcoming, the literature proposes using a proxy pattern with client side redirection [Marchetti]. Figure 7.5 illustrates how a client request interceptor is used to redirect an operation from the original server object to a locally hosted proxy object. The Proxy pattern mechanism is similar to a Wrapper (see also paragraph 5.1.6), in the fact that it wraps the server interface, and calls the original server interface through another connection. However, in contrast to Wrapping, the redirection is not achieved by renaming the server but by redirecting the message with a forward exception thrown by request interceptors. The proxy object would then be able to log all missing data by simply storing the input parameters received from client and by storing the parameters and by reading the result of the server invocation.

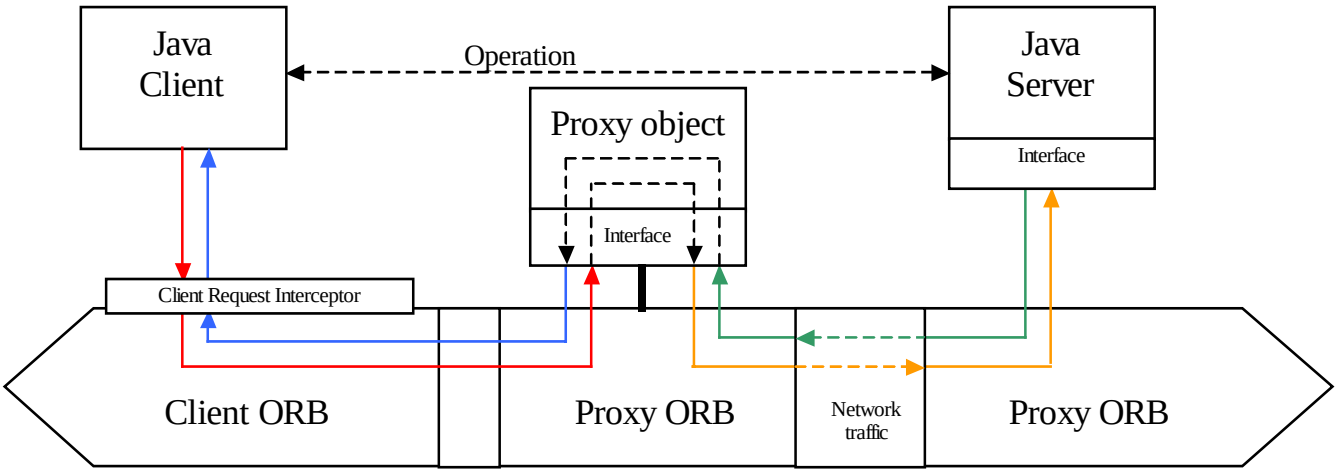


Figure 7.5: Proxy Pattern

Although this might seem the solution to a Java problem, there are however several problems with this Proxy Pattern technique:

- *Information Loss*. Once the re-invoked message receives at the Java server, it has lost all security, fault tolerance, or any context information. That is because this information gets lost once the original request

message enters the proxy object. Therefore, it would become incompatible with any Orb, which sends additional data trough the GIOP message.

- *Overhead*. A defensive strategy would require using a proxy pattern for all communication between components, which would slows down communication. In order to minimise communication overhead, the client Request Interceptor should only forward the message to a proxy object when both client and server Interceptors are unable to read parameters (e.g. both Java). However, this requires that the client Request Interceptor have knowledge whether or not any Request Interceptor installed on the server-side is able to read and store all Request Information.
- *Complexity* Because a proxy object is hosted locally, the Proxy Pattern requires a separate proxy object between every client and server. To prevent the usage separate proxy objects between every connection, the tracing mechanism needs to know which client and server object communicate with each other, which will make tracing deployment more difficult.

In the next sub paragraph, we will introduce a message contents tracing technique, which solves the problems of the Proxy Pattern.

7.1.5.1 A New Transparent Proxy Mechanism for tracing message contents

To overcome this loss of message transparency, we need solution that allows our tracing mechanism to access the message contents but also ensure that the message does not lose information. Although request Interceptors can forward a request only to a one alternative address, there no limitation on the number of times a message can be forwarded. We can exploit this functionality to transparently position a proxy object between client and server without losing message information. In figure 7.6 we can see how forwarding mechanism can be exploited to forward the same message to multiple servants by link them like a daisy chain each forwarding the message to the next servant in the chain. Instead of using a proxy that wraps the interface of the server, we use C++ dummy object with a server request interceptor installed. In contrast to the proxy object that re-invokes the server object, the Dummy object is an empty object that will not do anything. Because the Dummy Server is a C++ Orb, the C++ request interceptor is able to access all message contents. After logging the missing attributes, the request interceptor throws a second forward exception, which diverts the message back to its original destination, the Java server. Because the message received from the client is not altered (except the forward address fields), the server will receive the same message send by the client. The request interceptor knows the address of the intended servant by reading the `target` attribute located in the GIOP message.

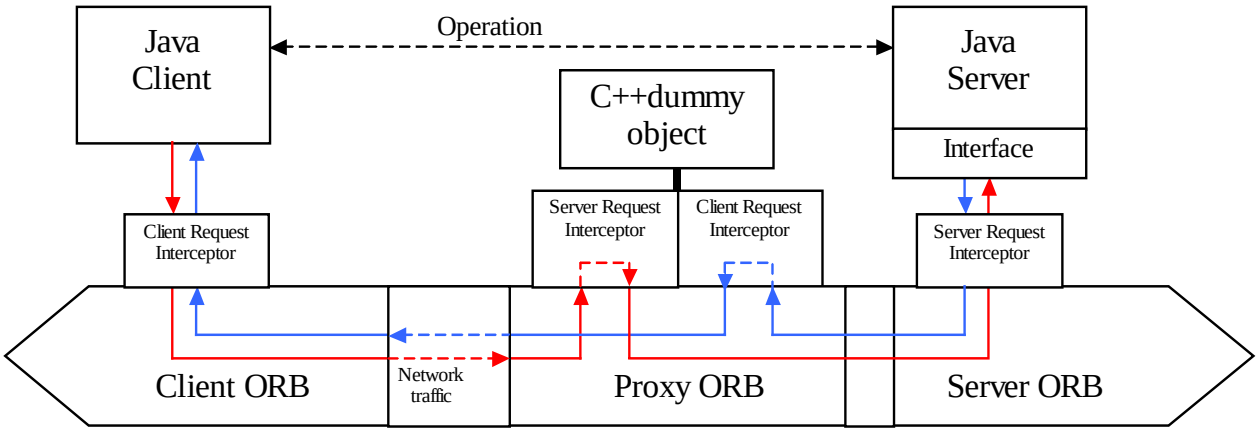


Figure 7.6: Transparent Proxy Mechanism

The request interceptor can use two redirection methods techniques to divert a message to an alternative destination, by calling a permanent forward exception, or a non-permanent forward exception. A permanent forward exception is in comparison with a non-permanent forward exception a lot faster because it only has to be thrown once to divert all messages automatically to an alternative location. The non-permanent forward technique is on the other hand a lot more flexible because it can redirect message on a per-message basis. Non-permanent forwarding allows the construction of a tracing interceptor, which could be used to trace all clients to server connections. However, for performance reasons, it is advised to use a separate trace object with interceptors for every hosted servant object. Figure 7.7 shows how by deploying the Trace object on the same host as the target object, the tracing mechanism only requires a single Trace object for every Java Servant and multiple Java clients can use the same Trace object

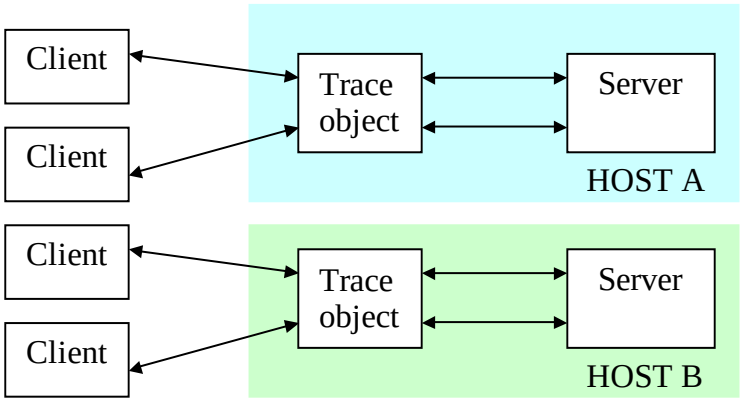


Figure 7.7: Configuration Proxies

Because of performance reasons and the problems associated with forwarding (see paragraph 5.2.1), a tracing mechanism should only use the forwarding technique when both client and server are unable to access the message contents information. Although this can be achieved by instructing the request interceptor statically in the source code, it can also be achieved at run time with minimum overhead. In order to allow the client request interceptor make this decision, it can use Corba callback mechanism to determine whether a server request interceptor is able to access the contents data in a message. By sending the success of a server request interceptor along with the reply service context, a client request interceptor will know if forwarding to the proxy object is required. Note that when the client receives a message that does not contain any service context, it know indicate that none of our server request interceptors touched the reply message.

Figure 7.8 illustrates the interceptor stack model. The interceptor stack is a last-in-first-out (LIFO) buffer mechanism that guarantees that all interceptors in the stack will always have their ending interception point called in reverse order. Whenever the Orb (without the occurrence of a user exception) successfully calls a Request Interceptor, the interceptor is placed on a Flow Stack. However, if an interceptor raised a `ForwardRequest` exception in response to a call of an interceptor, no other interception is called for that interception point. This means that forward exception can only be called once per interception point. The remaining interceptors in the Flow Stack (which are already successfully called once by the Orb) have their appropriate ending interception point called. On the client, this will be `receive_other` interface and on the server `send_other` interface.

This interceptor behaviour has the following consequences for any interceptor using forwarding:

- *Proceeding Forwarding Problem*. Any proceeding interceptors (that would normally called after our interceptor) are consequently never called by the Orb; thereby disrupting their proper functionality. Although, all preceding interceptors (that are already called are called before our interceptor) are notified of the forward exception, the Corba Portable Interceptor specification does not specify any mechanism to warn interceptors that they missed a Orb interception call.
- *Preceding Forwarding Problem*. The other-way could also be true. When another request interceptor, closer to the client object has already thrown interrupt exception. In that case, our client request interceptor that wants to divert the message to the proxy will never be called, causing our client side tracing mechanism to fall. Our client request interceptor will therefore be unable intercept the message and to re-route the message to our proxy Orb.

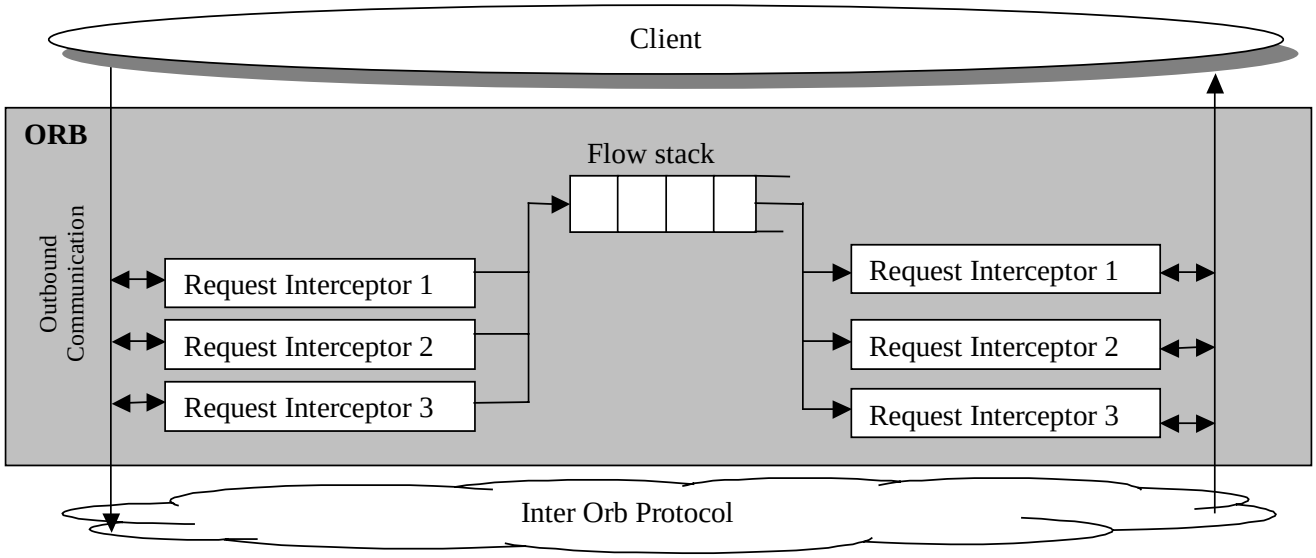


Figure 7.8: Request Interceptor Stack Model

Although we can prevent the proceeding forward problem from occurring by installing our request interceptor after other request, when another Client Request Interceptor with a higher priority in the stack, throws a forward exception (for example a fault tolerance service), the Orb will never call our client request interceptor.

To solve client side forward problem we have to forward a message at the server side to Proxy Orb. On the server side, the server request interceptor can detect whether or not the parameters have been read, by inspecting the service context. The server can detect this because every request interceptor, insert context data concerning the success on reading the parameters. After request interceptor has detected that the parameters have not been read it can employ server forward proxy technique.

The server proxy forward technique consist the following steps:

- The first time a server request interceptor receives an unread message it pushes the `request_id` on a stack buffer and throws a forward exception that diverts the message to the Proxy Orb.
- After the diverted messages arrive at the Proxy Orb, it triggers the `receive_request` interface of the server request interceptor. The Proxy Orb request interceptor reads the parameter data and throws a second forward the message
- The second time the same message is intercepted by the message, the request interceptor is able to recognise the same message by it `request_id` stored in it stack buffer. The request interceptor pops the request_id from its stack buffer and permits access to the target object like normal.
- After the target object returns the reply message on the request, the message travels back in reverse order triggering `send reply` interface, allow server request interceptor on the Proxy Orb to access the output parameters.

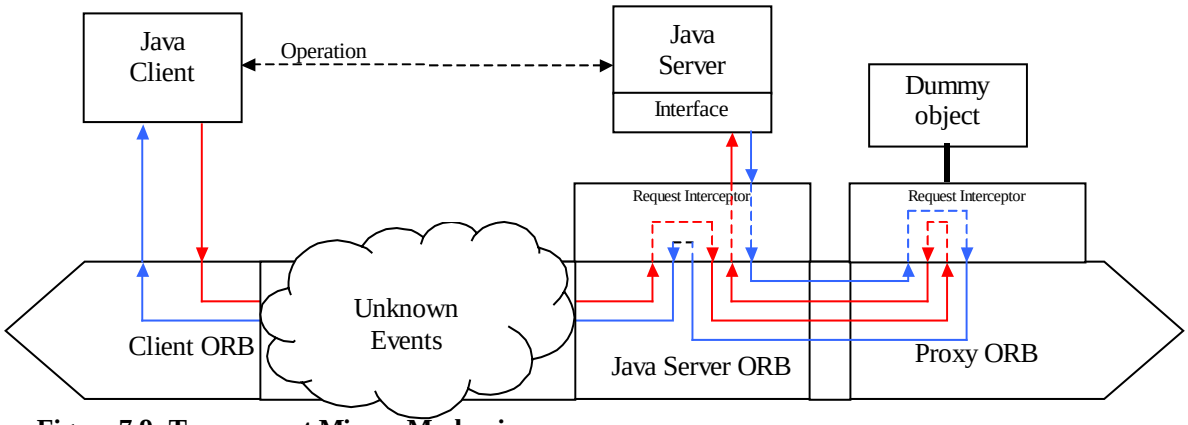


Figure 7.9: Transparent Mirror Mechanism

The disadvantage of this redirection technique is that is a slower than the client proxy forward technique. This is because the target request interceptors cannot use the permanent forwarding technique applied with the transparent proxy technique. Instead, the request interceptor on the target Orb must use per request forwarding, which is a lot slower than permanent forwarding.

7.2 Tracing Mechanism Deployment

In this paragraph, we explain how we could to deploy the tracing mechanism. Although we will not give a full design for deployment of the tracing framework, we will discuss several techniques for deployment of request interceptors.

We differentiate tree technique for deploying portable interceptors:

- *Parameters*: By calling the Orb executable with a portable interceptor as a parameter. The advantage of this method of deployment is that it requires no modifications to the component source code and is therefore fully transparent. All Java Orbs are able to use parameters; in fact, it is the only way Java Orbs can install interceptors (for details see appendix L). Although this installation technique is proprietary, most vendors have realised the significance of this problems and support some the installation of interceptors by parameters in their Orb. To solve the portability problems when switching vendors, the tracing framework could maintain a single configuration for each Orb vendor containing the proprietary usage of the Orb parameters.
- *Statically*: By registering the interceptor in the Orb before initialisation. The current Corba specification only allows the installation of interceptors by statically registering an interceptor in the Orb before it becomes initialised. The Problem of statically registering interceptors is that it is requires the manually modification the part that initialises the Orbs. Normally this means, programmers must either change all components/object source code to allow the Tracer to intercept its communication, or modify the Orbs source code itself by registering the interceptors just before the Orb calls it final initialisation method.
- *Dynamically*: By default, compiling a Corba object with a request interceptor requires the availability of the implementation of request interceptor. As an alternative, we could use a dynamic registration technique that allows programmers/testers to install interceptors on a Orb without recompiling the Corba object source code. By packaging the registration code and interceptor implementation code into a DLL file, the registration code can be executed at run time. With the C++ function `DLLOpen` and `DLLSim`, a generic initialisation function can load and execute the interceptor registration code just before Orb initialisation. Figure 7.10, illustrates (in pseudo code) how an interceptor logging service, packaged as a plug-in, is registered dynamically into the Orb. The advantage of this mechanism is that it allows the transparent registrations of request interceptors without using propriety Orb parameters initialisation. Although this technique is perfect for components that are build in house, fact remains that it still requires the instrumentation of the component source code with a single initialisation method call and will therefore never be fully transparent until Corba specifies a standardised mechanism for registering interceptors transparently.

Although interceptor installation by parameters is proprietary solution, it is still the best method for installing our Tracers. The main reason is that installation by parameters it currently is the only technique, which offers complete transparency. The second reason is that installation by parameters also allows control over the order of the interceptors which is important to prevent forwarding exception problems.

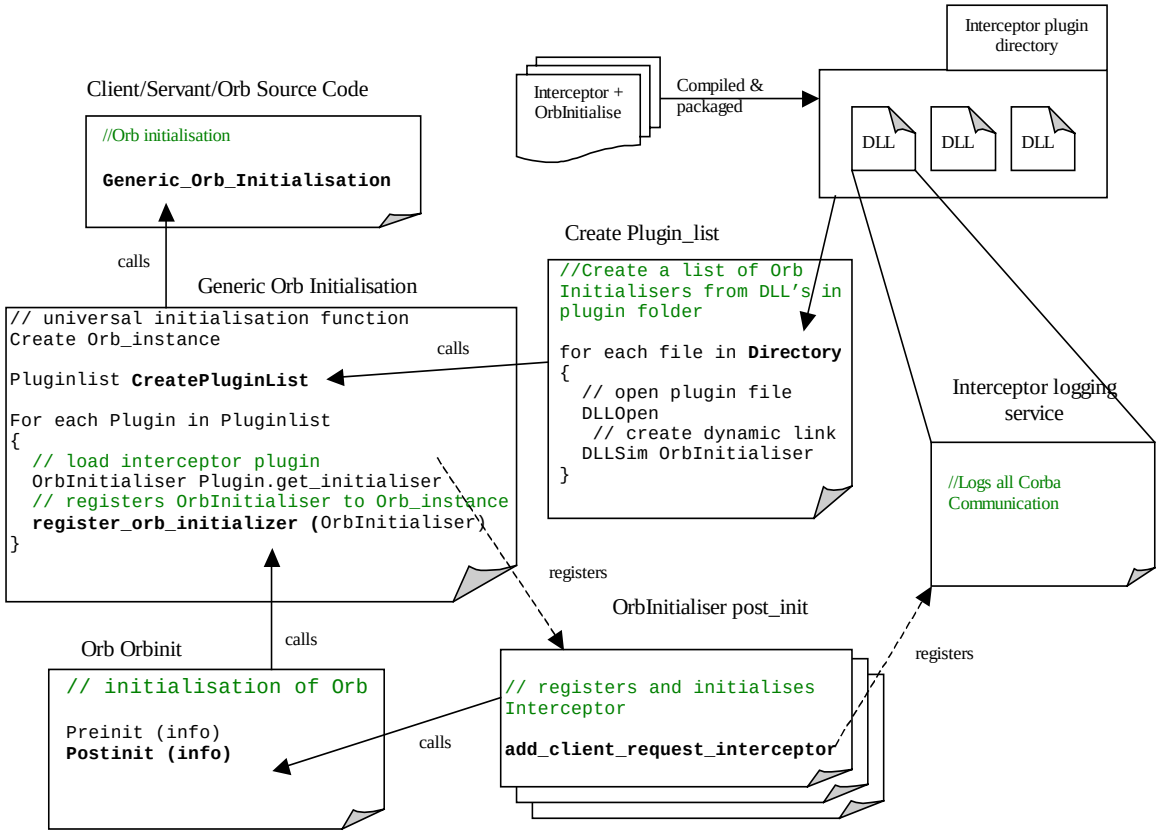


Figure 7.10: Dynamic Interceptor Installation Mechanism

7.3 Tracing Mechanism Management

Because our tracing Interceptors contains profiles, our tracing framework must be to modify the active profile settings. However as already explained, Corba does not support any interface which could be used for managing installed request interceptor.

There are several techniques to consider for managing the tracing interceptors:

- Call-back mechanism. Instead of attempting to manage the Tracer from the outside, The Trace could be made responsible for keeping its profile settings up to date. The tracer could use Corba call-back mechanism to transfer management data between tracing interceptors and Collector component. The management data would update its profile data after it received the reply received from the Collector. Although the Collector component is located on the same host, this would still introduce an extra latency. This is because the interceptor must wait until the reply message from Collector component returns. Also because the reply message will not contain any new management most of the time, will therefore be a waste latency and resources.
- Event Filter mechanism. The Corba notification service standard defines event filters. These filters encapsulate the request of a consumer application for particular events. Because event filters are propagated from consumers through the notification service, to the source of the events. Tracers can use the event filters for retrieving management data from the Collector without introducing a large overhead. However, the information in the filter is limited to a single format.
- Actor Piggyback mechanism. By using the Actor and Corba piggyback mechanism, management data can be send to all used component transparently (see figure 7.11). This information flow would have to be initiated by the first interceptor activated during invocation chain through a Corba application. For every time new management data becomes available at the Collector, it would update the profile setting of the Actor component. The interceptor installed on the Actor orb would then receive the management data and send a copy of the new management data along the first request message send to another target. The main advantage of this technique is that there is no additional latency overhead during the execution of the trace interceptor. A disadvantage of piggybacking is that extra management context data send along message temporarily increase the size of the message, degrading performance. However, because management data will rarely change, the extra overhead will only effect during the change of management data.

In all techniques, the Collector Component is used to locally buffer management data at the host. In order for the Collector to keep its management data up to date, the Collector could use Corba publish-subscribe mechanism,

to received update management data. The Processor component, which is also used for the processing and storage of retrieved, tracing data, could be fitted with an event source, publishing management data events.

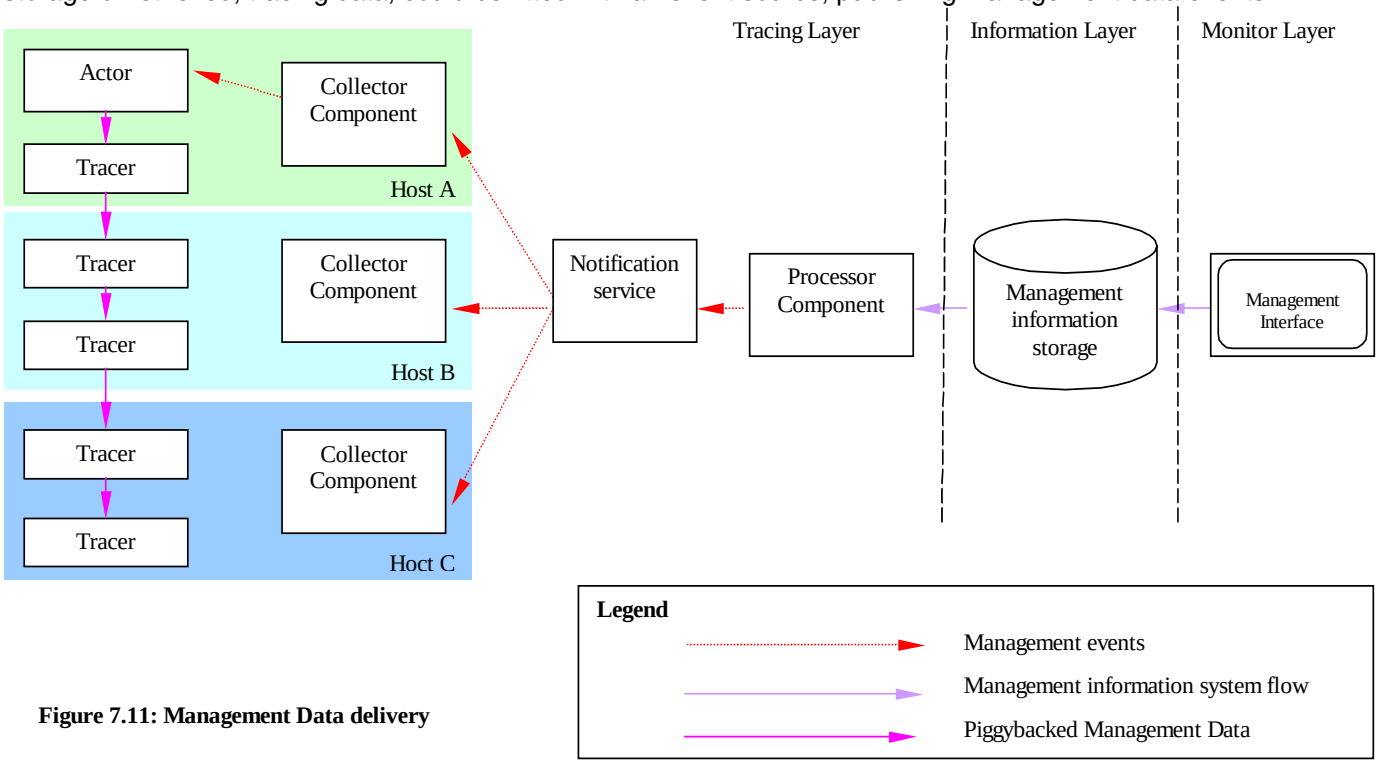


Figure 7.11: Management Data delivery

7.4 Tracing Mechanism Processing

The tracing data intercepted by the Tracers must somehow be communicated and processed through the tracing layer to their eventual destination in the information system. The Processing of intercepted Corba tracing data is achieved by the collaboration of the Collector and the Processor. In this paragraph, we describe how the Collect and Processor Component can be implemented using Corba technology.

In paragraph 4.2.4 we compared several Corba communication techniques. The best communication technique, which is most suited for transmitting communication between Collector and Processor is event based communication. The reason is that event communication is easier for distribution of tracing data and because it is faster. Although there are several ways to realise Collectors and Processes with event based communication, we will limit ourselves to describing the most interesting technologies, which is Telecom Logging Service. Telecom Logging services service has many build in facilities, which makes the implementation of a lot easier. In figure 7.12 illustrates how the Telecom logging service can be using to implement in the construction of the tracing mechanism.

There are several reasons why the telecom services are useful for implementation:

- *Storage*: The Telecom Log service provides all mechanism used for storing and retrieving tracing events. The Collector can use this service for the temporally storage raw intercepted tracing data and the Processor can use it to pertinently store intercepted tracing data in generic trace information system
- *Forwarding.* The Collector component can use the forwarding capability of the Telecom Log Service to directly forward tracing event received by Tracer component to the Processor Component. This increases the performance of the framework simplifies the tracing layer implementation.
- *Filtering.* Both Collector and Processor can make use of the filtering capabilities of the notification services derived by the Log Service. The filter can be set to only allow event trough in which the Collector or the Processor are interested in.
- *Communication*. The Notification service, which derived by the Log services, simplifies the event communication as there is no need for packing intercepted tracing information into an Any at the sending begin or unpacking at receiving end.

For more details about Telecom Logging Service, see appendix C

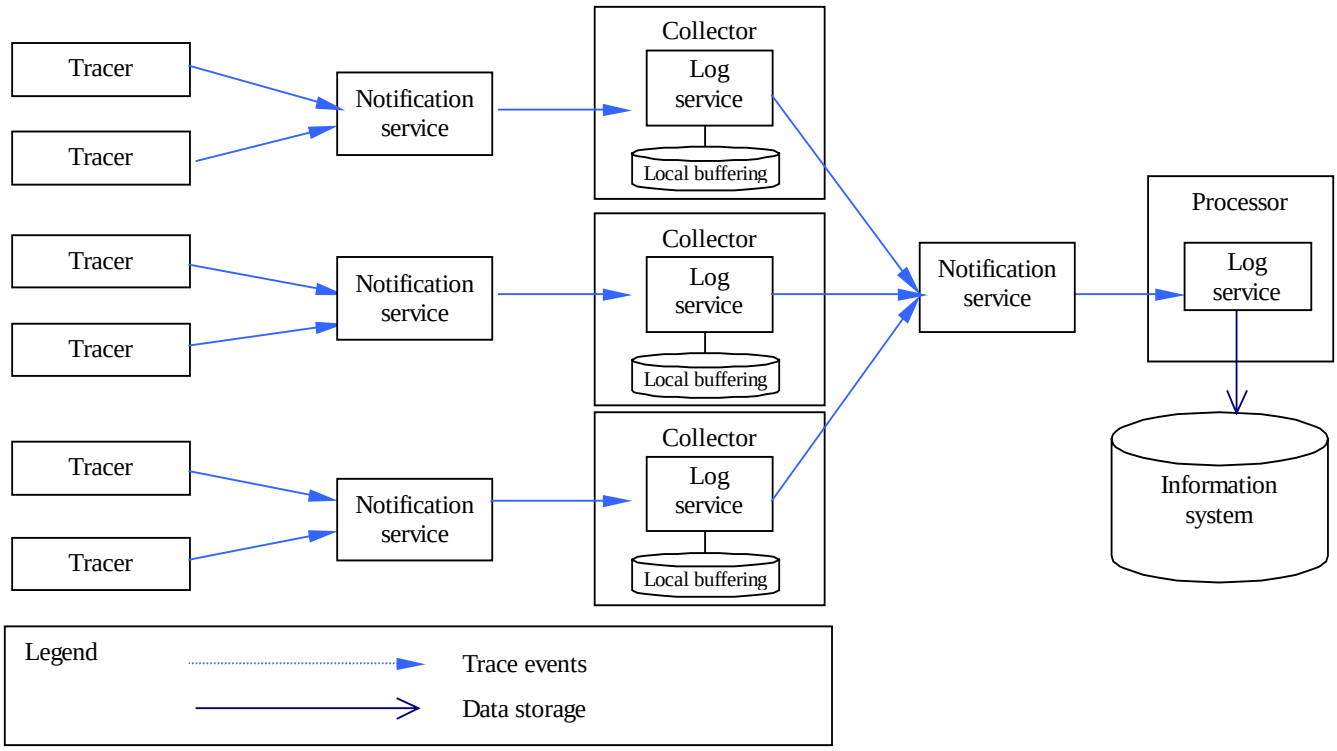


Figure 7.12: Using Telecom logging service for Tracing Framework.

7.5 Evaluation

Although we were unable to implement our transparent tracing mechanism, we will evaluate problems and value of our transparent tracing mechanism. We will evaluate at the advantages of our tracing framework, and then the disadvantages.

The tracing mechanism has the following advantages:

- High compatibility. The techniques used by our tracing mechanism are compatible with most Orb implementations.
- Language transparency. The tracing framework will be able to retrieve all tracing data written in all programming languages, including Java.
- Maximum Component source transparency. Neither the client nor server needs to be instrumented.
- Maximum Middleware source transparency. The middleware does not require any modifications.
- High Invocation transparency. Thanks to the forwarding techniques, a server will receive the exact same messages as the client has send.
- Dynamic Invocation Traceability. The tracing mechanism is able to trace dynamic invocations.
- Simple deployment. By using exclusively dynamic techniques, the same request interceptor implementation can be used to trace all Corba objects in a distributed system.
- Centralised real-time tracing with minimal overhead. Thanks to the event based communication, local buffering and minimum service context data usage, the tracing mechanism will be able to collect tracing at real time data with minimal impact on the distributed system.
- Centralised real-time Management with minimal overhead. The management mechanisms proposed will allows real-time management of tracing mechanism with minimal overhead.

The main disadvantage is that our tracing framework will not be able to trace collocated objects. Although we have shown that a Corba object with own unique Orb is able to collect a tracing data required by a tracing framework, not all Corba Objects have the luxury of a unique Orb for themselves. It is technically possible that multiple Corba Object share the same Orb. In figure 7.13, we can see how multiple components in a CCM container (see also paragraph 4.2.1) share a single Orb. Although we have seen that the request interceptor can derive the identity server is from target_id field located in the GIOP message (see paragraph 7.1.3), the request interceptor support no functionality to determine the identity of client (for details appendix H). Because the Orb in a container is shared by multiple components, we cannot link the usage of the request interceptor with a single client. We can therefore conclude that our tracing mechanism will not be able to uniquely identify Corba clients, which share a single Orb.

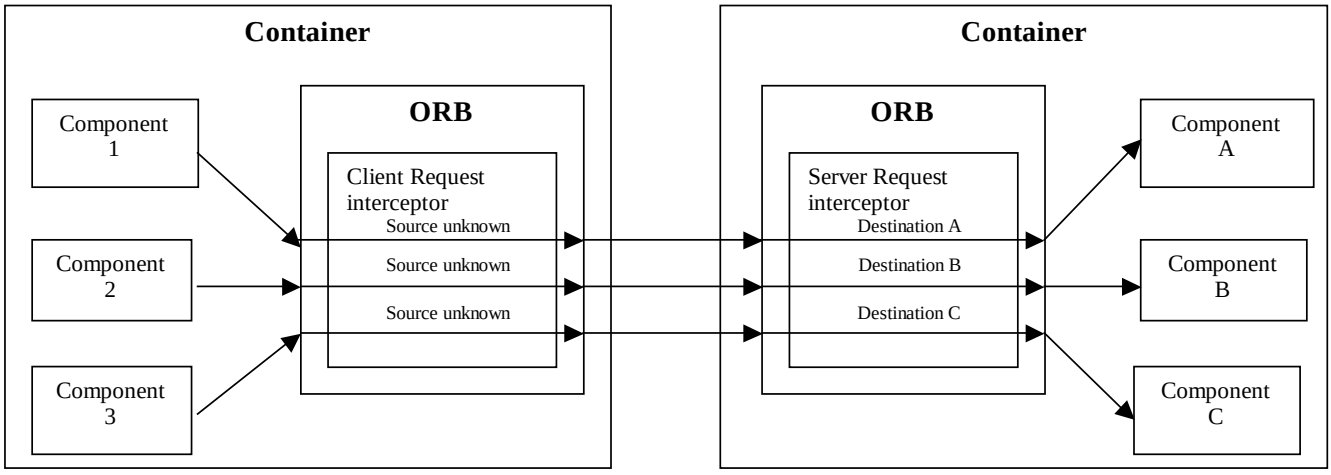


Figure 7.13 Component source identification problem

In order to solve the client identification problem we must some proxy object, which is uniquely identifiable in relation with the client (see figure 7.14). The proxy object, which must have a one-to-one or many-to-one association with the client, collects client context data of the client en transmits it further to the request interceptor after every communication event. The transmission of client context data between proxy object and the request interceptor can be accomplished by sending the missing data to the to the request interceptor with a slot in the PICurrent.

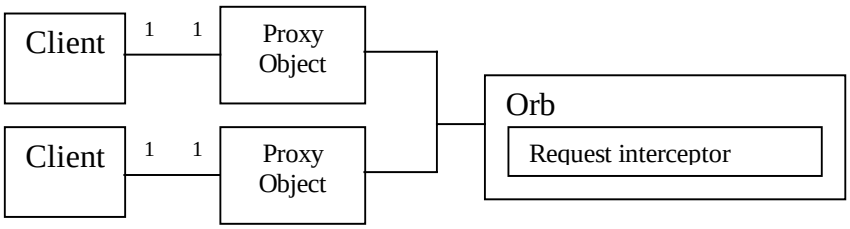


Figure 7.14: cardinality Proxy Object with Client

For the implementation proxy object of the proxy object, we can either use one of the instrumentation techniques described in paragraph 5.1 or the meta programming techniques described in paragraph 5.2. Our preference would be to use a transparent technique like stub/skeleton instrumentation or smart proxies. The reason we did not use them in our tracing mechanism solution is because stub/skeleton instrumentation does not allow tracing of dynamic invocation, and smart proxies is not supported by many Orb vendors, and therefore not portable.

7.6 Tool Development recommendations

Whether or not we should implement a generic tracing framework with our transparent tracing mechanism, we would advise against it. Although the ability to trace Corba application relying exclusively request intercepts in useful in some circumstances (Corba objects from which the source code is unavailable or Corba application that use dynamic connections)

Looking at other Corba tracing tools that took several years, it would be ineffective use of resources to create a tracing framework from scratch. In chapter 6, we have seen several tools that are already offer adequate tracing framework and which have great potential to be fitted with additional tracing functionality. Although they might not be fully transparent and interoperable with multiple middleware solutions, they fulfil the most import functionality, which is to trace regular Corba applications.

DSC Toolkit and Corba Trace could both adapted with new tracing functionality either by Thales themselves or in cooperation with the toolkit developers (Lucent). Both tools for could be made more transparent by using other transparent tracing mechanisms. Once TAO Smart Proxies or pluggable protocol become standardised, they could be used to optionally replace a tools existing trace data interception technique by the more effective or transparent data interception technique.

8 Conclusion

Tracing currently does not receive the attention it deserves. The OMG Corba specification documents do not mention the words monitoring/tracing anywhere and neither is there any OMG special interest group investigating the problem. Instead, the OMG Corba community is mostly focused on adding additional middleware functionality like fault tolerance and security services. Although the OMG attempts to make the development of distributed systems al tot easier with their introduction of the new CCM specification, they neglect to specify any services that can help testers to trace them properly.

This lack of interest from the Corba community is also reflected by market for Corba testing tools. The market for tracing tools is bad to almost non-existent. There are not any established Corba testing tools available, which allow the tester to trace a Corba application at the component level. Most commercial Corba testing tools available are limited to a interface testing tool and use proprietary solutions, which only support one specific Vendor Corba middleware solution.

The only established commercially available Corba testing tool with relevant tracing functionality, Silk Observer, died quietly due to lack of sufficient interest from the industry. Although Silk Observer successor, Silk Performer Component Edition, currently does not support Corba, Segue Software roadmap includes plans to extend their tool with future Corba support. However, it remains to be seen whether it will be a generic Corba testing tool or becomes a proprietary tool just like its predecessor. Silk Observer tracing mechanism depended on propriety Orbix message filters renders their tool unsuitable for Thales, which uses Orb implementations that are unsupported by Silk Observer.

Fortunately, a new tool has become available on the market. Lucent Technologies, the developer of the DSC toolkit, has recently decided to offer their product under licence. Lucent DSC Toolkit has shown in our tools evaluation to offer the best testing solution for Thales Corba testing problem. Although the DSC Testing Tool was originally made for a proprietary CCM middleware solution, it can be made compatible with most Orb implementations. The toolkit offers user-friendly scriptable actor/reactor utility, which allows the construction of a controllable testing environment, is flexible enough to evaluate any test case. The DSC tracing framework contains a powerful monitor tool, which allows both online and offline analysis of intercepted tracing datae. The tracing framework automatically retrieves trace data intercepted by the Corba request interceptor and instrumented stub/skeletons without causing too much overhead to the system. Because the DSC Toolkit tracing mechanism is based on the collaboration of two best interoperable and transparent trace interception techniques available, the resulting framework is a transparent and interoperable tracing solution. Although the toolkit current version only fully supports Java Orbs, Lucent Technologies has shown us a workable C++ prototype version of their tracing tool.

Another tool, which possesses some of qualities of a good tracing framework, is the freely available Corba Trace tool. Although the tools does not support advanced framework features like online tracing and an integrated framework, it is open source, it can intercept Corba communication and generate a sequence diagram in multiple formats. Note however, that the tracing mechanism uses an Adapter instrumentation technique, which must be initialised with an object reference and Orb. A consequence is that the Adapter must be instrumented in the Corba object source code or integrated in CCM Component home and CCM Container. Because the development documentation is available, it could be used for the construction of a tracing framework. Although Thales could integrate the tool into PERCO, the tool would still not provide sufficient functionality to satisfy Thales testing requirements.

Because the market was quite disappointing in the number of mature Corba tracing utilities, we also investigated the possibility of developing a tracing framework ourselves. We soon discovered that traditional debugging techniques would not work for testing distributed Corba Applications. We therefore analysed several Corba trace data interception techniques and compared them based the characteristics important for creating a good tracing mechanism. One particular group of these techniques, the usage of meta-programming mechanisms, promised to be most preferable solution because they allow the modification the middleware behaviour without sacrificing transparency. Although TAO Smart Proxy and Pluggable Protocol framework are very promising tracing technologies for the near future, the Portable Request Interceptor is currently the only transparent meta-programming mechanism that is supported by most Orb implementations. Although Request Interceptors were never intended for tracing, they offer many features that are useful for development of a transparent tracing mechanism. Two of the most interesting tracing features of the Portable Interceptor is the ability to transparently send service context information along with normal messages and its ability transparently intercepts Corba message from both client and server side. Message interception allows a tracing mechanism to collect contents data while service context data allows a tracing mechanism to sort messages chronologically. Unfortunately, the Portable Interceptor also exhibits many limitations that become a problem when using them as a main tracing mechanism. One of these problems is inability to access the contents of a message on a Java Orb. We have

shown that by forwarding the message to a C++ request interceptor, which can access the message contents before forwarding it to its original target, we partially solve the Java problem. Unfortunately, this technique also introduces new problems that can make the tracing framework incompatible with other Corba services, which use Request Interceptors.

Although we have shown that we can push the Corba technology to create a transparent tracing mechanism based solely on the request interceptors, its application remains limited by its main flaw. The request interceptor main flaw is inability to detect the source of a message in an environment where multiple clients share a single Orb. The reason why interceptors cannot detect a client source is that a Corba message simply does not contain any information that can be used to derive a client address. That is because Corba callback mechanism is based on the lower network protocol, which contains a reply network (IP/port) address and an opaque message ID field, which only has meaning to client Orb.

There is however light on the other side of the tunnel. The Smart Proxy meta-programming mechanism, which allows access to the same tracing data as the stub/skeleton instrumentation technique could fill the information gap that the request interceptor lacked. Another meta-programming mechanism, the Pluggable Protocol has the potential to completely replace a request interceptor because it allows access to the same tracing as middleware instrumentation, which is practically all data. Further study should be done one their effective usage as a tracing mechanism. Both meta-programming mechanism are expected become a standard specification just like Portable Interceptors. Once they become standardised by the OMG and supported by implemented by Orb vendors, they allows the construction of a portable, fully transparent tracing mechanism. However, looking back at the development of the Portable Interceptors, which took many years before anything useful became available, the same history likely to repeat for the Smart Proxy specification.

We also introduced a high level design for a generic tracing framework, which can trace a distributed application crossing multiple middleware boundaries. However, Thales needs a good tracing tool soon and simply does not have the luxury to wait until a new tracing tool is developed. The costs of developing a new tracing framework far outweigh the advantages of building tracing framework scratch when there are already good tracing framework available that can fulfil most of Thales testing requirements. If Thales want to take the tracing seriously, we recommend licensing Lucent DSC Toolkit and use it for all Corba testing. In our view, Lucent Technologies has proven being the leading authority on Corba tracing tooling. Although Thales would be one of the first companies to licence the DSC toolkit, we are confidant that with support of Lucent technologies, Thales will be able to integrate the DSC toolkit in their testing strategy in no time.

Bibliography

[CBSEbook] Component Based Software Engendering

[Instant] Robert Orfali, Dan Harkey, Jeri Edwards, Instant Corba, 1997, ISBN 0-417-18333-4

[Reference] Alan Pope, *The Corba reference Guide, Understanding the Common Object Request Broker Architecture*, 1997, ISBN 0-201-63386-8

[Bill] Bill Councill, George T.Heineman, *Definition of Component and its Elements*, Chapter 1, located in *Component-Based Software Engineering, Putting the pieces together,* George T.Heineman, William T.Counsill

[Whitehead] Katharine Whitehead, *Component-based Development, principles, and planning for Business Systems*, ISBN 0-201-6728-5, 2002

[Schmidt] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrency and Distributed Objects, Volume 2*. New York, NY: Wiley & Sons, 2000.

[MCITT] *The Manufacture's Corba Interface Testing Toolkit*, http://www.mel.nis.gov/msidstaff/flater/mcitt/

[DSCDoc] Documentation DSC 3.2

[Corba Trace] http://corbatrace.tuxfamily.org/

[Corba 2.4] Object Management Group, The Common Object Request Broker: Architecture and Specification, 2.4 ed., Oct. 2000

[IDLScript] OMG, Corba Scripting Language, Specification, February 2003, Version 1.1, formal/03-02-01

[Proposal] OMG portable interceptors request for proposals, ftp://ftp.omg.org/pub/docs/orbos/98-09-11.

[Revised] Portable interceptors joint revised submission, ftp://ftp.omg.omg/pub/docs/orbos/99-12-02, 1999.

[Jerry] Jerry Gao, Ph.D. *Testing Component-Based Software,* San Jose, State University, One Washington Square, San Jose, CA 95192-0180, gaojerry@email.sjsu.edu

[OMG2000] Object Magement Group, *Telecom Log Sevice Specification*, Version 1.0 January 2000, OMG Headquaters, 250 First Anenue, Suite 201, Needham, MA 02494 USA, Tel: +1-781-444-0404, Fax: +1-781-444-0320, pubs@omg.org, http://www.omg.org

[Priya] Priya Narasimhan, Louise E. Moser, P.M. Melliar-Smith, University of Carlifornia, Santa Barbara, *Using interceptors to Enhance Corba*, 0018/9162/99/$10.00 © 1999 IEEE

[Szyperski] Szyperski, 1998; Brown and Wallnau, 1998; Gartner Group, Butler Group, 1998]

[Testability] Roy S. Freedman, "Testability of Software Components" IEEE Transactions on Software Engenering. Vol. 17, No.6, June 1991

[Gamma] E.Gamma, R.Helm, R.Johnson, and J.Vlissides*, Design Pattern: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995

[Nanbour] Nanbor Wang, Kirthaka Parameswaran, Dougles Schmidt, Ossama Othman, *The Design and Performance of Meta-Programming Mechanisms for Object Request Broker Middleware, kirthika@cs.wustl.edu, nanbor@cs.wustl.edu* Department of Computer Science Wachington University, St.Louis, schmidt@uci.edu, ossama@uci.edu Electrical& Computer Engenering University of California, Ivrine

[Zimmerman] C.Zimmermann, "Metalevels, MOPs and What the Fuzz is All About", on Advances in Object-Oriented Metalevel Architechtures and Reflection (C.Zimmerman, ed), Boca Raton, FL:CRC Press, 1996)

[Mann] Zoltan Adam Mann and Karoly Kondorosi*, Tracing system-level communication in distributed systems*, Budapest University of Technology and Economics Department of Control Engenering and Information Technology, H-111 Budapest, Magyar Tudosok koruta 2, Hungary E-mail mann@iii.bme.hu, kondor@iii.bme.hu, Phone: +36 1 4631425, Fax: +36 1 4632204, Copyright 2000 John Wisly & Sons, Ltd.

[ExtTrans] Object Management Group, Extensible Transport Framework for Realtime Corba, Request for Proposal. Object Management Group, OMG Document orbos/2000-09-12 ed. Feb. 2000

[Gamma] E.Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994

[Kuhns] Fred Kuhns, Carlos O'Ryan, Douglas C. Schmidt, Ossama Othman, and Jeff Parsons, *The Design and Performance of a Pluggable Protocols Framework for Object Request Broker Middleware,* fredk@cs.wustl.edu, coryan@cs.wustl.edu, schmidt@cs.wustl.edu, othman@cs.wustl.edu, parsons@cs.wustl.edu Department of Computer Science, Washington University St. Louis, MO 63130, USA

[Scallan] Todd Scallan, *Monitoring and Diagnostics of Corba Systems*, Demystifying the Corba communication bus to enable 'distributed debugging', June 2000, Java Developers Journal.com

[Rosenblum] David S.Rossenblum, Adequate Testing of Component Based Software, Department of Information and Computer Science, University of Carlifornia, Irvine, CA 92697-3425, +1 714.824.6534, dsr@ici.uci.edu/~dsr/

[MDA] Miller, Joaquin; Mukerji, Jishnu, ed. (2003), *MDA Guide Version 1.0.1*, Object Management Group, document number omg/2003-06-01

[Schmidt] D.C. Schmidt, M.Stal, H.Rohnert, and F.Buschmann, *Pattern-Orient Software Architecture*: Patterns for Concurrent and Networked Objects, Volume 2. New York, NY: Wiley & Sons, 2000

[Philip] Mathieu Vadet* ó Philippe Merle**, Conteneurs adaptables: du modele l'architecture

[Douglas] Douglas C. Schmidt, Carlos O'Ryan, and Ossama Othman Fred Kuhns and Jeff Parsons
Applying Patterns to Develop a Pluggable Protocols Framework for ORB Middleware

[Koster] Rainer Koster and Thorsen Kramp, Loadable Smart Proxies and Native-Code Shipping for Corba, Distributed System Groups, Dept. of Computer Science, University of Kaiserslauteren, P.O.Box 3049, Kaiserslauteren, Germany

Appendixes

In the appendixes we describe some detailed information about Corba in which has relevance to our Thesis but were too technical for out Thesis. Although they do not have to be read, they can help the reader get a better insight in Tracing Corba technology.

.

Appendix A: IOR

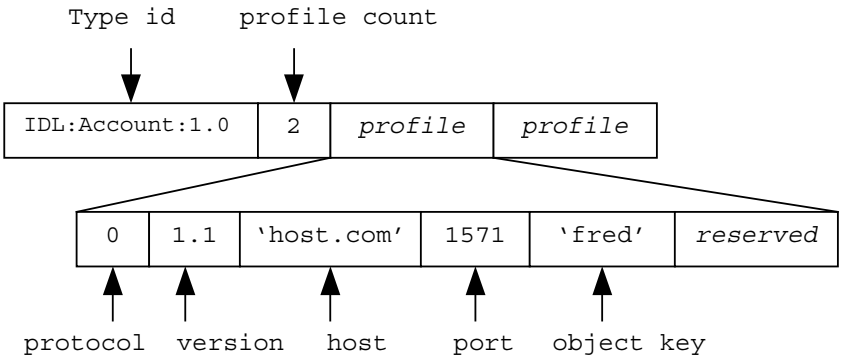
In this paragraph, we will take a closer look at the Interoperable Object Reference (IOR). Although an IOR is used by an application in the exact same way that an object reference is used, they are not exactly the same. While the IOR is a specialised object reference that can be universally understood by all Corba applications, an object reference only has meaning within the current Corba application. An object reference identifies one instance of an object and associates one or more paths by which that object can be accessed. An object reference therefore allows an application to establish a connection and make remote method calls on a Corba object.

The same object may be located by different object references, *e.g.*, if a server is re-started on a new port or migrated to another host. Likewise, multiple server locations can be referenced by one IOR, *e.g.*, if a server has multiple network interfaces connecting it to distinct networks, there may be multiple network addresses. However, because every Corba applications can convert an IOR string to an object reference and back from object reference to IOR string, they can effectively be used interchangeably.

Once an application used the IOR to establish a connection, the client can transparently access the methods of a remote Corba object via Corba communication protocol, GIOP (see also paragraph 0). Just as a telephone number contains information of the whereabouts of the number, the IOR contains information of the whereabouts of a servant object. The IOR is normally created by the POA when it starts a Corba servant. During creation of the IOR, IOR Interceptors (see also paragraph 0) can be used to modify the information in the IOR.

Because the IOR plays an important role in the messages we intend to trace, we take a closer look at what information is contained in the IOR format. An IOR contains one or more profiles. Profiles are essentially a one-way communication mechanism that allows a server to tell a client which methods of communications are available. A profile is a reference to a server location and provides an opaque, protocol-specific representation of an object location. Profiles can be used to annotate the server location with QoS information, such as the priority of the thread serving each endpoint or redundant addresses to increase fault tolerance.

Each profile contains all the information required to dispatch request messages and must be stand-alone. That is, when sending a request, the client is allowed to use information from only one profile. When a client gets an IOR, it tries to bind the request using one of the profiles. If that fails, it goes to the next profile, and so on, until it either succeeds or runs out of profiles.



The figure above illustrates the IOR format:

- An IOR begins with the string type_id which gives the type of the object, equivalent to the name of the IDL interface defining the object
- A `profile_count` specifies the total number of profiles that the IOR Contains.
- An IOR profile starts by indicating which underlying protocol is used. In our case 0 indicates that the TCP protocol will be used, which effectively means the IIOP communication format will be used.
- This is followed by field reserved for the Version of the protocol, which consists of a major and a minor version number.
- The next two fields provide the address appropriate for transport protocol needed to establish communication with the remote server. In our case, a host IP address and port number.
- The `object_key` is a field, which is used by the remote server to locate the object being accessed. Although a client needs to have a copy of the `object_key`, as far as the client is concerned, the key is just an opaque code which passes to a server in order to identify an object referred to

- There can also be additional fields at the end but these are currently not used and are reserved for future expansion to the protocol.

Typically, the server who hosts the corresponding object creates an IOR. The IOR is then published in order to make it available to other client processes. To assist publication of an IOR, it must be possible to convert it into a alphanumeric string format which is not subject to any conversions when communicated from place to place. For this reason, Corba specifies a standard alphanumeric string format (see IOR string example below).

```
IOR:010000001300000049444c3a546573742f48656c6c6f3a312e3000000100000000000006c0000000
10102cd0f000000706323835312e7369676e61616c00cdfa05cdcd1b00000014010f005253549893d43f
600e0e0000000000010000000100000cd020000000000000008000000001cdcdcd004f415401000000140
0000001cdcdcd0100010000000000009010100000000000
```

As seen in the IOR string above, it consists of the characters “IOR” followed by a series of hexadecimal numbers. Every byte of the original IOR is translated into a two-digit hexadecimal number. Although this string format is simple and resistant to corruption, interpreting the content of the IOR is difficult.

Appendix B: GIOP/IIOP

Because our tracing mechanism must trace Corba message traffic, we take a closer look at Corba message communication protocols, GIOP and IIOP. To allow Orb-to-Orb communication, the Orb uses the General Inter-Orb Protocol (GIOP), which was designed to work directly over any connection-oriented transport protocol like TCP/IP. The GIOP specification provides a general framework for protocols to be built on top of specific transport layers. The Internet Inter Orb Protocol (IIOP) protocol is a special case of the General Inter-Orb Protocol (GIOP), which is built on top of TCP/IP transport layer. IIOP makes it possible for distributed programs written in different programming languages to communicate over the Internet.

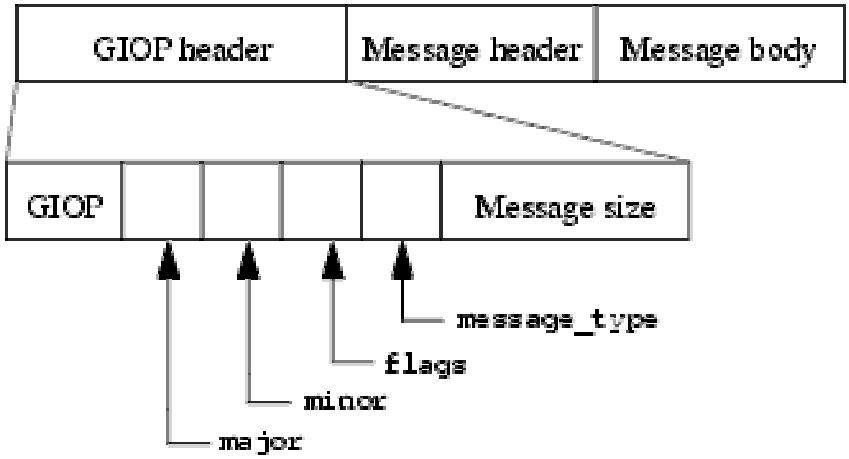
GIOP also specifies a Common Data Representation (CDR) for communications between Orbs. The Common Data Representation (CDR) maps common data types defined in OMG IDL into a flat message representation. This transfer syntax specifies a coding mechanism for all IDL types: including basic types, structured types, object references (in the form of IORs), and pseudo-object types such as TypeCodes. Another feature of CDR is its ability to deal with the different kinds of byte ordering required by different hardware types: both big-endian and little-endian byte ordering is supported. The convention adopted is that the sender of a message sends data using its native byte ordering (and sets a flag in the message header to indicate the ordering used). The receiver of a message is obliged to detect the byte ordering used and carry out any conversion, if it is required. The advantage of this convention is that when both sender and receiver use the same byte ordering, no conversion is required resulting in considerable gain in efficiency.

GIOP specifies seven message format types that cover all Orb request/reply semantics. These message types allow clients to pass invocations to servers and receive replies which can be either normal or indicate some error. Some additional messages are available to help manage the connection. They are:

- *Request Message* allows a client application to invoke an operation on a remote server.
- *Reply Message* is normally sent by a server in response to a client Request message
- *Cancel Request Message* is sent by the client to the server to indicate that the client is no longer interested in receiving a Reply to a particular message.
- *Locate Request Message* can be sent from client to server to probe for the location of a remote object. It is advantageous to send this message before sending a large Request on a connection which has just been opened
- *Locate Reply* Message is sent from server to client in response to a Locate Request message. The sever can answer with UNKNOWN_OBJECT, OBJECT_HERE or OBJECT FORWARD:
- *Close Connection Message* is sent by the server to tell the client that it intends to close the connection
- *Message Error Message* can be sent by the client or by the server Orb. It is used within the IIOP protocol to indicate that the last message received was either corrupted or incorrectly formatted in some way. It consists only of a GIOP header with the message type set to MessageError.

Depending on the message type, a message can consist out of one, two or three parts. All message types consist at least of the GIOP message header (see figure above). The fields in the header can be described as follows:

- The four characters "GIOP" serve to identify the protocol.
- The GIOP version number (major and minor) is used to create the message.
- A flag byte is currently only used to indicate the byte ordering.
- An integer is used to indicate the message type.
- The message size (excluding the GIOP header itself).

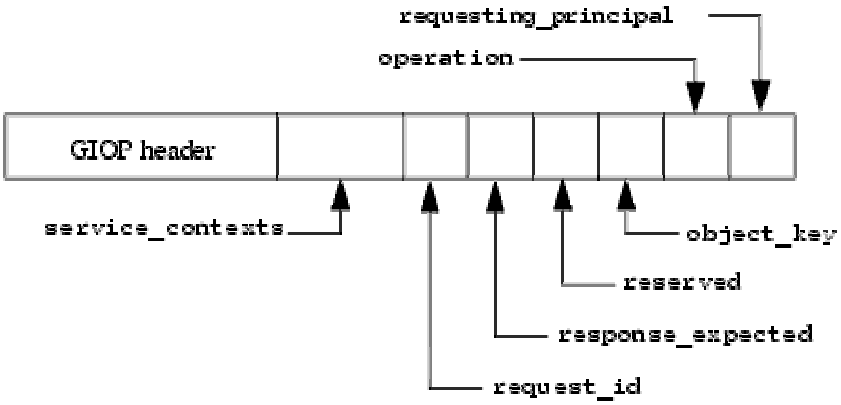


Note that although not illustrated above, the GIOP message format also provides fields known as the Service Context that is used to transfer context information between request interceptors (see also 5.2.1). Because the request and reply messages are of special interest for a tracing mechanism, we will take a closer look at these important message types in following sub paragraphs.

The Orb creates a new request message, every time client does a remote operation on the servant. The client usually waits for a Reply message from the server (unless the operation has been declared to be one-way), which normally contains a return value, or possibly an error condition. Since the request and reply messages are of special interest for a tracing mechanism, we will take a closer look at these important message types in the next two paragraphs. The Request message contains all the information needed for the invocation including the identity of the object, the operation name, and any parameters associated with the operation. Because a Request message is designed specifically to invoke operations declared in an IDL interface, the message format is designed to support all of the syntax that can appear in an IDL operation definition. The message consists of a Request header followed by a Request body. The body of the Request consists essentially of a list of the operation parameters followed by any context strings for the operation. It is possible for the body of the Request to be empty.

An outline of the Request header is shown in the figure below consists of the following fields:

- The service_contexts field allows service specific context-information to be passed along with a Request. Intended for use in conjunction with the Corba services to carry extra information along with the Request[2], the service contexts are not needed in the core specification of Corba.
- The request_id field is used to uniquely identify a Request emanating from a client so that the client can later match a received Reply with its corresponding Request (e.g. the corresponding Reply is tagged with the same request_id).
- The response_expected flag is used to indicate whether the Request is oneway or not. A normal Request has response_expected set equal to TRUE.
- The next field is an array of three bytes reserved for future use.
- The object_key field is used at the server end to identify the object that is being invoked by the client
- The operation field is simply a string giving the name of the operation being invoked.
- The requesting_principal field identifies the user making the request. That is, it is simply the user name of the person running the client.

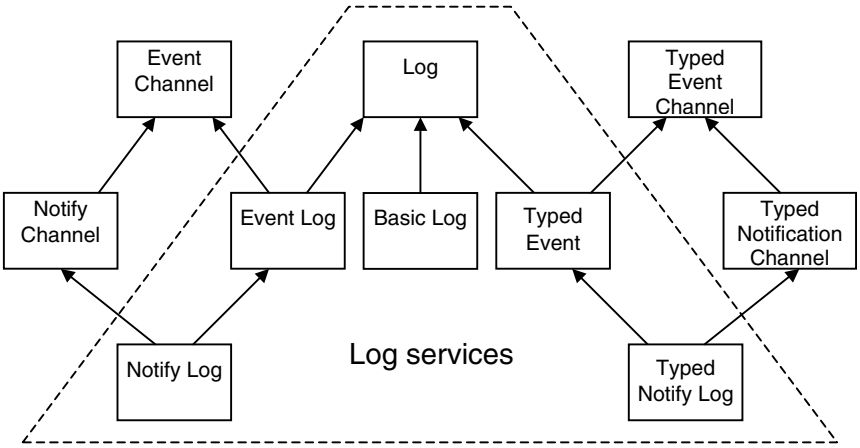


The Reply message consists of a GIOP header followed by a Reply header and a Reply body. The usual intent of a Reply message is to pass back a return value for an operation and to indicate the completion status for the operation. The Reply header does not pass as much information as a Request header and typically consists of the following three fields:

1. The service_context field that is similar to the service context field used by the Request message.
2. The request_id field is used to match this Reply message with the client Request that gave rise to this reply. That is, all replies messages are paired off with their corresponding Request and the request_id is a unique identifier for active messages between Orbs.
3. The reply_status field is used to indicate whether this is a normal Reply or if some error condition occurred in the server. The reply_status is used to toggle between a number of different Reply types so that a Reply message is almost like four messages rolled into one. The possible values for reply_status are NO_EXCEPTION, USER_EXCEPTION, SYSTEM_EXCEPTION or LOCATION_FORWARD

Appendix C: Corba Telecom Logging Services

The Object Management Group (OMG) proposed a specification for a Corba Telecom Logging Service [OMG2000] for logging events in a pure Corba environment. The proposed Application Programming Interface (API) services supports on top of the functionality defined in the ITU-T X.735 specification extra Corba event logging functionality. The Telecom Log Services can be accessed by six different Log objects, which inherit functionality from each other or from other object services (see figure below).



Each log interface object can store events as entries in an individual log repository. This Log repository can store any type of event as long as the underlying storage, one or more files or databases can support their storage. Conceptually an Event/Notification Log object and can be modelled as an Event/Notification Channel (see figure below) which support both the push or pull models of event communication (see event communication). Event/Notify Log objects can operate as an event consumer, event supplier or as an event forwarder. Log objects can also be used to form a complex “log and forward” network. A “log and forward” network could potentially be used to create a testing environment for event communication.

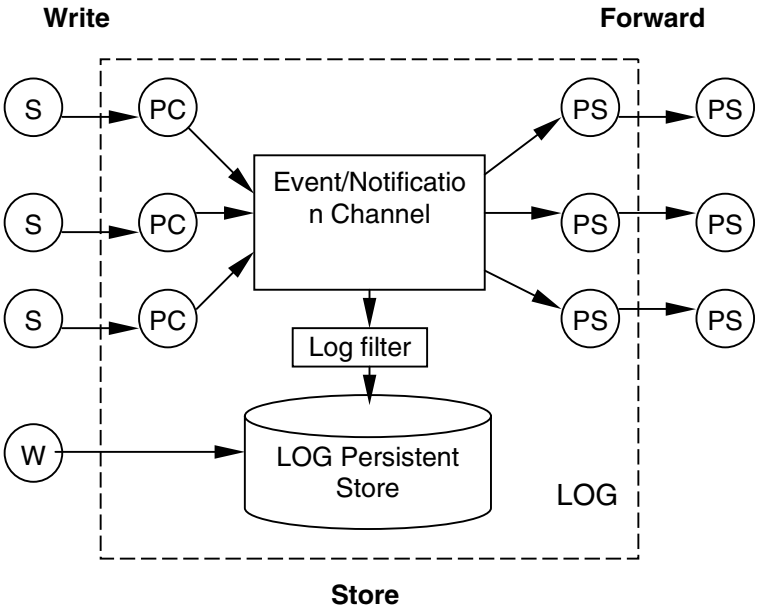


Figure Key:

- S: event/notification supplier
- W: event/notification unaware writer
- PC: proxy consumer
- PS: proxy supplier
- C: event/notification

Appendix D: Corba Scripting Language

The OMG has specified the IDLScript language [IDLScript] designed specifically for testing Corba applications. IDLscript is new general-purpose object-oriented scripting language that allows any user to develop their activities by simply and interactively accessing objects available on the Corba core middleware object, the Orb. Therefore the user is completely free to operate, administrate, configure, connect, create, and delete distributed objects on the Orb.

IDLscript is a true high-level language comprising programming concepts such as structured procedures, modularity, and object-oriented programming. The IDLscript language provides various syntactical constructions such as basic values and types, expressions, assignments, control flow, statements, procedures, classes, modern exception handling and modules. The binding between Corba and IDLscript is achieved through the Dynamic Interface (DII) and the Interface Repository. The DII is used to construct requests at runtime and the Interface Repository is used to check parameters types of requests. Moreover, using the DSI, IDLscript allows one to implement OMG IDL interfaces through scripted objects. IDLscript allows scripts to invoke IDL operations, access IDL attributes of remote Corba objects/components. The interpreter automatically does all type checks and conversions and parameter coercions are automatically done according to IDL signatures. IDLscript provides a simple Java-like exception mechanism that allows one to catch users' defined IDL exceptions and also standard Corba system exceptions. The Dynamic Invocation Interface sends Corba requests.

An IDLscript engine is the mechanism that interprets the IDLscript. It provides three execution modes: the interactive one, the batch one, and the embedded one. In the first mode, users provide their scripts interactively. In the second one, the interpreter loads and executes file scripts allowing batch processing or server implementations. In the last one, the interpreter can be embedded in another program and then interprets strings as scripts. The IDLscript engine allows the introspection of any scripting object. The introspection encompasses object displaying and dynamic attribute, method, and type discovering. All OMG IDL concepts such as basic types, modules, constants, enumeration's, structures, unions, typedefs, sequences, arrays, interfaces, exceptions, TypeCode, and Any types are directly and transparently available to scripts. The integration between IDLscript and the Orb is fully dynamic which means there is no stubs/skeletons generation required. The IDLscript engine discovers OMG IDL specifications through the Interface Repository. When scripts invoke Corba objects, the Dynamic Invocation Interface and the Dynamic Skeleton Interface are internally used to send and receive requests and the IFR is used to check parameter types at runtime. But users never use directly these Corba dynamic mechanisms, they are totally hidden by the scripting engine.

Appendix E: Tcl Scripting Language

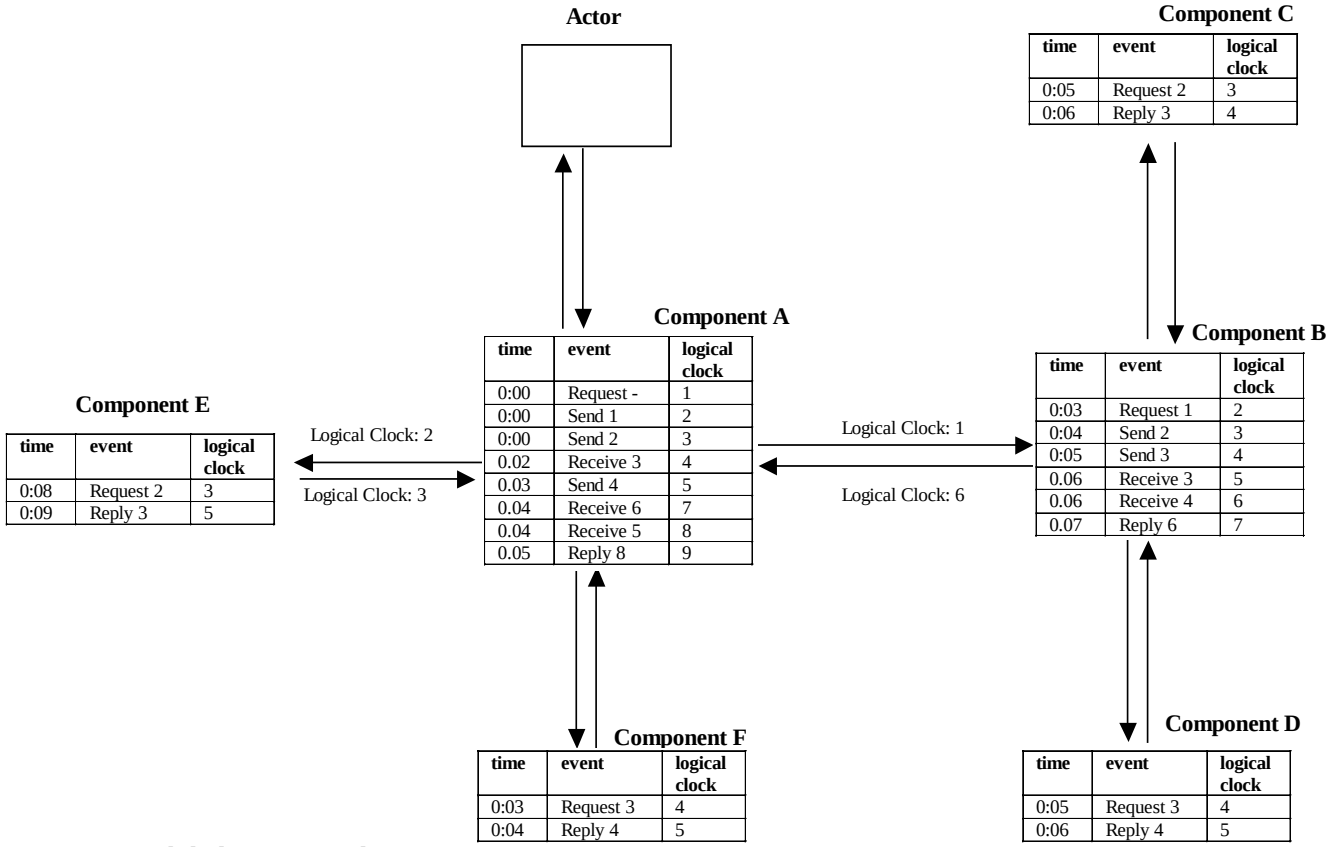
Tcl scripting is a popular open source scripting languages, which enjoys a large availability of development tools. The Tcl Scripting Language simplifies the access and the use of computer system resources like files and processes in the context of an operating system shell, relational database query requests in the context of SQL, and graphic widgets in the context of Tcl/Tk. Combat is an extension to the Tcl scripting language that allows accessing and providing Corba objects at the script level. Combat allows Tcl scripts to access Corba services or become Corba servers themselves. The interpreted language Tcl can therefore serve as a prototype scripting language for implementing functional test on a software component. By constructing a small set of scripts, a tester could validate a system component in a relatively short time. Because Tcl interpreted language can be executed without recompilation, its code can even be send across a network. This would allow a Tcl script to do remote interactive modifications or send an extension to another server. A disadvantage of interpreted languages is that they are slower than their equivalent code in a static programming language. The speed decrease might be acceptable for temporary prototype, test reactor, or low priority components but a programmer is advised to reprogram a component in a statically linked component for deployment.

There are two versions of Combat available.

- Combat/C++, does not itself include an Orb, but is a glue package that builds upon an existing Orb C++ Orb such as Mico or Orbacus. Therefore, Combat/C++ inherits your Orbs properties such as configuration options and protocol support.
- Combat/Tcl also includes an Orb written in pure Tcl. By not depending on any compiled code, this version works on all platforms where Tcl is available, and can be used where compiled code is not acceptable, such as in the Tcl browser plug-in. Naturally, packaging and deployment is much easier than with compiled code.

Appendix F: Logical Clock Limitations

Although Logical Clock is ideal for chronologically ordering communication events between two distributed processes, the mechanism will start lacking in distributed systems where component instances maintain communication links with multiple components at the same time. Figure N illustrates a possible communication scenario where this problem manifests. Although a logical clock value will allow the tracing framework to order events between two communicating components, logical clock values that are not connected by a message cannot be logically ordered. For example, there is no telling whether logical clock event 3 on Component A was before or after logical clock event 3 on Component B. However, sometimes we can estimate it by using the available timing data. Let us assume that physical clock on Component A is the real clock time. Notice that Component A send a request message (with Logical clock value 1) at time 0:00 and receives the reply message (containing logical clock value 6) at time 0.04. Now compare it the times on component B. Component B receives Logical Clock value 1 at time 0.04 and returns Logical Clock 6 at time 0:07. Notice that that both component A and Component B require 4 seconds between Logical clock 1 and Logical clock 2. We can clearly see now that there is an offset of 3 seconds. Now we can use this knowledge to sort logical clock 3. Because 0.05 – 0.03 > 0.00 we can conclude that logical clock event 3 at Component B happened approximately 2 seconds after logical clock event 3 at Component B



time	event	logical clock
0:05	Request 2	3
0:06	Reply 3	4

time	event	logical clock
0:00	Request -	1
0:00	Send 1	2
0:00	Send 2	3
0.02	Receive 3	4
0.03	Send 4	5
0.04	Receive 6	7
0.04	Receive 5	8
0.05	Reply 8	9

time	event	logical clock
0:03	Request 1	2
0:04	Send 2	3
0:05	Send 3	4
0.06	Receive 3	5
0.06	Receive 4	6
0.07	Reply 6	7

time	event	logical clock
0:08	Request 2	3
0:09	Reply 3	5

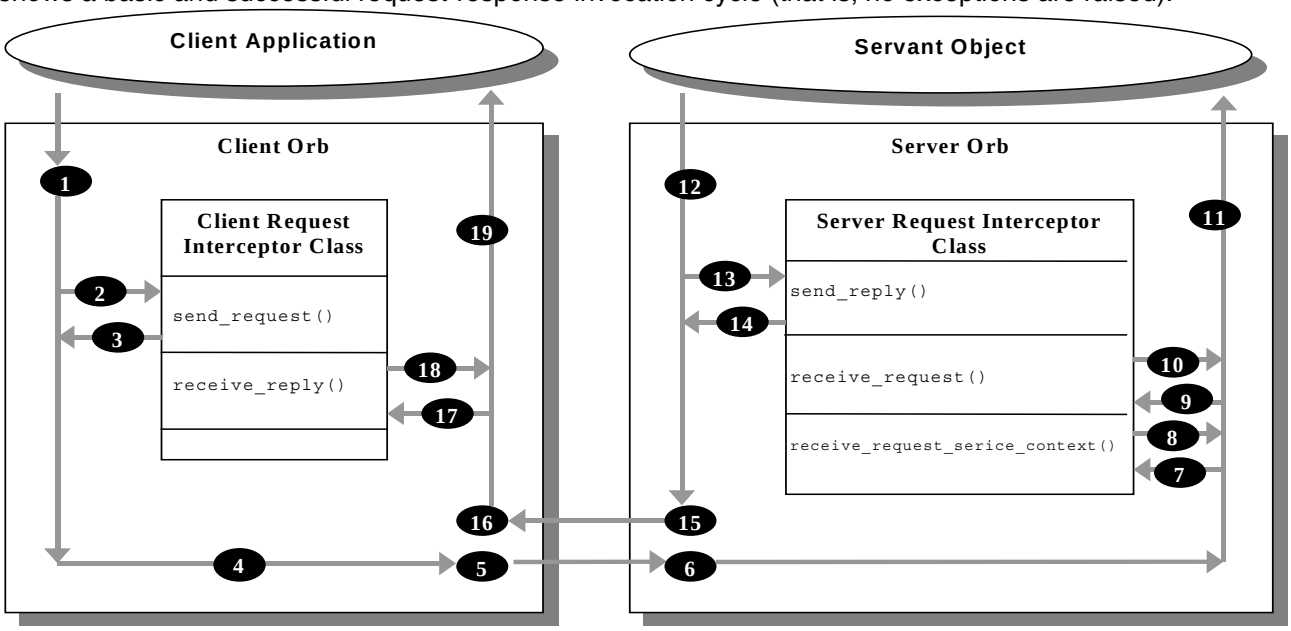
time	event	logical clock
0:03	Request 3	4
0:04	Reply 4	5

time	event	logical clock
0:05	Request 3	4
0:06	Reply 4	5

Figure N: Logical clock usage example

Appendix G: Request & Reply Flow

Although there are seven message types in GIOP, we focus our attention on two of the most interesting message types for tracing purposes, which are the request and reply message. The picture below illustrates the information flow through a client and server Orb both with active request interceptor. A client application starts an invocation by sending a request message and receives a reply message back from the servant. This figure shows a basic and successful request-response invocation cycle (that is, no exceptions are raised).



In figure above, the following events take place:

1. The request leaves the client and arrives at the client Orb.
2. The Orb calls the `send_request` method in the `ClientRequestInterceptor` Class implementation with a `RequestInfo` object as in parameter
3. The client request interceptor processes the request and returns to the Orb.
4. If no user exception is returned during execution of the `send_request` method, the request resumes its path toward the target object.
5. The Orb marshals the Client request into a IIOP message and sends it to the server Orb
6. IIOP message arrives at the server Orb and gets unmarshalled into client request
7. The Orb calls the `receive_request_service_context` interface with a `RequestInfo` object which only allows access to the message service context fields (see paragraph 7.4)
8. The `receive_request_service_context` allows the request interceptor to read available service context (see paragraph 7.5) and transfer it to available Portable Interceptors current slots (see paragraph 7.6) before returning to the Orb
9. The Orb calls the `receive_request` interface with a `RequestInfo`, which allows the request interceptor access to all outbound message fields.
10. The server request interceptor processes the request and returns to the Orb
11. If no exception is raised during the execution of the `receive_request` operation, the request resumes its path toward the target object.
12. The servant object processes the request and issues a response.
13. The target-side Orb calls the `send_reply` method in the `ClientRequestInterceptor` Class implementation with a `RequestInfo` object as in parameter
14. The request interceptor processes the response and returns to the Orb.
15. The response is sent to back to the client Orb.
16. The response arrives back at the client Orb
17. The Orb calls the `receive_reply` method in the `ClientRequestInterceptor` Class implementation with a `RequestInfo` object as in parameter
18. The interceptor processes the response and returns to the Orb.
19. The client eventually receives the servant reply message back from the servant

Appendix H: Request Information

When the Orb activates the request interceptor, it receives a RequestInfo object as parameter. Depending on the message types and contents of the intercepted message, the RequestInfo object is populated with a different set of message information.

The RequestInfo object allows the Request Interceptor to get access to the following attributes:

- *Request ID*, does in contrast to the intuition, not specify the source of the message. Instead, it only uniquely identifies an active request reply sequence between two Orb instances. Once a request reply sequence is concluded the Orb might reuse the ID. The request ID can therefore not serve as a unique message identifier in the Corba System.
- *Client/Server Service Contexts* attribute allows the request interceptor to transparently transfer context information between client and servant. The service context field is a new reserved field in the IIOP 1.1 message format (see also appendix B). Service Contexts can be used for out-of-band communication between interceptors installed in different Orb instances. Depending on the interceptor interception point, the client service context can be access, or modified. In the next paragraph, we will explain how Interceptors can use this piggybacking mechanism to transparently sent context information between services.
- *Reference to the target object,* is the effective IOR address to the object that is being invoked by the client. Although this information is normally not available in the GIOP, the interceptor is able to derive this address from the Orb. This information is very useful for tracing services that want to determine the effective Corba Object a client interacts with. Note that the reference on the reply message is the same reference in the request message.
- *Argument* and attributes returns a parameter list containing the input and output arguments on the operation being invoked. Note however that the parameters list is not supported by all Orb implementations. Due to enforced security rules, Java Orb implementations are not able to read the arguments.
- *Method* attribute allows services to determine the name of method that was remotely called on a servant by the client object. The method name is the same name defined in the IDL file. The Combination of method and argument attributes collected on both server request interceptor and client request interceptor would allow a tester to verify the input with output of a object (see figure below).
- *Result* or *Exception* list is the result of a method call on a Cobra object. This information is only available if the interceptor captures a reply message from the servant. The return value is of special interest to tester to verify a methods output with the expected output. Exceptions can give a service an indication what went wrong. Fault tolerant services can use this information to recover from a fault without the client noticing any problems.
- *Tagged components* attribute is a new read-only field in the IIOP message format, which can be modified by IORinterceptors (see also appendix L). They general usage is for establishing secure or fault tolerance connections. Although they cannot be used to determine the identity of a client, the information might be useful for verifying the system fault tolerance.

Note that there is no field that returns a reference to the source of the message. The GIOP message protocol simply does not contain any source ID. This is because Corba communication protocol is mainly focused on establishing and maintaining a between two objects connected by an Orb, which already have network connection on a lower communication level like TCP/IP. The Orb process IP number and port number therefore determine a message return path. Note also that process that can initialise an Orb instance can effectively invoke any Corba object. Clients therefore do not even have to be an object to call remote function.

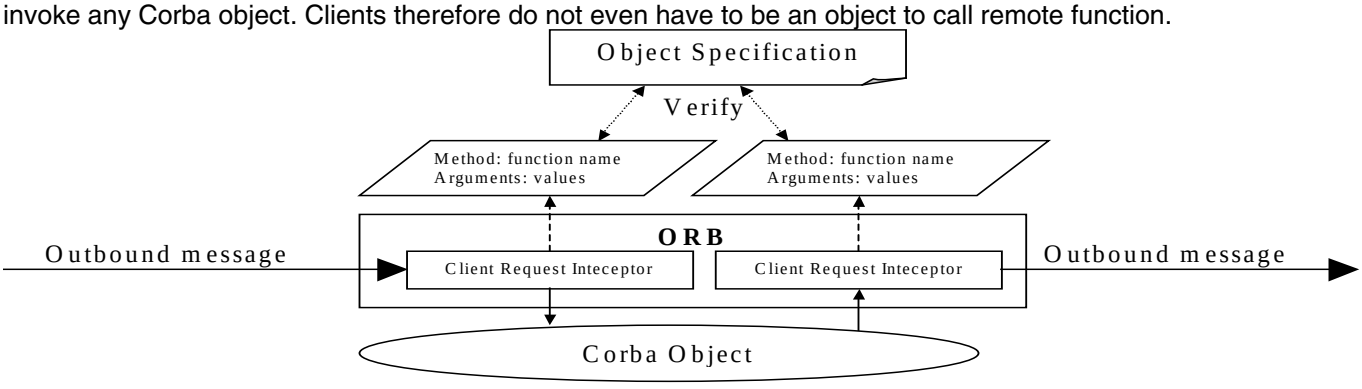


Figure: Verify an object input argument and output arguments

Appendix I: Service Context Data

Service Context Data is data send inside reserved field along normal Corba communication. In order to transfer context data from client to server, there must be at least one client request interceptor installed in the client Orb and server request interceptor installed in the server Orb at the same time. The client interceptor inserts context data into the request message and the server request interceptor retrieves the data from the message. This mechanism could for example be used to transparently send authentication data between client and server security services. On the client side a security service intercepts a request message and adds authentication data to the message, on the server side, after validating the authentication, the security service can either confirm access, by doing nothing or refuse access by raising an exception. The flow of piggybacked information can also be bi-directional. In bi-directional piggybacking, context data both flows from client to servant and back from servant to client. In order to resent the same or modified context data back to the client, the client request interceptor can make use of the Interceptor shared memory mechanism, called PICurrent (see appendix J).

Unfortunately everything comes at a price, so does flowing service context data through distributed system. Due to the extra overhead caused by populating and retrieving the service context and transmitting the larger than normal messages between Orbs, the context data flow will have a negative effect on the overall middleware performance. Depending on the performance of the Orb, and the amount of bytes service context being piggybacked, both message response time and throughput will suffer. The tables below illustrate the increment in latency and decrement in throughput in terms of percentage caused by the interceptor flowing context data in comparison to normal data flow without interceptors. Note that latency table also includes a column where we can see the minimum latency overhead caused by a non-operating request interceptor. A non-operating interceptor does nothing except slowing down an Orb through the default overhead caused by the internal parameter conversions.

	Latency Increase				
Orb	0 bytes	10 bytes	100 bytes	1000 bytes	10.000 bytes
Orbacus v.4	6,36%	13,64%	28,18%	47,23%	145,45%
Orbix 2000	1,39%	9,03%	19,44%	36,83%	91,67%
JacOrb v1.3	10%	32%	50%	69%	171%

	Throughput Decrease			
Orb	10 bytes	100 bytes	1000 bytes	10.000 bytes
Orbacus v.4	15,49%	29,89%	50,32%	149,18%
Orbix 2000	9,81%	15,28%	33,20%	92,24%
JacOrb v1.3	37,54%	57,58%	79,62%	185,71%

Although the above tested Orbs are currently largely outdated, they still give us a good impression what to expect from other Orbs. We can clearly see (in the range 10 to 10000 bytes) that latency approximately doubles after the number of bytes is increased by a factor of ten (e.g. the latency grows at $^{10}\log n$, $10 < n < 10000$, with n the number of bytes in the service context data). A similar trend can also be seen with the Throughput that also drops at approximately the same rate.

Appendix J: PICurrent

The Portable Interceptor Current (PICurrent) is a data sharing mechanism that is specifically used by Request Interceptors to transfer context information between interception points. The PICurrent is a slot table from which the slots are used by each service to transfer their data between their data context and their request or reply's service context. This mechanism is especially useful for transferring Context data from Client services to Server services. Each service that wishes to use PICurrent, reserves a slot or multiple slots during initialisation time and uses those slots during the processing of request and replies.

The PICurrent contains two different context scopes:

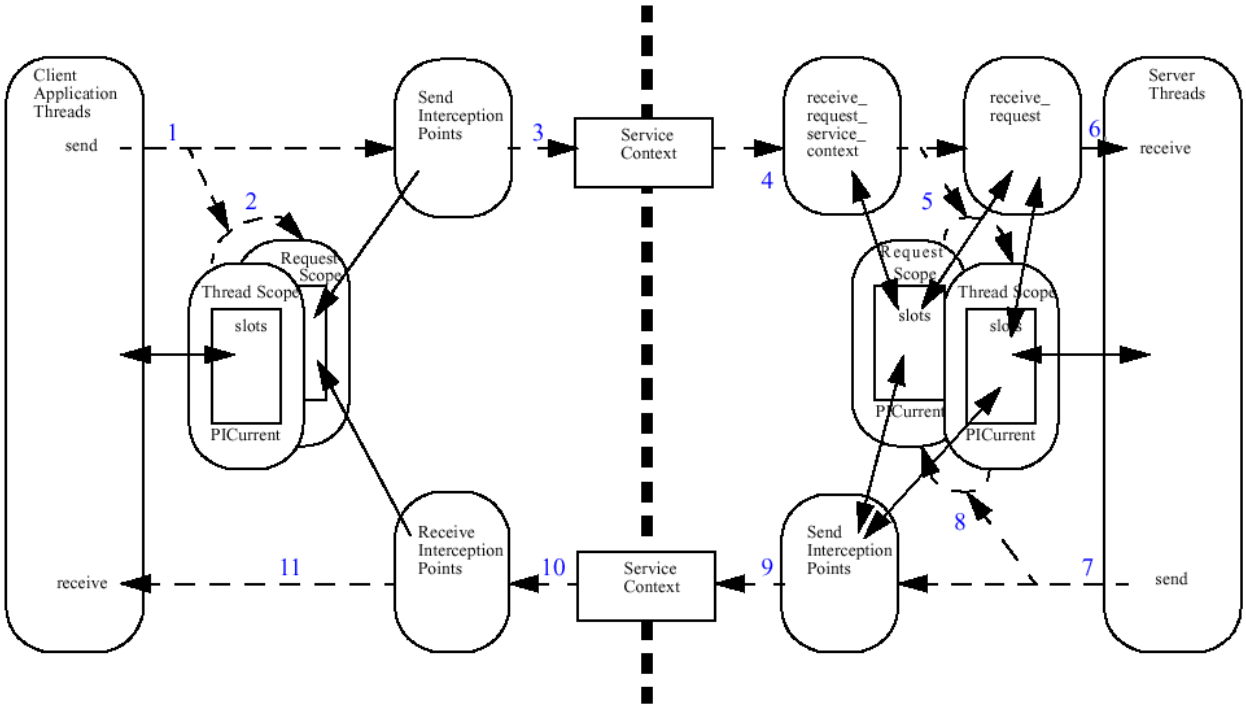
- *Tread scope*; a thread scope PICurrent is the PICurrent that exist within a thread's context. The thread scope is active during the execution of all local processes. This includes the local stub or skeleton code, which the client uses for communication.
- *Request scope*; a request scope PICurrent is the PICurrent associated with the request. The request scope is active during the execution of a request interceptor.

In the figure below, we can see how Context data is transferred between the tread scope and the request scope and used to transfer service context between client and server.

Depending on the state of the current message, the thread scope and request scope exchange information in a different way:

- On the client-side, the thread scope is logically copied to the request scope PICurrent from the threads's context when a request begins and is attached to the ClientRequestInfo object.
- On the server side, the request scope PICurrent is attached to the ServerRequestInfo and follows the request processing. It is logically copied to the threads scope PICurrent after the lists of receive_request_sercive_context interception points are processed.

The request scope PICurrent is a logical copy of the thread scope PICurrent at the point the invocation began. Even if a client side interceptor happens to be running in the same thread from which the invocation was made, the request scope PICurrent and thread scope are still different. PICurrent allows portable service code to be written regardless of the Orbs threading model. Interceptors assume that each client interception point logically runs its own thread, with no context between it and any other thread. While an Orb implementation may not actually behave in this manner, it is up to the Orb implementation to thread PICurrent as if it did.



Appendix K: Installing Interceptors

Before any interceptor can do their work, they must first be registered with a local Orb object before initialisation. This is because interceptors must become part of the Orb architecture. In order for a client to establish a connection with a servant, the Corba Orb architecture requires both client and servant to initialise a local Orb object in their source code. The Orb architecture also requires that all clients use a separate local Orb object for each connection with another servant instance.

Request interceptors are executed in a stack flow model. This means their request interceptor request methods are invoked one after other, and pushed onto a stack by the Orb. When the reply arrives back at the client, the interceptors are popped of the stack, and their ending method is invoked, but this time in the opposite order. The model also guarantees the same interceptors intercept the request and reply message invocation, except when an exception is thrown during invocation by either a client or server, then this may only be a subset of all registered interceptors

Although activated interceptors can communicate with the outside world, once an Orb with one or more interceptors registered is initialised, Corba offers no facilities that can alter the interface calling behaviour of an interceptor from the inside or outside. By interface calling behaviour, we mean the Orb functionality that calls the interfaces of each installed interceptor. In other words, there is no way of disabling or enabling an installed interceptor, except during initialisation or killing the Orb completely.

The Official OMG Interceptor specification states that an Interceptor is registered by registering an associated *OrbInitializer* that implements the *OrbInitializer* interface (see interface definition below).

```
module PortableInterceptor {
      local interface OrbInitializer {
            void pre_init (in OrbInitInfo info);
            void post_init (in OrbInitInfo info);
      };
};
```

Unfortunately, the activation of *OrbInitializer* is different for each programming language. In the case of C++, the OrbInitializer is activated by calling *register_orb_initializer* defined by the *PortableInterceptor* name space (see C++ specification below)

```
Namespace PortableInterceptor {
      Static void register_orb_initialiser (
            PortableInterceptor::OrbInitializer_ptr init);
};
```

To register the service, an application would first call to the global operation, register_orb_initialiser, passing in the service OrbInitialiser as parameter (see C++ source code example below).

```
PortableInterceptor::register orb initializer (initializer.in ());
```

After this is complete, the application would make an instantiating Orb_init call that produces an active Orb object (see C++ source code example below).

```
Corba::Orb var orb = Corba::Orb init (argc, argv, "" ACE ENV ARG PARAMETER);
```

Although the registration of request interceptors into Orb by function callsis well defined, most Orbs implementations allow the registration of the request interceptors through external means. In fact, some Orbs do not offer any corresponding internal method for Orb initialisation. Java Orb implementation only allows registration of an *OrbInititiaializer* object by means of Orb properties. Java Orb properties can be provided for instance by using command line arguments or by setting a special initialisation file. Although Orb properties allow us to install interceptors without altering any client source code, it is currently Orb vendor dependant and therefore a proprietary solution.

Once every OrbInitializer method is registered in the local Orb, a client can start the initialisation of the Orb instance by calling the Orb_init method of the Orb object. During Orb object initialisation, it calls each registered

OrbInitializer once, passing it an OrbInitInfo object, which is used by its methods to register an interceptor. Note however, that the registration code executed during initialisation should avoid using the Orb because invocations during this state are undefined in some Corba implementations.

The *OrbInitialiser* interface only contains two methods; pre_init and post_init. Both interface methods are called only once during Orb initialisation and therefore interceptors cannot be registered on an Orb after it has been returned by a call to Orb_init . The only difference between them is that pre_init must register initial services itself while post_init can assume that all initial reference are available. Both methods obtain an *OrbInitInfo* object as in parameter, which contains methods for registering interceptors (see interface definition below).

```
Module PortableInterceptor  {

   local interface OrbInitInfo {
      typedef string ObjectID;
      exception DuplicateName {string name;};
      exception InvalidName {};

      readonly attribute Corba::StringSeq arguments
      readonly attribute string orb_id;
      readonly IOP::CodecFactory codec_factory

      void register_initial_reference (in ObjectID id, in Object obj) raises (Invalidname)
      void resolve_initial_references (in ObjectId id) raises (InvalidName)
      void add_client_request_inteceptor (in ClientRequestInterceptor interceptor) raises (DuplicateName)
      void add_server_request_interceptor (in ClientRequestInterceptor interceptor) raises (DuplicateName)
      void add_ior_interceptor (in IORInterceptor interceptor) raises (DuplicateName)
      void register_policy_factory (in Corba::PolicyType type, in PolicyFactory policy_factory);

      Slotid alocate_slot_id
   }
```

In practice, we only have to implement either the pre_init or the post_init method and initialise and register the interceptor with methods available in *OrbInitInfo* (see example below)

```
  Void Client OrbInitializer::pre init (
      PortableInterceptor::OrbInitInfo_ptr
      ACE_ENV_ARG_DECL_NOT_USED)
    ACE_THROW_SPEC ((Corba::SystemException))
  {
  }

  Void Client_OrbInitializer::post_init (
      PortableInterceptor::OrbInitInfo_ptr info ACE_ENV_ARG_DECL)
    ACE_THROW_SPEC ((Corba::SystemException))
  {
    PortableInterceptor::ClientRequestInterceptor_ptr interceptor =
      PortableInterceptor::ClientRequestInterceptor::_nil ();

    PortableInterceptor::ClientRequestInterceptor_var
      client_interceptor = interceptor;

    info->add_client_request_interceptor (client_interceptor.in ());
  }
```

Appendix L: IOR interceptors

An IOR interceptor is an object, which is invoked by the Orb when the Programmable Object Adapter (POA) creates the IOR. The IOR interceptor allows an IOR to be customised by appending special tagged components. These tagged components can be used in conjunction with Request Interceptors to establish Quality of Services (QoS)

IOR interceptors can provide the following functionality:

- *Tracing IORs*. IOR interceptors can be used to trace IORs as they start their journey trough a system after they are generated by the POA. This could be useful for security services that guard the spreading of IORs or by logging services that want to track the birth of IOR.
- *Modifying IORs*. In some cases, a portable Orb service implementation may need to add information describing the server's or object's Orb service related capabilities to object references in order to enable the Orb service implementation in the client to function properly. For example, by associating information about security policies with an object reference, a client and server Orb can establish a secure connection. Another example is to augment the object reference with additional profiles that provide an alternative Internet address for the object when the object is reachable by multiple different TCP/IP paths. In these cases, IOR Interceptors can be used to insert service-specific information into the IOR. The IOR Interceptor does this by establishing tagged components within an Interoperable IOR.
- *Re-routing IORs*. Some systems may require published object references that escape from the system to be references to a firewall or gateway than the real object. IOR Interceptors can be used to replace the IOR of an object with an IOR of an entirely different object, like Orb daemons.

Policies are used for specifying component security policies and can be created by implementing a special Policy Factory. The picture below illustrates how a set of policies influences the set of tagged components contained within the profiles of any IOR created by that POA [Interceptor v2.6.1].

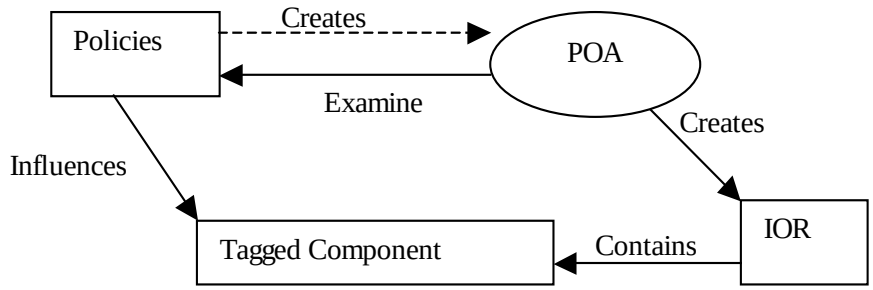


Figure 5.6: Relation ship between IOR and other Corba concepts

In order to add tagged components to the IOR, special field must be modified embedded inside the new Corba message format. Version 1.1 of the Corba Internet Inter Orb Protocol (IIOP) introduces an IDL attribute called `components`. The new attribute contains a list of tagged components to be embedded within an IOR. This list of tagged components provides a placeholder for an Orb to store extra information pertinent to the objects in a component. This information can contain various types of Quality of Service (QoS) related information pertaining to security, server thread priorities, network connections, Corba policies, or other domain-specific data. Modifying the IOR attributes is supported through the IORInterceptor and IORInfo interfaces. The IOR interceptor uses IORInfo object to access the several attributes contained in the IOR. The old Corba message format (IIOP version 1.0) provided no standard way for applications or services to add tagged components into an Orb. Services and applications that required this field were therefore forced to use proprietary Orb interfaces, which limited their portability. The new PI specification resolves this problem by specifying IOR interceptors that provide a meta-programming mechanism to customise components IOR.

Appendix M: History of the Portable Interceptor

Before Corba 2.3.1, interceptors were under specified. In contrast to the current specification, which consists out of xxx pages, the first definition only contained nine pages. It was therefore not surprising that the first interceptor implementations were often lacking, bugged, or simply not implemented at all. This motivated many frustrated Orb developers to discard the original OMG specification, and develop their own interpretation of the Interceptor. This resulted into in a large number of propriety solutions that was non-portable between the different Orb implementations. The problem was recognised by the OMG, which issued a Request for Proposal (RFP) in September 1998 [proposal]. After some iteration of proposals and discussions, the leading vendors of the field came to an agreement, and handed in their Joint Submission in December 1999 [revised]. Many Orb vendors, but not all, now have adopted the latest OMG Interceptor specification [Interceptors 2.6.1] in their list of features.

The original PI definition also contained a specification for the message interceptor. In the new Interceptor specification, the functionality of the message interceptor is embedded in the request interceptor. Message interceptors were used to transfer context information transparently between different execution environments. Message interceptors could treat the message as a structureless buffer. This means that message interceptors could operate on messages in general without understanding how these messages relate to requests. A hooked message interceptor is able to examine and modify a context field located in the IIOP message structure travelling between Corba components.

To allow the examination and modification of the context field at both client side and server side, the message interceptor differentiates between a client message interceptor and the server message interceptor. The client message interceptor was hooked in Orb that acted as the client while the server message interceptor is hooked in the Orb, which acted as the server (see picture below). As seen in the image, the message interceptor operated on a lower Orb level than the request interceptor. On the client side, the message interceptor is called after the Client request interceptor. This level of interface would be useful for performing operations on fragmented messages or add encryption for increased security.

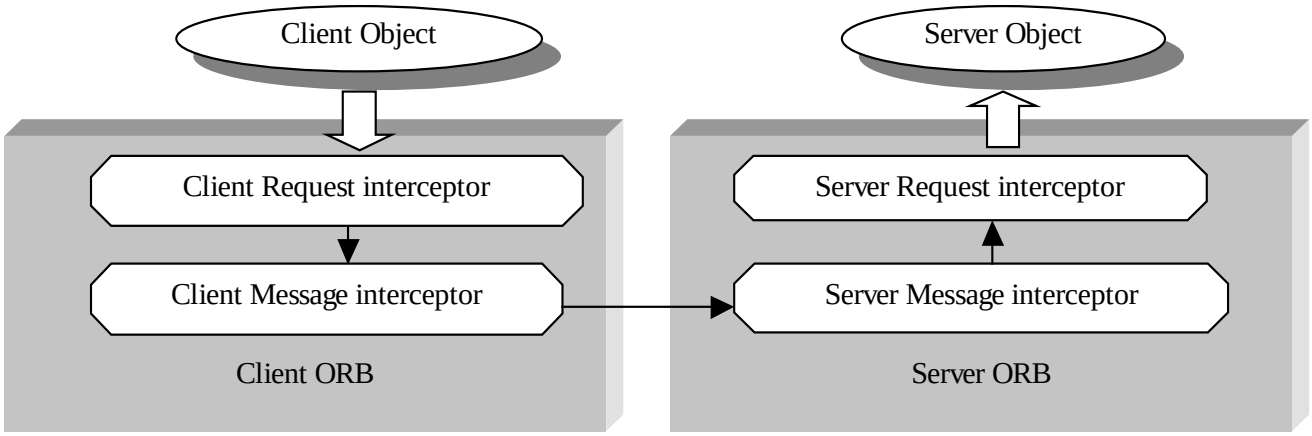


Figure M: Message and Request level interceptors