An approach for performance measurements in distributed CORBA applications

Author: Marcus Holgersson
marcus.ho@telia.com
Supervisor: Håkan Grahn
Abstract

Distributed computing systems are becoming more and more important in everyday life as well as in industrial and scientific domains. The Internet and its capabilities enable people to communicate and cooperate all over the world. One way to construct distributed systems is to use a communication model with distributed objects as CORBA (Common Object Request Broker Architecture). Distributed objects give many advantages, but suffer from some performance problems.

In order to handle the performance problem it is important to find where in the event chain the delays occur. Therefore a tool for performance measurement and for identifying the bottlenecks in a distributed system should be a great help. This report answers the question: Can a profiling tool for CORBA applications, constructed with Interceptors as instrumentation for measuring points, give sufficient information for identify performance problems?

This report investigates the possibilities to measure performance in a distributed system and if it’s possible to automatically find the bottlenecks in a distributed system. The needs of a profiling tool are discussed and analyzed. The different ways of constructing a tool for distributed profiling is discussed. For verifying the ideas evolved in the investigation a prototype tool for profiling and performance measures is constructed. The profiling tool is constructed with Interceptors as instrumentation of the different nodes in the distributed system. A presentation program is also constructed for making the captured information more readable.

The tool and presentation program constructed give the flow of the system in different callgraphs and also produces some call statistics in different levels. The constructed tool is tested and verified in a distributed environment. In the experiments we shows that the principle of the tool can work in a distributed environment and gives sufficient information for finding the bottlenecks and identifying the performance problems of the system.
1 Introduction .......................................................... 4
2 BACKGROUND ......................................................... 5
  2.1 Distributed system ........................................... 5
  2.2 Distributed Object Computing................................. 5
  2.3 CORBA ............................................................ 6
  2.4 Summary of related work .................................... 9
3 Performance handling in a distributed system ........ 10
  3.1 Demands for high performance. ............................. 10
  3.2 Improving Performance ....................................... 10
  3.3 Diagnosing a distributed system ............................. 11
  3.4 Interceptor. ....................................................... 11
4 Profiling tool ......................................................... 12
  4.1 Requirements ................................................... 12
  4.2 Design ............................................................. 13
  4.3 Implementation ................................................ 16
  4.4 Running the profiling-tool. .................................. 21
5 Experimental framework ......................................... 22
  5.1 Server ............................................................. 22
  5.2 Components ....................................................... 23
  5.3 Client ............................................................... 24
  5.4 Running the experiment ..................................... 24
6 The Result ........................................................... 26
  6.1 Experiment 1 ..................................................... 26
  6.2 Experiment 2 ..................................................... 26
  6.3 Experiment 3 ..................................................... 29
  6.4 Experiment 4 ..................................................... 33
  6.5 Experiment 5 ..................................................... 36
7 Discussion and Future work ................................. 39
8 Conclusion .......................................................... 40
9 Bibliography ........................................................ 41
Appendix A
Appendix B
Appendix C
1 Introduction

Distributed computing systems are becoming more and more important in everyday life as well as in industrial and scientific domains. The Internet and its capabilities enable people to communicate and cooperate all over the world. The technique has made it possible to totally change the way we do businesses and given borne to the expression e-commerce. One way to construct distributed systems is to use a communication model with distributed objects. Distributed objects give many advantages, but suffer from some performance problems. A distributed object system can be constructed using CORBA (Common Object Request Broker Architecture [10]). One great benefit with CORBA is that it can function in a heterogenic environment with many different program languages and different operating systems. With the use of CORBA can an application handle distributed objects as they where local object.

Performance problems can occur in several phases of the event change in a distributed system. Many applications are time critical as for booking systems and exchange dealings. These have demands for short response times, which make great demands upon the performance of the system.

When developing a distributed system it is desirable to know its limitations and bottlenecks for preventing a performance problem. If it were possible to make performance tests early in the development potential performance problem could be prevented. In order to handle the performance problem it is important to find where in the event chain the delays occur. Therefore a tool for performance measurement and for identifying the bottlenecks in a distributed system should be a great help.

The aim of this report is to answer the following question: Can a profiling tool for CORBA applications, constructed with Interceptors as instrumentation for measuring points, give sufficient information for identify performance problems? That question demands answers to a lot of other questions. What performance criteria are important for a distributed system and how does one measure the performance of a distributed system. Is it possible to construct a profiling tool so general that it will be usable on different applications without changing the application?

For answering the questions I will collect and analyze information concerning performance in distributed systems. This will be used for investigating the possibilities to measure performance and to finding out what performance criteria’s is important for a distributed system. It will also be investigated however it is possible to automatically find the bottlenecks in a distributed system. The investigation will be used for constructing a prototype tool for profiling and performance measures of a test system. The test system that has been uses uses JacORB as CORBA implementation for the server side and for the client side. Test system will be a simulation of a distributed application. The result of the implementation and the experiments will be evaluated. It will also be analyzed for giving design hints for designing distributed system with distributed objects.

This report is written for readers interested in distributed object systems and with basic knowledge in object oriented system development. This paper evaluates the use of interceptors for instrumentation of a distributed object system in the purpose of finding the bottlenecks of the system. The best way to evaluate if it is possible to make such a tool is to try making one. First the report will give the background to distributed systems and the techniques used in the experiments. The next section analyzes the need of such a profiling tool. Then the next section will work through the process of constructing the tool. After the construction of the tool the next two chapters will explain the test environment and the experiments done for evaluating the prototype.
2 BACKGROUND

2.1 Distributed system

Distributed computing systems are becoming more and more important in everyday life as well as in industrial and scientific domains. The Internet and its capabilities enable people to communicate and cooperate all over the world. The technique has made it possible to totally change the way we do businesses and given borne to the expression e-commerce.

The use of distributed systems as the Internet has enabled people to communicate and cooperate independently of geographic location. This kind of systems have become important

Today it is possible to construct an application that can be used by different users all over the world. It has become possible to cutout the middleman in commercial systems and reducing the cost and making business more efficient.

Distributed computing can be seen as an effort to bring multiple network machines, that will cooperate with each other in such a way that information and other resources can be shared by all of these connected computers. The reason for building the application distributed is one or more of the following reasons:

- The data used by the application are distributed
- The computation is distributed
- The users of the application are distributed

There are a lot of techniques to choose from for constructing distributed systems. The current trends in distributed computing shows a movement towards distributed middleware environments based on standardized communication protocols and networks [5]. The most common programming paradigm used for distributed computing is called distributed object computing (DOC)

2.2 Distributed Object Computing

Distributed object computing as system approach integrates object orientation, client/server architecture, and distributed computing. Applications are decomposed into a set of objects, either providing server functionality or acting as clients. Every object can act as both a client and a server for a set of other objects. Objects interact by invoking methods on requested target objects, identical to the traditional paradigm of object-orientation.

Distributed object computing is a popular paradigm for object-oriented distributed applications. Since the application is modeled as a set of cooperating objects, it maps very naturally to the services of the distributed system. The remote access is implemented by making use of proxy and broker patterns.

The purpose with distributed object technology is to make an object location transparent. It aims at making it just as easy to access an object on a remote node as an object on a local node. Location transparency involves these functions [2]:

- Locating and loading remote classes
- Locating remote objects and providing references to them
- Enabling remote method calls, including passing of remote objects as arguments and return values
- Notifying programs of network failures and other problems
The first function, locating and loading remote classes has to be satisfied. A distributed system demands a flexible function that can locate and download classes from several hosts. Such a capability lets system developers store classes on whatever system that can provide the class most efficient.

The second function, locating and providing references to remote objects, requires a database of objects. The database has to be managed by a server that provides access to the database and some kind of lookup service. When a service is needed the server has to provide a reference to a suitable object. The reference must be legal on any node in the system and it has to be possible to transport it across the network.

The third function, support method calls requires a mechanism for obtaining a reference to a target method as well as a mechanism for transporting end return values across the network. This is a rather complex mechanism because a parameter and a return value may contain other objects.

The fourth function, notifying the programs of network failure is necessary for enabling recovering functionalities and fault tolerance. This is usually made with some kind of exception handling.

If all four functions are present and works properly a distributed system is enabled. There exist several DOC systems the most commonly known are OMG’s CORBA, Sun’s Java RMI, Sun’s Enterprise Java Beans and Microsoft’s DCOM systems.

2.3 CORBA

CORBA stands for Common Object Request Broker Architecture. It is a standard for a class of products, not a product itself. The CORBA standard is the creation of the Object Management Group (OMG), a non-profit organization founded in 1989 by a number of large corporations. They do not produce software or implementation guidelines; only specifications which are put together using ideas of OMG members who respond to Requests For Information (RFI) and Requests For Proposals (RFP). The strength of this approach comes from the fact that most of the major software companies interested in distributed object oriented development are among OMG members. The first CORBA standard was introduced in 1991, and it has undergone continual and significant revisions ever since. The latest approved version is 2.2 and it weighs in at 962 pages [10].

The aim of CORBA is to define a general framework for construction of distributed object computing systems. CORBA is a form of middleware. The objective of middleware is to hide the details of network protocols from the application level, providing a software layer that wraps the network with a logical set of services. Programmers can then call upon these services in their applications, without worrying about network-related issues like connection setup, deciding on protocols, on-the-wire data format, and failure scenarios.

Since CORBA is just a specification, it can be used on diverse operating system platforms from mainframes to UNIX boxes to Windows machines to handheld devices as long as there is an ORB implementation for that platform.

CORBA is designed to work in a heterogeneous environment and is location and programming language independent.

The central component of CORBA is the Object Request Broker (ORB). It contains the entire communication infrastructure necessary to identify and locate objects, handle connection management and deliver data. In general, the ORB is not required to be a single component; it is simply defined by its interfaces. The ORB Core is the most crucial part of the Object Request Broker; it is responsible for communication of requests.
2.3.1 Interface Definition Language (Idl)
To allow CORBA to be programming language independent, server interfaces are specified in a declarative language called IDL (Interface Definition Language). The IDL Syntax and Semantics is defined by the OMG CORBAv2.3 definition [9] The IDL is separating object interface from implementation. The IDL can be described as the contract for communication between a client and a server. The IDL is a purely declarative language that means that it isn’t possible to write execution or state dependent code in IDL.

The OMG defines IDL language mappings for a number of common programming languages including C, C++, Ada, and Java.

IDL provides a set of built-in types that can easily be translated into most programming languages. The set of build-in types can be augmented by user-defined types, such as structures and sequences. IDL provide object orientation through interface inheritance, which in turn establishes type compatibility and polymorphism.

The IDL is translated in an IDL-compiler to client side stubs and server side skeletons. The stub is an API for the client to access the distributed object. It acts as a proxy for the distributed object. The skeleton is an up-call interface from the ORB into the developer written code in the server. It is a connection between the networking layer of the ORB and the application code. The server’s implementation of the distributed object is termed servent.

The interface declaration of the components is the key to how it can be distributed. Functionality can only be distributed if there is an interface to access the functionality. So the way an application is broken into interfaces determine how it can be distributed over physical address spaces.

2.3.2 Interoperable Object Reference (IOR)
CORBA’s way for uniquely addressing objects within the system is the usage of so called interoperable object references (IOR). An IOR is an opaque handle for an object that contains all the necessary information for any other ORB on the network to locate the object. When a new object implementation for a given service type is instantiated, it is the object adapter’s task to choose a globally unique object reference. The ORB interface provides operations for converting IORs to strings and back, so that a simple mechanism for passing around IORs is given. As objects are only accessible by their object reference,
CORBA objects are generally passed by reference when used as parameters in method calls.

An IOR contains three major pieces of information: Repository ID, Endpoint Info, and Object key. The repository ID is a string that identifies the most derived type of the IOR at the time the IOR was created. Endpoint Info contains information required by the ORB to establish a physical connection to the server implementing the object. The Object key is an unstandardized object ownership information. The information is only understood by the ORB that has created the object reference. The client simply sends the key as a blob of binary data with every request made on the object.

2.3.3 CORBA Interoperability Protocols

The CORBA standard guarantees interoperability between ORB implementations as well as applications built with different vendors ORBs. CORBA interoperability provides a gateway infrastructure that makes different ORB implementations compatible. It sits in the transport layer of the OSI model. The General Inter-ORB Protocol (GIOP) defines standard message formats, a common data representation for mapping IDL data types to flat messages, and a format for interoperable object references. The Internet Inter-ORB Protocol, commonly known as IIOP, defines GIOP message exchange over TCP/IP networks. In other words, IIOP is the CORBA wire level protocol for the Internet and intranets.

2.3.4 Portable Object Adapter (POA),

The POA provides a comprehensive set of interfaces for managing object references and servants. Then an object is constructed using the POA interfaces it gets portable across ORB implementations and has the same semantics in every ORB that is compliant to CORBA 2.2 or above. This makes the objects porting between different ORB’s very easy.

The POA defines standard interfaces to do the following:
- Map an object reference to a servant that implements that object
- Allow transparent activation of objects
- Associate policy information with objects
- Make a CORBA object persistent over several server process lifetimes

2.3.5 Corba service

Another important part of the CORBA standard is the definition of a set of distributed services to support the integration and interoperation of distributed objects. The services are defined on top of the ORB. That is, they are defined as standard CORBA objects with IDL interfaces.

2.3.5.1 Object location mechanism

For object location the CORBA standard defines a naming service. It provides a mapping from names to object references: given a name, the service returns an object reference stored under that name. The naming service provides a number of advantages. Clients can use meaningful names for objects instead of having to deal with stringified object references. By changing the value of a reference registered under a name, the client’s starts use a new implementation of an interface without having to change the source code. The clients use the same name but get a different reference. The Naming Service can be used to solve the problem of how application components get access to the initial references for an application. Advertising these references in the naming services eliminates the need to store them as stringified references in files. The Naming Service is a CORBA object that also has a IOR. The IOR of the naming service is the only location dependent information.
needed in a distributed application. All other objects can be found through the naming service if they are registered in the service.

2.3.5.2 Support for alternate communication paradigms.

The standard also defines some services for alternative communications paradigm. There is a service for Event Channels and one for asynchronous measuring. These services can be very useful for decoupling components in a distributed application. They can also be used for making loadbalancing and fault tolerant applications.

2.3.6 JacORB

JacORB is an object request broker written in Java - an implementation of OMG's CORBA 2.0-2.3 standard. JacORB is a fully multithreaded ORB with support for IIOP, POA, Portable Interceptors and OMG Interoperable Naming Service. I also include an IDL compiler that supports OMG IDL/Java language mapping rev. 2.3. JacORB can be obtained at http://www.inf.fu-berlin.de/~jacob/

2.3.7 Performance considerations In CORBA

There are some basic factors that have a distinct impact on the performance of a CORBA application. First one is the number of remote method invocations that are made within the system. Each request sent over a network connection imposes a "net latency." This delay adds a considerable amount of time for the processing of each CORBA remote invocation. This factor is the reason that the number of remote invocations is often more significant than the amount of data transferred with each request.

The second one is the amount of data that is transferred with each remote method invocation. If the message gets too big, the throughput rate decreases due to the limitation of the network as TCP buffering issues and growing process sizes.

The third one is the marshalling/unmarshalling costs of the different IDL data types used by the system. Marshalling/unmarshalling is the procedure of translating the data from the program representation to a portable and transportable format and vice versa. This factor is highly dependent of the ORB implementation used.

The cost of call dispatch varies considerably among environments and depends on a large number of variables, such as the underlying network technology, the CPU speed, the operating system, the efficiency of the TCP/IP implementation and the efficiency of the ORB run time itself. The only way to really find out is to create a benchmark [8]

2.4 Summery of related work

Most of the papers on CORBA performance actually compare the performance and scalability of competing CORBA-compliant ORBS (an example of this is [3, 1]). And many studies on the performance of CORBA objects focused mainly on identifying the performance constraints of an Object Request Broker (ORB). Schmidt analyzed the performance of Orbix and VisiBroker over high speed ATM networks and pointed out several key sources of overhead in middleware ORBs [4]

In the article "Benchmark metrics for enterprise Object Request Brokers" [7] says the author’s that a pilot study with the actual ORB in the actual application environment is the only measurement that gives a trustworthy result. They experience that an automated testing is the key factory. The article provides some guiding on what measures are to use then estimating an ORB’s performance. It also gives some design-guiding on how to build CORBA applications.
IBM has developed a performance tool for distributed applications using RMI called javiz [6]. It uses an instrumented JVM and catches the profiling data in a file in every node. After the execution is done the tool merge the files and presents the result. The profiling is presented with one graph for each client call.

3 Performance handling in a distributed system

This chapter will discuss the performance demands on a distributed system and analyze how to cope with the performance problems in a distributed system. It will discuss what can be done for improving the performance and the need of a profiling tool. This chapter will also introduce interceptor’s principles and techniques. Interceptors are one possible technique for constructing a profiling tool.

3.1 Demands for high performance.

What kind of performance is important for a large distributed system? It depends on the purpose of the system. There are a lot of different aspects on performance response time, through put and workload. If it is an e-commerce system the throughput is important to the system owner and the response time for the user. If the system is a bid / ask system the performance must be the same for all users. If it isn’t some users may get commercial benefits depending on higher performance. As a System developer I want to know what load the system can handle and were the bottlenecks are.

The distributed systems to day have to be able to handle a lot of concurrent users. A big commercial system can have several thousands of users. If the information in the distributed system is rapid changing as in an exchange system it must be able to handle high load peaks. The performance in the system is directly dependent of the invocation response time. By shorting the invocation response time other performance parameters will benefit the change.

3.2 Improving Performance

There is different way for improving the performance of a distributed system. Optimize code, optimize the design and the distribution and improve hardware and network. If there are a performance problem, the first thing to do is to find were the problem is in the system. And then it is found decide on the counter action to reduce it. For finding the problem is it necessary to get an overview of the flow and performance of the system. The bottlenecks have to be identified.

It is important to find the steps in the critical path of the application. The critical path can be different from time to time. It could be common to use the system different during the workday. For example if the system is a exchange and the costumer is used to trade before lunch and doing transaction handling after lunch the critical path of the system change over the workday. When the critical path is found it is much easier to improve the performance of the system.

If it was possible to measure the time used in the step in the critical path it would give the bottlenecks of the system in the current run. For improvement of the performance in a distributed system a measurement tool could be in great help. The measurement tool should give the paths in the system and the time spent in every step. And the measurement tool should be plugeable to the system with out the need of rewriting any code of the application.

If CORBA is used for all communication between all autonomous components the system can be fully distributed. The design and the distribution over several processes or even
several machines have a great impact on the performance of the system. If it was possible to profile different design solutions the best distribution can be found and give hints for the best design of a distributed system.

### 3.3 Diagnosing a distributed system

What measurement is important in diagnosing distributed system. The response time is the symptom for that there is a performance problem. It would be possible to measure the response time under different workload. It is necessary to make a reasonable workload on the system. Some kind of workload program has to be constructed. This is very hard or even impossible to construct this generally so it could be used on different systems. But this kind of measurements would only give a value for if there were some performance problems not what to do about it. It doesn’t give any clues for where in the system the problem resides.

For getting a more precise measurement some kind of profiling information is necessary. Profiling information can be retrieved using an existing profiling tool as gprof or java prof. A profiling tool is only giving information about the current process it is running in. Profiling refers to the measurement of program execution characteristics of interest, e.g., execution time, message propagation time, and memory consumption. Ordinary profiling tools cannot give the information needed for finding the bottlenecks in a distributed system.

One good and effective way of diagnosing distributed systems is through monitoring communication between the various distributed components. The problem is how to capture the details of messages passed between distributed objects. Such monitoring lets you observe and record method invocations and exceptions. The information retrieved should help in avoiding or eliminating bottlenecks and other potential failures. Measuring the application-level communication should give request-reply details. Details captured about each message could include request ID, interface name of the target object, method being invoked, timing data, process IDs and host IDs. This measurement should have all the necessary information for constructing graph representing the different components and calls in the system. It should give the time spent in every method and the interprocess latency. The measurement should give useful information for finding the best design and distribution of the system.

Other profiling principles can be very useful when improving the performance of a certain component but not for the overall application. The best profiling of the system could be achieved by combining different measurement techniques. Use different tools for different levels of the system diagnosing.

### 3.4 Interceptor.

The interceptor architectural pattern allows services to be added transparently to a framework and trigged automatically when a certain event occur [4].

The OMG specifies CORBA Portable Interceptors [11]. Portable Interceptors are hooks into the ORB through which ORB services can intercept the normal flow of execution of the ORB. In the OMG specification a number of interceptor types is specified Request interceptors, IOR Interceptors and Registering Interceptors. This paper will only look in to the Request interceptors.

#### 3.4.1 Request interceptors

A request Interceptor is designed to intercept the flow of a request/reply sequence through the ORB. This makes it possible to query the request information and manipulate the service contexts, which are transported between clients and servers. The primary use of request Interceptors is to enable ORB services to transfer context information between
clients and servers. The context information is added by one interceptor and read by another
interceptor. This is information that is transported without being declared in any IDL. There
are two types of request Interceptors: client-side and server-side

![Diagram of request path using interceptors in a method call]

The specification includes a set of design principles that specifies the interceptor’s
behavior. This paper will only present and describe a few principles. For further reading see
the Interceptor specification [11]. An interceptor can affect the outcome of a request by
throwing a system exception or by directing a request to a different location. An interceptor
can’t change the parameters of a request only read them. One interceptor is independent of
an other interceptor. This means that a call can go through a number of interceptors on it’s
path to the server and back again.

Each request interceptor is called at a number of interceptor points. The following figure
shows the flow of control for a request/reply cycle.

![Diagram of a visualization of the interceptor-points of a Portable interceptor]

4 Profiling tool

For answering to the questions discussed earlier in this report a profiling tool has to be
constructed for measure and analyze the application-level communication. The tool will be
a prototype and its main propose is to answer the question if it is possible to implement
such a tool and if the chosen technique is the right way to do it. This chapter will describe
how the profiling tool is constructed. It will discuss what design decisions and what
tradeoffs made.

4.1 Requirements

There is a set of requirements that the tool must live up to. These requirements are based on
the demands of how to diagnosing a distributed system for finding the performance
bottlenecks. The requirements are divided into three categories, requirements for functionality and requirements for portability and requirements for performance.

**Requirements for functionality:**
The tool shall have the ability to trace and record elapsed time for all distributed method calls in the target application.

The tool shall have the ability to merge the trace information from different nodes into execution tree or call-graph.

**Requirements for portability:**
The tool shall be easy to port to other ORB implementations.

**Requirements for performance:**
An implementation that does not excessively distort the performance profile of the applications being traced. This means that the tool should have a low impact on the application execution time and not reducing the performance of the system.

### 4.2 Design

There are a lot of aspects to take in consideration when designing the tool. This chapter will explain and discuss the design decisions taken in constructing the tool. For making it possible to trace a call through several nodes instrumentation for catching information has to be added at every node. The tool shall only profile the system, not monitory it. This means that there is no need for merging information from several nodes in real time. The merging of the information can be done after the experiment is done. This makes it feasible to write the information to a file and later on do the merge. The presentation of the information will be done after the merging. The tool will be working in three phases information registering, information merging and information presentation.

The tool should not burden the system because that should have impact on the result. For minimize the load of the tool a producer consumer technique should be used see figure 4. The information register should produce an internal event representation that could be put in a datastructure that the consumer could read from and write to file.

![Diagram](image)

*Figure 4: A schematic explanation of the design of the profiling tool instrumentation.*

### 4.2.1 Instrumentation
For catching the information in the application an instrumentation of the nodes has to be done. Every node will contain one or several objects. These objects have to be instrumented for getting the request-response information needed. Measuring points has to be added in the application.

![Diagram](Servent_client_P1_request_P2_reply_P3_P4 Client)

*Figure 5: A visualization of where the measuring points is to be added in a request response execution.*

For registering the information in a request-reply four measuring points have to be added. In figure 5 the four points is marked P1, P2, P3 and P4. The start of the request (P1) is the first measuring point. The start of the execution (P2) of the method is the second one. The finish of the method execution (P3) is the third one and the arrival of the response (P4) is the fourth one. This points give the information wanted. The simple calculation principles used with the four measuring points are shown in table 1.

<table>
<thead>
<tr>
<th>Calculation principles</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution of method</td>
<td>P2-P3</td>
</tr>
<tr>
<td>Network latency</td>
<td>P1-P4-(P2-P3)</td>
</tr>
<tr>
<td>Total remote call</td>
<td>P1-P4</td>
</tr>
</tbody>
</table>

*Table 1: The principle for calculation of profiling data.*

Measure points could be added by adding code to the application. It could be possible to parse in profiling measurepoint into the auto generated code. The IDL file representing the distributed interfaces is compiled for generating the stubs and skeletons. The stub and skeletons could be run through a script that added measurement code. This could be done as general as it could be used on different systems. All that had to be done for adding the measurement is to recompile the IDL files, run the generated files in a parser and recompile the source code the application including the parsed files. For making this possible a big effort to construct the parser script has to be done. A framework for added code to use has to be developed. And another drawback is that the parse script shouldn’t be portable between different IDL compilers.

One other way to add measuring points is to use interceptors. The use of interceptors limits the tool to be used on CORBA systems supporting interceptors. The Interceptors could be easy hooked onto the system. A great advantage is that it only registers the CORBA specific events. Interceptors is a complete framework with all hook on facilities already developed and portable. Interceptors can be added to the application without the need of recompile the object source code. The Portable interceptors in the OMG standard [11] include all necessary measuring points.

In this prototype of a profiling tool, will the instrumentation be added to the application with the use of OMG Portable interceptors.

**4.2.2 Information processing**

This section will discuss the design of the information-processing component. The measurement will result in several files containing the data of run of the system. There will be one file for each node in the system. For reducing the complexity is the information processing divided into two phases parsing and merging.
For making the information manageable an internal representation has to be constructed. The first phase parsing is responsible for translating the file information into the internal representation.

The internal representation have to support the merging of information and work as input for the presentation. For answering to these demands and due to the lack of time in constructing the tool, a tree representation is chosen for the parsing phase. A tree representation is not to prefer in handling a lot of data but for this prototype is it sufficient. When constructing a large-scale commercial tool this has to be replaced with a database backbone.

The tree consists of following entities see Figure 6. The root of the tree is the system. A system has a number of nodes. A node is representing one ORB and that means one instrumentation set and one data-file. The node has Objects and unbound threads. An Orb can act as a client or a server or both. When an ORB acts as a client can it make requests from code outside a distributed object. This kind of requests will be placed under the object UnboundThread. If an ORB only acts as a client all its calls will be in this entity. An UnboundThread has a list of Calls. Calls will be explained later. Objects are the distributed objects in the node. An Object has a list of methods. A method entity is created for every method executed on the Object. The Method entity has a list of executions. An execution is created for every time the method is executed. An execution entity has a list of calls, this calls is the same entity that an UnboundThread holds. The call entity represent a distributed call made from the current execution or from the current UnboundThread.

![Figure 6: The internal representation of the executed system](image)

After the parsing is done the whole execution of the system is represented as shown in figure 6 and the merging phase can start. The merging phase is responsible for finding the relationship between calls and executions. For every Call entity will the target Execution be
searched for. If it is found a relation to it is made seen Figure 7. There can be nodes in the system that is not instrumented and that result in that the execution of the target method may not be found. After the merging is performed all distributed calls in the instrumented system will be registered.

Figure 7: The additional reference of the merging phase

After the merging is done there is a complete internal representation of the execution in the system. This internal representation is easily parsed and searched in for all kind of information not only callgraph profiling.

### 4.2.2 Result presentation

This section will discuss the design of the presentation component of the profiling tool. The presentation component will not be fully implemented in the scope of this work. With the use of the internal representation it is easy to write presentation programs that make a lot of different information extraction. The purpose of the paper is to evaluate tool construction principle so due to the lack of time the only presentations is done for illustrating the possibilities and for answering the specific questions of the experiments. The results will only be outputted as flat files no graphic representation will be implemented.

### 4.3 Implementation

This section will explain how the tool is implemented and why certain implantation decisions has been taken. The source code for the client and the server interceptor of the instrumentation is presented in Appendix A.

#### 4.3.1 File writing

For handling the file writing an application called log4j is used. It can be retrieved at http://jakarte.apache.org/log4j/. log4j includes a lots of functionalities for logging.

One of the features of log4j is asynchronous logging. The asynchronous logging will collect the events sent to it and then dispatch them to all the appenders that are attached to it. An appender is a class that writes the event to some media or stream. There are appenders for writing to the consol, to a file or sockets. The asynchronous logging uses a separate thread to serve the events in its bounded buffer. This does not automatically increase performance of the tool. But in an application that has operations with long blocking and non CPU-intensive operations, such as I/O, network access, sleeping threads, the asynchronous
logging would have tremendously reduced the cost of logging in terms of overall application runtime.

The Log4J also have features for logging message design with functionalities for time stamps and thread information registering. The use of log4j has improved the measuring tool construction and removed the need of implementing a threaded producer-consumer communication described earlier. This functionality already exists in the log4j asynchronous logging

4.3.2 Call registration

Information for making it possible to connect client and server to each other has to be registered. This is implemented with the use of the CORBA Portable Interceptors [11].

The purpose with the instrumentation is to add measuring points for registering the distributed calls. When the interceptor hooks on and catches the information for a client request, it gets a ClientRequestInfo object (see Appendix B). In ClientRequestInfo the target is represented as an Object reference. For resolving the object reference to an object the interceptor has to call the naming service. This results in a new distributed call. This is not acceptable because it have a sever impact on the measurement. It results in two distributed calls instead of one. What is even worse is if the calls to the namingservice also are intercepted this call will generate a new namingservice call and a recursive loop is started with no end.

For making the callgraph it is necessary to know which client is calling which server. The time for the call isn’t enough because the nodes have individual clocks end execution environments.

The only way to connect the client interceptor send_request to the server interceptor receive_request is to register the IOR in both measuring points. The IOR can be retrieved in the client interceptor from target attribute in the ClientRequestInfo object. And in the server side can the attribute object_id in the ServerRequestInfo object be used for getting the IOR from the POA that has created it.

It isn’t enough to know which client that calls which method for making a call graph. It is necessary to know which request on a method that is made from which client. For making it possible to connect the client request to the server execution of the request the request ID in the struct RequestInfo can be recorded see Appendix B. According to the OMG specification shall the request ID be a unique id for the request on a particular request/response sequence [11]. It also says, “ Once a request/reply sequence is concluded this ID may be reused”. In the CORBA implementation used (JacORB1_3_11) has the developer chosen to make the requestID unique only for a request/response between two ORB’s. This means that the requestID is only unique for the request between two nodes. If a new nod calls the same server the id will be reset to zero. This makes it impossible to know which node makes which request.

It is absolutely necessary to record which client request is connected to which execution of a method for making a callgraph. For connecting the client side send_request with the server side receive_request the ServiceContext function is used. The ServiceContext functionality is for letting data being added to the request in one Interceptor and become read in an other [11]. A unique instrumentationRequestID is created in the ClientRequestInterceptor and added to the ServiceContext. The ServerRequestInterceptor reads the instrumentationRequestID and record the information. This makes it possible to register which send_request is connected to which receive_request.

In the client side two measuring points are added using the send_request interception point and receive_reply. Table 2 shows which data recorder at the points.
Table 2: The Data recorded in the clientside measuring points.

<table>
<thead>
<tr>
<th>Data recorded</th>
<th>Send_request</th>
<th>receive_reply</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target Address</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Target IOR</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Method name</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>RequestID</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>InstrumentationRequestID</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Time point</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread name</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

In the serverside two measuring points are added using the receive_request interception point and send_reply interception point. Table 3 shows which data recorder at the points.

Table 3: The Data recorded in the clientside measuring points.

<table>
<thead>
<tr>
<th>Data recorded</th>
<th>Server Measuring points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>Recive request</td>
</tr>
<tr>
<td>IOR</td>
<td>send_reply</td>
</tr>
<tr>
<td>Interface</td>
<td>Yes</td>
</tr>
<tr>
<td>Method name</td>
<td>Yes</td>
</tr>
<tr>
<td>RequestID</td>
<td>Yes</td>
</tr>
<tr>
<td>InstrumentationRequestID</td>
<td>Yes</td>
</tr>
<tr>
<td>Time point</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread name</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The time point recorded is made with a built in function in java System.currentTimeMillis. The functions gives the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC. But the resolution of the function on a Windows NT machine is 10 ms.

4.3.3 Information presentation.

The internal representation described earlier gives the possibilities to make a lot of calculations and presentation of the execution of the system. For showing that the profiling principles of the constructed tool can give sufficient information some presentation programs have been implemented. The presentation programs can also be seen as examples of what kind of calculations and presentations that can be performed on the internal representation.

The first presentation implemented is a client call graph. This presentation writes the different call graphs to a file called CallGraphTyp1.txt. Only the calls that start from an UnboundThread will be traced and recorded in this presentation. This will trace down the calls and the execution of distributed methods recursively. Every new step in the call graph will be visualized with an insertion. The end of an execution will be marked with a line with the same insertion as the start of the execution and with the text “End execution” see figure 8.
This presentation gives the ability to get some profiling information from some specific calls. A drawback of this presentation is that if there are a lot of calls in the system it gives a lot of information that can be hard to get a grip on. Every call made from a client will give one call graph.

The next presentation is some call statistics. It gives a summarized picture of what calls that have been done in the system. It presents the information in ordinary profiling style see figure 9. The information presented is the number of calls made from one method to another, the total time this calls took and the total latency for this calls. When the presentation program is run, the presentation will be written to a file named callStatistics.txt.
information increases the resolution of the presented information see figure 10. This presentation will be written to a file called callStatisticsPerNode.txt.

<table>
<thead>
<tr>
<th>count</th>
<th>callee caller time latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>162</td>
<td>IDL:Comp_5:1.0 127.0.0.1:1408 Comp_5_Meth_1</td>
</tr>
<tr>
<td>81</td>
<td>IDL:Comp_4:1.0 127.0.0.1:1408 Comp_4_Meth_1</td>
</tr>
<tr>
<td>50</td>
<td>IDL:Comp_2:1.0 127.0.0.1:1408 Comp_2_Meth_1</td>
</tr>
<tr>
<td>42</td>
<td>IDL:Comp_3:1.0 127.0.0.1:1408 Comp_3_Meth_1</td>
</tr>
<tr>
<td>39</td>
<td>IDL:Comp_3:1.0 127.0.0.1:1408 Comp_3_Meth_1</td>
</tr>
<tr>
<td>32</td>
<td>IDL:Comp_3:1.0 127.0.0.1:1408 Comp_3_Meth_2</td>
</tr>
<tr>
<td>28</td>
<td>IDL:Comp_5:1.0 127.0.0.1:1408 Comp_5_Meth_1</td>
</tr>
<tr>
<td>21</td>
<td>IDL:Comp_5:1.0 127.0.0.1:1408 Comp_5_Meth_1</td>
</tr>
<tr>
<td>7</td>
<td>unknown 127.0.0.1:1407 to_name UnBoundThread</td>
</tr>
<tr>
<td>6</td>
<td>unknown 127.0.0.1:1407 bind UnBoundThread</td>
</tr>
<tr>
<td>4</td>
<td>IDL:Comp_5:1.0 127.0.0.1:1408 Comp_5_Meth_2</td>
</tr>
<tr>
<td>2</td>
<td>unknown 127.0.0.1:1407 to_name IDL:Comp_1:1.0 Comp_1_Meth_1</td>
</tr>
<tr>
<td>2</td>
<td>unknown 127.0.0.1:1407 resolve IDL:Comp_1:1.0 Comp_1_Meth_1</td>
</tr>
<tr>
<td>1</td>
<td>unknown 127.0.0.1:1407 is_a UnBoundThread</td>
</tr>
<tr>
<td>1</td>
<td>unknown 127.0.0.1:1407 to_name IDL:Comp_2:1.0 Comp_2_Meth_1</td>
</tr>
<tr>
<td>1</td>
<td>unknown 127.0.0.1:1407 resolve IDL:Comp_2:1.0 Comp_2_Meth_1</td>
</tr>
<tr>
<td>1</td>
<td>unknown 127.0.0.1:1407 to_name IDL:Comp_3:1.0 Comp_3_Meth_1</td>
</tr>
<tr>
<td>1</td>
<td>unknown 127.0.0.1:1407 resolve IDL:Comp_3:1.0 Comp_3_Meth_1</td>
</tr>
<tr>
<td>1</td>
<td>unknown 127.0.0.1:1407 to_name IDL:Comp_3:1.0 Comp_3_Meth_2</td>
</tr>
<tr>
<td>1</td>
<td>unknown 127.0.0.1:1407 resolve IDL:Comp_3:1.0 Comp_3_Meth_2</td>
</tr>
<tr>
<td>1</td>
<td>unknown 127.0.0.1:1407 to_name IDL:Comp_4:1.0 Comp_4_Meth_1</td>
</tr>
<tr>
<td>1</td>
<td>unknown 127.0.0.1:1407 resolve IDL:Comp_4:1.0 Comp_4_Meth_1</td>
</tr>
</tbody>
</table>

Figure 10: An example of the presentation of a call statistics per node.

The fourth presentation is call statistics summarized on node level. It shows how many calls that have been done between two nodes. It also shows the summary of all calls done within a node see figure 11. This presentation has been constructed for giving hints on what distribution will give the best performance. All three call-statistic presentations have been implemented in a similar way. The difference is in what level the summery of the calls will be performed. This presentation will be written to a file called nodeStatistics.txt. This kind of calculation becomes misleading for the calls inside a node. If several components are placed within the node and they are calling each other some execution times will be double registered. This phenomenon doesn’t arise for the latency only for the total call time. This presentation should be mainly used for getting an overview of the inter node communication.

<table>
<thead>
<tr>
<th>count</th>
<th>callee caller time latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>459</td>
<td>Server1 127.0.0.1:1408 Server1 127.0.0.1:1408 459770 70190</td>
</tr>
<tr>
<td>112</td>
<td>Server1 127.0.0.1:1408 Client1 222290 8830</td>
</tr>
<tr>
<td>27</td>
<td>unknown 127.0.0.1:1407 Server1 127.0.0.1:1408 2750 unknown</td>
</tr>
<tr>
<td>3</td>
<td>unknown 127.0.0.1:1407 Client1 660 unknown</td>
</tr>
</tbody>
</table>

Figure 11: An example of the presentation of a node statistics.

This last implemented presentation shows the summarized execution paths from one method. The presentation program walks all paths evolved from the start method and summarizes the data. The information is presented as a callgraph see Figure 12. This presentations shows of where calls through the start method spends it time and what is the usual path for this method. This presentation is done for all executed distributed methods and is written to a file named CallGraphTyp2.txt. If the interface of the called method is unknown the method is described as address of the component. The interface is unknown if the target node for a call isn’t instrumented and due to that isn’t it possible to calculate the effective time of the execution.
The merging and presentation program is constructed so that it is easy to implement new calculations and presentations for further measurements and experiments.

4.3.4 Tools used for implementation.

No commercial development environment has been used. The source code has been written in an ordinary text editor. For managing compiling and distribution Ant has been used. Ant is a free compiling tool that can be obtained at http://jakarta.apache.org. Ant uses an xml file for configuring compilation instructions. It can call other programs as the IDL-compiler. Ant has been very helpful in the handling of all compilation phases and distribution handling.

4.4 Running the profiling-tool.

This section explains how the tool is hooked on to a node and what properties that have to be configured.

Configuration of the instrumentation is done using Java system properties. The easiest way to hook on and configure the instrumentation is to add following on the command line when starting the application-node that shall be profiled presented in figure 13.

```
-DInstrumentLogFile=ServerLog.txt -DInstrumentNodeName=Server1
```

The property “org.omg.PortableInterceptor.ORBInitializerClass” tells the ORB to hook on the interceptors. And the second property “InstrumentLogFile” configure the name of the log file for the current node. The third property “InstrumentNodeName” is for give the node a name. This property isn’t necessary since the node will get a auto generated name if no name is given.

The applications are often started by using some kind of script as windows bat-file or Unix shellscript. All that has to be done to add the instrumentation is putting the instrument.jar file in the classpath and adding the three properties explained earlier.
After the execution of the system is done all data-files are moved to a directory for information processing. The information processing and presentation is run with following command presented in figure 14.

```
java exam.profTool.Runner c:\exam\datafiles
```

*Figure 14: The commando for starting the information handling and the presentation of the result of the execution of a distributed system.*

The parameter passed to the program is the directory for the collected log files from the instrumentation. The run of the program produces five files containing the presentations (callGraphTyp1.txt, callGraphTyp2.txt, callStatistics.txt, callStatisticsPerNode.txt, nodeStatistics.txt), one containing a printout of the entire tree (tree.txt), and one containing the internal representation in a sterilized format (distrSystem.dc). The file distrSystem.dc can be used ass in parameter for other programs performing calculations on the current execution of the distributed system.

## 5 Experimental framework

For answering the questions discussed earlier in this report an experiment environment for evaluating the tool has to be constructed. This section explains how and why the test environment is constructed. It will discuss the design decisions taken for constructing the test application. The purpose of the test environment is to test the profiling tool. To test if the principles of the profiling tool is the right for profiling a distributed application. Due to the lack of time and the access to a real application the first test of the instrumentation is run on a fictive application.

For modeling a distributed application a set of components have to be constructed. The components shall use methods on each other. All intercomponent calls shall be done by CORBA and the components shall be distributed over several nodes. A server framework for registering the components on has to be developed and a client to trigger the requests in the distributed system has to be constructed. The ORB used for the test environment is JacORB described earlier.

### 5.1 Server

The server has to be implemented in such way that it is easy to distribute the components on different JVM or different machines. An activation and registration framework has to be design see figure 15.

*Figure 15: The activation and registration design of the test environment*
Server principle is as follows. The server class instantiates the ORB and tell the ComponentRegister to register the distributed objects. The ComponentRegister reads a properties-file containing a list of the components that shall be running on this server. It instantiates a ComponentService for every component on the list. The ComponentService instantiates the component and register it to the POA and the naming service. This framework makes it possible to distribute the component on different servers by only changing the list of components in the properties-file.

5.2 Components

The components shall model real components in a real distributed application. The components shall each be an implementation of an IDL interface. The test application is composed of 5 components. The components are both client and servant, and use other components for solving the tasks. The components are named comp_1 – comp_5. The components dependencies are shown in figure 16.

![Figure 16: The dependencies of components in the test environment.](image)

For simulating a real application are the outcome of the distributed methods random based on a random number generation and a percentage give to the different outcomes. The way this is coded is shown in figure 17.

```java
int random = (int)(Math.random() * 100);
if(random>0 && random<=50)
{
    getComp_2().Comp_2_Meth_1("tjingeling parameter message");
    return "result from method_1";
}
if(random>50 && random<=80)
{
    getComp_3().Comp_3_Meth_1("tjingeling parameter message");
    return "result from method_1";
}
if(random>80 && random<=100)
{
    try
    {
        thisThread.sleep(300);
    }
    catch(InterruptedException e)
    {
        System.err.println("exception "+e);
    }
    return "result from method_1";
}
```

![Figure 17: A part of the code of Comp_1 that shows the principle for random outcome of the distributed methods.](image)

The percentage and the different outcomes are shown in the description for the specific component made below. The references to other components are made with the help of a
factory method. This makes the instantiation and naming-service advertising order unimportant as long as the needed component is alive and kicking when asked for.

In Appendix C, the design of the components is described.

5.3 Client

This section describes the client in the test environment. It also describes the purpose of the client and how to operate it. A client has to be constructed for trigging the requests in the application. A new version of the client is implemented for every experiment. The different client implementations will be discussed in the description of the actual experiment that it is used in.

5.4 Running the experiment

The purpose is to evaluate the tool not to test a real application. This section will describe several test scenarios. It will explain how the test is to be run and what modifications that has been done in the environment, the instrumentation and the client.

5.4.1 Tool overhead, experiment 1

This test is called experiment 1, and measures the tool overhead. The test is performed by making request to the same server started with the instrumentation hooked on and without the instrumentation on. There are three different degrees of instrumentation tested one with client and server instrumented, one with only the client instrumented and one with no instrumentation at all. There is no need for measuring the network latency in this test. The application is configured with all server components on the same server and a client making the request on the same machine. The server components have to be modified for this test. The random execution has to be removed. This is done by letting the first condition always be true. The dependences between methods for this experiment are as shown in table 8.

<table>
<thead>
<tr>
<th>Caller</th>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp_1_Meth_1</td>
<td>Comp_2_Meth_1</td>
</tr>
<tr>
<td>Comp_2_Meth_1</td>
<td>Comp_3_Meth_1 and Comp_3_Meth_2</td>
</tr>
<tr>
<td>Comp_3_Meth_1</td>
<td>Comp_4_Meth_1</td>
</tr>
<tr>
<td>Comp_3_Meth_2</td>
<td>Comp_5_Meth_1</td>
</tr>
<tr>
<td>Comp_4_Meth_1</td>
<td>2*Comp5_Meth_1</td>
</tr>
</tbody>
</table>

*Table 8: Dependences in the tool overhead test*

The first call made isn’t a part of the test because it involves a lot of factory methods and references lookup. The first call has to contact the naming service for getting a reference to the remote object. It will have a much higher latency. The client has a timing function coded into it requests. The client makes calls on the method Comp_1_Meth_1 on the component Comp_1. First it makes two calls that are timed individually. Then it makes 10 calls that are timed together and finely it makes 100 calls that are timed together. The test is run three times on every degree of instrumentation. All three tests are done on the same instance of the server this means that the server will only be restarted for changing the instrumentation. The Experiment is run on a machine with a Pentium II CPU and 320 MB RAM. The operation system used is Windows 98.

5.4.2 Profiling with simulated methods latency

The purpose of this test is to test if the instrumentation works in an application with different nodes. It also tests if the information processing and presentation works with different nodes. A client will make 100 calls on the method Comp_1_Meth_1 on the component Comp_1. The log files will be collected and parsed with the merging and
presentation program. The experiment will be run on three different distributions shown in table 9.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Node 1</th>
<th>Node 2</th>
<th>Node 3</th>
<th>Node 4</th>
<th>Node 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Comp [1,2,3,4,5]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 9: The different distributions in experiment 2 –4.

All nodes are run on the same machine. The machine used is running windows 98 and it has a Pentium II CPU and 320 MB RAM.

5.4.3 Profiling in a distributed environment

The purpose of this experiment is to test if the instrumentation and presentation can function in a distributed system in a real distributed environment. The nodes are on one machine each connected to a 100 MBit LAN. The operation system of the machines used is Windows NT. The machines used are Pentium III 660 Hz with 128 RAM. In a real application should the network used probably be the Internet.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Node 1</th>
<th>Node 2</th>
<th>Node 3</th>
<th>Node 4</th>
<th>Node 5</th>
</tr>
</thead>
</table>

Table 10: The distribution of components in experiment 5.

The naming service will be placed on the same computer as node 1 and the client will placed on a sixth computer. The client will make 100 calls on the method Comp_1_Meth_1 on the component Comp_1.
6 Result

6.1 Experiment 1
The result of the performed tests in experiment 1 is presented in Table 11. The result indicates that the tool has no registerable overhead in the test application. If there is an overhead it is lower when one percent. The instrumentation doesn’t excessively distort the performance profile.

<table>
<thead>
<tr>
<th>Number of requests</th>
<th>Instrumentation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Client and Server</td>
</tr>
<tr>
<td>First call</td>
<td>2470 2090 2140</td>
</tr>
<tr>
<td>10</td>
<td>20210 19990 20100</td>
</tr>
<tr>
<td>100</td>
<td>200860 201580 201140</td>
</tr>
</tbody>
</table>

Table 11: The result of experiment 1.

6.2 Experiment 2
The result of the experiment is presented as the all five presentations implemented. The result shown is one of several calls from the client to the method Comp_1_Meth_1. Due to the random outcome of the methods will the callgraphs differ and it will be one per client call. In figure 18 is one of the callgraph presented, it shows that the instrumentation can capture the calls and register the time spent.

```
Call Comp_1_Meth_1 on node 127.0.0.1:1591 time 2740 network latency 50
Execution Comp_1_Meth_1 on IDL:Comp_1:1.0 time 2680 effective time 60
Call Comp_2_Meth_1 on node 127.0.0.1:1591 time 2630 network latency 220
Execution Comp_2_Meth_1 on IDL:Comp_2:1.0 time 2410 effective time 110
Call Comp_3_Meth_1 on node 127.0.0.1:1591 time 1700 network latency 220
  Execution Comp_3_Meth_1 on IDL:Comp_3:1.0 time 1480 effective time 160
Call Comp_4_Meth_1 on node 127.0.0.1:1591 time 660 network latency 110
  Execution Comp_4_Meth_1 on IDL:Comp_4:1.0 time 550 effective time 110
Call Comp_5_Meth_1 on node 127.0.0.1:1591 time 220 network latency 0
  Execution Comp_5_Meth_1 on IDL:Comp_5:1.0 time 220 effective time 220
End execution
Call Comp_5_Meth_1 on node 127.0.0.1:1591 time 220 network latency 0
  Execution Comp_5_Meth_1 on IDL:Comp_5:1.0 time 220 effective time 220
End execution
End execution
Call Comp_4_Meth_1 on node 127.0.0.1:1591 time 660 network latency 170
  Execution Comp_4_Meth_1 on IDL:Comp_4:1.0 time 490 effective time 110
Call Comp_5_Meth_1 on node 127.0.0.1:1591 time 170 network latency 0
  Execution Comp_5_Meth_1 on IDL:Comp_5:1.0 time 170 effective time 170
End execution
Call Comp_5_Meth_1 on node 127.0.0.1:1591 time 210 network latency 0
  Execution Comp_5_Meth_1 on IDL:Comp_5:1.0 time 210 effective time 210
End execution
End execution
End execution
Call Comp_3_Meth_2 on node 127.0.0.1:1591 time 600 network latency 160
  Execution Comp_3_Meth_2 on IDL:Comp_3:1.0 time 440 effective time 270
Call Comp_5_Meth_1 on node 127.0.0.1:1591 time 170 network latency 0
  Execution Comp_5_Meth_1 on IDL:Comp_5:1.0 time 170 effective time 170
End execution
End execution
End execution
End execution
End execution
```

Figure 18: The result of experiment 2 presented as callgraph type1

The first line represents the initiating call from the client. If the target node is instrumented the execution of the called method will be registered as in line 2 in figure 18. If the execution of the method involves new distributed calls it will be written with the same
indentation as in line 3 in figure 18. The call graph in figure 18 shows that the instrumentation works for several components.

For getting a better overview of the execution of the application a summarized call statistics is presented in figure 19.

<table>
<thead>
<tr>
<th>count</th>
<th>callee</th>
<th>caller</th>
<th>time</th>
<th>latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>166</td>
<td>IDL:Comp_5:1.0 Comp_5_Meth_1</td>
<td>IDL:Comp_4:1.0 Comp_4_Meth_1</td>
<td>34170</td>
<td>550</td>
</tr>
<tr>
<td>100</td>
<td>IDL:Comp_1:1.0 Comp_1_Meth_1</td>
<td>UnBoundThread</td>
<td>135060</td>
<td>600</td>
</tr>
<tr>
<td>83</td>
<td>IDL:Comp_4:1.0 Comp_4_Meth_1</td>
<td>IDL:Comp_3:1.0 Comp_3_Meth_1</td>
<td>55300</td>
<td>12390</td>
</tr>
<tr>
<td>50</td>
<td>IDL:Comp_2:1.0 Comp_2_Meth_1</td>
<td>IDL:Comp_1:1.0 Comp_1_Meth_1</td>
<td>88630</td>
<td>9720</td>
</tr>
<tr>
<td>42</td>
<td>IDL:Comp_3:1.0 Comp_3_Meth_1</td>
<td>IDL:Comp_2:1.0 Comp_2_Meth_1</td>
<td>49860</td>
<td>8280</td>
</tr>
<tr>
<td>36</td>
<td>IDL:Comp_5:1.0 Comp_5_Meth_1</td>
<td>IDL:Comp_3:1.0 Comp_3_Meth_1</td>
<td>7240</td>
<td>50</td>
</tr>
<tr>
<td>36</td>
<td>IDL:Comp_3:1.0 Comp_3_Meth_1</td>
<td>IDL:Comp_2:1.0 Comp_2_Meth_1</td>
<td>21480</td>
<td>5040</td>
</tr>
<tr>
<td>29</td>
<td>IDL:Comp_3:1.0 Comp_3_Meth_1</td>
<td>IDL:Comp_1:1.0 Comp_1_Meth_1</td>
<td>32000</td>
<td>5720</td>
</tr>
<tr>
<td>9</td>
<td>IDL:Comp_5:1.0 Comp_5_Meth_1</td>
<td>IDL:Comp_3:1.0 Comp_3_Meth_1</td>
<td>1800</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>unknown 127.0.0.1:1590 to_name</td>
<td>UnBoundThread</td>
<td>110 unknown</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>unknown 127.0.0.1:1590 bind</td>
<td>UnBoundThread</td>
<td>0 unknown</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>unknown 127.0.0.1:1590 _is_a</td>
<td>UnBoundThread</td>
<td>440 unknown</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>unknown 127.0.0.1:1590 to_name</td>
<td>IDL:Comp_1:1.0 Comp_1_Meth_1</td>
<td>0 unknown</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>unknown 127.0.0.1:1590 resolve</td>
<td>IDL:Comp_1:1.0 Comp_1_Meth_1</td>
<td>60 unknown</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>unknown 127.0.0.1:1590 to_name</td>
<td>IDL:Comp_3:1.0 Comp_3_Meth_1</td>
<td>0 unknown</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>unknown 127.0.0.1:1590 resolve</td>
<td>IDL:Comp_3:1.0 Comp_3_Meth_1</td>
<td>60 unknown</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>unknown 127.0.0.1:1590 to_name</td>
<td>IDL:Comp_4:1.0 Comp_4_Meth_1</td>
<td>50 unknown</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>unknown 127.0.0.1:1590 resolve</td>
<td>IDL:Comp_4:1.0 Comp_4_Meth_1</td>
<td>0 unknown</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>unknown 127.0.0.1:1590 to_name</td>
<td>IDL:Comp_2:1.0 Comp_2_Meth_1</td>
<td>50 unknown</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>unknown 127.0.0.1:1590 resolve</td>
<td>UnBoundThread</td>
<td>170 unknown</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 19: The result of experiment 2 presented as call statistics**

The call statistics gives information about what method that has been called and what time spent in them. In line 3 in figure 19 is the summery of the calls made from the client to the first component. The time this call spent is near the total time spent in the execution of the application. The total time for the execution was according to figure 19 135060 ms.. The caller UnBoundThread is the Client in this application. If it hade been several client all hade been registered under unbound thread in this execution. The callee in unknown 127.0.0.1:1590 is the naming service. The naming service isn’t instrumented.

In figure 20 is the same information as in figure 19 but with a higher resolution, and it is divided per node. As seen in figure 20, there is only two instrumented nodes in this execution of the system “Node Server1 127.0.0.1:1591” and “Node Client2 “.
== Node Server1 127.0.0.1:1591 ==

count callee caller time latency
166 IDL:Comp_5:1.0 127.0.0.1:1591 Comp_5_Meth_1 IDL:Comp_4:1.0 Comp_4_Meth_1 34170 550
83 IDL:Comp_4:1.0 127.0.0.1:1591 Comp_4_Meth_1 IDL:Comp_3:1.0 Comp_3_Meth_1 55300 12390
50 IDL:Comp_2:1.0 127.0.0.1:1591 Comp_2_Meth_1 IDL:Comp_1:1.0 Comp_1_Meth_1 88630 9720
42 IDL:Comp_3:1.0 127.0.0.1:1591 Comp_3_Meth_1 IDL:Comp_2:1.0 Comp_2_Meth_1 49860 8280
36 IDL:Comp_5:1.0 127.0.0.1:1591 Comp_5_Meth_1 IDL:Comp_3:1.0 Comp_3_Meth_1 7240 50
36 IDL:Comp_3:1.0 127.0.0.1:1591 Comp_3_Meth_1 IDL:Comp_2:1.0 Comp_2_Meth_1 21480 5040
29 IDL:Comp_3:1.0 127.0.0.1:1591 Comp_3_Meth_1 IDL:Comp_1:1.0 Comp_1_Meth_1 32000 5720
9 IDL:Comp_5:1.0 127.0.0.1:1591 Comp_5_Meth_1 IDL:Comp_3:1.0 Comp_3_Meth_1 1800 0
7 unknown 127.0.0.1:1590 to_name UnBoundThread 110 unknown
7 unknown 127.0.0.1:1590 bind UnBoundThread 0 unknown
2 unknown 127.0.0.1:1590 to_name IDL:Comp_1:1.0 Comp_1_Meth_1 0 unknown
2 unknown 127.0.0.1:1590 resolve IDL:Comp_1:1.0 Comp_1_Meth_1 60 unknown
2 unknown 127.0.0.1:1590 to_name IDL:Comp_3:1.0 Comp_3_Meth_1 0 unknown
2 unknown 127.0.0.1:1590 resolve IDL:Comp_3:1.0 Comp_3_Meth_1 60 unknown
1 unknown 127.0.0.1:1590 _is_a UnBoundThread 330 unknown
1 unknown 127.0.0.1:1590 to_name IDL:Comp_4:1.0 Comp_4_Meth_1 50 unknown
1 unknown 127.0.0.1:1590 resolve IDL:Comp_4:1.0 Comp_4_Meth_1 0 unknown
1 unknown 127.0.0.1:1590 to_name IDL:Comp_2:1.0 Comp_2_Meth_1 50 unknown
1 unknown 127.0.0.1:1590 resolve IDL:Comp_2:1.0 Comp_2_Meth_1 0 unknown

Figure 20: The result of experiment 2 presented as call statistics per node.

For getting a macro view of the execution of the application a call statistics is made summarized on the calls made between nodes and within nodes see figure 21. This kind of presentation doesn’t give much value then the application isn’t distributed over several servers. The result in figure 21 shows that much time is spent in internal latency with in the server “ Server1 127.0.0.1:1591”.

== Node Client2 ==
count callee caller time latency
100 IDL:Comp_1:1.0 127.0.0.1:1591 Comp_1_Meth_1 UnBoundThread 135060 600
1 unknown 127.0.0.1:1590 _is_a UnBoundThread 110 unknown
1 unknown 127.0.0.1:1590 to_name UnBoundThread 0 unknown
1 unknown 127.0.0.1:1590 resolve UnBoundThread 170 unknown

Figure 21: The result of experiment 2 presented as call statistics between nodes.

An example of the summarized callgraph is presented in figure 22. The call graph chosen is for method Comp_1_Meth_1 on component Comp_1 see Appendix C for further description of the components. All execution paths in experiment 2 runs through method Comp_1_Meth_1 so the graph in figure 22 gives a overview of all execution paths in the current run of the application.
The results presented here shows that the instrumentation works and gives valuable information for a distributed system based on several components on a single server.

### 6.3 Experiment 3

The result of experiment 3 is presented here and examples of all the five different presentations are included here in figure 23 to figure 27.

The result in figure 23 is one of several callgraph produced by the presentation program. The result shows that the tool can register and trace calls and executions in an application consisted by several servers, three in this experiment. The different servers are presented as different addresses in the Call lines. As example is Node1 represented by the address “127.0.0.1:1617” and Node4 as “127.0.0.1:1621”.

```
<table>
<thead>
<tr>
<th>Call</th>
<th>Method</th>
<th>Node Address</th>
<th>Time (ms)</th>
<th>Network Latency (ms)</th>
<th>Execution Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call Comp_1_Meth_1 on node 127.0.0.1:1617</td>
<td>time 1540</td>
<td>60</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution Comp_1_Meth_1 on IDL:Comp_1:1.0</td>
<td>time 1480</td>
<td>effective time 50</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Call Comp_2_Meth_1 on node 127.0.0.1:1617</td>
<td>time 1270</td>
<td>effective time 110</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Call Comp_3_Meth_1 on node 127.0.0.1:1617</td>
<td>time 660</td>
<td>effective time 110</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Call Comp_4_Meth_1 on node 127.0.0.1:1621</td>
<td>time 550</td>
<td>network latency 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution Comp_4_Meth_1 on IDL:Comp_4:1.0</td>
<td>time 550</td>
<td>effective time 110</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Call Comp_5_Meth_1 on node 127.0.0.1:1619</td>
<td>time 220</td>
<td>network latency 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution Comp_5_Meth_1 on IDL:Comp_5:1.0</td>
<td>time 220</td>
<td>effective time 220</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>End execution</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Call Comp_5_Meth_1 on node 127.0.0.1:1621</td>
<td>time 220</td>
<td>network latency 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution Comp_5_Meth_1 on IDL:Comp_5:1.0</td>
<td>time 220</td>
<td>effective time 220</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 23: The result of experiment 3 presented as callgraph type1.

The next result presentation in figure 24 shows the overall call statistics of the execution of the application. This result is very similar to the result of experiment 2. The difference is in the latency times and in the number of calls on each method due to the random behavior coded in the components.
In figure 25 is the call statistics per node represented. This presentation gives a rough picture of the amount of work performed on each node. It shows the number of call initiated from the node and their execution time and latency’s. This figures can be compared between the different nodes for getting a estimation of the load on different nodes.
The next presentation figure 26 is call statistics based on communications between nodes. The result presented shows that the latency is greater for calls between distributed components on the same server than between components on different servers. Calls made within Node1 had an average latency of \( \frac{7070}{142} = 49 \) ms and calls between Node1 and Node4 had an average latency of \( \frac{500}{66} = 7.5 \) ms. This result indicates that the thread pool and the handling of internal distributed calls isn’t optimized in the used CORBA implementation.

\[
\begin{array}{c|c|c|c|c|c}
\text{count} & \text{callee} & \text{caller} & \text{time} & \text{latency} \\
66 & IDL:Comp_4:1.0:1621 Comp_4_Meth_1 & IDL:Comp_3:1.0 Comp_3_Meth_1 & 35540 & 500 \\
45 & IDL:Comp_2:1.0:1617 Comp_2_Meth_1 & IDL:Comp_1:1.0 Comp_1_Meth_1 & 58930 & 6610 \\
39 & IDL:Comp_3:1.0:1617 Comp_3_Meth_1 & IDL:Comp_2:1.0 Comp_2_Meth_1 & 30430 & 60 \\
33 & IDL:Comp_5:1.0:1621 Comp_5_Meth_1 & IDL:Comp_3:1.0 Comp_3_Meth_2 & 6930 & 340 \\
33 & IDL:Comp_3:1.0:1621 Comp_3_Meth_1 & IDL:Comp_2:1.0 Comp_2_Meth_1 & 15470 & 290 \\
13 & IDL:Comp_5:1.0:1621 Comp_5_Meth_1 & IDL:Comp_3:1.0 Comp_3_Meth_1 & 2510 & 0 \\
3 & unknown 127.0.0.1:1616 to_name UnBoundThread & 60 & unknown \\
2 & unknown 127.0.0.1:1616 bind UnBoundThread & 100 & unknown \\
1 & unknown 127.0.0.1:1616 to_name IDL:Comp_1:1.0 Comp_1_Meth_1 & 0 & unknown \\
1 & unknown 127.0.0.1:1616 resolve IDL:Comp_1:1.0 Comp_1_Meth_1 & 50 & unknown \\
1 & unknown 127.0.0.1:1616 to_name IDL:Comp_3:1.0 Comp_3_Meth_1 & 0 & unknown \\
1 & unknown 127.0.0.1:1616 resolve IDL:Comp_3:1.0 Comp_3_Meth_1 & 170 & unknown \\
1 & unknown 127.0.0.1:1616 to_name IDL:Comp_3:1.0 Comp_3_Meth_2 & 0 & unknown \\
1 & unknown 127.0.0.1:1616 resolve IDL:Comp_3:1.0 Comp_3_Meth_2 & 0 & unknown \\
1 & unknown 127.0.0.1:1616 to_name IDL:Comp_2:1.0 Comp_2_Meth_1 & 50 & unknown \\
1 & unknown 127.0.0.1:1616 resolve IDL:Comp_2:1.0 Comp_2_Meth_1 & 0 & unknown \\
\end{array}
\]

Figure 25: The result of experiment 3 presented as call statistics per node.

Figure 26: The result of experiment 3 presented as call statistics between nodes.
In figure 27 is the summarized call graph of method Comp_1_Meth_1 on component Comp_1 presented. As in experiment 2 is all calls initiated through this method and due to that becomes this call graph a summary of all calls in the execution of the system.

<table>
<thead>
<tr>
<th>IDL: Comp_1:1.0 Comp_1_Meth_1</th>
<th>Execution time: 92810</th>
<th>Effective time: 15910</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 unknown 127.0.0.1:1616 to_name</td>
<td>Execution time: 0</td>
<td>Effective time: unknown</td>
</tr>
<tr>
<td>2 unknown 127.0.0.1:1616 resolve</td>
<td>Execution time: 50</td>
<td>Effective time: unknown</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IDL: Comp_3:1.0 Comp_3_Meth_1</th>
<th>Execution time: 17810</th>
<th>Effective time: 3720</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 unknown 127.0.0.1:1616 to_name</td>
<td>Execution time: 0</td>
<td>Effective time: unknown</td>
</tr>
<tr>
<td>1 unknown 127.0.0.1:1616 resolve</td>
<td>Execution time: 170</td>
<td>Effective time: unknown</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IDL: Comp_4:1.0 Comp_4_Meth_1</th>
<th>Execution time: 12870</th>
<th>Effective time: 2730</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 unknown 127.0.0.1:1616 to_name</td>
<td>Execution time: 0</td>
<td>Effective time: unknown</td>
</tr>
<tr>
<td>1 unknown 127.0.0.1:1616 resolve</td>
<td>Execution time: 12870</td>
<td>Effective time: 2730</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IDL: Comp_5:1.0 Comp_5_Meth_1</th>
<th>Execution time: 9590</th>
<th>Effective time: 9590</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 IDL: Comp_5:1.0 Comp_5_Meth_1</td>
<td>Execution time: 770</td>
<td>Effective time: 770</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IDL: Comp_2:1.0 Comp_2_Meth_1</th>
<th>Execution time: 52320</th>
<th>Effective time: 6370</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 unknown 127.0.0.1:1616 to_name</td>
<td>Execution time: 50</td>
<td>Effective time: unknown</td>
</tr>
<tr>
<td>1 unknown 127.0.0.1:1616 resolve</td>
<td>Execution time: 0</td>
<td>Effective time: unknown</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IDL: Comp_3:1.0 Comp_3_Meth_1</th>
<th>Execution time: 30370</th>
<th>Effective time: 6240</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 unknown 127.0.0.1:1616 to_name</td>
<td>Execution time: 0</td>
<td>Effective time: unknown</td>
</tr>
<tr>
<td>1 unknown 127.0.0.1:1616 resolve</td>
<td>Execution time: 30370</td>
<td>Effective time: 6240</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IDL: Comp_4:1.0 Comp_4_Meth_1</th>
<th>Execution time: 22170</th>
<th>Effective time: 4560</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 IDL: Comp_5:1.0 Comp_5_Meth_1</td>
<td>Execution time: 1740</td>
<td>Effective time: 1740</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IDL: Comp_5:1.0 Comp_5_Meth_1</th>
<th>Execution time: 17320</th>
<th>Effective time: 17320</th>
</tr>
</thead>
<tbody>
<tr>
<td>33 IDL: Comp_5:1.0 Comp_5_Meth_1</td>
<td>Execution time: 6590</td>
<td>Effective time: 6590</td>
</tr>
</tbody>
</table>

| IDL: Comp_5:1.0 Comp_5_Meth_1 | Execution time: 6590 | Effective time: 6590 |

Figure 27: The result of experiment 3 presented as call graph type 2.

The results presented here of experiment 3 shows that the instrumentation works and gives valuable information for a distributed system based on several components on several servers. Due to the random behavior built in to the components is it possible to do exact comparison with experiment 2. But the experiment indicates that it is better performance with a more distributed application over several servers. One of the indications is that the overall execution time for the calls from the client to the server is lower for experiment 3 than experiment 2. This time is found in line three in figure 24 (93540 ms) and in line three in figure 19 (135060 ms). This is very surprising, that the treadling pool and the internal distributed call of the application has a lower performance then calling another server.
6.4 Experiment 4

The result of experiment 4 is presented here and examples of all the five different presentations are included here in figure 28 to figure 32.

Call Comp_1_Meth_1 on node 127.0.0.1:1029 time 1380 network latency 0
Execution Comp_1_Meth_1 on IDL:Comp_1:1.0 time 1380 effective time 60
Call Comp_2_Meth_1 on node 127.0.0.1:1031 time 1320 network latency 0
Execution Comp_2_Meth_1 on IDL:Comp_2:1.0 time 1320 effective time 160
Call Comp_3_Meth_1 on node 127.0.0.1:1033 time 660 network latency 0
  Execution Comp_3_Meth_1 on IDL:Comp_3:1.0 time 660 effective time 170
Call Comp_4_Meth_1 on node 127.0.0.1:1035 time 490 network latency 0
  Execution Comp_4_Meth_1 on IDL:Comp_4:1.0 time 490 effective time 110
Call Comp_5_Meth_1 on node 127.0.0.1:1037 time 160 network latency 0
  Execution Comp_5_Meth_1 on IDL:Comp_5:1.0 time 160 effective time 160
End execution
Call Comp_5_Meth_1 on node 127.0.0.1:1037 time 220 network latency 0
  Execution Comp_5_Meth_1 on IDL:Comp_5:1.0 time 220 effective time 220
End execution
End execution
End execution
End execution
Call Comp_3_Meth_2 on node 127.0.0.1:1033 time 500 network latency 60
Execution Comp_3_Meth_2 on IDL:Comp_3:1.0 time 440 effective time 220
Call Comp_5_Meth_1 on node 127.0.0.1:1037 time 220 network latency 0
  Execution Comp_5_Meth_1 on IDL:Comp_5:1.0 time 220 effective time 220
End execution
End execution
End execution
End execution
End execution

Figure 28: The result of experiment 4 presented as callgraph type1.

The call graph presented in figure 28 is one of the callgraphs produced in experiment 4. The callgraph shows that the tool functions in an environment; there all components are distributed on a server each. The address field of the Call clause is different for every component showing that it resides on server each.

In figure 29 is the call statistics for experiment 4 presented. The difference in this statistics compared to experiment 2 and experiment 3 is in the latency times and in the number of calls.

<table>
<thead>
<tr>
<th>count</th>
<th>callee caller</th>
<th>time latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>120</td>
<td>IDL:Comp_1:1.0 Comp_2_Meth_1</td>
<td>24830 390</td>
</tr>
<tr>
<td>100</td>
<td>IDL:Comp_1:1.0 Comp_2_Meth_1</td>
<td>86110 430</td>
</tr>
<tr>
<td>60</td>
<td>IDL:Comp_2:1.0 Comp_3_Meth_1</td>
<td>32080 270</td>
</tr>
<tr>
<td>49</td>
<td>IDL:Comp_3:1.0 Comp_4_Meth_1</td>
<td>50950 360</td>
</tr>
<tr>
<td>35</td>
<td>IDL:Comp_4:1.0 Comp_5_Meth_1</td>
<td>16720 380</td>
</tr>
<tr>
<td>33</td>
<td>IDL:Comp_5:1.0 Comp_6_Meth_1</td>
<td>7180 0</td>
</tr>
<tr>
<td>24</td>
<td>IDL:Comp_6:1.0 Comp_7_Meth_1</td>
<td>25960 240</td>
</tr>
<tr>
<td>14</td>
<td>IDL:Comp_7:1.0 Comp_8_Meth_1</td>
<td>18350 50</td>
</tr>
<tr>
<td>6</td>
<td>unknown 127.0.0.1:1028 is_a</td>
<td>1150 unknown</td>
</tr>
<tr>
<td>6</td>
<td>unknown 127.0.0.1:1028 to_name</td>
<td>60 unknown</td>
</tr>
<tr>
<td>5</td>
<td>unknown 127.0.0.1:1028 bind</td>
<td>110 unknown</td>
</tr>
<tr>
<td>2</td>
<td>unknown 127.0.0.1:1028 to_name</td>
<td>110 unknown</td>
</tr>
<tr>
<td>2</td>
<td>unknown 127.0.0.1:1028 resolve</td>
<td>170 unknown</td>
</tr>
<tr>
<td>1</td>
<td>unknown 127.0.0.1:1028 resolve</td>
<td>220 unknown</td>
</tr>
<tr>
<td>1</td>
<td>unknown 127.0.0.1:1028 to_name</td>
<td>60 unknown</td>
</tr>
<tr>
<td>1</td>
<td>unknown 127.0.0.1:1028 resolve</td>
<td>160 unknown</td>
</tr>
<tr>
<td>1</td>
<td>unknown 127.0.0.1:1028 to_name</td>
<td>50 unknown</td>
</tr>
<tr>
<td>1</td>
<td>unknown 127.0.0.1:1028 resolve</td>
<td>170 unknown</td>
</tr>
<tr>
<td>1</td>
<td>unknown 127.0.0.1:1028 to_name</td>
<td>0 unknown</td>
</tr>
<tr>
<td>1</td>
<td>unknown 127.0.0.1:1028 resolve</td>
<td>60 unknown</td>
</tr>
<tr>
<td>1</td>
<td>unknown 127.0.0.1:1028 to_name</td>
<td>0 unknown</td>
</tr>
<tr>
<td>1</td>
<td>unknown 127.0.0.1:1028 resolve</td>
<td>220 unknown</td>
</tr>
</tbody>
</table>

Figure 29: The result of experiment 4 presented as some call statistic.
The next presentation in figure 30 is call statistics per node. This presentation purpose is to show if there is different amount of workload on the different servers. This presentation doesn’t add more value to experiment 4 when some further proof of the functionalities of the instrumentation in a distributed environment.

<table>
<thead>
<tr>
<th>Node</th>
<th>Call Statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client2</td>
<td>count callee caller time latency</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>unknown</td>
</tr>
<tr>
<td>1</td>
<td>unknown</td>
</tr>
<tr>
<td>1</td>
<td>unknown</td>
</tr>
</tbody>
</table>

| Node1 | count callee caller time latency |
| 46 | IDL:Comp_2:1.0 | 127.0.0.1:1031 | Comp_2_Meth_1 | IDL:Comp_1:1.0 | Comp_1_Meth_1 | 50950 360 |
| 24 | unknown | 127.0.0.1:1028 | to_name | IDL:Comp_1:1.0 | Comp_1_Meth_1 | 110 unknown |
| 2 | unknown | 127.0.0.1:1028 | resolve | IDL:Comp_1:1.0 | Comp_1_Meth_1 | 170 unknown |
| 1 | unknown | 127.0.0.1:1028 | _is_a | UnBoundThread | 600 unknown |
| 1 | unknown | 127.0.0.1:1028 | to_name | UnBoundThread | 60 unknown |
| 1 | unknown | 127.0.0.1:1028 | bind | UnBoundThread | 0 unknown |

| Node2 | count callee caller time latency |
| 35 | IDL:Comp_3:1.0 | 127.0.0.1:1033 | Comp_3_Meth_2 | IDL:Comp_2:1.0 | Comp_2_Meth_1 | 16720 380 |
| 33 | IDL:Comp_3:1.0 | 127.0.0.1:1033 | Comp_3_Meth_1 | IDL:Comp_2:1.0 | Comp_2_Meth_1 | 25960 240 |
| 1 | unknown | 127.0.0.1:1028 | _is_a | UnBoundThread | 110 unknown |
| 1 | unknown | 127.0.0.1:1028 | to_name | UnBoundThread | 0 unknown |
| 1 | unknown | 127.0.0.1:1028 | bind | UnBoundThread | 0 unknown |
| 1 | unknown | 127.0.0.1:1028 | to_name | IDL:Comp_2:1.0 | Comp_2_Meth_1 | 60 unknown |
| 1 | unknown | 127.0.0.1:1028 | resolve | IDL:Comp_2:1.0 | Comp_2_Meth_1 | 160 unknown |

| Node3 | count callee caller time latency |
| 60 | IDL:Comp_4:1.0 | 127.0.0.1:1035 | Comp_4_Meth_1 | IDL:Comp_3:1.0 | Comp_3_Meth_1 | 32080 270 |
| 35 | IDL:Comp_5:1.0 | 127.0.0.1:1037 | Comp_5_Meth_1 | IDL:Comp_3:1.0 | Comp_3_Meth_2 | 7180 0 |
| 14 | IDL:Comp_5:1.0 | 127.0.0.1:1037 | Comp_5_Meth_1 | IDL:Comp_3:1.0 | Comp_3_Meth_1 | 2920 60 |
| 1 | unknown | 127.0.0.1:1028 | _is_a | UnBoundThread | 110 unknown |
| 1 | unknown | 127.0.0.1:1028 | to_name | UnBoundThread | 0 unknown |
| 1 | unknown | 127.0.0.1:1028 | bind | UnBoundThread | 50 unknown |
| 1 | unknown | 127.0.0.1:1028 | to_name | IDL:Comp_3:1.0 | Comp_3_Meth_1 | 50 unknown |
| 1 | unknown | 127.0.0.1:1028 | resolve | IDL:Comp_3:1.0 | Comp_3_Meth_1 | 170 unknown |
| 1 | unknown | 127.0.0.1:1028 | to_name | IDL:Comp_3:1.0 | Comp_3_Meth_2 | 0 unknown |
| 1 | unknown | 127.0.0.1:1028 | resolve | IDL:Comp_3:1.0 | Comp_3_Meth_2 | 60 unknown |

| Node4 | count callee caller time latency |
| 120 | IDL:Comp_5:1.0 | 127.0.0.1:1037 | Comp_5_Meth_1 | IDL:Comp_4:1.0 | Comp_4_Meth_1 | 24830 390 |
| 1 | unknown | 127.0.0.1:1028 | _is_a | UnBoundThread | 110 unknown |
| 1 | unknown | 127.0.0.1:1028 | to_name | UnBoundThread | 0 unknown |
| 1 | unknown | 127.0.0.1:1028 | bind | UnBoundThread | 0 unknown |
| 1 | unknown | 127.0.0.1:1028 | to_name | IDL:Comp_4:1.0 | Comp_4_Meth_1 | 0 unknown |
| 1 | unknown | 127.0.0.1:1028 | resolve | IDL:Comp_4:1.0 | Comp_4_Meth_1 | 220 unknown |

| Node5 | count callee caller time latency |
| 1 | unknown | 127.0.0.1:1028 | _is_a | UnBoundThread | 110 unknown |
| 1 | unknown | 127.0.0.1:1028 | to_name | UnBoundThread | 0 unknown |
| 1 | unknown | 127.0.0.1:1028 | bind | UnBoundThread | 60 unknown |

Figure 30: The result of experiment 4 presented as call statistics per node.

The next presentation, figure 31, is call statistics based on communications between nodes. This presentation is for investigate if the distribution of components is satisfying. This kind of presentation shows that the tool can capture and present the information necessary for finding bottlenecks caused by poor distribution design.
The summarized callgraph of the initializing method in experiment 4 is presented in figure 32. The information found here indicates the same as other presentations before. There are performance benefits to gain with reducing the distributed calls within the servers. The total execution of method Comp_1_Meth_1 in experiment 5 (85680 ms) is significantly lower than the same method in experiment 2 (134460 ms).

```
100 IDL:Comp_1:1.0 Comp_1_Meth_1 Execution time:85680 Effective time:16100
  2 unknown127.0.0.1:1028 to_name Execution time:110 Effective time:unknown
  2 unknown127.0.0.1:1028 resolve Execution time:170 Effective time:unknown
  49 IDL:Comp_2:1.0 Comp_2_Meth_1 Execution time:50590 Effective time:7690
  1 unknown127.0.0.1:1028 to_name Execution time:60 Effective time:unknown
  1 unknown127.0.0.1:1028 resolve Execution time:160 Effective time:unknown
  33 IDL:Comp_3:1.0 Comp_3_Meth_1 Execution time:25720 Effective time:5020
  1 unknown127.0.0.1:1028 to_name Execution time:50 Effective time:unknown
  1 unknown127.0.0.1:1028 resolve Execution time:170 Effective time:unknown
  35 IDL:Comp_4:1.0 Comp_4_Meth_1 Execution time:18500 Effective time:3860
  1 unknown127.0.0.1:1028 to_name Execution time:0 Effective time:unknown
  1 unknown127.0.0.1:1028 resolve Execution time:60 Effective time:unknown
  70 IDL:Comp_5:1.0 Comp_5_Meth_1 Execution time:14080 Effective time:14080
  8 IDL:Comp_5:1.0 Comp_5_Meth_1 Execution time:1650 Effective time:1650
  35 IDL:Comp_5:1.0 Comp_5_Meth_1 Execution time:7180 Effective time:7180
  24 IDL:Comp_3:1.0 Comp_3_Meth_1 Execution time:13310 Effective time:13310
  6 IDL:Comp_5:1.0 Comp_5_Meth_1 Execution time:1210 Effective time:1210
  25 IDL:Comp_4:1.0 Comp_4_Meth_1 Execution time:13310 Effective time:13310
  50 IDL:Comp_5:1.0 Comp_5_Meth_1 Execution time:10360 Effective time:10360
```

Figure 32: The result of experiment 4 presented as callgraph type2.

The results presented here of experiment 4 show the same as for experiment 3. The instrumentation and information presentation works for an application consisted of several servers or nodes run on the same machine with different JVM.
6.5 Experiment 5

The result of experiment 5 is presented here and examples of all the five different presentations is included here in figure 33 to figure 37.

<table>
<thead>
<tr>
<th>Count</th>
<th>Caller</th>
<th>Call</th>
<th>Time</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>178</td>
<td>IDL:Comp_5:1.0</td>
<td>Comp_5_Meth_1</td>
<td>35767</td>
<td>103</td>
</tr>
<tr>
<td>100</td>
<td>IDL:Comp_1:1.0</td>
<td>Comp_1_Meth_1</td>
<td>90560</td>
<td>160</td>
</tr>
<tr>
<td>89</td>
<td>IDL:Comp_4:1.0</td>
<td>Comp_4_Meth_1</td>
<td>44891</td>
<td>44</td>
</tr>
<tr>
<td>50</td>
<td>IDL:Comp_1:1.0</td>
<td>Comp_1_Meth_1</td>
<td>52516</td>
<td>109</td>
</tr>
<tr>
<td>39</td>
<td>IDL:Comp_2:1.0</td>
<td>Comp_2_Meth_1</td>
<td>31686</td>
<td>21</td>
</tr>
<tr>
<td>34</td>
<td>IDL:Comp_3:1.0</td>
<td>Comp_3_Meth_1</td>
<td>26055</td>
<td>31</td>
</tr>
<tr>
<td>34</td>
<td>IDL:Comp_3:1.0</td>
<td>Comp_3_Meth_2</td>
<td>13357</td>
<td>186</td>
</tr>
<tr>
<td>23</td>
<td>IDL:Comp_5:1.0</td>
<td>Comp_5_Meth_1</td>
<td>4618</td>
<td>-1</td>
</tr>
<tr>
<td>11</td>
<td>IDL:Comp_5:1.0</td>
<td>Comp_5_Meth_2</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>IDL:Comp_5:1.0</td>
<td>Comp_5_Meth_1</td>
<td>1603</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>unknown</td>
<td>194.47.139.228:1053</td>
<td>540</td>
<td>unknown</td>
</tr>
<tr>
<td>6</td>
<td>unknown</td>
<td>194.47.139.228:1053</td>
<td>41</td>
<td>unknown</td>
</tr>
<tr>
<td>5</td>
<td>unknown</td>
<td>194.47.139.228:1053</td>
<td>30</td>
<td>unknown</td>
</tr>
<tr>
<td>2</td>
<td>unknown</td>
<td>194.47.139.228:1053</td>
<td>0</td>
<td>unknown</td>
</tr>
<tr>
<td>1</td>
<td>unknown</td>
<td>194.47.139.228:1053</td>
<td>110</td>
<td>unknown</td>
</tr>
<tr>
<td>1</td>
<td>unknown</td>
<td>194.47.139.228:1053</td>
<td>80</td>
<td>unknown</td>
</tr>
<tr>
<td>1</td>
<td>unknown</td>
<td>194.47.139.228:1053</td>
<td>0</td>
<td>unknown</td>
</tr>
<tr>
<td>1</td>
<td>unknown</td>
<td>194.47.139.228:1053</td>
<td>10</td>
<td>unknown</td>
</tr>
<tr>
<td>1</td>
<td>unknown</td>
<td>194.47.139.228:1053</td>
<td>0</td>
<td>unknown</td>
</tr>
<tr>
<td>1</td>
<td>unknown</td>
<td>194.47.139.228:1053</td>
<td>70</td>
<td>unknown</td>
</tr>
<tr>
<td>1</td>
<td>unknown</td>
<td>194.47.139.228:1053</td>
<td>0</td>
<td>unknown</td>
</tr>
<tr>
<td>1</td>
<td>unknown</td>
<td>194.47.139.228:1053</td>
<td>20</td>
<td>unknown</td>
</tr>
<tr>
<td>1</td>
<td>unknown</td>
<td>194.47.139.228:1053</td>
<td>0</td>
<td>unknown</td>
</tr>
<tr>
<td>1</td>
<td>unknown</td>
<td>194.47.139.228:1053</td>
<td>60</td>
<td>unknown</td>
</tr>
</tbody>
</table>

The result in figure 33 is one of several callgraph produced by the presentation program. The result shows that the tool can register and trace calls and executions in an application that is distributed over several machines. In the address field of the calls one can read that the components resides on different machines because they have different IP. The network used in the experiment has very low latency. It is less then the resolution of the timing function used.

The next result presentation in figure 34 shows the overall call statistics of the execution of the application. In this presentations is some figures that can seems a little oide. Some summarized calls have a negative latency as in line five. This phenomenon arises due to
that the latency of the network is less than the resolution of the timing function. The latency is calculated by the principles presented in Table 1. If the target machine’s timing gives a little higher execution time than the calling machine and the network don’t give a recordable latency the calculated latency becomes negative.

In Figure 35 is a presentation of the call statistics summarized per node. The same negative latency occurs in this presentation.

---

**Figure 35: The result of experiment 5 presented as call statistics per node.**
In figure 36 is the call statistics between the nodes presented. The same negative latency occurs in this presentation as in the presentations before.

<table>
<thead>
<tr>
<th>count</th>
<th>callee</th>
<th>caller</th>
<th>time</th>
<th>latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>178</td>
<td>ServerComp5</td>
<td>194.47.139.223:1039</td>
<td>194.47.139.229:1033</td>
<td>35767 103</td>
</tr>
<tr>
<td>100</td>
<td>ServerComp1</td>
<td>194.47.139.228:1054</td>
<td>Client1</td>
<td>90560 160</td>
</tr>
<tr>
<td>89</td>
<td>ServerComp4</td>
<td>194.47.139.228:1033</td>
<td>ServerComp3</td>
<td>194.47.139.225:1036</td>
</tr>
<tr>
<td>73</td>
<td>ServerComp3</td>
<td>194.47.139.228:1036</td>
<td>ServerComp2</td>
<td>194.47.139.231:1064</td>
</tr>
<tr>
<td>50</td>
<td>ServerComp2</td>
<td>194.47.139.228:1054</td>
<td>ServerComp1</td>
<td>194.47.139.228:1054</td>
</tr>
<tr>
<td>42</td>
<td>ServerComp5</td>
<td>194.47.139.228:1039</td>
<td>ServerComp3</td>
<td>194.47.139.225:1036</td>
</tr>
<tr>
<td>34</td>
<td>ServerComp3</td>
<td>194.47.139.228:1036</td>
<td>ServerComp1</td>
<td>194.47.139.228:1054</td>
</tr>
<tr>
<td>7</td>
<td>unknown</td>
<td>194.47.139.228:1053</td>
<td>ServerComp1</td>
<td>194.47.139.228:1054</td>
</tr>
<tr>
<td>7</td>
<td>unknown</td>
<td>194.47.139.228:1053</td>
<td>ServerComp3</td>
<td>194.47.139.225:1036</td>
</tr>
<tr>
<td>5</td>
<td>unknown</td>
<td>194.47.139.228:1053</td>
<td>ServerComp2</td>
<td>194.47.139.231:1064</td>
</tr>
<tr>
<td>5</td>
<td>unknown</td>
<td>194.47.139.228:1053</td>
<td>ServerComp4</td>
<td>194.47.139.229:1036</td>
</tr>
<tr>
<td>3</td>
<td>unknown</td>
<td>194.47.139.228:1053</td>
<td>Client1</td>
<td>161 unknown</td>
</tr>
<tr>
<td>3</td>
<td>unknown</td>
<td>194.47.139.228:1053</td>
<td>ServerComp5</td>
<td>194.47.139.223:1039</td>
</tr>
</tbody>
</table>

Figure 36: The result of experiment 5 presented as call statistics between nodes.

The next presentation is the summarized callgraph of method Comp_1_Meth_1. This type of presentation doesn’t calculate latency’s so the odd negative values don’t occur in this presentation.

| 100 | IDL:Comp_1:1.0 Comp_1_Meth_1 | Execution time:90400 Effective time:11719 |
| 2   | unknown | 194.47.139.228:1053 to_name | Execution time:0 Effective time:unknown |
| 2   | unknown | 194.47.139.228:1053 resolve | Execution time:110 Effective time:unknown |
| 34  | IDL:Comp_3:1.0 Comp_3_Meth_1 | Execution time:26024 Effective time:5226 |
| 1   | unknown | 194.47.139.228:1053 to_name | Execution time:0 Effective time:unknown |
| 39  | IDL:Comp_4:1.0 Comp_4_Meth_1 | Execution time:19718 Effective time:4004 |
| 1   | unknown | 194.47.139.228:1053 to_name | Execution time:0 Effective time:unknown |
| 1   | unknown | 194.47.139.228:1053 resolve | Execution time:60 Effective time:unknown |
| 76  | IDL:Comp_5:1.0 Comp_5_Meth_1 | Execution time:15632 Effective time:15632 |
| 5   | IDL:Comp_5:1.0 Comp_5_Meth_1 | Execution time:1001 Effective time:1001 |
| 50  | IDL:Comp_2:1.0 Comp_2_Meth_1 | Execution time:52625 Effective time:7572 |
| 1   | unknown | 194.47.139.228:1053 to_name | Execution time:0 Effective time:unknown |
| 1   | unknown | 194.47.139.228:1053 resolve | Execution time:10 Effective time:unknown |
| 39  | IDL:Comp_3:1.0 Comp_3_Meth_1 | Execution time:31665 Effective time:5899 |
| 50  | IDL:Comp_4:1.0 Comp_4_Meth_1 | Execution time:25129 Effective time:5016 |
| 100 | IDL:Comp_5:1.0 Comp_5_Meth_1 | Execution time:20032 Effective time:20032 |
| 3   | IDL:Comp_5:1.0 Comp_5_Meth_1 | Execution time:601 Effective time:601 |
| 34  | IDL:Comp_3:1.0 Comp_3_Meth_2 | Execution time:13171 Effective time:8523 |
| 1   | unknown | 194.47.139.228:1053 to_name | Execution time:0 Effective time:unknown |
| 23  | IDL:Comp_5:1.0 Comp_5_Meth_1 | Execution time:4619 Effective time:4619 |
| 11  | IDL:Comp_5:1.0 Comp_5_Meth_2 | Execution time:10 Effective time:10 |

Figure 37: The result of experiment 5 presented as callgraph type2.

The resolution for the trimming function used (currentTimeMillis) is 10 ms for the platform used (windows NT). Then the application is run on a LAN is the latency less then the resolution of the timing function. That’s explains why some latency’s is less then 0 in the results presented. For proving that it isn’t a systematic error are the same experiment run three additional times. The result is presented in figure 38,39 and 40 as the call statistics between nodes.

<table>
<thead>
<tr>
<th>count</th>
<th>callee</th>
<th>caller</th>
<th>time</th>
<th>latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>126</td>
<td>ServerComp5</td>
<td>194.47.139.223:1044</td>
<td>ServerComp4</td>
<td>194.47.139.229:1037</td>
</tr>
<tr>
<td>108</td>
<td>ServerComp1</td>
<td>194.47.139.228:1088</td>
<td>Client1</td>
<td>75587 198</td>
</tr>
<tr>
<td>65</td>
<td>ServerComp3</td>
<td>194.47.139.228:1053</td>
<td>ServerComp2</td>
<td>194.47.139.231:1067</td>
</tr>
<tr>
<td>63</td>
<td>ServerComp4</td>
<td>194.47.139.229:1037</td>
<td>ServerComp3</td>
<td>194.47.139.225:1043</td>
</tr>
<tr>
<td>49</td>
<td>ServerComp2</td>
<td>194.47.139.231:1067</td>
<td>ServerComp1</td>
<td>194.47.139.228:1068</td>
</tr>
<tr>
<td>43</td>
<td>ServerComp5</td>
<td>194.47.139.223:1044</td>
<td>ServerComp3</td>
<td>194.47.139.225:1043</td>
</tr>
<tr>
<td>29</td>
<td>ServerComp3</td>
<td>194.47.139.228:1043</td>
<td>ServerComp1</td>
<td>194.47.139.228:1068</td>
</tr>
<tr>
<td>7</td>
<td>unknown</td>
<td>194.47.139.228:1067</td>
<td>ServerComp1</td>
<td>194.47.139.228:1068</td>
</tr>
<tr>
<td>7</td>
<td>unknown</td>
<td>194.47.139.228:1067</td>
<td>ServerComp3</td>
<td>194.47.139.225:1067</td>
</tr>
<tr>
<td>5</td>
<td>unknown</td>
<td>194.47.139.228:1067</td>
<td>ServerComp2</td>
<td>194.47.139.231:1067</td>
</tr>
<tr>
<td>5</td>
<td>unknown</td>
<td>194.47.139.228:1067</td>
<td>ServerComp4</td>
<td>194.47.139.229:1037</td>
</tr>
<tr>
<td>3</td>
<td>unknown</td>
<td>194.47.139.228:1067</td>
<td>Client1</td>
<td>140 unknown</td>
</tr>
<tr>
<td>3</td>
<td>unknown</td>
<td>194.47.139.228:1067</td>
<td>ServerComp5</td>
<td>194.47.139.223:1039</td>
</tr>
</tbody>
</table>

Figure 38: The result of the second run of experiment 5 presented as call statistics between nodes.
The latency for calls between the nodes differs between the different runs of the application. This result indicates that it isn’t a systematic error. The negative latency occurs randomly and is due to the low resolution of the timing and the very low network latency of the test environment.

The results of experiment 5 shows that the tool presented in this paper can function in a distributed environment that is physically distributed over several machines. The result also shows that all necessary data is recorded and that it is possible to get sufficient information for profiling a distributed system.

7 Discussion and Future work

The profiling gives a good picture of the distributed system. If the application profiled is run on a high performance network the timing resolution is to low as seen in experiment 5. For getting a correct picture of the distributed system on a high performance network a timing principle with higher resolution has to be developed.

For getting an even better understanding of profiling in distributed environments further experiments could be performed. It could be tested in a more distributed application or on an application with high workload. The next natural step would be to test the tool on a real live application. This would also gives proof for, if the presentations are sufficient for finding the bottlenecks.

On aspect of the tool overhead that isn’t investigated is however the tool gives different latency in different parts of the application. It is possible that the instrumentation has different latency depending on the data type and the amount of data. Some future efforts have to be made for investigating this for evaluating the overhead of the instrumentation.

In real distributed systems is it common to use asynchronous calls and event channels. The instrumentation doesn’t support these features. Some work should also be put down to expand the tool for handling asynchronous calls and event channels.
One interesting future path is to extend the approach to make a monitoring tool of the profiling tool. This could be done if the instrumentation points write to a socket instead of to a file, and a monitoring server has to be constructed.

8 Conclusion

For handling performance problems of a distributed environment a distributed profiling tool could be to great help. There is a lot different ways such a tool can be constructed. One way to construct a distributed profiling tool for CORBA application is to use interceptors as instrumentation. In this paper, we investigate if a tool constructed with interceptors can give the sufficient information needed for identifying and handling performance problems.

The design and implementation sections in this paper show that it is possible to construct a profiling tool with the use of interceptors. It has many benefits, the registration framework is implemented, the interception points are sufficient placed, portable between CORBA implementations and doesn’t require changes in the source code of the application to be profiled.

The tool and presentation program constructed give the flow of the system in two different types of callgraphs. One callgraph type presents the flow of the system with one callgraph for each client call from a non-distributed object. The other type of callgraph summarizes all the execution paths evolved from each distributed method. The data from the execution paths presented with one graph for each executed distributed method in the application. These two types of graphs give an overview of the flow in the system.

The presentation program also produces some call statistics in three different levels. One for all the calls in the whole system, one for the calls within a node and one for calls between nodes. This statistics make it possible to find bottlenecks on different levels in the system and pinpoint the problem down to which type of distributed method that causes it and on which node it resides. The possibility to make several different presentations of the captured data indicates that sufficient information needed for finding bottlenecks and getting an overview of the performance of the application is captured.

When the tool was run on a test application it showed that it was possible to hook on it without changing the target application. The overhead tests run showed that the tool had less then one percent overhead on the application. The experiments run with different degrees of distribution showed that a profiling tool using interceptors works in distributed system and gives sufficient information. The implementation experience and the performed experiments show that interceptors are the right instrumentation technique for building a profiling tool for distributed systems based on CORBA.

Due to the easy way of hooking on the tool and the amount of information presented by it, the tool can be very useful. It can be useful in developing, debugging and maintaining a distributed system. I think that even if the resolution is poor, profiling tools of the type presented in this paper will become common in the software industry for handling distributed systems.
9 Bibliography


5 Günther Rackl, Ivan Zoraja, Arndt Bonde, Distributed object computing:Concepts and Trends, LRR_TUM, Institut für Informatik, Technische Universität München,Germany

6 J. H. Kazi, D. P. Jose B. Ben-Hamida , JaViz: A client/server Java profiling tool IBM SYSTEMS JOURNAL, VOL 39, NO 1, 2000


9 Object Management Group. CORBA v2.3 jun 1999


Appendix A
This appendix contains the Java code for the instrumentation of a distributed system built on CORBA.

A.1 Instrumentation initializer

package exam.profTool.instrumentation.interceptors;

import org.omg.PortableInterceptor.*;
import org.omg.CosNaming.*;
import org.apache.log4j.PatternLayout;
import org.apache.log4j.AsyncAppender;
import org.apache.log4j.FileAppender;
import org.apache.log4j.Category;
import java.io.*;
import jacorb.util.Environment;
import jacorb.poa.util.ParsedIOR;
import jacorb.poa.util.*;
import jacorb.poa.*;

public class InstrumentInitializer  extends jacorb.orb.LocalityConstrainedObject
implements ORBInitializer
{
    public InstrumentInitializer() { }

    public void post_init(ORBInitInfo info)
    {
        // read system properties
        String logFileName = null;
        try
        {
            logFileName = System.getProperty("InstrumentLogFile");
        }
        catch (Exception ex)
        {
            System.err.println("error "+ex);
        }
        if(logFileName==null)
        logFileName= "log_file.txt";

        String NodeName = null;
        try
        {
            NodeName = System.getProperty("InstrumentNodeName");
        }
        catch (Exception ex)
        {
            System.err.println("error "+ex);
        }
        AsyncAppender AAppender = new AsyncAppender();
        Category root = Category.getRoot();
        try
        {
            AAppender.addAppender(new FileAppender(
            new PatternLayout("%m [%t] %c %x - %n"),logFileName,false));
        }
        catch(IOException e)
        {
            System.out.println("error "+e);
        }

        root.addAppender(AAppender);
        Category cat = Category.getInstance("InitInfor");
        try
        {
            jacorb.orb.portableInterceptor.ORBInitInfoImpl ORBInitInfoImpl =
            (jacorb.orb.portableInterceptor.ORBInitInfoImpl) info;
            org.omg.CORBA.ORB orb = ORBInitInfoImpl.getORB();
            if(NodeName==null)
        }
A.2 Client instrumentation

package exam.profTool.instrumentation.interceptors;

import java.io.*;
import org.omg.Dynamic.*;
import org.omg.PortableInterceptor.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.Any;
import jacorb.orb.ParsedIOR;
import org.apache.log4j.Category;
import org.omg.CORBA_2_3.portable.*;
import org.omg.IOP.ServiceContext;
import org.omg.IOP_N.Codec;

/**
   * @author Marcus Holgersson
   */
public class ClientInstrumentInterceptor extends jacorb.orb.LocalityConstrainedObject
    implements ClientRequestInterceptor {

    static Category cat = Category.getInstance("ClientIc");
    org.omg.CORBA.ORB orb;
    String NodeID;
    private static int requestNr=0;
    private int InstrumentionContexID=12;

    public ClientInstrumentInterceptor(org.omg.CORBA.ORB orb,String p_NodeID) {
        System.err.println("instansiated ClientInstrumentInterceptor");
        cat.info("%COMENT%% instansiated ClientInstrumentInterceptor.");
        this.orb= orb;
        NodeID= p_NodeID;
    }

    public String name() {
        return "ClientInstrumentInterceptor";
    }

    public void send_request(ClientRequestInfo ri) throws ForwardRequest {
        String IOR = ri.target().toString();
        // this is jacorb dependent
    }
}
ParsedIOR ParIOR = new ParsedIOR(IOR, (jacorb.orb.ORB) orb);
String Address = ParIOR.getAddress();
String ReqNodeID = NodeID + "::getRequestNr();

// send captured data to the logger
cat.info("%%SEND_REQUEST\%% \%Address\% ++
 IOR+" +
 ri.operation()+"++
 ri.request_id() + "++
 + ReqNodeID ++
 + System.currentTimeMillis());

// add a ServiceContext for connection send_request to receive_request.
ServiceContext context= new
 ServiceContext(InstrumentationContextID, ReqNodeID.getBytes());
ri.add_request_service_context(context, false);
}

public void send_poll(ClientRequestInfo ri)
{
 String IOR = ri.target().toString();
// this is jacorb dependent
ParsedIOR ParIOR = new ParsedIOR(IOR, (jacorb.orb.ORB) orb);
String Address = ParIOR.getAddress();
cat.info("%%SEND_POLL\%% ++Address++ ++
 IOR++
 ri.operation()+"++
 ri.request_id() + "++
 + System.currentTimeMillis());
}

public void receive_reply(ClientRequestInfo ri)
{
 String IOR = ri.target().toString();
// this is jacorb dependent
ParsedIOR ParIOR = new ParsedIOR(IOR, (jacorb.orb.ORB) orb);
String Address = ParIOR.getAddress();
cat.info("%%RECEIVE_REPLY\%% ++Address++ ++
 IOR++
 ri.operation()+"++
 ri.request_id() + "++
 + System.currentTimeMillis());
}

public void receive_exception(ClientRequestInfo ri) throws ForwardRequest
{
 String IOR = ri.target().toString();
// this is jacorb dependent
ParsedIOR ParIOR = new ParsedIOR(IOR, (jacorb.orb.ORB) orb);
String Address = ParIOR.getAddress();
cat.info("%%RECEIVE_EXCEPTION\%% ++Address++ ++
 IOR++
 ri.operation()+"++
 ri.request_id() + "++
 + System.currentTimeMillis());
}

public void receive_other (ClientRequestInfo ri) throws ForwardRequest
{
 String IOR = ri.target().toString();
// this is jacorb dependent
ParsedIOR ParIOR = new ParsedIOR(IOR, (jacorb.orb.ORB) orb);
String Address = ParIOR.getAddress();
cat.info("%%RECEIVE_OTHER\%% ++Address++ ++
 IOR++
 ri.operation()+"++
 ri.request_id() + "++
 + System.currentTimeMillis());
}

private static synchronized int getRequestNr()
{
 int returnType=requestNr;
 requestNr++;
 return returnType;
}
A.2 Server instrumentation

package exam.profTool.instrumentation.interceptors;

import org.omg.Dynamic.*;
import org.omg.PortableInterceptor.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.Any;
import org.apache.log4j.Category;
import jacorb.orb.util.*;
import jacorb.orb.ParsedIOR;
import org.omg.IOP.ServiceContext;
import jacorb.util.Environment;

public class ServerInstrumentInterceptor extends jacorb.orb.LocalityConstrainedObject
    implements ServerRequestInterceptor {
    private org.omg.CORBA.ORB orb;
    private org.omg.PortableServer.POA poa;
    public static Category cat = Category.getInstance("ServerIc");
    private String NodeID;
    private int InstrumentationContexID=12;

    public ServerInstrumentInterceptor(org.omg.CORBA.ORB orb, String p_NodeID)
    {
        NodeID =p_NodeID;
        cat.debug("%%COMENT%% instansiated ServerInstrumentInterceptor");
        this.orb=orb;
        try
        {
            poa =
            org.omg.PortableServer.POAHelper.narrow(
            orb.resolve_initial_references("RootPOA"));
        } catch ( Exception e )
        {
            e.printStackTrace();
        }
    }

    public String name()
    {
        return "ServerInstrumentInterceptor";
    }

    public void receive_request_service_contexts(ServerRequestInfo ri) throws ForwardRequest
    {
        cat.info("%%COMENT%% receive_request_service_contexts "% +ri.operation()++ " +ri.request_id());
    }

    public void receive_request(ServerRequestInfo ri) throws ForwardRequest
    {
        String IOR=null;
        // Translate object_id to IOR
        try
        {
            IOR= poa.id_to_reference(ri.object_id()).toString();
        } catch ( Exception e )
        {
            cat.info("error "+e);
            e.printStackTrace();
        }

        // this is jacorb dependent
        ParsedIOR ParIOR= new ParsedIOR(IOR, (jacorb.orb.ORB) orb);
        String Address = ParIOR.getAddress();
    }
}
ServiceContext ctx = ri.get_request_service_context(InstrumentationContexID);
String InstrumentRequestID = "UNKNOWN";
if(ctx!=null)
{
    InstrumentRequestID = new String(ctx.context_data);
}
cat.info("%%RECEIVE_REQUEST%% Address="+Address+" IOR="+IOR+
    +ri.target_most_derived_interface()+" 
    +ri.operation()+" 
    +ri.request_id()+" 
    +InstrumentRequestID+" 
    +System.currentTimeMillis());
}

public void send_reply(ServerRequestInfo ri)
{
    String IOR=null;
    // Translate object_id to IOR
    try
    {
        IOR= poa.id_to_reference(ri.object_id()).toString();
    }
    catch ( Exception e )
    {
        cat.info("%%ERROR%% "%+e);
    }

    // get the address for the current node
    ParsedIOR ParIOR= new ParsedIOR(IOR, (jacorb.orb.ORB) orb);
    String Address = ParIOR.getAddress();

    // send captured data to the logwriter
    cat.info("%%SEND_REPLY%% Address="+Address+" IOR="+IOR+
        +ri.target_most_derived_interface()+" 
        +ri.operation()+" 
        +ri.request_id()+" 
        +System.currentTimeMillis());
}

public void send_exception(ServerRequestInfo ri) throws ForwardRequest
{
    String IOR=null;
    try
    {
        IOR= poa.id_to_reference(ri.object_id()).toString();
    }
    catch ( Exception e )
    {
        cat.info("error "%+e);
        e.printStackTrace();
    }

    // this is jacorb dependent
    ParsedIOR ParIOR= new ParsedIOR(IOR, (jacorb.orb.ORB) orb);
    String Address = ParIOR.getAddress();
    cat.info("%%SEND_EXCEPTION%% Address="+Address+" IOR="+IOR+
        +ri.target_most_derived_interface()+" 
        +ri.operation()+" 
        +ri.request_id()+" 
        +System.currentTimeMillis());
}

public void send_other (ServerRequestInfo ri) throws  ForwardRequest
{  
    String IOR=null;
    try
    {
        IOR= poa.id_to_reference(ri.object_id()).toString();
    }
    catch ( Exception e )
    {
        cat.info("error "%e);
        e.printStackTrace();
    }
}
// this is jacorb dependent
ParsedIOR ParlOR= new ParsedIOR(IOR, (jacorb.orb.ORB) orb);
String Address = ParlOR.getAddress();
cat.info("\%\%SEND\_OTHER\%\% " +Address+" \"+IOR+" "+
ri.target_most_derived_interface()+" "+
+ri.operation()+" "+
+ri.request_id()+" "
+System.currentTimeMillis());
}
Appendix B

This appendix contains the IDL specification of the portable interceptors used in the instrumentation in the profiling tool:

B.1 Client-side Interceptor

The client-side interceptor interface is called ClientRequestInterceptor and looks like the following IDL declaration.

```
local interface ClientRequestInterceptor : Interceptor {
  void send_request (in ClientRequestInfo ri) raises (ForwardRequest);
  void send_poll (in ClientRequestInfo ri);
  void receive_reply (in ClientRequestInfo ri);
  void receive_exception (in ClientRequestInfo ri) raises (ForwardRequest);
  void receive_other (in ClientRequestInfo ri);
}
```

Figure B1: The IDL definition for ClientRequestInterceptor.

B.2 Server-side Interceptor

The Server-side interceptor interface is called ServerRequestInterceptor and looks like the following IDL declaration in figure B2.

```
local interface ServerRequestInterceptor : Interceptor {
  void receive_request_service_contexts (in ServerRequestInfo ri)
    raises (ForwardRequest);
  void receive_request (in ServerRequestInfo ri) raises (ForwardRequest);
  void send_reply (in ServerRequestInfo ri);
  void send_exception (in ServerRequestInfo ri) raises (ForwardRequest);
  void send_other (in ServerRequestInfo ri) raises (ForwardRequest);
}
```

Figure B2: The IDL definition for ServerRequestInterceptor.

B.3 Requestinfo Interfaces

The following IDL in figures B3 -B5, declare the Request interfaces used as parameters in request interceptors.

```
local interface RequestInfo {
  readonly attribute unsigned long request_id;
  readonly attribute string operation;
  readonly attribute Dynamic::ParameterList arguments;
  readonly attribute Dynamic::ExceptionList exceptions;
  readonly attribute Dynamic::ContextList contexts;
  readonly attribute Dynamic::RequestContext operation_context;
  readonly attribute any result;
  readonly attribute boolean response_expected;
  readonly attribute Messaging::SyncScope sync_scope;
  readonly attribute ReplyStatus reply_status;
  readonly attribute Object forward_reference;
  any get_slot (in SlotId id) raises (InvalidSlot);
  IOP::ServiceContext get_request_service_context (in IOP::ServiceId id);
  IOP::ServiceContext get_reply_service_context (in IOP::ServiceId id);
};
```

Figure B3: The IDL definition for the RequestInfo.

```
local interface ClientRequestInfo : RequestInfo {
  readonly attribute Object target;
  READONLY_ATTRIBUTE OBJECT EFFECTIVE_TARGET;
  readonly attribute IOP::TaggedProfile effective_profile;
  readonly attribute any received_exception;
  readonly attribute CORBA::RepositoryId received_exception_id;
};
```
IOR::TaggedComponent get_effective_component (in IOP::ComponentId id);
IOR_N::TaggedComponentSeq get_effective_components (in IOP::ComponentId id);
CORBA::Policy get_request_policy (in CORBA::PolicyType type);
void add_request_service_context (in IOP::ServiceContext service_context, in boolean replace);
);

Figure B4: The IDL definition for the ClientRequestInfo.

local interface ServerRequestInfo : RequestInfo {
  READONLY ATTRIBUTE ANY SENDING_EXCEPTION;
  readonly attribute CORBA::OctetSeq object_id;
  readonly attribute CORBA::OctetSeq adapter_id;
  readonly attribute CORBA::RepositoryId target_most_derived_interface;
  CORBA::Policy get_server_policy (in CORBA::PolicyType type);
  void set_slot (in SlotId id, in any data) raises (InvalidSlot);
  boolean target_is_a (in CORBA::RepositoryId id);
  void add_reply_service_context (in IOP::ServiceContext service_context, in boolean replace);
};

Figure B5: The IDL definition for the ServerRequestInfo.
Appendix C
This appendix contains a description of the server components used in the experimental environment. The components are presented with their IDL-Definition and a description of the different outcomes of the distributed methods.

C.1 Comp_1
This section describes the design of component Comp_1. In Figure C1 is the IDL for Comp_1.

```java
interface Comp_1
{
    string Comp_1_Meth_1(in string a);
    string Comp_1_Meth_2(in string a);
};
```

*Figure C1: The IDL definition for Comp_1*

The Component is dependent of an instance of component Comp_2 and Comp_3. The different outcomes of the methods are shown in table C1. The figures are the percentage for the chance that it will be called. The three outcomes are exclusive so only one is run per execution of the method.

<table>
<thead>
<tr>
<th>Method</th>
<th>Comp_2_Meth_1</th>
<th>Comp_3_Meth_1</th>
<th>Delay for 300 ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp_1_Meth_1</td>
<td>50 %</td>
<td>30 %</td>
<td>20 %</td>
</tr>
<tr>
<td>Comp_1_Meth_2</td>
<td>0 %</td>
<td>0 %</td>
<td>0 %</td>
</tr>
</tbody>
</table>

*Table C1: The different outcomes of the methods in component Comp_1.*

In the beginning of the method Comp_1_Meth_1 a 70 milliseconds delay is inserted and in method Comp_1_Meth_2 is no delay inserted.

C.2 Comp_2
This section describe the design of component Comp_2. In figure C2 is the IDL for Comp_2.

```java
interface Comp_2
{
    string Comp_2_Meth_1(in string a);
    string Comp_2_Meth_2(in string a);
};
```

*Figure C2: The IDL definition for Comp_2*

The Component is dependent of an instance of component Comp_3. The different outcomes of the methods are shown in table below.

<table>
<thead>
<tr>
<th>Method</th>
<th>Comp_3_Meth_1 and Comp_3_Meth_2</th>
<th>Comp_3_Meth_1</th>
<th>Comp_3_Meth_2</th>
<th>Delay for 150 ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp_2_Meth_1</td>
<td>60 %</td>
<td>20 %</td>
<td>10 %</td>
<td>10 %</td>
</tr>
<tr>
<td>Comp_2_Meth_2</td>
<td>0 %</td>
<td>0 %</td>
<td>0 %</td>
<td>0 %</td>
</tr>
</tbody>
</table>

*Table C2: The different outcomes of the methods in component Comp_2.*

In the beginning of the method Comp_2_Meth_1 a 130 milliseconds delay is inserted and in method Comp_2_Meth_2 is no delay inserted.

C.3 Comp_3
This section describe the design of component Comp_3. In figure C3 is the IDL for Comp_3.
The Component is dependent of an instance of component Comp_4 and an instance of component Comp_5. The different outcomes of the methods are shown in table below.

<table>
<thead>
<tr>
<th></th>
<th>Comp_3_Meth_1</th>
<th>2 * Comp_4_Meth_1</th>
<th>Comp_5_Meth_1</th>
<th>Comp_5_Meth_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp_3_Meth_1</td>
<td>50 %</td>
<td>30 %</td>
<td>20 %</td>
<td>0 %</td>
</tr>
<tr>
<td>Comp_3_Meth_2</td>
<td>0 %</td>
<td>0 %</td>
<td>70 %</td>
<td>30 %</td>
</tr>
</tbody>
</table>

Table C3: The different outcomes of the methods in component Comp_3.

In the Component is a delay for 150 milliseconds inserted in method Comp_3_Meth_1 and a 250 milliseconds delay is inserted in method Comp_3_Meth_2.

C.4 Comp_4

This section describes the design of component Comp_4. In figure C4 is the IDL of component Comp_4 shown.

<table>
<thead>
<tr>
<th></th>
<th>2 * Comp_5_Meth_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp_4_Meth_1</td>
<td>100 %</td>
</tr>
<tr>
<td>Comp_4_Meth_2</td>
<td>0 %</td>
</tr>
</tbody>
</table>

Table C4: The different outcomes of the methods in component Comp_4.

In method Comp_4_Meth_1 a 100 milliseconds delay is inserted and in method Comp_3_Meth_4 a 200 milliseconds delay is inserted.