An approach for performance measurements in distributed CORBA applications

HÅKAN GRAHN

Dept. of Software Engineering and Computer Science
Blekinge Institute of Technology
P.O. Box 520, SE-372 25 Ronneby, Sweden

MARCUS HOLGERSSON

Velocity AB
Campus Gräsvik 26
SE-371 75 Karlskrona, Sweden

ABSTRACT

One way to construct distributed systems is to use a communication model with distributed objects such 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 performance bottlenecks in a distributed system should be a great help.

In this paper we present an approach for performance measurements in distributed CORBA applications. Our approach is based on Interceptors, which is the technique we use for insertion of measurement points. This approach gives sufficient information for identifying many performance problems. In order to verify our approach, a prototype tool for profiling and performance measurements is constructed. A presentation program is built for making the captured information more readable. The tool and presentation programs show the execution flow of the system in different call graphs and also produces some call statistics at different levels. Finally, the tool is tested and verified in a distributed environment.

1. INTRODUCTION

Distributed systems are becoming more and more important in everyday life as well as in industrial and scientific domains. One way to construct large distributed systems is to use a communication model with distributed objects. Distributed objects have many advantages, but suffer from some performance problems. A distributed object system can be constructed using CORBA (Common Object Request Broker Architecture [10]). One large benefit with CORBA is that it works in a heterogeneous environment with different programming languages and different operating systems. By using CORBA, an application handles distributed objects just as if they where local objects.

Performance problems can occur in several places in the event chain in a distributed system. When developing a distributed system it is important to know its limitations and bottlenecks in order to prevent performance problems. If it is possible to make performance tests early in the application development, potential performance problem can be prevented. In order to handle the performance problem it is important to identify where in the event chain the delays occur. Therefore a tool for performance measurements and for identifying performance bottlenecks in a distributed system should be a great help.

In order to build such a tool, we need to identify what performance criteria that are important, how much information shall the tool gather, and where the important measurement points are. Further, the tool much have low overhead in order to disturb the application execution as little as possible. Finally, we would like the tool to be so general that it can be used on different applications without changing the application.

In this paper, we collect and analyze information concerning performance in distributed systems. This information is used for investigating the possibilities to measure performance and finding out what performance aspects that are important in a distributed system. We also investigate whether it is possible to automatically find the performance bottlenecks in a distributed system. The investigation is used as a basis for a prototype tool for performance profiling and measurement. The approach taken is based on interceptors as a means to transparently introduce measurement points in a distributed system.

In order to evaluate our approach, we have implemented a prototype of the performance measurement tool that uses interceptors for instrumentation of a distributed object system in the purpose of finding the bottlenecks of the system. Further, we have evaluated the prototype tool using a number of distributed test applications. The test applications and the distributed test system have been implemented using JacORB as CORBA implementation. The results from the experiments are analyzed in order to give design hints for distributed system with distributed objects.

The rest of the paper is organized as follows. In sections 2 and 3, we give some background information. In Section 4, we discuss performance aspects of distributed systems. Then, section 5 presents the design of the prototype tool. Sections 5 and 6 presents the experimental framework and the experimental results, respectively. Finally, the results and related work are discussed in section 6, and the paper is concluded in section 7.

2. DISTRIBUTED SYSTEMS

Distributed systems are becoming more and more important in everyday life as well as in the industrial and scientific domains. The Internet and its capabilities enable people to communicate and cooperate all over the world. Distributed computing can be seen as an effort to connect multiple 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 can be 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 system. The current trends in distributed computing shows a movement towards distributed middleware environments based on standardized communication protocols and networks [5]. One of the most common programming paradigms used for distributed computing is called distributed object computing (DOC)

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.

If all four functions are present and work properly, a distributed object computing system is enabled.

3. CORBA

CORBA (Common Object Request Broker Architecture) [10] is a standard created by the Object Management Group (OMG). The aim of CORBA is to define a general framework for construction of distributed object computing systems. In addition, CORBA is a specification designed to work in a heterogeneous environment, and is location and programming language independent. CORBA hides 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 use these services in their applications, without worrying about network related issues like connection setup, protocol decisions, on-the-wire data format, and failure scenarios.

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.

Clients request services from objects (which will also be called servers) through a well-defined interface. This interface is specified in IDL (Interface Definition Language) [9], a declarative language which means that it is not possible to write execution or state dependent code in IDL. The IDL separates object interface from implementation. The IDL can be described as the contract for communication between a client and a server. IDL provides a set of built-in types, which can be augmented by user-defined types, such as structures and sequences.

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 to the 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 servant.

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.

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.

The CORBA standard guarantees interoperability by providing a gateway infrastructure that makes different ORB implementations compatible. 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.

The portable object adapter (POA) provides a set of interfaces for managing object references and servants. When 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 ORBs very easy.

The CORBA standard also defines a set of distributed services to support the integration and interoperation of distributed objects. The services are defined on top of the ORB, i.e., they are defined as standard CORBA objects with IDL interfaces For example, for object location the CORBA standard defines a naming service by providing a mapping from names to object references: given a name, the service returns an object reference.

The standard defines some services for alternative communication paradigms. 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 load balancing and fault tolerant applications.

4. PERFORMANCE IN DISTRIBUTED SYSTEMS

4.1. Demands for high performance.

Depending on the purpose of the system the important performance measure can be different, e.g., response time or throughput. As a system developer I want to know what load the system can handle and were the bottlenecks are. Distributed systems today 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 rapidly 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.

4.2. Improving Performance

There is different way for improving the performance of a distributed system, e.g., optimize code, optimize the design and the distribution, and improve hardware and network. If there is a performance problem, the first thing to do is to find were the problem is in the system, i.e., the bottlenecks have to be identified. One way can be to find the steps in the critical path of the application. The critical path can be different from time to time. 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 steps 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. The measurement tool should be possible to use in the system without 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.

4.3. Performance considerations in CORBA

There are some basic factors that have a distinct impact on the performance of a CORBA application. The first one is the number of remote method invocations that are made within the system. Each request sent over a network connection imposes a network 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 a vises versa. This factor is highly dependent of the ORB implementation used.

4.4. Diagnosing a distributed system

What measurements are important when diagnosing distributed system? A long response time is one symptom that there is a performance problem. Therefore, measuring the response times under different workloads may give an indication that a performance problem exists. However, this kind of measurements cannot give any advise regarding what to do about problem. It simply can’t give any clues to 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 inter-process latency. The measurement should give useful information for finding the best design and distribution of the system.

Other profiling principles can be very useful then 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.

4.5. Interceptors

The interceptor architectural pattern allows services to be added transparently to a framework and trigged automatically when certain events 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.

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 is two types of request Interceptors: client-side and server-side.

The specification includes a set of design principles that specify the interceptors [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.

5. PROFILING TOOL

5.1. 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 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.
- The tool shall be easy to port to other ORB implementations.
- The tool shall have a low impact on the application execution time and not reducing the performance of the system.

5.2. Design

There are a lot of aspects to take in consideration then designing 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 monitor it. This means that there is no need for merging information from several nods in real time. 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 work 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. The information register should produce an internal event representation that could be put in a data structure that the consumer could read from and write to file.

5.2.1. 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.

For registering the information in a request-reply four measuring points has to be added. 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. These points give the information wanted. The simple calculation principles used with the four measuring points are shown in table 1.

Table 1: The principle for calculation of profiling data.

Calculation principles	
Execution of method	P2-P3
Network latency	P1-P4 –(P2-P3)
Total remote call	P1-P4

We have chosen to add measuring points by using interceptors. The interceptors can easily be hooked onto the system. An 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 recompiling the object source code. The Portable interceptors in the OMG standard [11] include all necessary measuring points.

5.2.2. Information processing

The measurement results in several files containing the data of run system. There is one file for each node in the system. For reducing the complexity the information processing is divided into two phases: parsing and merging.

In order to make the information manageable an internal representation is constructed. The first phase, i.e., parsing, is responsible for translate 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 a tree representation is chosen. A tree representation is not preferable 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 root of the tree is the system. A system has a number of nodes. A node represents one ORB, i.e., one instrumentation set and one data file. The node has Objects and unbound threads. An Orb can act as a client, 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 of 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.

After the parsing is done, the merging phase can start. The merging phase is responsible for finding the relationship between calls and executions. For every Call entity the target Execution will be searched for. If it is found a relation to it is made. There can be nodes in the system that aren’t 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.

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 call graph profiling.

The presentation component will not be fully implemented in the scope of this work. With the use of the internal representation is it easy to write presentation programs that make a lot of different information extraction. However, in this paper the results will only be outputted as flat files since no graphic representation is implemented.

5.3. Implementation

In this study, we have used JacORB, an object request broker written in Java which implements 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. It also includes an IDL compiler that supports OMG IDL/Java language mapping rev. 2.3. JacORB can be obtained at http://www.inf.fu-berlin.de/~jacorb/

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. Log4J also has 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

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 the registration of the distributed calls. When the interceptor hooks on and catches the information for a client request, it gets a ClientRequestInfo object. 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 severe impact on the measurement. It results in two distributed calls instead of one. What is even worse is if the calls to the naming service also are intercepted this call will generate a new naming service 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 recorder 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 nods. 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 is added using the send_request interception point and receive_reply. Table 2 shows which data recorder at the points.

Table 2: Data recorded at the client side.

	Client Measuring points	
Data recorded	Send_request	receive_reply
Target Address	Yes	Yes
Target IOR	Yes	Yes
Method name	Yes	Yes
RequestID	Yes	Yes
InstrumentationRequestID	Yes	No
Time point	Yes	Yes
Thread name	Yes	Yes

At the server side two measuring points are added using the receive_request interception point and send_reply interception point. Table 3 shows which data that are recorded at the points.

Table 3: The Data recorded at the server side.

	Server Measuring points	
Data recorded	Receive_request	send_reply
Address	Yes	Yes
IOR	Yes	Yes
Interface	Yes	Yes
Method name	Yes	Yes
RequestID	Yes	Yes
InstrumentationRequestID	Yes	No
Time point	Yes	Yes
Thread name	Yes	Yes

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.

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 has been implemented. The presentation programs are also examples of what kind of calculations and presentations that can be performed on the internal representation. In section 6, where we present our experimental results, we use the following presentation views.

The first presentation implemented is a client call graph. This presentation writes the different call graphs to a file. 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". 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 second presentation is some call statistics. It gives a summarized picture of what calls that has been done in the system. It presents the information in ordinary profiling style. 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.

The third presentation program is call statistics grouped into one call statistics per nod. Some additional information has been added to the identification of the callee. The added information is the node address of the node that holds the called object. This extra information increases the resolution of the presented information.

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. This presentation has been constructed for giving hints on that distribution will give the best performance. All three call-statistic presentations have been implemented in similar way. The difference is in what level the summery of the calls will be performed. 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.

This last implemented presentation shows the summarized execution paths from on method. The presentation program walks all paths evolved from the start method and summarizes the data. The information is presented as a call graph. 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. 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. For example it should be straight forward to implement a graphical user interface to our system.

5.4. Running the profiling tool

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 commando line then starting the application-node that shall be profiled presented in figure 1.

```
-Dorg.omg.PortableInterceptor.ORBInitializerClass.Instrument=
exam.profTool.instrumentation.interceptors.InstrumentInitializer
-DInstrumentLogFile=ServerLog.txt
-DInstrumentNodeName=Server1
```

Figure 1: The additional properties for adding the instrumentation on a node.

The first property tells the ORB to hook on the interceptors. The second property configures the name of the log file for the current node. The third property is for giving the node a name. This property isn't necessary since the node will get an auto generated name if no name is given.

After the execution of the system is done all data-files is moved to a directory for information processing. The run of the program produces five files containing the presentations (callGraphTyp1.txt, callGraphTyp2.txt, callStatistics.txt, callStatisticsPerNode.txt, and nodeStatistics.txt), one containing a printout of the entire tree (tree.txt), and one containing the internal representation (distrSystem.dc). The file distrSystem.dc can be used as input parameter for other programs performing calculations on the current execution of the distributed system.

6. EXPERIMENTAL FRAMEWORK

In order to test and evaluate our prototype tool, a test environment is constructed. The purpose of the test environment is to figure out whether the principles of the profiling tool are the right for profiling a distributed application. Note that the purpose is to evaluate the tool and not to test a real application.

6.1. Test application

Due to the lack of access to a real application, these first tests of the tool are run on a fictive application. For modeling a distributed application a set of components is constructed. The components use methods on each other. All inter-component calls are done using CORBA and the components are distributed over several nodes.

A server framework for registering the components is developed as well as a client to trigger the requests in the distributed system. The ORB used in the test environment is JacORB described earlier. The server is implemented in such way that it is easy to distribute the components on different JVMs or different machines. In addition, an activation and registration framework is also designed.

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, as shown in figure 2. The components is both client and servant the use other components for solving the tasks. The components are named comp_1 – comp_5. The components dependencies are shown in figure 2. In order to simulate a real application, the method calls are distributed randomly. For example, comp_1 calls comp_2 with a certain probability and comp_3 with another probability.

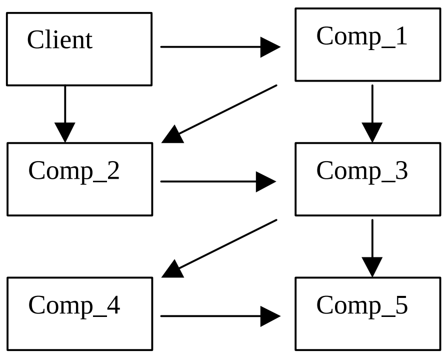

Figure 2: Dependencies between components in the test application.

A client in the test environment is constructed for trigging the requests in the application. The client will not have any GUI and is run using the console. The client is constructed in such way that the profiling tool can be hooked on to it. 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.

6.2. Defining the experiments

6.2.1. Tool overhead, experiment 1

Experiment 1 measures the tool overhead. The test is performed by making request to the same server started with the instrumentation and without the instrumentation. There is three different degrees of instrumentation tested: both client and server instrumented, only the client instrumented, and 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 dependencies between methods for this experiment are as shown in table 4.

Table 4: Dependencies in the tool overhead test.

Caller	**Calls**
Comp_1_Meth_1	Comp_2_Meth_1
Comp_2_Meth_1	Comp_3_Meth_1 and Comp_3_Meth_2
Comp_3_Meth_1	Comp_4_Meth_1
Comp_3_Meth_2	Comp_5_Meth_1
Comp_4_Meth_1	2 *Comp5_Meth_1

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. Experiment 1 is run on a machine with a Pentium II CPU, 320 MB RAM, and Windows 98.

6.2.2. Profiling with method 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 makes 100 calls on the method Comp_1_Meth_1 on the component Comp_1. The log files is collected and parsed with the merging and presentation program. The experiment is run on three distributions shown in table 5, i.e., experiment 2, 3, and 4. All nodes run on the same machine, which runs Windows 98 and has a Pentium II CPU and 320 MB RAM.

Table 5: The different distributions in experiment 2 – 4.

Exp.	Node 1	Node 2	Node 3	Node 4	Node 5
2	Comp [1-5]				
3	Comp [1-3]			Comp [4]	Comp [5]
4	Comp [1]	Comp [2]	Comp [3]	Comp [4]	Comp [5]

6.2.3. 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 distributed on one machine each, and the machines are connected to a 100 Mbit/s LAN. The machines used are Pentium III 660 Hz with 128 RAM running Windows NT.

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

7. EXPERIMENTAL RESULTS

7.1. Experiment 1

The results of the tests in experiment 1 indicate that the tool has negligible overhead in the test application. The measured overhead is lower then one percent, both in the case with both client and server as well with only the client. Therefore, we conclude that the instrumentation does not excessively distort the performance profile.

7.2. Summary of experiments 2, 3, and 4

Due to the limited number of pages, we do not present the results from experiments 2, 3, and 4 here. The results from all our experiments are available in [12]. In essence, those results are complementary to those from experiment 5. The difference is that in experiment 5, the objects are distributed on several different machines, while in experiments 2, 3, and 4 the objects reside on one single machine. The results from experiments 2, 3, and 4 have convinced us that the instrumentation works and gives valuable information for a distributed system based on several components on a single server. As expected, we found that the experiments indicate that it results in 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 then experiment 2.

7.3. Experiment 5

The results from experiment 5 is presented in this section and examples of all five presentation views are shown.

The result in figure 3 is one of several call graphs produced by the presentation program. The first line represents the initiating call from the client. If the target node is instrumented the execution of the called method will be registred as shown at line 2. If the execution of the method involves new distributed calls it will be written with the same indentation, e.g., see line 3.

The result shows that the tool can instrument, register and trace calls and executions in an application that is distributed over several machines. In the address field of the calls can one read that the components residues 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 as we will discuss later.

```
Call Comp_1_Meth_1 on node 194.47.139.228:1054 time 1332
network latency 0
  Execution Comp_1_Meth_1 on IDL:Comp_1:1.0 time 1332
effective time 70
  Call Comp_2_Meth_1 on node 194.47.139.231:1064 time 1262
network latency 0
    Execution Comp_2_Meth_1 on IDL:Comp_2:1.0 time 1262
effective time 130
    Call Comp_3_Meth_1 on node 194.47.139.225:1036 time
671 network latency 10
      Execution Comp_3_Meth_1 on IDL:Comp_3:1.0 time 661
effective time 150
      Call Comp_4_Meth_1 on node 194.47.139.229:1033 time
511 network latency 0
        Execution Comp_4_Meth_1 on IDL:Comp_4:1.0 time
511 effective time 100
        Call Comp_5_Meth_1 on node 194.47.139.223:1039
time 200 network latency 0
          Execution Comp_5_Meth_1 on IDL:Comp_5:1.0 time
200 effective time 200
          End execution
        Call Comp_5_Meth_1 on node 194.47.139.223:1039
time 211 network latency 10
          Execution Comp_5_Meth_1 on IDL:Comp_5:1.0 time
201 effective time 201
          End execution
        End execution
      End execution
    Call Comp_3_Meth_2 on node 194.47.139.225:1036 time
461 network latency 10
      Execution Comp_3_Meth_2 on IDL:Comp_3:1.0 time 451
effective time 250
      Call Comp_5_Meth_1 on node 194.47.139.223:1039 time
201 network latency 1
        Execution Comp_5_Meth_1 on IDL:Comp_5:1.0 time
200 effective time 200
        End execution
      End execution
    End execution
  End execution
```

Figure 3:A example of the result of experiment 5 presented as call graph type 1.

For getting a better overview of the execution of the application a summarized call statistics is presented in figure 4. The call statistics gives information about what method that has been called and what time spent in them. In line 3 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 caller UnBoundThread is the Client in this application. If it had been several clients, all had 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 this presentation view, some figures can seem a little odd. Some summarized calls have a negative latency as in line five. This phenomenon arises due to that the latency of the network is less then the resolution of the timing function. The latency is calculated by the principles presented in table 1. If the target machines timing gives a little higher execution time then the calling machine and the network does not give a measurable latency the calculated latency becomes negative.

```
count callee caller time latency
178    IDL:Comp_5:1.0 Comp_5_Meth_1    IDL:Comp_4:1.0
Comp_4_Meth_1    35767 103
100    IDL:Comp_1:1.0 Comp_1_Meth_1    UnBoundThread
90560 160
89    IDL:Comp_4:1.0 Comp_4_Meth_1    IDL:Comp_3:1.0
Comp_3_Meth_1    44891 44
50    IDL:Comp_2:1.0 Comp_2_Meth_1    IDL:Comp_1:1.0
Comp_1_Meth_1    52516 -109
39    IDL:Comp_3:1.0 Comp_3_Meth_1    IDL:Comp_2:1.0
Comp_2_Meth_1    31686 21
34    IDL:Comp_3:1.0 Comp_3_Meth_1    IDL:Comp_1:1.0
Comp_1_Meth_1    26055 31
34    IDL:Comp_3:1.0 Comp_3_Meth_2    IDL:Comp_2:1.0
Comp_2_Meth_1    13357 186
23    IDL:Comp_5:1.0 Comp_5_Meth_1    IDL:Comp_3:1.0
Comp_3_Meth_2    4618 -1
11    IDL:Comp_5:1.0 Comp_5_Meth_2    IDL:Comp_3:1.0
Comp_3_Meth_2    10 0
8    IDL:Comp_5:1.0 Comp_5_Meth_1    IDL:Comp_3:1.0
Comp_3_Meth_1    1603 1
6    unknown 194.47.139.228:1053 _is_a    UnBoundThread    540
unknown
6    unknown 194.47.139.228:1053 to_name    UnBoundThread
41 unknown
5    unknown 194.47.139.228:1053 bind    UnBoundThread    30
unknown
2    unknown 194.47.139.228:1053 to_name    IDL:Comp_1:1.0
Comp_1_Meth_1    0 unknown
2    unknown 194.47.139.228:1053 resolve    IDL:Comp_1:1.0
Comp_1_Meth_1    110 unknown
1    unknown 194.47.139.228:1053 resolve    UnBoundThread
80 unknown
1    unknown 194.47.139.228:1053 to_name    IDL:Comp_2:1.0
Comp_2_Meth_1    0 unknown
1    unknown 194.47.139.228:1053 resolve    IDL:Comp_2:1.0
Comp_2_Meth_1    10 unknown
1    unknown 194.47.139.228:1053 to_name    IDL:Comp_3:1.0
Comp_3_Meth_1    0 unknown
1    unknown 194.47.139.228:1053 resolve    IDL:Comp_3:1.0
Comp_3_Meth_1    70 unknown
1    unknown 194.47.139.228:1053 to_name    IDL:Comp_3:1.0
Comp_3_Meth_2    0 unknown
1    unknown 194.47.139.228:1053 resolve    IDL:Comp_3:1.0
Comp_3_Meth_2    20 unknown
1    unknown 194.47.139.228:1053 to_name    IDL:Comp_4:1.0
Comp_4_Meth_1    0 unknown
1    unknown 194.47.139.228:1053 resolve    IDL:Comp_4:1.0
Comp_4_Meth_1    60 unknown
```

Figure 4: The result of experiment 5 presented as some call statistics.

In figure 5 is the call statistics per node presented. This presentation gives a rough picture of the amount of work performed on each node. It shows the number of calls initiated from the node and their execution times and latencies. This figures can be compared between the different nodes for getting a estimation of the load on different nodes. Also in this presentation view, the negative latency occurs occasionally.

```
== Node Client1  ==
count callee caller time latency
100    IDL:Comp_1:1.0 194.47.139.228:1054 Comp_1_Meth_1
UnBoundThread    90560 160
1    unknown 194.47.139.228:1053 _is_a    UnBoundThread    70
unknown
1    unknown 194.47.139.228:1053 to_name    UnBoundThread
11 unknown
1    unknown 194.47.139.228:1053 resolve    UnBoundThread
80 unknown
=====================

== Node ServerComp1 194.47.139.228:1054 ==
count callee caller time latency
50    IDL:Comp_2:1.0 194.47.139.231:1064 Comp_2_Meth_1
IDL:Comp_1:1.0 Comp_1_Meth_1    52516 -109
34    IDL:Comp_3:1.0 194.47.139.225:1036 Comp_3_Meth_1
IDL:Comp_1:1.0 Comp_1_Meth_1    26055 31
2    unknown 194.47.139.228:1053 to_name    IDL:Comp_1:1.0
Comp_1_Meth_1    0 unknown
2    unknown 194.47.139.228:1053 resolve    IDL:Comp_1:1.0
Comp_1_Meth_1    110 unknown
1    unknown 194.47.139.228:1053 _is_a    UnBoundThread    190
unknown
1    unknown 194.47.139.228:1053 to_name    UnBoundThread
10 unknown
1    unknown 194.47.139.228:1053 bind    UnBoundThread    10
unknown
=====================

== Node ServerComp2 194.47.139.231:1064 ==
count callee caller time latency
39    IDL:Comp_3:1.0 194.47.139.225:1036 Comp_3_Meth_1
IDL:Comp_2:1.0 Comp_2_Meth_1    31686 21
34    IDL:Comp_3:1.0 194.47.139.225:1036 Comp_3_Meth_2
IDL:Comp_2:1.0 Comp_2_Meth_1    13357 186
1    unknown 194.47.139.228:1053 _is_a    UnBoundThread    80
unknown
1    unknown 194.47.139.228:1053 to_name    UnBoundThread
0 unknown
1    unknown 194.47.139.228:1053 bind    UnBoundThread    0
unknown
1    unknown 194.47.139.228:1053 to_name    IDL:Comp_2:1.0
Comp_2_Meth_1    0 unknown
1    unknown 194.47.139.228:1053 resolve    IDL:Comp_2:1.0
Comp_2_Meth_1    10 unknown
=====================

== Node ServerComp3 194.47.139.225:1036 ==
count callee caller time latency
89    IDL:Comp_4:1.0 194.47.139.229:1033 Comp_4_Meth_1
IDL:Comp_3:1.0 Comp_3_Meth_1    44891 44
23    IDL:Comp_5:1.0 194.47.139.223:1039 Comp_5_Meth_1
IDL:Comp_3:1.0 Comp_3_Meth_2    4618 -1
11    IDL:Comp_5:1.0 194.47.139.223:1039 Comp_5_Meth_2
IDL:Comp_3:1.0 Comp_3_Meth_2    10 0
8    IDL:Comp_5:1.0 194.47.139.223:1039 Comp_5_Meth_1
IDL:Comp_3:1.0 Comp_3_Meth_1    1603 1
1    unknown 194.47.139.228:1053 _is_a    UnBoundThread    60
unknown
1    unknown 194.47.139.228:1053 to_name    UnBoundThread
0 unknown
```

```
1    unknown 194.47.139.228:1053 bind    UnBoundThread    10
unknown
1    unknown 194.47.139.228:1053 to_name    IDL:Comp_3:1.0
Comp_3_Meth_1    0 unknown
1    unknown 194.47.139.228:1053 resolve    IDL:Comp_3:1.0
Comp_3_Meth_1    70 unknown
1    unknown 194.47.139.228:1053 to_name    IDL:Comp_3:1.0
Comp_3_Meth_2    0 unknown
1    unknown 194.47.139.228:1053 resolve    IDL:Comp_3:1.0
Comp_3_Meth_2    20 unknown
=====================

== Node ServerComp4 194.47.139.229:1033 ==
count callee caller time latency
178    IDL:Comp_5:1.0 194.47.139.223:1039 Comp_5_Meth_1
IDL:Comp_4:1.0 Comp_4_Meth_1    35767 103
1    unknown 194.47.139.228:1053 _is_a    UnBoundThread    70
unknown
1    unknown 194.47.139.228:1053 to_name    UnBoundThread
10 unknown
1    unknown 194.47.139.228:1053 bind    UnBoundThread    0
unknown
1    unknown 194.47.139.228:1053 to_name    IDL:Comp_4:1.0
Comp_4_Meth_1    0 unknown
1    unknown 194.47.139.228:1053 resolve    IDL:Comp_4:1.0
Comp_4_Meth_1    60 unknown
=====================

== Node ServerComp5 194.47.139.223:1039 ==
count callee caller time latency
1    unknown 194.47.139.228:1053 _is_a    UnBoundThread    70
unknown
1    unknown 194.47.139.228:1053 to_name    UnBoundThread
10 unknown
1    unknown 194.47.139.228:1053 bind    UnBoundThread    10
unknown
=====================
```

Figure 5: The result of experiment 5 presented as call statistics per node.

In figure 6, the call statistics based on the communication between the nodes are presented. This presentation view is for investigating 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 same negative network latency occurs in this presentation as in the presentations before.

```
count callee caller time latency
178    ServerComp5 194.47.139.223:1039    ServerComp4
194.47.139.229:1033    35767 103
100    ServerComp1 194.47.139.228:1054    Client1    90560 160
89    ServerComp4 194.47.139.229:1033    ServerComp3
194.47.139.225:1036    44891 44
73    ServerComp3 194.47.139.225:1036    ServerComp2
194.47.139.231:1064    45043 207
50    ServerComp2 194.47.139.231:1064    ServerComp1
194.47.139.228:1054    52516 -109
42    ServerComp5 194.47.139.223:1039    ServerComp3
194.47.139.225:1036    6231 0
34    ServerComp3 194.47.139.225:1036    ServerComp1
194.47.139.228:1054    26055 31
7    unknown 194.47.139.228:1053    ServerComp1
194.47.139.228:1054    320 unknown
7    unknown 194.47.139.228:1053    ServerComp3
194.47.139.225:1036    160 unknown
5    unknown 194.47.139.228:1053    ServerComp2
194.47.139.231:1064    90 unknown
5    unknown 194.47.139.228:1053    ServerComp4
194.47.139.229:1033    140 unknown
3    unknown 194.47.139.228:1053    Client1    161 unknown
3    unknown 194.47.139.228:1053    ServerComp5
194.47.139.223:1039    90 unknown
```

Figure 6: The results of experiment 5 presented as call statistics between nodes.

The resolution of the timing function used (currentTimeMillis) is 10 ms for the platform used (Windows NT). When the application is run on a LAN, the latency is less then the resolution of the timing function. That explains why some latencies are less then 0 in the results presented. We have also run the experiment several times and found that this effect is not a systematic error. Instead we have found that the latencies for calls between the nodes differ between the different runs of the application. This result indicates that it is not a systematic error. The negative latency occurs randomly and is due to the low resolution of the timing function 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.

8. DISCUSSION AND RELATED 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 is sufficient for finding the bottlenecks

In real distributed systems it is common to use asynchronous calls and event channels. The instrumentation doesn’t support these features today. Some work should also be put down to expand the tool for handling asynchronous calls and event channels.

Most of the papers on CORBA performance compare the performance and scalability of competing CORBA-compliant ORBS (an example of this is [1, 3]). Further, many performance studies of CORBA objects focus mainly on identifying the performance constraints of an Object Request Broker (ORB). For example, 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 [7], the authors say 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 factor. The article provides some guidelines on what measures to use when estimating an ORB's performance and 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 present the result. The profiling is presented with one graph for each client call.

9. CONCLUSIONS

Finding performance problems in a distributed application is often a very difficult problem. Therefore, a distributed profiling tool is very useful and a great help. There is many different ways such tool can be constructed. One approach to construct a profiling tool for distributed CORBA applications is to use interceptors to capture performance information. In this paper, we propose an approach based on interceptors that can give sufficient information for identifying and handling performance problems in distributed CORBA applications.

In order to evaluate our approach, we designed and implemented a profiling tool based on interceptors. It has many benefits; the registration framework is implemented, the interception points are sufficiently placed, it is portable between CORBA implementations, and it doesn't require source code changes in the profiled application.

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

The presentation program also produces call statistics at 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. These statistics make it possible to find performance bottlenecks at different levels in the system, track the problem to which type of distributed method that caused it, and on which node the method resides. The possibility to make several different presentations of the data indicates that sufficient information needed for finding bottlenecks and getting an overview of the performance of the application is gathered.

When running the tool on a number of test application, we showed that it was possible to use the performance tool without changing the target application, not even a recompilation was needed. The overhead tests showed that the tool had less then one percent overhead on the application. Our experiments with distributed test applications showed that a profiling tool using interceptors works in a distributed system and gives sufficient profiling information. The implementation experience and the experiments performed show that interceptors is a promising instrumentation technique for profiling tools for distributed CORBA applications.

Due to the easy way of using 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. We believe 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.

REFERENCES

1. A. Buble, "Comparing CORBA Implementations," Master Thesis, Charles Universitet, Parg 1999.
2. B. McCarty and L. Cassady-Dorion. "Java Distributed Objects The Authoritative solution," Sams,1999.
3. Distributed Systems Research Group, "*CORBA Comparison Project,"* Dept. Software Eng, Charles Univ., Prague, 1998. http://nenya.ms.mff.cuni.cz/thegroup/COMP/index.html.
4. D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, "Pattern-oriented software architecture, Volume 2, patterns for concurrent and networked object," John Wiley & Sons, 2000 England.
5. G. Rackl, I. Zoraja, and A. Bonde, *"Distributed object computing:Concepts and Trends,"* LRR_TUM, Institut für Informatik, Technische Universytät München,Germany.
6. I.H. Kazi, D.P. Jose, and B. Ben-Hamida , *"JaViz: A client/server Java profiling tool,"* IBM SYSTEMS JOURNAL, VOL 39, NO 1, 2000.
7. M.Vilich, s. Aslam-Mir, *"Benchmark metrics for enterprise Object Request Brokers*," http://www.expersoft.com/Resources/WPapers/bencmetric.htm
8. M. Henning and S. Vinoski, *"Advanced CORBA Programming with C++*," Reading, MA: Addison-Wesley, 1999.
9. Object Management Group. CORBA v2.3 June 1999
10. Object Management Group. The Common Object Request Broker: Architecture and Specification, 2.2 ed., Feb. 1998.
11. Object Management Group, "*Portable Interceptors*," OMG TC Document orbos/99-12-02.
12. M. Holgersson, "An approach for performance measurements in distributed CORBA applications," Bachelor thesis, Blekinge Institute of Technology, June 2001.