CORBA Tracer

Execution Trace of a
CORBA Based Distributed System

M a t t i a s A n u é l l
D a n i e l S a n d b e r g

Examiner
Dr.-Ing. Christian Schulte

Master of Science Thesis
Stockholm, Sweden 2005

IMIT/LECS-2005-60

Abstract

Ericsson AB is developing a telecommunication Operation and Maintenance system based on CORBA. Testing of the system includes using a tool called the CORBA Tracer, which is used to perform distributed tracing on the CORBA call level using CORBA Interceptors. Collected data is used to compute and present statistics about the communication between system components.

One part of this thesis project was to create documentation describing the CORBA Tracer implementation, and what functionality it contains, including a complete user guide. This activity derives from the fact that no detailed documentation exists regarding the implementation of the tool.

The main purpose of the thesis was to explore new areas of using the tool. This lead to two new features being implemented and integrated with the CORBA Tracer. The first is used to create statistics used for analysis of parallel CORBA events in the system. The second is used to reconstruct CORBA call chains, which shows the sequence of communication performed as a consequence of a CORBA call made in the system.

Acknowledgements

We would like to thank the following persons for making this thesis work possible and for supplying us with invaluable help and support:

Thomas Friberg, Henric Lovén, Johan Moe, Henrik Samuelsson, Peter Lundin, Göran Ihs, Jonny Sääf and the rest of RGF/V.

We would also like to thank our examiner, Christian Schulte for invaluable advice and feedback.

Table of contents

Table of Figures

Table of Tables

1 Introduction

The development of computer systems today is more and more aimed towards distribution. Distribution of a system is a highly desirable property, allowing information and communication to be shared, stored and used by a multitude of machines and users. With higher performance and connectivity comes also an increased demand for ways of controlling, supervising and evaluating the systems. The inner workings of a distributed system are often complex and difficult to understand and measure. From this fact arises the need for tools able to give a picture of the execution.

The CORBA Tracer is a tool used at Ericsson RGF in testing a CORBA based Operation and Maintenance platform used for supervision of telecommunication networks. The CORBA Tracer tool was developed by Johan Moe as a part of a doctoral thesis in collaboration with Ericsson. It is used to create an overview of how a distributed system behaves during execution and to identify parts of the execution where abnormal behavior occurs. Examples of usage are verifying performance requirements by monitoring response times, finding occurrences of when a normal case raises an unexpected exception, identifying a client making an excessive number of calls to a server and measuring coverage of interfaces used during test.

Since the creation of the CORBA Tracer in the late 1990´s until present day, it has been maintained and evolved by the developer. These premises are about to change and therefore the maintenance and further development of the tool must be handled by an organization. Currently, knowledge at RGF is limited to the usage of the CORBA Tracer. Knowledge regarding the actual implementation is sparse or non-existent. This is much due to the lack of detailed documentation.

Goals

The goals of the thesis are to examine and analyze the CORBA Tracer in order to produce a detailed description of the implementation and functionality for use in further development. Produced documentation should also contain a detailed usage description to be used by testers new to the tool or as reference material. Furthermore, a description of the targeted integrated, common environment, Extended Platform [2], should be produced for the purpose of learning the components in the system. Also, to enable further analysis of the system, two new features are to be implemented. The first should enable the CORBA Tracer to perform measurements of parallel CORBA events in the system. This should give a tester the possibility to relate the degree of concurrency of a CORBA server to the changes in response times. The second feature should contain functionality to extract CORBA call chains. This implies means to see the actual communication sequence spawned by a single call in the system.

Reading guidelines

Section 2 gives some background information on distributed systems, an introduction to distributed tracing, defines concurrency and parallelism and discusses visualization.

Section 3 introduces the basics of CORBA and CORBA interceptors.

Section 4 briefly describes the Ericsson Operation and Support System (OSS), which is the environment for which the CORBA Tracer was developed.

Section 5 describes the CORBA Tracer implementation and functionality.

Section 6 presents the new CORBA Tracer feature; Analysis of Parallel Events

Section 7 presents the new CORBA Tracer feature; Extraction of CORBA Call Chains.

Section 8 presents our conclusion and suggestions on future work.

2 Background

This section discusses background topics relevant to the thesis work. First we give a brief overview of the concept of distributed systems. Second, we discuss the techniques used for distributed tracing related to the solution used by the CORBA Tracer. Finally, we state the difference between concurrent and parallel execution.

2.1 Distributed System Overview

A distributed system is defined as a collection of independent computers working together to appear to its users as a single coherent system.

The definition can be split into two major aspects: the machines comprising the system are autonomous and users of the system should think that they are dealing with a single system. From these aspects can be derived a number of desirable characteristics of a distributed system. The differences between the computers in the system and the means by which they communicate should be hidden from users. This should also be true for the internal organization of the system. Users and applications should be able to interact with a distributed system consistently and uniformly, regardless of the time and place for the interaction.

Expansion of the system should be relatively easy. Also, a distributed system should be available at any time, even if one or more parts of the system are out of order or being replaced or upgraded. These characteristics can be summarized in the following four goals that a distributed system should be able to fulfill.

Connectivity

Users of a distributed system should be able to connect to remote resources easily. Also, it should be possible to share resources without much effort. The sharing of resources in the system must be performed in a controlled way. Resources shared in a proper way make the exchange of information in the system faster and easier. Another advantage may be purely economical. A physical resource, as a printer, may be easily shared by several users instead of each user having its own.

With increased connectivity and sharing of resources, comes the need for increased security in the system. This is an area where there is still room for much improvement, but these aspects will not be discussed in this report.

Transparency

A distributed system is said to be transparent if it can hide the fact that resources are distributed across multiple computers and appear to the user as a single system. There are several different kinds of transparency:

- Access transparency
  is the hiding of differences in data representation and how resources are accessed. Computers in a distributed system may be running different operating systems with different rules and conventions, which should be hidden from the user.

- Location transparency
  means that a user does not have to know where in the system a resource is located. This is achieved through different naming techniques.

- Migration transparency
  means that a resource in the system can be moved without the need to change the way it is accessed.

- Relocation transparency
  is when a resource can be relocated while it is being accessed.

- Replication transparency
  hides the fact that several copies of a resource exists in the system. Replication is a technique used to increase availability, and for this to be realized, all copies must have the same name. Consequently, a system supporting replication transparency should also support location transparency.

- Concurrency transparency
  hides the fact that two or more users are accessing the same resource. Concurrent access to resources raises the need for techniques ensuring consistency in the system.

- Failure transparency
  hides the occurrence of the failure and possible recovery of a resource.

- Persistence transparency
  hides whether a resource is located in volatile memory or is stored on disk.

The transparency of a distributed system varies and a system providing full transparency in all aspects is hardly feasible. This is due to the fact that solutions providing transparency often includes some impact on performance.

Openness

A distributed system is open if the services it offers are defined according to standard rules describing syntax and semantics. This ensures that the system can be created using components from different developers and platforms. Services in a distributed system are generally specified by interfaces, using an Interface Definition Language (IDL). An IDL specification only includes the names of the functions included in the interface, together with parameters, return values and possible exceptions. This gives the possibility to create the actual implementation of the interface functions according to a specific platform or programming language, while a process using the interface functions can be running on a whole different platform.

For this to be possible, the interface specifications must possess the properties of completeness and neutrality. This means that all information needed to create an implementation is included and that the specification has no restrictions regarding the actual implementation structure. These two properties are vital for system interoperability and portability.

Scalability

Scalability is the ability of a system to continue to function well when it is changed in size or volume. Also, it is the ability to take advantage of the rescaling.

A system's scalability can be measured along (at least) the following three dimensions:

- Scalability with respect to size
  users and resources can be added to the system without loss in performance.

- Geographical scalability
  users and resources may be located far apart.

- Administrative scalability
  the system remains easy to manage although spanning many different administrative organizations.

Middleware

Middleware is a general term for functionality to mediate between two separate programs. It typically works as glue, providing messaging services so that different applications can communicate. In a distributed system, middleware refers to an additional layer between the applications and the operating system, hiding the heterogeneity and communication mechanisms of the underlying platform. This way, a user is offered a higher level of abstraction. Two examples of middleware approaches are Remote Procedure Calls (RPC) and Distributed Objects.

- Remote Procedure Calls
  RPC is a protocol putting emphasis on hiding communication mechanisms, letting an application send a request to a procedure located on a remote machine without having to understand network details. Parameters to the procedure are transparently sent to the remote machine. The procedure is executed and the result is sent back to the calling application. To the requesting application, the procedure appears as a local execution.

- Distributed Objects
  in this approach, each object implements an interface hiding the object's internal functionality from users. This interface specifies the methods supplied by the object. An application is only aware of the object's interface and issues its requests accordingly.

Distributed system models

Most distributed systems are based on some variant of the client-server model. A system is structured as a set of services offered by a server process which are requested by client processes. One variant is the proxy-server model, in which the system provides replicas of the resources being managed by several servers. This way performance and availability can be increased by reducing remote accesses to distant or heavily loaded servers.

A main issue regarding client-server architecture is where to draw the distinction between clients and servers. A system is divided into three levels:

- User-interface level
  Constitutes the application's interface to the user.

- Processing level
  Contains application core functionality.

- Data level
  Contains persistent data on which the applications operate and the programs maintaining this data.

In a basic client-server model, clients only implement the user-interface level and servers implement the processing and data level. This is called a two-tiered architecture. In a three-tiered (multi-tiered) architecture, machines are defined as clients and servers, with the addition that a server can also act as a client. For example, an application server executes an incoming request by subsequently issuing a request to a database server.

Apart from these obvious divisions of the three system levels, a system may also implement any subset of functionality from any level on clients or servers. For example, a client can consist of both user-interface and processing level, only using servers as data storage. The above described division of applications into three levels is called vertical distribution.

Horizontal distribution means that clients and servers may be physically split up into logically equivalent parts. Each part is operating on its own share of the complete data set. This way, the load in the system is balanced.

Finally, in peer-to-peer distribution, all components have the same status and similar tasks. There are no distinctions between components as clients or servers. Components may have both client and server capabilities. This technique eliminates the need for a centralized server, but high demands are put upon the coordination and synchronization of resources.

2.2 Distributed trace

Understanding the execution behavior of a distributed system is a difficult task. It is often very hard to predict the dynamics of a distributed system during runtime, which might lead to poor performance, low quality or faulty behavior of the system. A good tool for performing distributed execution trace is not only valuable for the researchers and designers, but also for administrators that performs maintenance of systems in use.

A lot of research has been done about tracing, primarily on debugging during development and in a laboratory setting. There exist different approaches for analyzing the properties of a distributed system. One simple example for a general distributed system is to simply halt the execution of the system at a specific point in time and to analyze the state of each local process in the system. This solution is not always feasible, especially for a distributed system and certainly not for one in commercial use.

The interesting approach for this work is for distributed systems using a communication model with distributed objects, such as CORBA, to perform execution trace at a remote-procedure call level. This means that the communication between the processes in the system is captured to gain understanding of its distributed behavior. This approach has a property that is favorable when wanting to trace systems under different circumstances. It is possible to add trace points without having to instrument the code under test, which might be crucial for a commercial system in use. Other important aspects to support this model are for example that a trace must interfere with the system as little as possible to produce legitimate result data, system analysis should involve little work for the people performing the trace and the visualization of the produced trace data should be easy to use. The CORBA Tracer tool for distributed execution trace supports this approach.

2.3 Concurrent vs. Parallel Execution

In the early days of computing, all programs had only one thread of control. This is what we refer to as sequential execution. A concurrent program is a program with more than one thread of control. The concurrent parts of such a program are those that can be executed independent of each other. In a similar way, the applications in a distributed system can be independently executed. The parallel parts of a program are the concurrent parts being executed at the same time on separate processors. Similarly, distributed system applications may access a resource at the same time, where the resource itself handles the separate requests on separate processing elements or machines. When a concurrent program is executed on a single processing element, the execution is said to be interleaved. The processor is then alternating between executing the concurrent parts of the program. Although the execution is not parallel, it may appear as such to the user.

From this distinction we can say that concurrency is a property of the program being executed, and parallelism is a property of the machine, on which a concurrent program is executed. The following figures illustrate the different concepts.

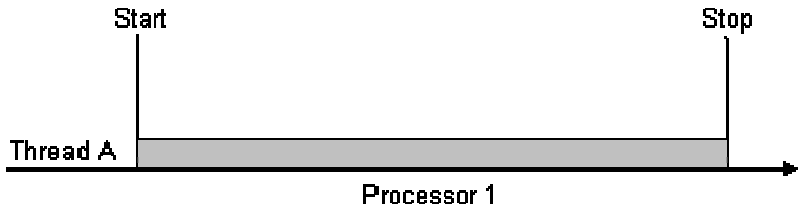

Figure 2.1: *Sequential execution. A single thread is executed on a single processor.*

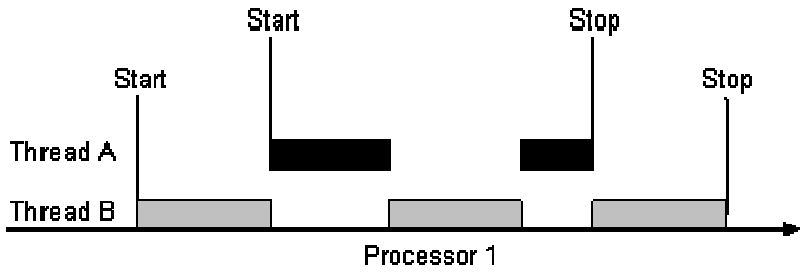

Figure 2.2: *Concurrent interleaved execution. Two threads are alternately executed on a single processor. To a user, execution appears parallel.*

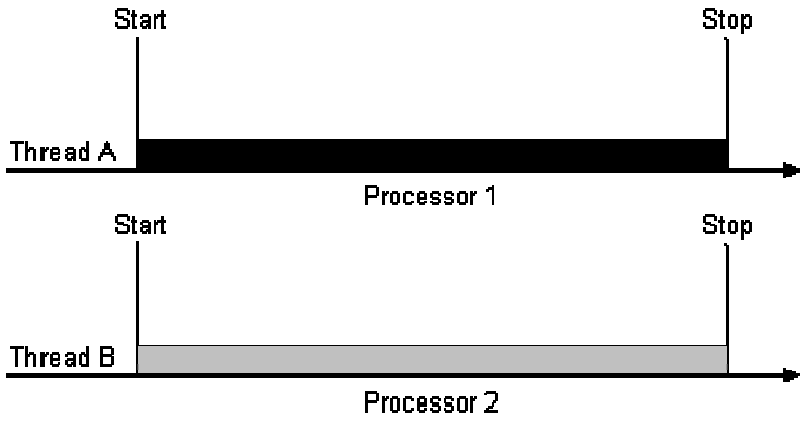

Figure 2.3: *Parallel execution. Two threads are simultaneously executed on separate processors.*

2.4 Visualization

To be able to analyze data in an easy and correct way, visualization is essential. The original version of CORBA Tracer makes use of a customized version of the StarDOM [5] visualization tool to analyze trace summary statistics. The tool is developed at the University of Maryland and implements a subset of functions from Spotfire (www.spotfire.com), a commercial visual data analytics application. StarDOM allows one to create scatter plots of the data, where the user can specify and change the graphical encoding of parameters. For example if the user is interested in the relationship between calls ending in an exception and those ending normally, these parameters can be chosen as the x- and y-axes. StarDOM also implements a technique called dynamic queries, which allows real-time filtering by specifying different values and ranges on the plot axes. The strength in StarDOM lies in the simplicity to build a custom visualization of the trace data.

For the two new features in CORBA Tracer, StarDOM is not suitable since it only can create scatter plots. A desirable visualization tool in this case would be able to show different line plots with functionality to zoom on for example time granularity. A good example would be a line plot of parallel CORBA events where the x-axis represents time and the y-axis represents different parallel statistics. The user could then start with a low time granularity to identify time intervals where the data shows suspicious behavior and to be able to zoom in on those areas to understand the cause. A possible candidate could be the IBM Data Explorer (www.research.ibm.com), which is a system of tools and user interfaces for visualizing data.

Due to the fact that it takes a lot of time to find and adapt to a good visualization tool we decided not to spend any considerable time on this matter. It did not fit in to the time frame of the thesis. Instead we produce white space separated data in the meaning that it should be possible to use it with a variety of visualization tools and different forms of graphs. For example Microsoft Excel (www.microsoft.com) or a script based tool such as Ploticus (http://ploticus.sourceforge.net/).

3 CORBA

The CORBA Tracer is used to monitor CORBA based distributed system. To be able to understand the functionality of the CORBA Tracer, one must first have some basic knowledge regarding the CORBA specification. The following sections briefly describe the parts of CORBA essential to the reader of this report. The complete CORBA specification can be found in [7].

3.1 CORBA overview

The Common Object Request Broker Architecture (CORBA) is an architecture and specification for a distributed system that allows applications to interoperate in a network, regardless of vendor, programming language or location. The architecture and specification of CORBA was developed by a consortium of vendors through the Object Management Group (OMG) with the goal to define a distributed system that could overcome many of the interoperability problems that appears when integrating networked applications. **Figure 3.1** shows the primary components in the CORBA global architecture, specified by the OMG Reference Model architecture.

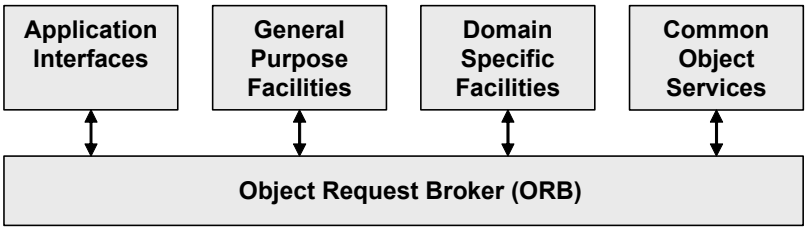

Figure 3.1: *Primary components of the CORBA global architechture.*

The reference model consists of four groups of services connected to the Object Request Broker (ORB).

Application Interfaces

These interfaces are developed specifically for given applications. They are not standardized, because the OMG only handles specifications.

General Purpose (Horizontal) Facilities

These services are horizontally oriented, meaning that they are general purpose, high-level services that are used regardless of the application domain. Still, they are oriented towards end-user applications. They include services for user interfaces, information management, system management and task management.

Domain Specific (Vertical) Facilities

These services have similar roles as the Horizontal Facilities but are targeted towards specific application domains. For example telecommunications, medical, banking and manufacturing domains.

Common Object (CORBA) Services

These are domain/application-independent, general-purpose interfaces that are used by many distributed object programs. They are an important part of the CORBA Reference Model architecture and each service is briefly described below:

- **Collection service**
  is used to group objects into lists, queues and sets

- **Query service**
  is used to query collections of objects using a declarative query language

- **Concurrency service**
  offers locking mechanisms to allow concurrent access to shared objects

- **Transaction service**
  used to define method invocations over multiple objects in a single transaction

- **Event service**
  facilities for asynchronous communication and interrupts

- **Notification service**
  used for filtering of asynchronous communication based on events

- **Externalization service**
  used for marshaling and unmarshaling of objects for disk storage or sending objects over a network

- **Life cycle service**
  provides means to create, destroy, copy or move objects

- **Licensing service**
  allows attachment of a license to an object

- **Naming service**
  used to give an object a human-readable name

- **Property service**
  used for associating attribute-value pairs with objects

- **Trading service**
  used to publish and find services offered by an object

- **Persistence service**
  used to persistently store objects on disk

- **Relationship service**
  used for expressing relationships between objects

- **Security service**
  provides mechanisms for authentication, authorization, auditing, secure channels, nonrepudiation and administration

- **Time service**
  gives the current time within specified error ranges

The Object Request Broker (ORB)

The ORB forms the core of the CORBA distributed system. It is a logical entity that may be implemented in various ways, as one or more processes or a set of libraries and network resources, enabling communication between objects and their clients while hiding issues related to localization and compatibility. ORB support in a network means that a client program can request services from a server program or object without having to understand where the server is in a distributed network or how it is implemented. The ORB provides a mechanism for transparently communicating client requests to target object implementations. The client only needs to know the name of the object and how to use the object's interface. This makes client requests appear to be local procedure calls. The ORB handles localization of the object, routing of the request and the returning of the result.

The CORBA object model

In CORBA, an object is defined as an entity residing in the address space of a server. It consists of an identity, an interface and an implementation. The object implementation part is also known as the servant, and is the part of the object implementing the methods available for client invocation. The structure of a servant is dependent of the programming language used. Object interfaces are specified in the Interface Definition Language (IDL) and are maintained by the ORB.

The object interface is a specification of which methods an object implements and what parameters to use with these methods. No semantics are described in the IDL interface. The object identifier is supplied by the Portable Object Adapter (POA), to which the servant is registered. A POA is a component responsible for making server-side code appear as CORBA objects to clients. When a servant is registered with a POA, it returns an object identifier for that servant. This identifier is encapsulated into a structure functioning as a system wide object reference, also identifying the POA and the server where it is located. This identifier is known as an Interoperable Object Reference (IOR).

When a client issues a request to an object, the initial object reference is fetched by the ORB. The request is sent via a proxy, a client side stub implementing the IDL specified interface of the targeted object. The stub marshals the invocation request and sends the request to the server. The response from the server is unmarshaled and passed back to the client. On the server side, the POA forwards the incoming request to the correct object. Unmarshaling on the server side is done by using stubs, called skeletons in CORBA.

Client/Server communication in CORBA is performed using a standardized protocol, named the General Inter-Orb Protocol (GIOP). GIOP, similar to CORBA itself, is not an actual protocol, but merely the specification of a protocol. The realization of the protocol is executed on top of an existing transport protocol, for example TCP. Running GIOP on top of TCP is called the Internet Inter-Orb Protocol (IIOP).

3.2 CORBA Interceptors

An interceptor is a mechanism by which an invocation can be intercepted on its way from client to server (**Figure 3.2**), and adapted as necessary before letting it continue. There may be various interceptors added to an ORB. Which one is actually activated depends on the object or server that is referenced in an invocation request.

Portable interceptors are an OMG standardized feature that allows writing of portable code as interceptors, which can be used with different ORB vendors. CORBA Tracer uses VisiBroker Interceptors, which are specific to the VisiBroker ORB implementation.

CORBA Tracer interceptors

The interceptors used in the CORBA Tracer are implementations of the following VisiBroker interceptor interfaces:

- **POALifeCycleInterceptor**
  A POALifeCycleInterceptor object is an Interceptor, which is called every time a POA is created.

- **BindInterceptor**
  A BindInterceptor object is an Interceptor, which is called on the client side before and after binds.

- **ServerRequestInterceptor**
  A ServerRequestInterceptor object is called at various stages in the invocation of a server implementation of a remote object before the invocation and after the invocation both before and after the marshalling of the reply. This Interceptor may be registered by a POALifeCycleInterceptor object at POA creation time on a per-POA basis.

- **ClientRequestInterceptor**
  A ClientRequestInterceptor object may be registered when a BindInterceptor is notified that a successful client bind has occurred and it remains active for the duration of the connection. Two of its methods are called before the invocation on the client object; one before the parameters are marshaled and the other after the parameters are marshaled. The third method is called after the request has completed.

For more information about VisiBroker interceptors, see [3].

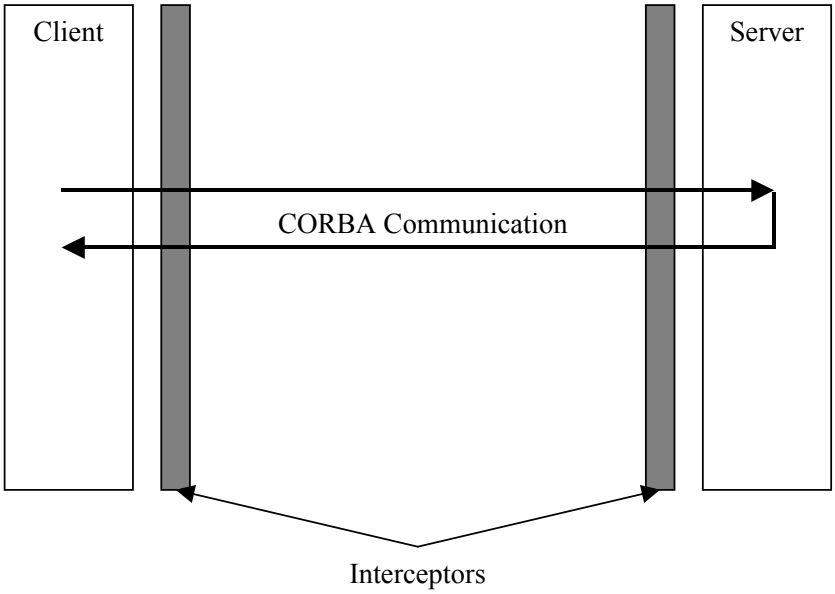

Figure 3.2: *CORBA interceptor overview.*

4 Ericsson Operation and Support System

This section gives a short introduction to Ericsson's Operations Support System (OSS), which is the main system using the CORBA Tracer for test purposes.

OSS is the Ericsson network manager supporting centralized operation and maintenance (O&M) of the radio access as well as the core networks of both Global System for Mobile Communications (GSM) and Wideband Code Division Multiple Access (WCDMA) systems. It provides a common framework for O&M handling of nodes within those networks and the system architecture is based on internationally accepted standards.

The OSS consists of a number of application packages, which are either common to all types of nodes in the managed system or related to specific nodes. These applications are built and integrated on a unified platform, which provides all support services required by an O&M application to manage a network.

The platform has a component based architecture based on CORBA communication. The components are developed using the Java language. It is on this component level that the CORBA tracer collects its data from the system.

5 CORBA Tracer

The following sections comprise the contents of the implementation and functionality documentation produced regarding the CORBA Tracer. First, a general overview of the CORBA Tracer tool and some related work is given. This is followed by a full description of the implementation, including a complete user guide.

5.1 CORBA Tracer general overview

CORBA Tracer is a tool used for tracing a distributed system based on Borland VisiBroker [3] CORBA implementation with the limitation that only Java implemented clients and servers are supported. The tool uses interceptors (section 3.2), which enables tracing of a system without having to alter its source code.

The CORBA Tracer was developed by Johan Moe [1] as part of a doctoral thesis in collaboration with Ericsson. It is owned by Ericsson and is used internally for test purposes, but is not yet feasible as a product.

A system under test can be traced on both client and server side. A client is defined as a process that calls a remote procedure and a server is defined as a process that invokes the remote procedure. System execution is traced at the remote procedure-call level, which means data is collected on the CORBA middleware layer. Trace points are set on client side before and after a remote method call and on server side before and after a remote method invocation. The data that is collected at the trace points is sent to a remote log server using asynchronous CORBA calls. Data on the log server is stored persistently for post analysis. To be able to control the CORBA Tracer, for example turning a trace on and off, each traced server provides an additional CORBA interface to handle this. The control interface for the user is provided by a trace controller application that communicates with the servers configuration interface. A client reads its configuration parameters from a file on its local host. This file is also handled by the trace controller, provided it is executed on the same host. The main components of the CORBA Tracer are illustrated in **Figure 5.1**.

The main areas of usage for CORBA Tracer is to get a picture of how the distributed system behaves during execution and thereby identify parts of the execution where abnormal behavior occurs. This could for example be when normal cases raise exceptions or excessive number of calls to a server.

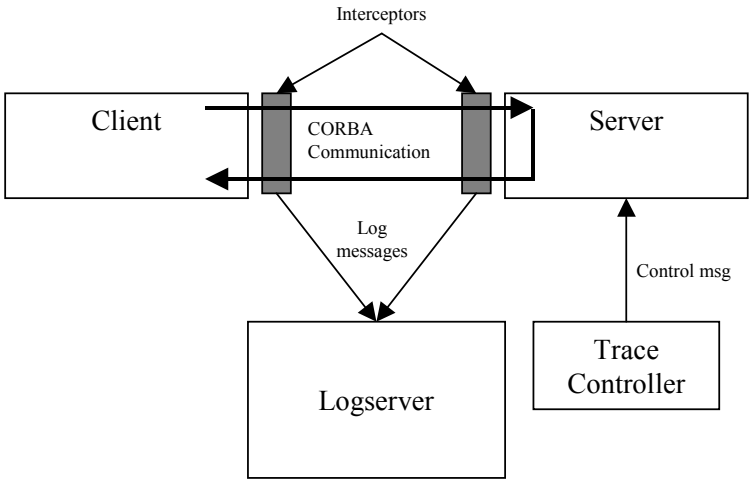

Figure 5.1: *CORBA Tracer component overview.*

5.2 Related work

There are some tracing tools based on similar techniques as the CORBA Tracer. Choi et al. have developed CSMonitor, a visual client/server monitoring and management tool, based on VisiBroker interceptors. The tool provides real time message monitoring of distributed CORBA objects, supplying the user with statistic information. It contains functionality to monitor network traffic and is mainly aimed at performance optimization. Cho et al. uses the information extracted from CORBA tracing to estimate the static and dynamic message flow in a system. The results are then used to improve the object clustering in the system. Application Response Measurement (ARM) is a commercial trace tool using a similar approach. It is an API establishing a standardized application instrumentation approach for distributed Client/Server implementations. An application can use the API to send trace information to a log server. This information can then be analyzed by commercial tools, supporting the ARM specification. A big difference from CORBA Tracer and CSMonitor is that the ARM API originally requires instrumentation of the application code, which is not needed when using interceptors.

5.3 CORBA Tracer Implementation

To describe the implementation of the CORBA Tracer, the execution flow is represented by cases. Each case describes a specific main event in the handling/execution of the applications involved in CORBA Tracer. The case description is made on a class-level. All IDL interfaces referred to in the text are specified in `Trace.idl`. The default package path is `com.ericsson.nms.cif.dt`. If a class is located in a sub directory to the default path, only the sub directory is specified.

5.3.1 Log Server

The Log Server is a standalone application, which is responsible for supplying persistent storage for traced events and trace status for servers. The Log Server implements the IDL interface `TraceLog`.

5.3.1.1 Case: Starting the Log Server

The main method is located in class `TraceLogImpl`. The orb is initialized and a new POA with persistent life span policy is created under the root POA.

A new servant is created which is an instance of the `TraceLogImpl` class that implements the interface `TraceLog`. The arguments set by the user and the orb instance is passed to the constructor, which also creates a default log file where traced events are stored when no session is active. The servant is activated on the created POA and also the POA manager for the new servant is activated. The Log Server is then registered in the CORBA Naming Service. If the system to be traced is distributed and thereby several Naming Services are used, the Log Server will be registered on all of them.

The method `orb.run()`is then executed to make the server start waiting for incoming requests.

5.3.1.2 Case: Incoming CORBA request to the Tracelog interface served by the TraceLogImpl class

Description of these methods will be included in the cases from which they are invoked.

5.3.2 Trace Loader

Trace Loader is not a standalone application; instead its code is executed when a client or server is restarted after the CORBA Tracer has been installed.

The Trace Loader creates a new CORBA object, which implements the interface `ObservationAndControl`. This object is added to the Naming Service. It creates interceptors for the restarted component, registers the component at the Log Server and starts a new thread responsible for sending events to the Log Server.

5.3.2.1 Case: Restart of Java application, client and server

When restarting a Java application after the CORBA Tracer has been installed, the line:
`Dvbroker.orb.dynamicLibs=com.ericsson.nms.cif.dt.TraceLoader` in `system.env` will cause the application to run the methods `TraceLoader.init()` and `TraceLoader.init_complete()` when its `orb.init()` method is executed.

The `TraceLoader` class will be instantiated and its constructor will run the constructor of its super class `ObservationAndControlImpl` that will get and store the server's identity.

The method `TraceLoader.init()` fetches a `VisiBrokerInterceptorControl` object reference through an initial reference. Using this object, two interceptor managers are created, a `BindInterceptorManager` and a `POALifeCycleInterceptorManager`. Instances of the implemented interceptors classes: `PerformanceBindInterceptor` and `PerformancePOAInterceptor` are created and registered with the interceptor managers.

After initialization the method `TraceLoader.init_complete()` is called. If a CORBA server was restarted a new POA with persistent life span policy called "ctrace-poa" is created under the root POA. The servant activated on the "ctrace-poa" is an `ObservationAndControlImpl` class instance that implements the interface `ObservationAndControl`. The newly created CORBA object is associated with its logical name and registered in the Naming Service.

A new instance of the threaded `EventSender` class is created. This class is responsible for sending events to the Log Server. The constructor instantiates the `EventPool` class which holds a linked list of "empty" and reused events. It also creates two event queues, one for adding new events and one for sending from. A "ping" thread is created that tells the Log Server that this server is alive.
The `EventSender` class then registers to the Log Server. It fetches the CORBA object reference to the Log Server using Naming Service. It invokes the method `TraceLogImpl.registrate()` with some initial data on the Log Server.

The remote method `TraceLogImpl.registrate()`is used when a Log Server client is started. The Log Server stores data about the client in a hash table and assigns it a log id. It also checks for clients that have not pinged for a specific time and removes them from the hash table.

An Event Sender (section 5.3.4) thread is then started which runs the ping thread, checks that the server is registered on the Log Server and then goes into a loop, checking if there are any events on the queue to send.

To fetch the trace status from the Log Server the remote method `TraceLogImpl.getPersistent()` is invoked on the Log Server. This method checks if there is a file containing trace status for the calling client. If the file exists the status is set accordingly, otherwise default values are set.

5.3.2.2 Sub Case: Traced client application binds to a CORBA object

When a traced client application succeeds in binding to a CORBA object the method `bind_succeeded()` in the `PerformanceBindInterceptor` class is executed. This method first checks that it is not the Log Server the application is talking to. If not, a `ClientRequestInterceptorManager` object is created and an instance of the `PerformanceClientInterceptor` class is created and registered with the interceptor manager.

5.3.2.3 Sub Case: Traced server application creates a new POA

When a traced server application creates a new POA the method `create()` in the `PerformancePOAInterceptor` class is executed. A `ServerRequestInterceptorManager` is created and an instance of the `PerformanceServerInterceptor` class is created and registered with the interceptor manager.

5.3.2.4 Case: Incoming CORBA request to the ObservationAndControl interface served by the ObservationAndControlImpl class

Description of these methods will be included in the cases from which they are invoked.

5.3.3 Request Interceptors

The request interceptors gather trace data at different points in the invocation of a remote object.

5.3.3.1 The Event class

When a request interceptor executes, the collected data is stored in an instance of the `evex.Event` class. Pre-allocated `evex.Event` instances are fetched from an Event-pool, created when the traced client or server is started. When the collected data is written to persistent storage on the Log Server, the `evex.Event` instance is returned to the Event-pool and reused.

The following data is collected and stored in the `evex.Event` object:

-Server host identity
-Client host identity
-IDL interface
-Method name
-POA identity
-Exception message
-Arguments
-Returned result
-Client start time
-Client stop time
-Server stop time
-Client response time
-Server response time
-Memory usage
-Number of server binds

5.3.3.2 Case: PerformanceServerInterceptor event

During invocation of a CORBA object on a traced server the interceptor will be called on three different occasions. Each call point is represented by a specific method in the interceptor class. These are `preinvoke()`, `postinvoke_premarshal()` and `postinvoke_postmarshal()`.

- `preinvoke()`
  This method is called before invocation of the remote object. First it checks its trace status to see if the application is to be traced. If so, it fetches a reference to an event object from the event pool. The interceptor closure object is set to the fetched event. A closure object is created by the ORB at the beginning of a sequence of interceptor method calls. The same closure object is used for all calls in that particular sequence. It contains a single public data field, which may be set by the interceptor to keep data that can be retrieved later by another interceptor method. Server data is stored in the event object. If a piggyback message is attached to the payload of the method invocation, the server knows that the client is also tracing. To reduce the number of calls to the Log Server, the collected data is sent back to the client by piggybacking on the result payload and the client sends the data to the Log Server.

- `postinvoke_premarshal()`
  This method is called after the invocation of the remote object but before marshalling of the reply. First it checks if an exception has been raised in the remote object during invocation. If so, the event reference is fetched from the closure object. Information about the exception is added to the event.

- `postinvoke_postmarshal()`
  This method is called after the invocation of the remote object and after marshalling of the reply. The event reference is fetched from the closure object. Data is added to the event object. If the client is not tracing or an exception occurred, the event is added to the event queue. If the client is tracing, the data is piggybacked with the reply to the client, which will add the event to its event queue. The Event Sender is responsible for sending events to the Log Server (section 5.3.4).

5.3.3.3 Case: PerformanceClientInterceptor event

During invocation of a CORBA object on a traced client the interceptor will be called on three different occasions. Each call point is represented by a specific method in the interceptor class. These are `preinvoke_premarshal()`, `preinvoke_postmarshal()` and `postinvoke()`.

- `preinvoke_premarshal()`
  This method is called before invocation of the client object and before marshalling of the parameters. First it checks its trace status to see if the application is to be traced. If so, it fetches a reference to an event object from the event pool. The interceptor closure object is set to the fetched event for use in the following interceptor methods. Client data is stored in the event object.

- `preinvoke_postmarshal()`
  This method is called before invocation of the client object but after marshalling of the parameters. The event reference is fetched from the closure object. Client data is stored in the event object. A message is added to the payload sent to the server to indicate that the client is tracing.

- `postinvoke()`
  This method is invoked after the request has completed. The event reference is fetched from the closure object. If an exception has been raised during the remote invocation, exception data is stored in the event object, the event is sent to the Log Server and the method returns. If no exception occurred, data is stored in the event object and if the server is also tracing its piggyback data is added to the event object. Finally the event is added to the event queue. The Event Sender is responsible for sending events to the Log Server.

5.3.4 Event Sender

The Event Sender is a thread, spawned by the Trace Loader. It is responsible for supplying an event queue and sending events to the Log Server.

5.3.4.1 Case: Event Sender delivers events to the Log Server

This case starts when an interceptor is done collecting data and wants to send it to the Log Server through a call to the `EventSender.send()` method. When the event is added to the event queue, the stop time of the event is compared to the current time offset for the queue. Time offset is set to the lower of these two values. If the event queue is larger than the specified size, the waiting loop in the Event Sender thread will be released. How to set the queue size is described in section 5.3.5.4.

The run method in the `EventSender` class is an infinite while loop. The loop starts by going into a waiting state until either a interceptor notifies it if the queue is full or the specified send interval timeout is reached. How to set the send interval is described in section 5.3.5.4. When the thread is notified it checks if there are any events in the queue to send, if not, it waits again. Note that Event Sender contains two event queues, one for storing and one for sending. When it is time to send the events in the current event queue, incoming events are stored in the other event queue. All the events in the sending queue are converted to the proper byte format (section 5.3.4.2) and sent to the Log Server using the remote method `TraceLogImpl.WriteLogStream()`. This method starts by checking the number of events stored in the current log file. If the maximum number of events stored is reached, a new log file is created. Info about the events is stored in the calling clients hash table entry, which was created when the client registered on the Log Server. Finally the byte coded event data is written to the current log file.

The thread then continues to loop, waiting for new events to send.

5.3.4.2 Case: Encoding events to byte code format

Before the Event Sender delivers the contents in the event queue to the Log Server, it is compressed into byte code in order to keep down the amount of data sent.

Information about all the events is encoded into two byte arrays. All event data that is in string format isencoded into one of the byte arrays using enhanced Java UTF-8 encoding. All the other event data (boolean, long, short and octet) are encoded to the other byte array in a compressed format (integers are written to shorts if possible and so on). To separate the data for the Log Reader (section 5.3.6), different tags are inserted to indicate what value the following byte(s) corresponds to.

5.3.5 ObservationAndControlHandler

The ObservationAndControlHandler is a standalone application, which provides a command interface to the user through which the CORBA Tracer can be configured. The main method starts by examining the arguments and setting parameters accordingly. The orb is initiated and an object reference to the Naming Service is fetched. If there are additional Naming Services in the system, information about these is also fetched. Next, different methods may be invoked depending on the arguments given by the user as described below.

5.3.5.1 Case: Listing traceable components

To list traceable components the main method in class ObservationAndControlHandler must be invoked with the corresponding option. If no name(s) is specified after the option, all traceable components will be listed. It is also possible to enter the name of one or more components or Name Services (all traceable components registered) to be listed. It is also possible to list only the client trace status.

For each Naming Service in the system, its binding list is fetched. From these binding lists, all traceable components as specified are filtered out. If the client trace status is to be listed, the `ctrace-clients` file is read and printed and then the method returns. The file contains the current trace settings and is located on the local host. If the file does not exist, default settings are used. Each of the traceable components implements the CORBA interface `ObservationAndControl`. To fetch the trace status from all traced servers the remote method `ObservationAndControlImpl.stat()` is invoked.

Finally all the fetched status data is printed and the application exits.

5.3.5.2 Case: Setting component trace mode

A component can be in one of three trace modes, activated by invoking the main method in class `ObservationAndControlHandler` with the corresponding option. The following trace modes are available:

- Performance mode, limited trace data is sent to the Log Server.
- Coverage mode, in addition to the regular trace data, information about the parameters sent during the trace is added.
- Off mode, no trace data is collected for the specified component.

If no name(s) is specified after the option, all traced servers will set to the specified mode. It is also possible to enter the name of one or more components or Naming Services (all traceable components registered) to be set to the specified mode. If trace is to be performed on the client side, the component name must be replaced with the corresponding option. When a trace has started, all events from that trace will be stored in a default directory at the Log Server until a session (section 5.3.5.3) is started.

Each of the traced components implements the CORBA interface `ObservationAndControl`. From the arguments specified, the corresponding components CORBA object reference is fetched. To set the components trace mode, the remote method `ObservationAndControlImpl.set()` is invoked. This method saves the new trace mode locally and also stores it at the Log Server by invoking the remote method `TraceLogImpl.persistent()`, which saves the new trace mode to the components corresponding trace status file. If no such file exists, one will be created.

If trace is to be performed on the client side, the new trace mode is set to the locally stored `ctrace-clients` file without using the CORBA interface.

Finally the application exits.

5.3.5.3 Case: Starting a session

If no session is set, all traced events are stored in a default directory. When a session is started, all traced events will be stored in a specific directory, which name is related to the name of the session. When a session is stopped, all traced events will again be stored in a default directory.

To start and stop a session the main method in class `ObservationAndControlHandler` must be invoked with the corresponding option.

A CORBA object reference to the Log Server is fetched using the Naming Service. The Log Server provides the IDL interface `TraceLog`. To start the session at the Log Server the remote method `TraceLogImpl.session()` is invoked. This method creates a directory with the name of the session and connects the Log Servers output stream to a file in that directory. If the session is stopped, the output stream is redirected to a new file in the default directory.

Finally the application exits.

5.3.5.4 Case: Setting queue size and send interval

It is possible to set the following parameters for the `EventSender` class for each traced component:

- Queue size
  This parameter decides the maximum number of events stored in the event queue before it has to be emptied by sending the events to the Log Server.

- Send interval
  This parameter decides how often the Event Sender checks its event queue if there are a sufficient amount of events there to send them to the Log Server.

To set the parameters, the main method in class `ObservationAndControlHandler` must be invoked with the corresponding options.

If no name(s) is specified, all traced components will set the specified parameters. It is also possible to enter the name of one or more components or Naming Services (all traceable components registered) to be set with the specified parameters. If to be set on the client side, the component name must be replaced with the corresponding client option.

Each of the traceable components implements the CORBA interface `ObservationAndControl`. From the arguments specified, the corresponding components' CORBA object reference is fetched. To set the components' parameter values, the remote method `ObservationAndControlImpl.setSend()` is invoked. This method saves the new parameter values locally and also stores them at the Log Server by invoking the remote method `TraceLogImpl.persistent()`, which saves the new parameter values to the components' corresponding trace status file. If no such file exists, one will be created.

If the client option is set, the new parameter values are locally stored to the `ctrace-clients` file without using the CORBA interface.

5.3.6 Log Reader

The Log Reader analyzes traced raw data (section 5.3.4.1) stored by the Log Server and creates output files containing data and statistics formatted to be visualized in StarDom [5], a presentation tool developed at University of Maryland. Passing different options to the Log Reader application can change the extent of the details in the files. The Log Reader is a standalone application.

The trace data is post analyzed. The main reason for this approach is that analyzing data in real time would have too much impact on the system under test. Using post analysis, it becomes possible to calculate all the wanted statistics without having to take memory usage into consideration.

The main method starts by examining the arguments and sets parameters accordingly. If not set to quiet mode, the Log Reader will try to contact the Naming Service, Log Server and Interface Repository in order to close any open files and fetch the IDL interface specification. The Log Reader processes one session at a time. If contact has been made with the Log Server, the remote method `TraceLogImpl.closeAndCreateFile()` is invoked before the data processing is started. This method closes any open trace file at the Log Server so that it becomes available for reading.

For each session the Log Reader starts by reading the trace file and restoring the raw data one event at a time. The `EventPool` class is used, which holds a list of allocated event objects that are reused when out of scope. The read events are added to a structure that is sorted accrding to the invocation time at the client or server depending on if the client side was traced. Events that are one-way or raised an exception are temporarily stored to see if they can be matched and combined with its corresponding event if both sides were traced. This procedure is done until either the sorted structure reaches its specified maximum size or that the time between the first and last event in the sorted structure guarantees that if the first event is removed, no added events in the future should precede the removed one.

When the above conditions are met, the first event is removed. It is possible to instruct the Log Reader to print the event's raw data to the file `trace.txt` and its arguments to the file `parameters.txt`. Then the event's statistics are calculated for each time window, which default are complete file, 24 hours and one hour. The time windows minute and second can optionally be set. A time window defines the time interval for which statistics are to be calculated. It is also possible to instruct the Log Reader to calculate additional client-server relation's statistics. When all the events for a time window have been processed, it is written to a result file, `stat-S.stardom`. Client –server relation statistics are stored in the file `stat-CS.stardom.`

When all events have been read, the remaining events in the sorted structure are processed and the data and statistics are written to the result files. If the coverage mode is set, all the methods that were not called compared to the interfaces and methods in the file `irep.db` will be added to the largest time window. This enables the user to see how many methods that were covered by the trace. To create the `irep.db` file the Log Reader must force a load of IDL data from the Interface Repository to the database file. This option cannot be combined with the quiet mode because the Log Reader has to communicate with the repository. It is also possible to set an option to the Log Reader with the path to the directory containing IDL files. This causes the files to be copied to the current session directory.

5.3.6.1 Result files contents

This chapter describes the two files that contain the trace data analyzed by the Log Reader. The data is structured in order to be visualized using StarDom, which requires tab-separated data, with the addition that the data type contained in each column is specified in the headers. Statistics are shown for each time window, which default are complete file, 24 hours, one hour and optionally, minute and second. Thus, each row in the output file corresponds to a specific time interval for the current time window.

The following is a description of each column in the default created file, `stat-S.stardom:`

- Date
  Specifies the time interval for the data in the corresponding row.

- Calls
  Number of calls made from client to server.

- Interface
  The server side IDL interface.

- Operation
  The invoked remote method.

- Object
  The name of the object implementation. The value is always NULL, since it is fetched using a deprecated method.

- Server Host
  Name of the host on which the invoked method is executed.

- Server Identity
  Server POA name.

- Time Window
  Size of the time interval of the current row.

- Exceptions
  Number of exceptions that occurred.

- Response Time tot,avg,min,max,variance,50perc,90perc
  Response time states the server invocation response time if only the server is traced. If the client is traced it states the time between a client request and the reply.

- Parameter Load tot,avg,min,max,variance,50perc,90perc
  The size in bytes of the arguments to the remote method.

- Return Load tot,avg,min,max,variance,50perc,90perc
  The size in bytes of the return value.

- Server Connect tot,avg,min,max,variance,50perc,90perc
  The number of times the client connects to the server.

- Memory tot,avg,min,max,variance,50perc,90perc
  Memory available in the Java Virtual Machine on the server.

- Specification
  The name of the IDL file specifying the used interface.

The following is a description of the additional columns in the file, `stat-CS.stardom`, which shows client-server relation statistics:

- Client Host
  Name of the host on which the remote call was made.

- Client Identity
  Identity of the traced client. Either default name or set with the –D option to the Java interpreter.

- Client Coupling
  The number of calls from the client to the specific server, divided by the number of calls from the client to all servers.

- Parameter Cohesion
  The total parameter load from the client to the specific server, divided by the total parameter load from the client to all servers.

- Return Cohesion
  The total return load to the client from the specific server, divided by the total return load to the client from all servers.

5.4 User Guide

This user guide describes the installation and usage of the CORBA Tracer. For the installation procedure it is assumed that all necessary files are located in the same directory.

5.4.1 Installation

To install CORBA Tracer the shell script `install.sh` is executed. It creates the following directories:

- `/opt/ericsson/nms_cif_dt/bin` containing the files:
  - `startCOHandler.sh`
  - `startCorrelator.sh`
  - `startFillIrep.sh`
  - `startIREP.sh`
  - `startlogreader.sh`
  - `startMakeIDLData.sh`
  - `startPresentation.sh`
  - `startTraceLogImpl.sh`
- `/opt/ericsson/nms_cif_dt/lib` containing the files:
  - `nms_cif_dt.jar`
  - `nms_cif_dt_3pp.jar`
- `/var/opt/ericsson/nms_cif_dt/idl` containing the files:
  - `IDLData.CIF`
  - `IDLData.FM`

Also the following lines are added to the file/etc/opt/ericsson/system.env:

- `JAVAFLAGS=\"\$JAVAFLAGS -Dvbroker.orb.dynamicLibs=com.ericsson.nms.cif.dt.TraceLoader\"\`

Every time a Java application is started, this option will tell the Java interpreter to run the methods in class `TraceLoader` when the application initializes the orb.

- `CLASSPATH=\$CLASSPATH:/opt/ericsson/nms_cif_dt/lib/nms_cif_dt.jar`

This option tells the Java interpreter where to find the CORBA Tracer classes.

If the message:

"ERROR: in system.env, please remove the following lines and rerun this script:"

is displayed, make a backup of the `/etc/opt/ericsson/system.env` file and then remove all the lines containing “nms_cif_dt.jar” from the original file. Run the installation script again.

5.4.2 Starting the Log Server

The Log Server has to be started before any tracing can be performed.

/opt/ericsson/nms_cif_dt/bin/traceLogImpl.sh [-z] [-y] [-v] [-c] [-n <*number of writes*>] [-?]

Description of the options:

`-z` Log files are stored in the zip format.

`-y` Prints the CORBA Tracer version (as defined in the .jar file manifest) and then exits.

`-v` Enables the verbose mode, additional output is printed.

`-c` Coverage (not applicable, only for C++ interceptors).

`-n` The following argument specifies the number of events written before creating a new log file.

`-?` Prints usage help.

5.4.3 Restarting components

To be able to perform a trace on a component (client or server), it has to be restarted after CORBA Tracer has been installed in order to load the Trace Loader.

The following properties must be set to the Java interpreter:

- `-Dvbroker.orb.dynamicLibs=com.ericsson.nms.cif.dt.TraceLoader`

In order to run the methods in class `TraceLoader` on ORB initialization.

- `-DSVCnameroot=NameService`

Bootstraps the application to obtain an initial object reference to a specified Naming Service.

The following properties to the Java interpreter are optional:

- `-DnameServices=value`

This property tells the Log Server in which Naming Services it should register its object implementation, `value` corresponds to one or more Naming Services separated with “,;”.

- `-Duser.home=path`

Change the default home directory to `path`.

- `-DTraceClient=true`

If the property is set to true, the client will get the identity "ctrace-clients" and no TraceController interface will be provided by the application.

- `-Didentity=name` or `–Ds=name`

The server’s identity is set to `name`.

5.4.4 Listing traceable components

/opt/ericsson/nms_cif_dt/bin/startCOHandler.sh –l
{<*component name*> | <*Naming Service name*> | [-a]}

Option description:

-a Only the current trace status for the clients is listed.

component name specifies the name of one or more components.

Only the specified components are listed, if they exist.

Naming Service name specifies the name of one or more Naming Services. Only the components registered in the specified Naming Service are listed, if it exists.

5.4.5 Setting queue size and send interval

/opt/ericsson/nms_cif_dt/bin/startCOHandler.sh
–q <*queue size*> -i <*send interval*> {<*component name*> |
<*Naming Service name*> | [-a]}

Option description:

-a Only the current trace status for the clients is changed.

queue size decides the maximum number of events stored in the event queue before it has to be emptied by sending the events to the Log Server.

send interval decides how often the Event Sender checks its event queue if there are a sufficient amount of events there to send them to the Log Server.

component name specifies the name of one or more components.

Only the specified components are changed, if they exist.

Naming Service name specifies the name of one or more Naming Services. Only the components registered in the specified Name Service are changed, if it exists.

5.4.6 Setting the trace mode

/opt/ericsson/nms_cif_dt/bin/startCOHandler.sh {-p | -c | -o} {<*component name*> | <*Naming Service name*> | [-a]}

Option description:

`-p` Sets the trace mode to performance.

`-c` Sets the trace mode to coverage.

`-o` Turns the tracing off.

`-a` Only the current trace mode for the clients is changed.

component name specifies the name of one or more components.

The trace mode is only changed for the specified components, if they exist.

Naming Service name specifies the name of one or more Naming Services. The trace mode is only changed for the components listed in the specified Naming Service, if they exist.

5.4.7 Creating sessions

By creating a session the traced data is stored in a specified directory.

/opt/ericsson/nms_cif_dt/bin/startCOHandler.sh –s {<*session name*> | stop}

session name specifies the name of the session and also the name of the directory where trace data will be stored (`/var/opt/ericsson/nms_cif_dt/sessions/session name`).

To stop a session, instead of a session name, “stop” is given as an operand. This causes the trace data to be written to the default directory.

5.4.8 Analyzing traced data with the Log Reader

`/opt/ericsson/nms_cif_dt/bin/startlogreader.sh [-m] [-n] [-v] [-I] [-?] [-d] [-c] [-p] [-r] [-a <script name>] [-q] [-s <session name>] [-i <IDL directory>] [-f]`

Option description:

`-m` A new time window is added to the output files. It shows statistics for each minute. Default time windows are complete file, 24 hours and hour.

`-n` A new time window is added to the output files. It shows statistics for each second. Default time windows are complete file, 24 hours and hour.

`-v` Enable verbose mode.

`-y` Prints the version number from `nms_cif_dt.jar`.

`-?` Prints usage help.

`-d` Create additional data for client – server relations, located in file `stat-CS.zip`.

`-c` Adds all methods in the irep.db that has not been called to the largest time window. If the `-f` flag is also used, the database file will be loaded from the Interface Repository.

`-p` If a coverage trace has been performed a separate file `parameters.txt` is created containing parameter values of all method calls.

`-r` Creates a separate text file `trace.txt,` containing all data for each event.

`-a` The following argument specifies the name of the script to be executed when the Log Reader has finished.

`-q` Enables silent mode. No contact will be established with remote objects.

`-s` The following argument specifies the session that will be processed. If no session is specified, all sessions will be processed.

`-i` The following argument specifies where to fetch the IDL file to be added in the session directory.

`-f` Forces a load of IDL data from the Interface Repository and creates a database file containing the fetched data.

5.4.8.1 Result files

The result files are located in the `/home directory/sessions/session` directory. The following files can be created depending on the arguments to the Log Reader:

- `stat-S.zip`, containing:
  - IDL files specifying the interfaces used.
  - `stat-S.stardom`

    File containing the trace result data (see chapter 6.6.1).
  - `scripts.scr`

    Script file used by StarDom making it possible to read IDL files or parameter files from within StarDom.
  - `readme.txt`

    Instructions on how to run StarDom.
- `stat-CS.zip`

Contains the same files as `stat-S.zip` except that `stat-S.stardom` is replaced by `stat-CS.stardom`, which contains client-server relation trace data (see chapter 6.6.1).

- `trace.txt`

Contains each individual event in the trace in its unprocessed form.

- `parameters.txt`

Contains data concerning all arguments sent during the trace for each method.

- `stat-Error.txt`

Contains information if the Log Reader failed during matching of events.

6 Analysis of Parallel Events

This section describes the implementation of the new CORBA Tracer feature used to create statistics on parallel CORBA events in a traced system. The section starts by describing the background leading to the need for this functionality. This is followed by an implementation description of the chosen solution. Finally we discuss our conclusions and suggestions for future work.

The CORBA Tracer is used at Ericsson RGF for verification and fault searching purposes. Originally, the CORBA Tracer possesses no means to analyze the collected data in terms of parallel events. The goal of this part of the thesis work is to use the structure of the existing CORBA Tracer implementation to produce additional output data describing parallel CORBA events in the traced system. The main purpose is to monitor the degree of concurrency on CORBA servers and mirror this on the change of response times. In this context, the concept of concurrency is interpreted as interleaved execution of two or more CORBA servant invocations on the same host. In other words, when the execution time intervals of two invocations coincide. A CORBA server is defined as a process serving remote method calls either by using threads or not, depending on the policy. According to our definition, concurrency can occur only when a method uses a multiple thread policy.

The major relevant changes to the implementation lie in the Log Reader, by adding a new parse mode. This new mode produces a number of data and statistics concerning concurrency from the collected trace events. No specific trace mode needs to be used when collecting data; instead a new measured value is collected by the CORBA Tracer in addition to the original measurements to be able to produce parallel statistics.

When parsing the trace data using the added mode, one parallel statistics file is created for each traced CORBA server. The data is tab separated, a format widely accepted by a variety of visualization tools. Each row in the files contains data for all traced methods for one second of the trace, based on measurements performed with millisecond precision. The following statistics are calculated:

Total measurements

Combined calculations for all traced methods for a specific server:

- Total number of events
- Maximum concurrency
- Maximum response time
- Average response time

Method specific measurements

Calculations for each traced method.

- Total number of events
- Concurrent states - max, 50p, 90p
- Total time for maximum concurrency during specific second
- Number of started events
- Response time - max, average, 50p, 90p
- Maximum concurrent state during maximum response time
- Average concurrency during maximum response time
- Total number of events during maximum response time

6.1 Problem definition

The goal of this activity is to use the structure of the Log Reader and the trace data to produce new data describing concurrent activities in the traced system.

To reach a feasible solution the following main issues are recognized:

- How to show parallelism of CORBA events and what data should be produced to meet the requirements?
- Is the data currently produced by the CORBA Tracer sufficient to create the wanted result?
- Can the current structure of the Log Reader be extended to create the wanted result or is an entirely new solution required?
- How to utilize the memory.
- How to visualize the produced data in a good way?

The original version of the Log Reader works as follows: all the traced data for a session is stored in one or more files under a session specific directory. The data is stored according to a predefined tag system and is compressed when possible. The Log Reader reads the data and restores the events one by one. The events are stored in a structure and are sorted according to the events' client start time. When the structure either has reached a certain size or represents a certain time window, statistics are calculated using the first event in the queue. The statistics are stored in a structure and when it reaches a certain size, the finished time windows are written to the result file. This process iterates until all events are processed.

6.1.1 How to show parallelism of CORBA events?

The reason for this activity was the wish to be able to monitor highly utilized CORBA servers and thereby see the degree of concurrency and parallelism and to mirror this on the change of response times. We recall the definition of concurrency and parallelism in section X.X. Concurrent interleaved execution appears when two threads are alternately executed on a single processor during the same time interval. Parallel execution is when two threads are simultaneously executed on separate processors.

In our solution we do not care if the execution is of either kind but instead we define two processes to be parallel when their execution time intervals coincide (**Figure 6.1**). The scheduling made by the operating system is not taken into consideration.

Second, on which level should the parallelism be measured? The relevant choices are either to measure the parallelism between different CORBA servers, or between method invocations in one specific server. Our choice is to monitor the parallelism between invocations of each method on one server. A CORBA server is defined as a process serving remote method calls either by using threads or not, depending on the policy. Only when a method uses a multiple thread policy can parallelism occur according to our definition.

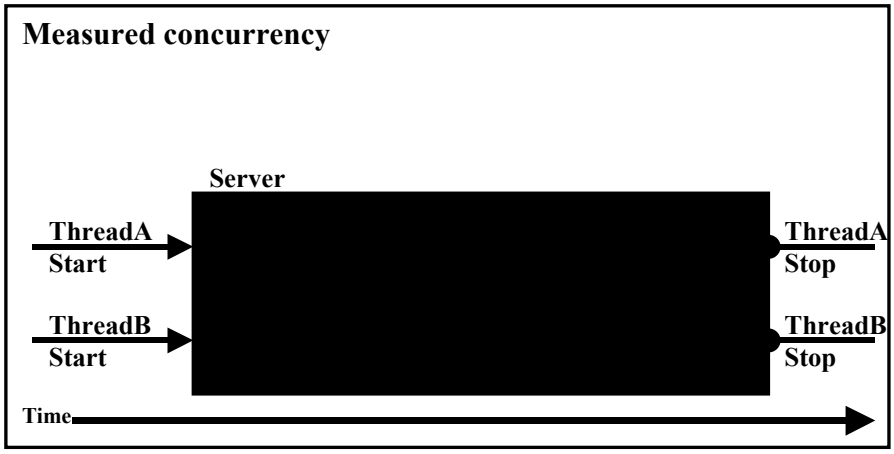

Figure 6.1: *Measured concurrency.*

We want to measure the parallelism as finely grained as possible regarding time. In this case, the CORBA Tracer restricts the accuracy to a millisecond level. Thus, for each method the number of concurrent executions is calculated every millisecond. However, to keep the output data manageable, it is not feasible to show the parallelism on this level. Instead, we choose to produce output data for each second corresponding to the values listed on page 47.

6.1.2 Is the data currently produced sufficient?

In the current version, two time values are measured for each CORBA event; the start time and the stop time. For our calculations, the data needed is the start and stop time of the method invocation on the server side. But currently, if the CORBA Trace is set to monitor client events as well as server events, the values measured are piggybacked with the server response and thereby set to represent the client side start and stop times. The corresponding server values are approximated. To always be able to extract the correct server start time and response time, a new event value is added to CORBA Trace. This value is set to the method invocation start time on the server and is set on an interceptor level.

6.1.3 Is the current structure of the Log Reader sufficient?

It is possible to add functionality to the current implementation, but there are a few issues that have to be considered. Currently, the Log Reader sorts the events according to client start time. In practice, this would be sufficient for our measurements if only a server trace is performed, since then the client and server start times are set to the same value. But if the corresponding client is also traced, the possibility exists that the sorting may not be correct for our purposes. When the data is read from file, it must be sorted according to the invocation start time for our calculations to be correct. If the sorting is made on client request time, the following scenario is possible (**Figure 6.2**):

Clients A and B both make a CORBA request to the same server. Client A makes its request before client B. If the request from client B reaches the server before the request from client A, the client start time for client A will be less than B: s client start time, but the server start time will be less for client B. Thus the sorting will be different.

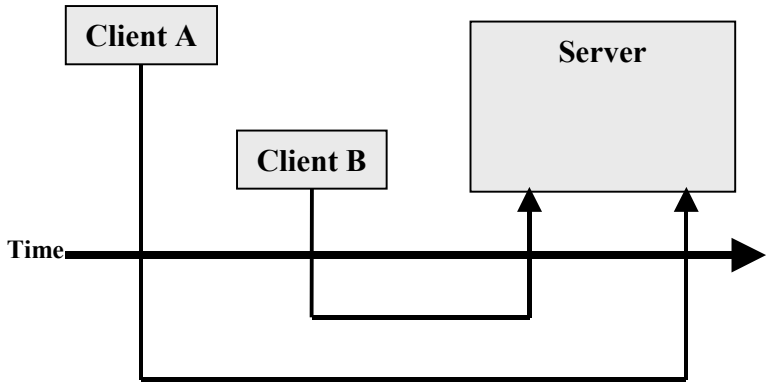

Figure 6.2: *Sorting of client A and client B on client start time will be different from sorting on server start time.*

Another issue is that the Log Reader writes data to file continuously. In our case, this is not possible, since our output structure demands that we know all monitored servers and methods before writing to files.

6.1.4 Memory usage

Since CORBA Trace is used to monitor systems producing a large number of CORBA events, it is vital to utilize the memory in a smart way. This is an issue that becomes even more important considering that the added functionality does not have the possibility to share all the memory resources currently used by the Log Reader.

6.1.5 How to visualize the produced data in a good way?

We want the result data to be in a general format, meaning that it should be possible to use it with a variety of visualization tools and different forms of graphs. A format that is widely accepted is white space separated data. In our implementation, we produce one tab-separated file for each monitored server. The output files contain one row for each second, where the first column contains the time. Each method is represented by a number of columns, containing a number of statistic values for both concurrency and response time. The exception is the first row, which contains the labels. One row represents the interval from the previous second up to the current (**Figure 6.3**).

Time	Total Calls	Total Con. max	Total Res. max	Total Res. avg	printString(Calls tot)
10:43:29	84	5	50	32	42
10:43:30	118	5	50	33	53
10:43:31	122	6	51	31	61
10:43:32	123	7	50	31	62
10:43:33	120	7	50	32	54
10:43:34	115	5	50	33	51
10:43:35	116	5	60	32	59
10:43:36	119	5	50	31	62
10:43:37	111	5	50	33	56

Figure 6.3: *Output file format.*

This format gives the possibility to visualize the data in a number of ways, depending on the needs of the user. The main thought is to put the time on the x-axis and produce different diagrams with the relevant values on the y-axis to be able to see the relations between concurrency and response times. An example is shown in **Figure 6.4**.

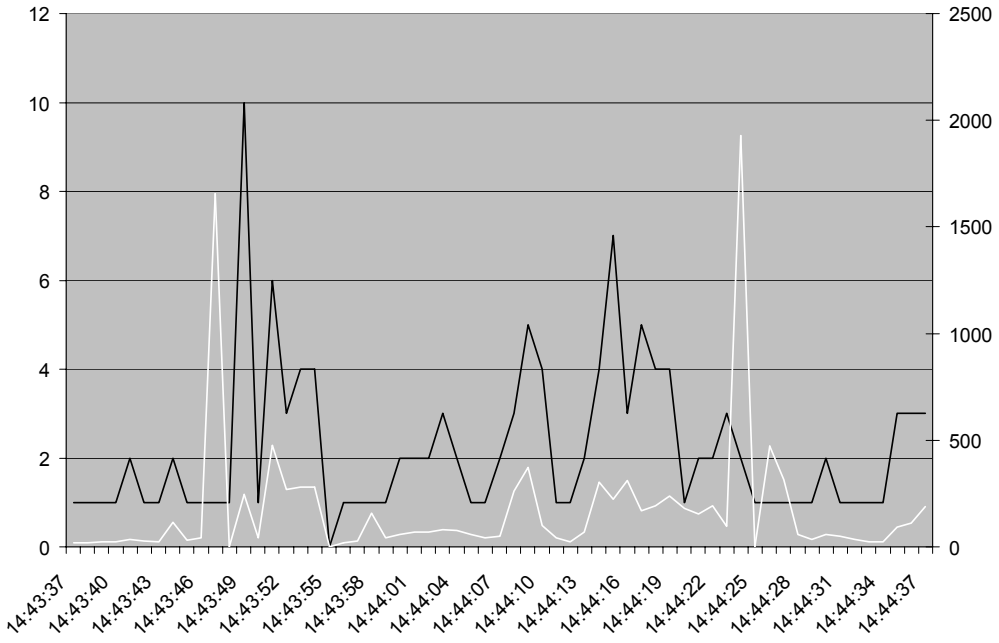

Figure 6.4: *The graph shows the maximum concurrency (black) and response time (white) for one method during a specific time interval.*

6.2 Implementation description

The new functionality for measuring concurrency is completely integrated with the CORBA Tracer implementation. The majority of the changes are realized by two new classes, which enables a new Log Reader parse mode for producing data and statistics concerning concurrency from collected trace event data. This new parse mode is activated through setting the `-l` option to the Log Reader (section 5.4.8). Also an additional value is collected by the interceptors, which is needed to be able to calculate the wanted statistics. The following subsections describe the implementation of the added feature.

6.3 Additional data collection

The CORBA Tracer stores all its monitored events in one or more files in the specified session directory. An event is defined as a method call to a specific process using CORBA. When a CORBA Tracer request interceptor executes, two time values are measured, the invocation start time and stop time. These values are collected on both the client side and the server side. On the client side, start time is when the client sends the request to the server. Stop time on the client side is when the client receives the reply from the server. On the server side, start and stop time defines as when the server starts the invocation and when the invocation returns.

If the CORBA Tracer monitors both sides of a remote invocation, the data collected by the server interceptor is piggybacked with the response to the client to minimize the amount of calls to the Log Server. This means that the data collected in the server side trace point is not sent individually to the log server, but instead it is attached to the reply to the client. The client in turn sends data for both itself and the server to the Log Server. The gain from this behavior is that the number of times data is sent to the Log Server is reduced. This in turn leads to less impact on the system under trace. By doing this the client and server event is combined and only the client side start and stop times are stored. Thus, the server start time can only be approximated and this is not sufficient for the new Log Reader parse mode, which needs to sort events according to this value. Thereby a new field is added. The new field, representing the server start time, is explicitly set in the interceptor and it is not changed if piggybacked. **Table 6.1** shows the values stored by the interceptors in each event class instance.

Table 6.1: *Values collected and stored by the interceptors.*

Field Type/Name	Description	Comment
boolean client	Set if client event	Client side only
String serverHostIdentity	Host identity	Server side only
String ClientHostIdentity	Host identity	Client side only
String poaOperationObjname	Repository identifier + Invoked operation + Object name	
boolean exception	Set if exception occurs	
String exceptionMesage	Exception description	If exception occurs
int messageId	ID of the current request	If one-way or exception
int clientThreadId	Client interceptor set thread ID	Only if coverage mode
int serverThreadId	Server interceptor set thread ID	Only if coverage mode
byte[] argument	Marshalled request	Only if coverage mode
int argumentOffset	Argument offset in marshalled request	Only if coverage mode
int argumentSize	Size of argument	Only if coverage mode
byte[] result	Marshalled result	Only if coverage mode
int resultOffset	Argument offset in marshalled result	Only if coverage mode
int resultSize	Size of result	Only if coverage mode
long stopTime	Client or server stop time. Client stop times if both are traced.	
long serverStartTime	Server start time	Server only
int clientResponseTime	Client response time	Client only
int serverResponseTime	Server response time	Server only
int serverBind	Number of client binds to the server	Server only
int totalMemory	Amount of memory in the JVM	Server only

The server start time field is of type long and represents the difference, measured in milliseconds, between the traced server's start time and midnight, January 1, 1970. The event field stop time is also represented in the same manner. In order to reduce the amount of data sent to the Log Server one would want to send as few fields as possible of type long. Therefore, every time one or more events are sent to the Log Server, the maximum stop time is extracted from the events and the server start time and stop time is recalculated as follow:

```
X = maximum stop time of the events to be sent
S = current event's stop time
T = current event's server start time
R = current event's server response time

S = X – S
T = X - (T + R)
```

The maximum stop time is sent as a separate value and the reversed procedure restores the values when parsed by the Log Reader. The gain by doing this is that one long field is sent instead of two for each event.

6.3.1 Reading and sorting trace data

The main part of the new implementation is located in two new classes. The classes implement a new Log Reader parse mode for producing data and statistics concerning concurrency from collected trace events. When parsing the trace data using the new parse mode, one parallel statistics file is created for each traced CORBA server. The created file has the name
“traced server name”-pdata
and is located in the same session directory as the log file from which the data was parsed. Note that executing the Log Reader with the new parse mode, does not exclude the creation of default statistics.

To create the default statistics, the Log Reader must restore the events from the raw byte format. This is done by reading events one by one from file and storing them in event information class instances. These instances are pre-allocated in a pool and are recycled when used. The events are sorted into a structure according to their client start time field.

To create the parallel statistics, from each event instance created, a new parallel event information class instance is created, containing a subset of the event data. These parallel events are sorted into a separate structure according to their server start time field. Only server side events are saved. **Table 6.2** shows the values stored in each parallel event class instance.

Table 6.2: *Values stored for parallel events.*

Field Type/Name	Description
String serverIdentity	Server identity
String serverHostIdentity	Host identity
String ClientHostIdentity	Host identity
String poaOperationObjname	Repository identifier + Invoked operation + Object name
boolean exception	Set if exception occurs
String exceptionMesage	Exception description
int messageId	ID of the current request

The Log Reader reads events one by one from file and sorts them into a sorted set according to the event's start time, which is calculated from the stop time and the response time. Thus, the start time represents the start of the method call at the client if a client trace is performed. This process continues until a specified number of events have been read or the sorted events represent a specified time gap. When one of these conditions is met, statistics are calculated for the first event. The structure containing this event is then reused, meaning that there exists a predefined number of allocated data structures representing events, which works as a pool.

In our first implementation, a read event was copied into a modified structure and sorted into a separate set, according to the server start time. Since we are only interested in events monitored on the server side, client events were skipped. All events were sorted into this separate set, and statistics were calculated when all events were read. In this way, no effort was made to keep down the memory usage. After testing this solution with large trace files, it became obvious that another approach was needed, since too much memory was used.

The second approach, tried to make use of the same allocated event structures as the original implementation. The events were sorted into two different sets, by using references to the same objects. One set was sorted according to the original implementation and the other was sorted according to the server start time. This solution became too complex and slow, due to the reuse of processed events. It is possible that an event removed from one set and reused, must still exist in the other set and thus be temporarily stored. This caused comparing and searching activities, slowing down the execution too much.

The final solution follows the same idea as the original implementation. When an event is read, it is copied to a new structure, which holds only data needed for concurrency statistics and then sorted into a separate set. When a specified number of events are read, ensuring that if the first event is removed there will not be a faulty sorting for any of the events to be read, statistics can be calculated for the first event. The processed events are deleted to free up resources. This may seem as a long way around, but the thing is that this solution causes problems further down in the processing.

6.3.2 Calculating statistics

When an event is ready for processing, its server and method name is checked against and added to a server map if not already present. The map is a Hash table, which holds server names as keys and stores arrays holding the corresponding servers' methods. This hash table is used when writing the result data to file. Also for each method an object (statistical data) is allocated, which is responsible for holding and calculating statistical data. It contains two arrays, one of length 1000 representing the concurrency for each millisecond; the other is of dynamical length and stores the different response times. These objects are stored in a separate hash table with a concatenated server and method name as keys. The event is then removed from the sorted set and added to a temporary holder. When all the events are read for the currently processed second the holder can be iterated to calculate the statistics. Each event in the holder goes through the following steps:

- A reference to the correct statistical data object is fetched.
- The indices of the millisecond array corresponding to the event's execution time are increased by one.
- If the event starts execution within the current second, the response time is added.
- If the event stops its execution within the current second, it is removed from the holder.

When all events in the holder have been processed, the statistical data objects compute the statistics for both concurrency and response time. The results are then stored as an array in a hash table, with a concatenated server and method name as keys, which in turn is added to a file output queue (**Figure 6.5**). The reason why we cannot write the result data directly to the result files is that we have to know the names of all monitored servers and methods in order to create the correct labels. When the output queue reaches its maximum allowed size, the contents are written to a temporary file. This is done to save memory. The statistical data objects are then reset and reused. The time is increased and the collection starts for the events executed in the next second.

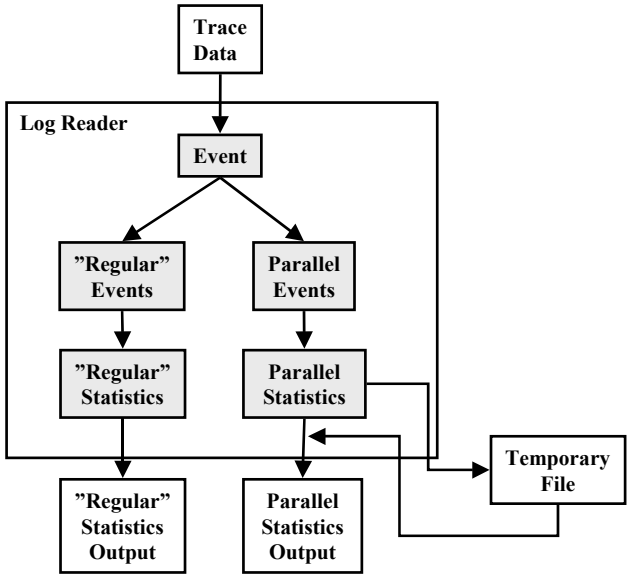

Figure 6.5: *Creation of output by the Log Reader.*

Figure 6.6 shows the memory usage during the execution of the Log Reader. It shows that the memory used is at a maximum when parsing a file containing around 130 000 events. Parsing a larger file does not increase the memory usage.

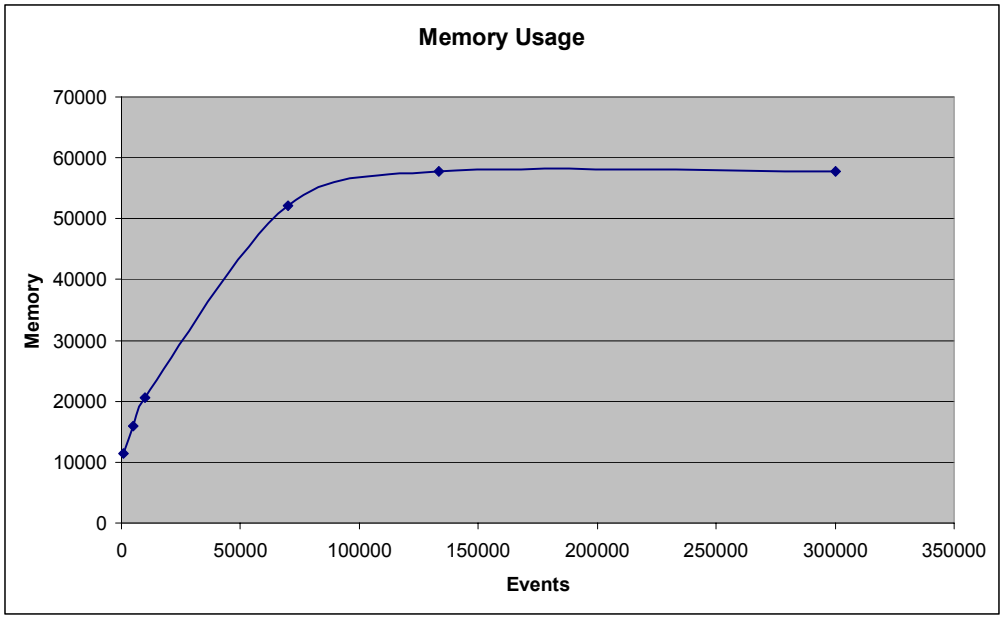

Figure 6.6: *Memory usage during execution of the Log Reader with files containing different number of events.*

6.3.3 Writing results to file

When all the events have been processed it is time to write the results to file. The server map tells us which files to create and what labels they should contain. The temporary file objects are then read and for each second the server map is iterated and for each server and method the stored statistics are written to the correct result file. This is also done with the result data left in memory.

A minor drawback with this method is that all the different result files have the correct start time but they all will have the same end time, even though no events occurred for a specific server. The following is a listing of the statistics contained in each file:

Total measurements

Combined calculations for all traced methods for a specific server:

- ***Total Calls***
  Total number of events that occurred during the specific second.

- ***Total Con. max***
  Maximum concurrency that occurred during the specific second.

- ***Total Res. max***
  Maximum response time of the events starting in the specific second.

- ***Total Res. avg***
  Average response time of the events starting in the specific second.

Method specific measurements

Calculations for each traced method.

- ***Calls tot***
  Total number of events that occurred during the specific second.

- ***Con. Max***
  Maximum concurrency that occurred in the specific second.

- ***Con. 50perc***
  The 50^{th} percentile of the different concurrency states in the specific second.

- ***Con. 90perc***
  The 90^{th} percentile of the different concurrency states in the specific second.

- ***TotTime max con.***
  The total amount of time the maximum concurrent state occurred in the specific second.

- ***Calls start tot.***
  The number of events started in the specific second.

- ***Res max.***
  Maximum response time of events started in the specific second.

- ***Res 50perc***
  The 50^{th} percentile of the different response times in the specific second.

- ***Res 90perc***
  The 90^{th} percentile of the different response times in the specific second.

- ***Res avg.***
  Average response time of events started in the specific second.

- ***MaxCon. max res time***
  Maximum concurrent state during the whole maximum response time starting in the specific second.

- ***AvgCon max res time***
  Average of the concurrent states during the whole maximum response time starting in the specific second.

- ***NEvents max res time***
  Total number of events that occurred during the whole maximum response time starting in the specific second.

6.4 Verification of results

To verify the produced statistics, the implemented functionality was executed using specifically designed test drivers. These test drivers consists of CORBA clients and servers, which performs requests and method invocations in a predefined pattern. The results from executing the test drivers can be derived in order to compare them with the CORBA Tracer output. In this way, the correctness of the implementation could be ensured.

6.5 Conclusions

The implementation of the analysis of parallel events increases the possibilities to understand the workings of a distributed system. Our solution does not measure true parallelism since this is not the wanted result. Instead we want to show time interleaved object invocations and how they affect each other regarding for example common resources, synchronization and execution sequences, and to mirror these measurements against response times to be able to pin down possible performance bottlenecks.

6.6 Future work

There are still many possible improvements. The ultimate tool would be able to show all CORBA calls on all servers in the system very finely grained not only regarding time, but also the utilization of processor and memory resources. Depending on different circumstances though, this is not reasonable. The output from such a trace when the number of events reaches hundreds of thousands would be immense. And to visualize such amounts of data in a good way would be a severely difficult task. One possibility is to either extend the CORBA Tracer Log Reader or implement scripts to search the output data for patterns corresponding to performance bottlenecks and extract a subset of the data to examine the specific time interval in more detail.

The possibility to monitor hardware utilization in relation to concurrency and response times would be possible through the use of the Java Native Interface (JNI) by implementing native code procedures to fetch the wanted data. Another possibility would be to use a separate tool (e.g. TeamQuest) to collect the wanted hardware statistics and merge it with the traced data.

7 Extraction of CORBA Call Chains

This section describes the implementation of the new CORBA Tracer feature used to produce output describing CORBA call chains in a traced system. The section starts by describing the background leading to the need for this functionality. This is followed by an implementation description of the chosen solution. Finally we discuss our conclusions and suggestions for future work.

In a CORBA based system, the distribution of objects is hidden from the calling client application. Still, for development and testing purposes, it is valuable to be able to see the dynamics of the system regarding the structure of communication between objects for different states of the system. That is, which remote object is invoked by a client and what other remote calls are spawned as a consequence of this invocation? This sequence of remote method invocations is referred to as a CORBA call chain.

The goal of this activity is to extend the implementation of the CORBA Tracer with functionality to extract CORBA call chains in a distributed system. The intent is to use the existing structure of the CORBA Tracer based on interceptors and thereby eliminating the need to instrument the code of the system under test.

The new feature is realized by adding a new trace mode to the CORBA Tracer. This mode only collects call chain data and requires that all applications that are part of a call chain must be under trace for the result to be complete. All the collected data is sent to the CORBA Tracer Log Server for persistent storage. The call chains can then be recreated at post analysis by using the CORBA Tracer Log Reader, which has been extended with a new parse mode to handle the call chain data. Figure 7.1 shows the general result format, which can easily be extended with more trace point data.

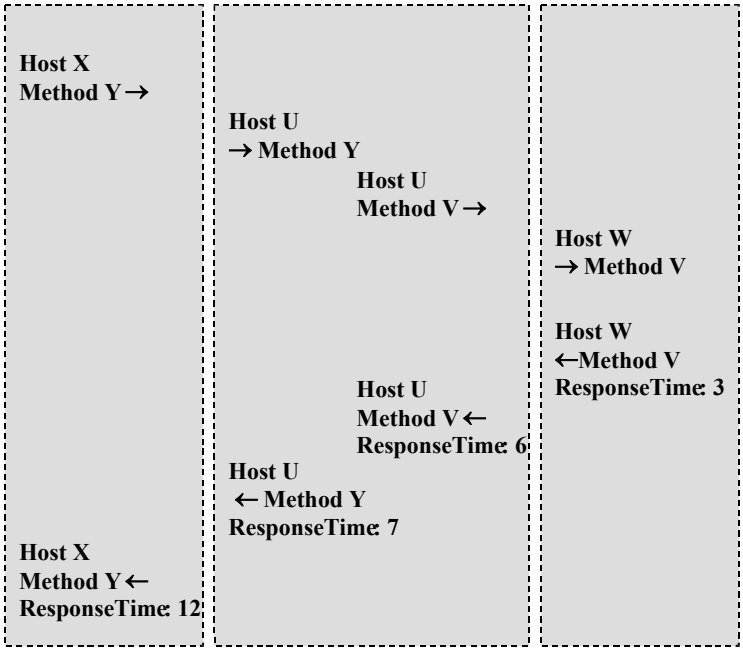

Figure 7.1: *Result output format describing a CORBA call chain.*

7.1 Problem definition

The following main issues are recognized to reach a feasible solution:

- All the interceptors on the traced nodes in the system must know to which call chain a specific call or invocation belongs. Thus, an identifier must be propagated along the call chain.

- The call chain identifier must be unique in the whole traced distributed system.

- Thread local data is needed. This is because a thread can execute several interceptors whose data belongs to the same call chain and thus they must be able to identify data local to their executing thread.

- The collected data from all interceptors must be stored on a single location in a common structure for post analysis.

- Collected data should be presented in a structured way.

7.1.1 Propagation of identifiers along call chains

To identify which call chain a specific call or invocation belongs to, an identifier must be propagated between the involved interceptors. This enables the interceptor to tag its collected data with the identifier, making it possible to reconstruct the call chain at post analysis. The easiest way would be to simply add an identifier as an argument to the remote method call, but this is not acceptable since it would mean that the method implementation must be altered. Another alternative would be to add the identifier to the payload. The chosen solution is to add the identifier as a service context object in the calling client interceptor. A service context object is used to identify the services assigned by the ORB and this information is passed along with the call and can be extracted by the receiving interceptor.

7.1.2 Creating unique call chain identifiers

The identifier propagated along the call chain must be unique in the distributed system in order to get the data stored correctly. This must be done on different levels. To create a unique identifier within a process, thread ID is used. To create a unique identifier within a host, process ID is used. To create a unique identifier in a system of distributed hosts, host ID is used. These three combined are used to create a truly unique identifier for use in the call chains. Java is limited by the inability to extract the process and thread ID assigned by the underlying operating system. Therefore the solution is implemented using the Java Native Interface (JNI). The JNI allows Java code that runs within a Java Virtual Machine (JVM) to operate with applications and libraries written in other, native languages, such as C, C++, and assembly. A shared library was written in C, containing functions to fetch process and thread ID using pthreads (**Figure 7.2**). Since the library must be compiled specifically to the underlying platform, the implementation becomes platform dependent. It is easy though, to recompile the library to another supported platform.

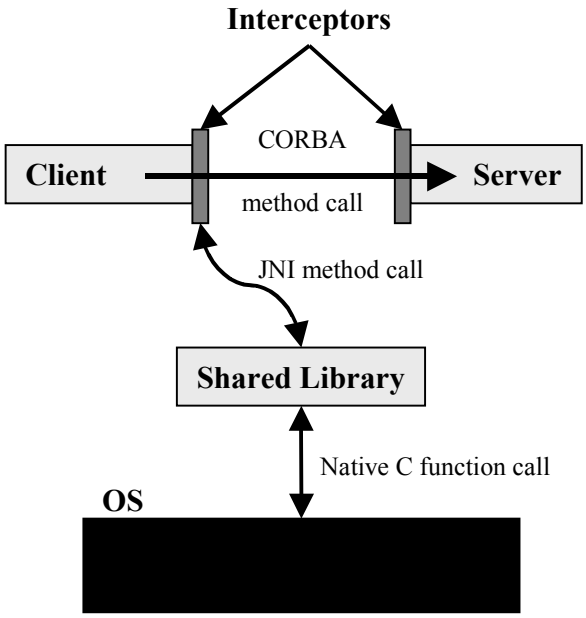

Figure 7.2: *Fetching process- and thread ID using a shared library with JNI.*

7.1.3 Thread local data

Consider the following scenario (**Figure 7.3**):

A client application A makes a remote call to a servant S. With the call it also sends a unique id that will identify the call chain started at A. If the POA handling S has a multithreaded policy the invocation request will be assigned to a new thread, thus it is possible that multiple instances of S are invoked at the same time. Before S invokes the request, its server interceptor creates a data structure containing the unique ID received from A. The data structure also contains the information gathered by the server interceptor. S then executes the method called by A. If this method in turn makes a remote call to another servant, a new trace point is handled by the current thread by invoking a client interceptor. For the call chain to be correct, the client interceptor must know the identifier sent from A. This information is contained in the data structure created by the server interceptor. Since there is a possibility that multiple concurrent threads are executing the servant and thereby the interceptors, this data structure must be local to each thread.

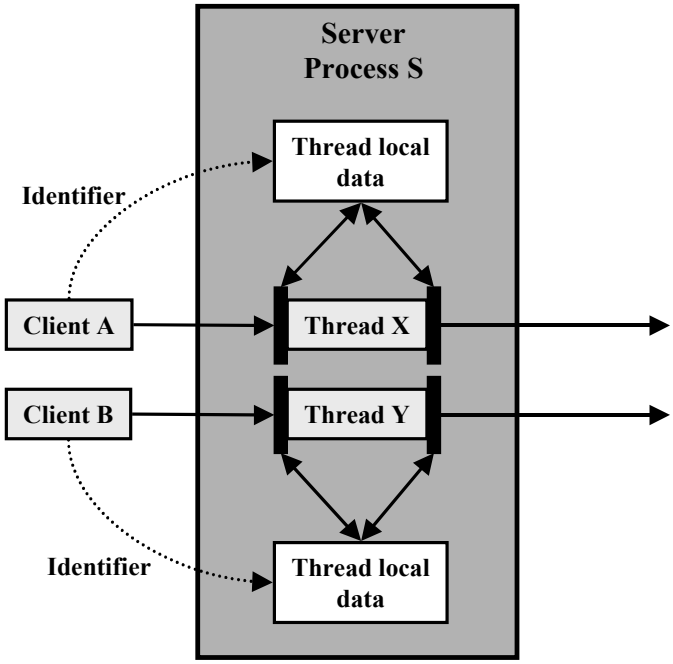

Figure 7.3: *Storing identifiers in thread local data structures.*

This is implemented using the java.lang.ThreadLocal [4] class. This class provides thread-local variables. These variables differ from their normal counterparts in that each thread that accesses one has its own, independently initialized copy of the variable.

7.1.4 Storing collected data

The CORBA Tracer Log Server provides the single location for storage. Traced data is stored and compressed using the same technique that is used in the original CORBA Tracer. The difference is that the call chain data is structured in a different way than regular trace data, resulting in modifications to the Log Server, but also to the Log Reader functionality.

7.1.5 Presenting the results

When the call chain data has been collected by a trace we want to be able to present the result in a structured way. For this we use the CORBA Tracer Log Reader, which is extended with a new standalone parse mode reconstructing the call chains. The different structure of the call chain data results in the inability to use the Log Reader's normal parsing procedure. Hence, the new parse mode is only able to handle call chain data, which is stored in separate files from the regular trace data.

7.2 Implementation description

The functionality for extracting call chains is integrated with the CORBA Tracer implementation. To produce the data you simply set the proper (`-t`) flag when starting the trace of an application (section 5.4.6). A result file will be created for each call chain in the corresponding session directory. The following sub-sections describe the major parts of the implementation.

7.2.1 Starting a call chain

A call chain is started in the interceptor of the client initializing the call. Here a thread local object is created. This object keeps track of the following values:

- Process ID, assigned by the OS.
- Thread ID, assigned by the OS.
- Depth, specifying in which level of the call chain this interceptor executes.
- Chain ID, which distinguishes between different call chains.
- Event count, which is used to recreate the chain in its correct order.
- Current call, representing the call made, including various trace data that can be specified by the user.
- A message buffer, containing the data to be sent to the Log Server.

The process and thread ID are fetched using two Java Native Interface methods, specified in a shared library written in C.

Before the client sends the request to the server, PID, ThreadID, depth, chainID and event count is put in a ServiceContext object, which is added to the marshaled payload of the call. The same data is then added to the message buffer, together with data describing the call.

7.2.2 Extending the call chain

When the server interceptor receives the request from the client, it creates its own thread local ChainContext instance. The values contained in this object are overwritten with the values contained in the ServiceContext object, extracted from the data included in the call from the client. Hence the event registered in the server interceptor is assigned the same call chain identifier as the requesting client event. The server interceptor increases the depth and event count values and adds its data to the message buffer. When this is done, the server invokes the method called by the client. This might in turn produce a request to another server, in which case a client interceptor is created. This client interceptor reads its identifier values from the thread local object overridden by the server interceptor and thus belongs to the same call chain. The course of events is described in **Figure 7.4**.

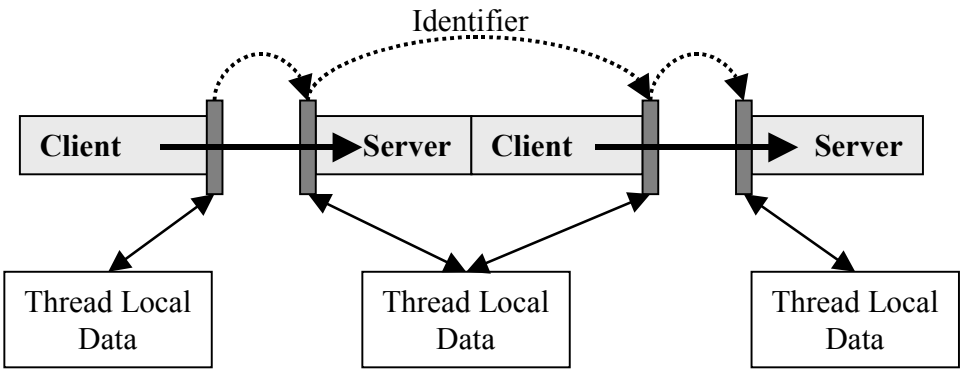

Figure 7.4: *A call chain is extended through propagating the call chain identifier between the thread local data objects of the clients and servers.*

7.2.3 Sending data to the Log Server

When a request returns in a client interceptor or an invocation is finished in a server interceptor, data regarding the event is added to the corresponding message buffer. On the client side, this happens after the event count value is extracted from the ServiceContext object returned from the server.

When the data has been added to the message buffer, the message buffer is sent to the Log Server using the Event Sender thread created by each interceptor. The information in the message buffer is compressed and contains data regarding each event traced in the interceptor.

Each message buffer starts with specifying the host ID, process ID, thread ID, chain ID and depth. These values set the identifier for the following events. The rest of the message buffer contains data describing each event with the following values:

- **Client/Server**
  Specifies if the event occurred on client or server side.

- **Out/In**
  Specifies if the event is a request or the receiving of a reply on the client side, or before or after invocation on the server side.

- **Exception**
  Specifies if the call raised an exception.

- **Method name**
  The name of the method called/invoked. The string representing the method name is stored on the Log Server in a separate structure; only the key representing the string in this structure is added to the event itself. This is to save space when several references are made to the same method.

- **Exception message**
  Is only stored if an exception occurred. The storage of the exception message is done in the same way as the method name.

- **Event count**
  The sequence number specifying where in the call chain this event occurred.

- **Response time**
  Only set on client side when a reply to a request is received, or on server side after a method invocation is finished.

The data sent is compressed using different optimization techniques. All types of data are preceded by one byte (tag) specifying if the following data represents a new message buffer (setting the identifier for the following events), an event, or a string representing a method name or exception message. Integers are written as shorts, if possible. Shorts are written as one byte, if possible. The Client/Server, Out/In, Exception and method name identifier are merged into two bytes, also containing the tag representing the beginning of a new event.

7.2.4 Presenting call chain data

The CORBA Tracer Log Reader is extended with a new parse mode, set with the –h flag (section 5.4.8), with functionality to read call chain data stored by the Log Server as described in section 5.3.1. The routine for restoring a call chain uses the tags set by the Log Server to identify, order and store the data correctly. There are three main tags:

- **Message Tag**
  Indicates the beginning of a sequence of events from a specific call chain, identified by host ID, process ID, thread ID, chain ID and depth.

- **Event Tag**
  Indicates the beginning of an event.

- **String Map Tag**
  Indicates that the following data is a method name or an exception message, to be stored in a string map, in order to be fetched when printing the call chain to file. A key for the string is also included.

The Log Reader reads one byte (tag), identifies the type of data that the tag represents, and takes the appropriate actions. If a Message Tag is read, the identifier data is stored, to be able to sort the following events (see below). If a String Map Tag is read, the string following the tag is stored in a string map structure, under its specific key.

If an Event Tag is read, the following data is read and stored in a ChainEvent object. The variables of the ChainEvent object are set according to a number of different flags and values, contained in the following bytes. An example is illustrated in **Figure 7.5**.

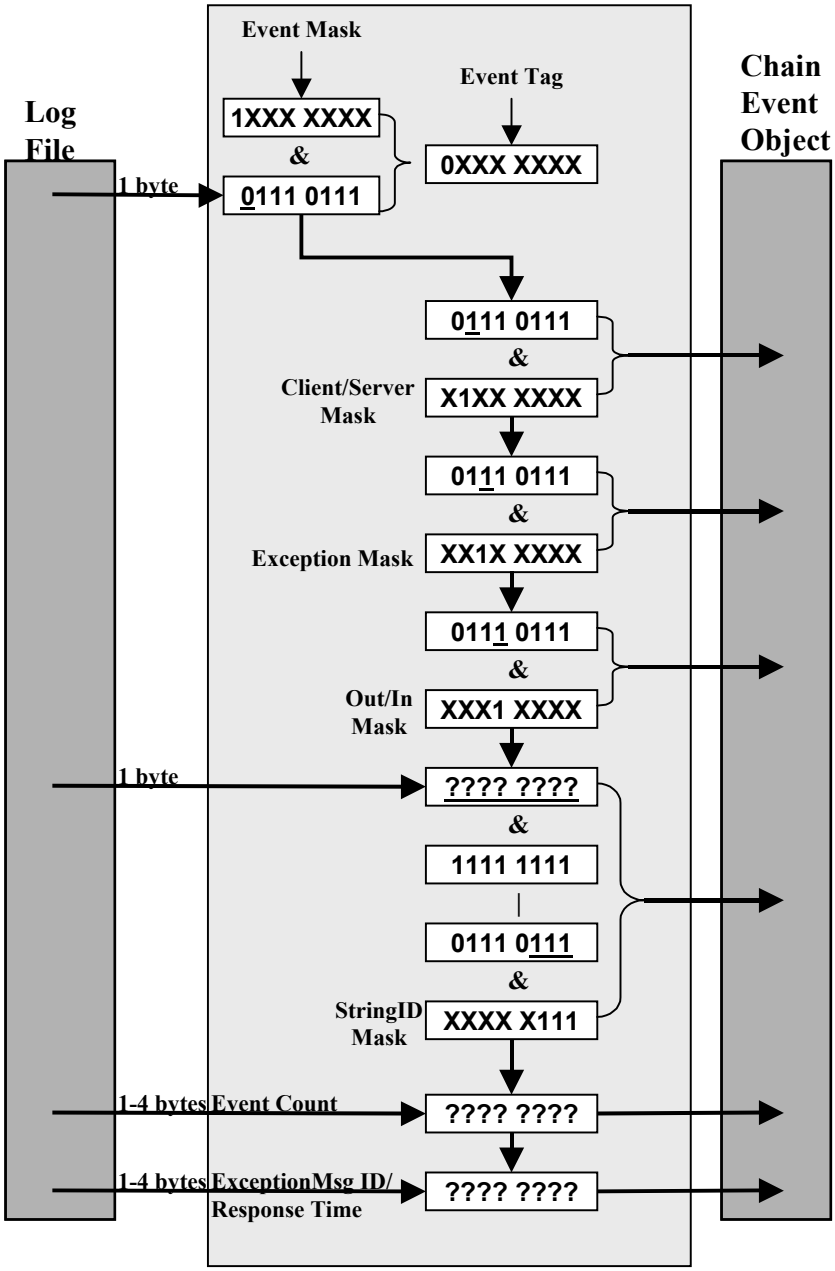

Figure 7.5: *Parsing of one event using the Log Reader.*

When the ChainEvent object's values are read and set, the object is stored until it's time to print the call chain. All Event objects are stored in a structure consisting of a hash table storing all different call chains, using host ID, process ID, thread ID and chainID as keys. For each entry in the hash table, the Events of the specific call chain are stored in a tree set; into which the Events are sorted according to their event count (sequence number) value.

When all events stored by the Log Server have been read, the contents of the hash table are printed to file, one call chain at a time. Since the Events of each call chain are stored in order, the tree sets are simply traversed and data is printed one event at a time, with the depth deciding the position in the output file (**Figure 7.6**). Method names and possible exception messages are fetched from the string map using the key stored in the ChainEvent and printed.

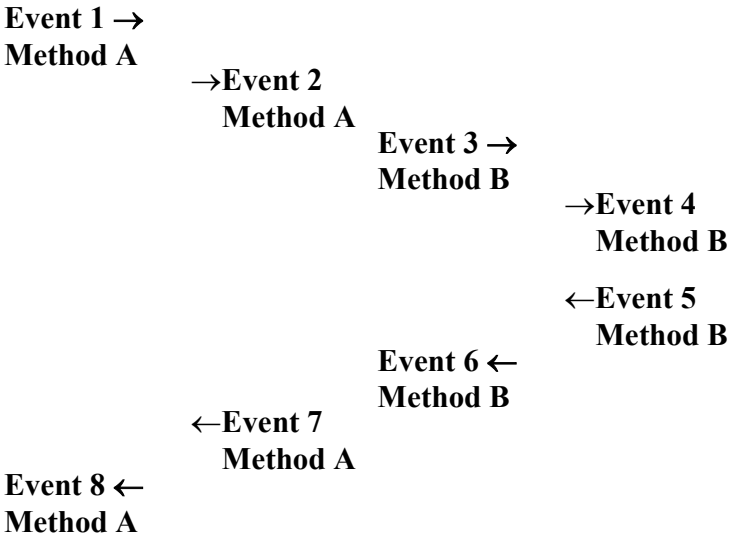

Figure 7.6: *Output format of one call chain.*

7.2.5 Handling asynchronous calls

While collecting data, each event is assigned a sequence number (event count), unique for the specific call chain. When reading from the Log Server, the Log Reader sorts the data according to this sequence number, in order to recreate the call chain correctly. There is however, a problem with this approach, asynchronous calls (**Figure 7.7**). In CORBA, these calls are called one-way calls. This means that when a client issues a request to a server method, it continues immediately without waiting for a reply from the server. If the numbering was simply to continue as usual, this would mean that the same sequence number would appear after the asynchronous call on both client and server side and thus, the reconstruction of the call chain in the Log Reader would become corrupt.

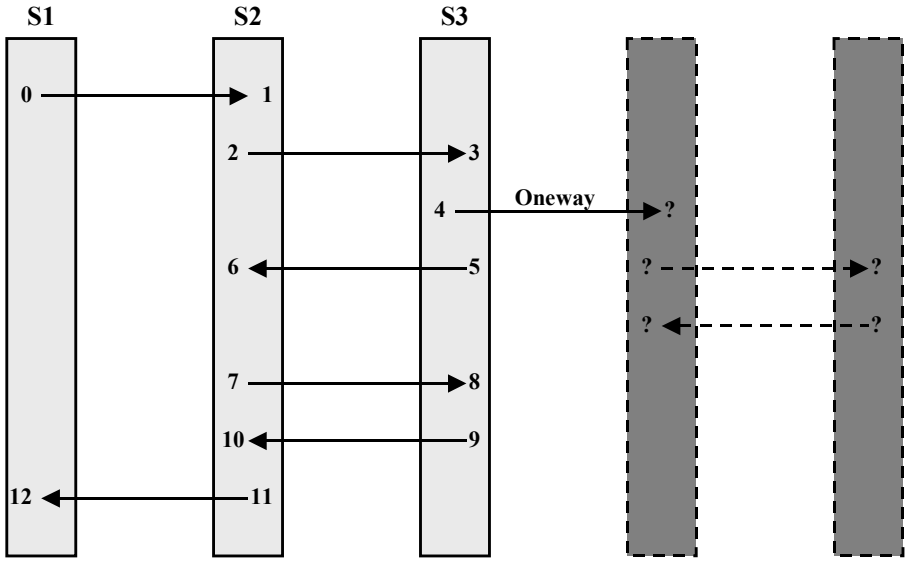

Figure 7.7: *An asynchronous (one-way) call, causes inconsistency in the sequence numbering.*

This is handled by only adding the one-way request to the call chain. The server side of the one-way call creates a new chain with its own unique ID and its own sequence numbering. Later, at reconstruction in the Log Reader, the “loose end” can be matched with the corresponding server side event to restore the whole call chain.

7.3 Verification of results

To verify the produced output, the implemented functionality was executed using specifically designed test drivers. These test drivers consists of CORBA clients and servers, which performs requests and method invocations in a predefined pattern. In this way, a predefined call chain can be executed and compared to the CORBA Tracer output. In this way, the correctness of the implementation could be ensured.

7.4 Conclusions

The added functionality is one step further on the way to understand the inner workings of a CORBA based distributed system. The produced results can be used to give a picture of the execution, different from earlier functionality implemented in the CORBA Tracer. A call chain gives an overview and a possibility to follow a specific call from the start to the end. This in turn may help finding the origin to abnormal behavior revealed by a regular or parallel trace.

7.5 Future work

Though useful, the implemented functionality is still to be seen as a prototype. Many issues are to be considered before the extraction of call chains can be made on a general system, with the ability to adapt to different kinds of execution structures. One example is discussed in section 7.2.5, but there are a multitude of different scenarios that must be considered and handled. For example, CORBA supports what is known as deferred synchronous requests. This is defined as an asynchronous (one-way) request with the difference that the server asynchronously sends the result back to the client. In this model, the client never knows if and when to expect a response from the server. Other examples of similar scenarios are connected to the CORBA Event-, Notification- and Messaging services. The curious reader is referred to the OMG CORBA specification [7].

Further improvements of the CORBA call chain extractor include the data collection and output format. In our implementation, the response times are measured, but it would be possible to add any value extracted in the interceptors to the chain event data.

8 Conclusions and Future work

In this section we discuss our thoughts on the results of the thesis work. How we met up with the goals we set up, what conclusions we have drawn and what insights we have made. We also discuss what we think should be done next in the further development of the CORBA Tracer.

8.1 Achievements

We performed a thorough investigation of the CORBA Tracer, covering both the implementation and present usage of the tool. Testers working with the tool gave their opinions on and requests for documentation and new functionality. The investigation phase resulted in documentation describing the structure of the CORBA Tracer implementation and also a detailed user guide. These documents should prove useful when introducing new users to the CORBA Tracer tool and also when applying further improvements to it.

The investigation of the CORBA Tracer also gave us the knowledge we needed to be able to implement new features to the CORBA Tracer. The functionality for analysis of parallel events produces output data that can give a user a good picture of the behavior of the system regarding parallel execution. The produced statistics contain a large number of different measurements. This gives the possibility to combine the results in many different ways to create an exhaustive analysis of the system under test. It also gives the possibility to vary the analysis to suit the current needs of the testers.

With the functionality for extraction of CORBA call chains, a user can follow the paths of the communication in the system. This way, abnormal or faulty behavior can easily be recognized.

8.2 Conclusions

Understanding the structure and implementation of a distributed system is a difficult task. There is a big need for tools that can give a comprehensive picture of the execution and at the same time describe the details of the complex inner workings. The CORBA Tracer is a big step towards achieving this kind of understanding. Our work shows that it is a tool with great potential that can be extended with additional functionality suiting specific needs.

The new functionality is, though fully functional, perhaps not optimized in all aspects. There are a few issues worth mentioning.

Parallel statistics are represented by a number of different values. All values give valuable information about the system, but only experience and extensive testing using the new functionality can decide which values are really useful in specific situations. It is difficult, if not impossible to create statistics suitable for all purposes and needs.

Call chains only include a part of the system. To extract a call chain, one must know which clients and servers to put trace on. The alternative is to put trace on all elements in the system, which may not be feasible in a live environment. Also, it is difficult to foresee all possible scenarios that may occur in a call chain.

Currently, the call chain functionality is entirely separated from the other parts of the CORBA Tracer. It is not possible to perform both kinds of traces simultaneously. The main reasons for this division is that one must trace all elements included in the call chain and also, the call chain data is stored in a totally different way from an ordinary or parallel trace.

8.3 Future work

As stated earlier, the CORBA Tracer is a fully functional, highly useable tool with great potential. Still, there is always room for improvement. One of the most important things to do is to measure and evaluate the impact of a trace on the system under test. It would be very useful in order to verify the correctness of trace data, but also to see if parts of the CORBA Tracer should be removed or redesigned. It is however, difficult to measure a tool's impact on a system. This is mainly because one cannot measure the impact on a system without having an impact on the system.

Another important thing would be to decide on a proper visualization tool to handle the output data. If a tool exists that can handle the different types of output from the CORBA Tracer in a good way, the format of the output data could easily be shaped to fit such a tool. Another approach would be to develop a new visualization tool, tailor made to handle the CORBA Tracer output data.

9 References

[1] Moe, J (2001). *Tracing of Large Distributed Systems*, Lic. Thesis, IDA, Sweden

[2] Sandberg D, Anuéll M (2005). Extended Platform, Overview & CORBA dependencies

[3] Borland Software Corporation (2004).*Borland Enterprise Server 6*, *Developers Guide* http://info.borland.com/techpubs/bes/v6/pdfs/index.html

[4] Sun Microsystems, Inc (2003). *Java 2 Platform, Standard Edition, v 1.4.2 API Specification*
http://java.sun.com/j2se/1.4.2/docs/api/index.html

[5] Cailleteau, L (1999). *Interfaces for Visualizing Multi-valued Attributes: Design and Implementation using Starfield Displays*
http://www.cs.umd.edu/hcil

[6] Andrew S. Tanenbaum, Maarten van Steen. (2002). *Distributed Systems, principles and paradigms*. Prentice Hall.

[7] The ObjectManagementGroup, Inc (2002). *TheCommonObject Request Broker: Architecture and Specification*
http://www.omg.org/docs/formal/02-06-33.pdf

[8] Johan Moe and David A. Carr. (2002).*Using Execution Trace Data to Improve Distributed Systems*. John Wiley & Sons.

[9] A. Ulricha, H. Hallalb, A. Petrenkob, S. Borodayb. *Verifying Trustworthiness Requirements in Distributed Systems with Formal Log-file Analysis.*

[10] Håkan Grahn Marcus Holgersson. *An approach for performance measurements in distributed CORBA applications.*
http://www.ide.hk-r.se/~nesse/Papers/pdcn02.pdf

10 Abbreviations

CORBA	Common Object Request Broker Architecture
OMG	Object Management Group
ORB	Object Request Broker
IDL	Interface Definition Language
POA	Portable Object Adapter
GIOP	General Inter Orb Protocol
IIOP	Internet Inter Orb Protocol
TCP	Transmission Control Protocol
JVM	Java Virtual Machine
JNI	Java Native Interface
OS	Operating System
PID	Process ID