Tracing system-level communication in distributed systems*

Zoltán Ádám Mann, Károly Kondorosi

Budapest University of Technology and Economics
Department of Control Engineering and Information Technology
H-1117 Budapest, Magyar tudósok körútja 2, Hungary
E-mail: zoltan.mann@cs.bme.hu, kondor@iit.bme.hu
Phone: +36 1 4632487, Fax: +36 1 4632204

Abstract

Traditional tracing mechanisms were usually developed for use in a single-computer environment. Moreover, they are bound to a specific programming language. Today's highly distributed and heterogeneous computing environments require new tracing methodologies. The paper addresses this problem by reviewing ways how the middleware might—and should—support tracing. In particular, CORBA meta-objects that can be applied for tracing, are studied. One of them, namely the *interceptor* concept is presented in more depth, followed by a detailed description of an interceptor-based tracing architecture for CORBA applications. Implementation details and evaluation experience is given.

1 Introduction

As computers are more and more interconnected, the target of software development also becomes a distributed, heterogeneous system, rather than a single computer [1]. As a result, new techniques, methodologies and tools are required to facilitate the development of distributed software. Also, as components of different platform, architecture and programming language are interconnected, *integration* becomes a major challenge. Since the object-oriented paradigm has provided a very good integration scheme, object-oriented distributed systems enjoy great prosperity. This is why we focus mainly on the tracing of object-oriented distributed systems and particularly on CORBA (Common Object Request Broker Architecture [2]), being the most widely used middleware system.

Besides these—rather technical—arguments, there are also some other factors to consider. The first is the wide spread of e-business, bringing along a boom for CORBA and similar middleware technologies as well. Another, not so widely recognized, but equally important fact is the emergence of *embedded* distributed systems. The idea of using standard middleware solutions in embedded environments is gaining popularity. Just consider the various intelligent electronic units that can be found on an airplane: a complex distributed system, where the individual components may communicate and co-operate using CORBA for instance. Obviously, tracing is a vital and complicated task in such an environment [3].

A third issue is the ever-growing competition on the market. Software development companies are supposed to create complex and reliable distributed systems. Parallel to these requirements, the time-to-market pressure also keeps growing. This, too, results in the need for methods and tools to make distributed software development an easier and quicker task, rendering tracing indispensable.

The aim of this paper is to investigate from a practical point of view the tracing possibilities for heterogeneous, distributed, object-oriented systems.

*This paper was published in Software: Practice and Experience, volume 34, issue 8, pages 727 - 755, Wiley, 2004.

By tracing, we mean *a step-by-step execution of a software system, conducted in order to gain extra information—i.e. information that is not part of the output in a normal execution—or insight on how the system works.*

Some notes on this definition:

- According to this definition, tracing is not the same as event recording.
- In the definition, it is not specified what a *step* in 'step-by-step execution' is. This is by intent so in order to make the definition scalable: steps may be very low-level (*e.g.* machine code instructions) but may also be high-level (*e.g.* communication events or messages in a distributed application).
- It is not specified either by whom tracing is carried out. This might be a human or a computer program. Note also that this entity does not participate usually in the normal execution of the system, at least not with this functionality.
- Although the definition allows arbitrary steps, in this paper we will regard communication events as the boundaries of steps. The reasons will be explained later in detail.
- Step-by-step execution does not necessarily mean that the system has to be stopped after each step, although it should be possible to stop it.

Tracing is sometimes used as a synonym for debugging. However, we consider it as a more general concept. Possible purposes of tracing include:

1. **Checking of correct behavior**. This is the most obvious usage: the programmer simply wants to make sure that the software does what it is supposed to do. Therefore, he or she runs the program step by step and looks at its output and internal state.
2. **Locating bugs**. If the software does not do what it is supposed to do, then again, tracing can help in identifying the nature and the location of the error.
3. **Monitoring crucial applications**. Even if the system seems to do correctly what it is supposed to, this will just not be enough in mission-critical applications. For instance, a power plant regulating software will have to be monitored constantly.
4. **Better understanding how the system works**. Running the software step by step can also serve demonstration purposes and thus be used in *e.g.* university lectures or other courses. Also, if a programmer, developing a new module for a complex software, would like to obtain a coarse picture of how the system works (*e.g.* how existing modules co-operate and communicate), they can use tracing.
5. **Extracting documentation**. There are some tools that can extract static documentation from the source code of the software. For instance, such tools can be used to extract an inheritance graph from the source code of object-oriented software. On the other hand, tracing could be used to extract *dynamic* documentation, *e.g.* a communication diagram, while the system is running.
6. **Performance analysis** (also known as profiling). In this case, it can usually be assumed that the software behaves correctly, but slowly. Therefore, the programmer—or a performance specialist—runs the system and at the same time measures the elapsed time in certain functions of the program, or more generally, the elapsed time between certain events. Typically, the goal is not a precise measurement but rather to identify bottlenecks.
7. **Detecting vulnerabilities**. An interesting application of execution monitoring was suggested in [4]. The idea is to use execution monitoring to automatically detect vulnerabilities of privileged programs by recognizing program states that exhibit potential danger for the integrity or security of the system.

8. **Aspect-oriented programming (AOP)**. Events generated by a tracing tool may be used to trigger the invocation of *aspects* (*i.e.* code implementing cross-cutting functionality that does not fit into any particular module of the system). Thus, tracing can be regarded as a mean for *dynamic weaving* of aspects (for more information, see [5]).

As shown later, all these aims can—and should—be provided for with essentially the same tracing architecture. Also, the usability of a tracing tool should be measured on how well it fulfills the requirements presented by the above aims. However, it has to be noted that the above scenarios need slightly different usage of the underlying tracing architecture. Therefore, a tracing tool should support several modes of operation. For example, the first four use cases may require some on-line user interaction, while the last four do not. It follows that tracing tools should provide at least an interactive and a non-interactive mode. Of course other distinctions are also possible. A variety of modes will be presented in section 4.4.

The paper is organized as follows. First we review the basics of tracing in distributed systems. Then, we survey several methods for the implementation of tracing, including meta-objects provided by the middleware (*i.e.* CORBA, in our case). The main contribution of the paper is the description of a tracing architecture based on one of the meta-objects, interceptors. We report on our experience with the implemented tracing tool as well as empirical results on the overhead caused by tracing. The paper is finished with a section on related work and our conclusions.

2 Distributed tracing model

In this section, we introduce the basic issues of tracing in a distributed, heterogeneous, object-oriented system: main difficulties, solution framework, and the role of the middleware. But first, the basic terminology has to be defined.

2.1 Terminology

With *distributed system*, we mean a software system that is distributed over a *network* of autonomous computing units (*nodes* or *computers*). The computers are interconnected through *communication channels*. A piece of software is running on each computer, and these pieces interact, co-operate and communicate in order to solve a given task or set of tasks. The distributed system is made up of *components*. A component is the smallest part of the system that can run on a separate computer. In the simplest case, each component runs on a different computer. However, some components can be colocated on the same node. Nevertheless, they could run on different computers, so their communication is logically location-independent.

Each component works on a sub-task of the system. Moreover, each component may consist of several *processes*, which work on sub-tasks of the task of the component. Note that the terms 'component' and 'process' are used in many other contexts as well, which are not relevant here. In particular, the sense in which we use the term 'component' is different from that of component-based software engineering. Similarly, the sense in which we use the term 'process' is different from that of operating system processes. In our terminology, a 'process' is a purely logical unit. It might be implemented for instance as an object (in the sense of OOP), an operating system thread or even an operating system process.

Note also that components may run in parallel and their operation is independent except for explicit communication and synchronization. Whether or not individual processes inside a component are also concurrent, does not matter from our point of view.

Enabling the co-operation of distributed components involves some common tasks, which are largely orthogonal to the business logic of the actual application. Such tasks include: client-server connectivity, platform and network transparency, remote method invocations, name-to-address translation, dynamic server invocation, locating services, load balancing, security, fault tolerance, *etc.* In modern distributed systems, there is usually a special layer of software—called *middleware*—that delivers this functionality [6, 7].

2.2 Difficulties of tracing in a distributed environment

Just as the design and implementation of distributed applications is more complex than that of applications for one computer, tracing is also much more challenging in a distributed system.

First of all, the place where trace information is created and the place where it is needed are generally not the same. Therefore, the following processes must be arranged separately, but not independently:

- Extraction of trace information
- Transport of trace information
- Processing and combining trace information
- Displaying trace information

This may also cause certain anomalies. For example, since the communication delays in the distributed system may vary over space and time, it is possible that the information that component B obtained a message from component A, becomes available earlier to an observer than the information that component A sent a message to component B. This phenomenon is called tachyon.

It is also possible that some components of the distributed system stop working correctly or stop working at all. Transitional network failures can also cause some trace information, that is just on its way, to be lost.

Time and time-related (such as performance) measurements are made difficult by the usual lack of a global clock.

If the system is not only distributed but also heterogeneous, this poses an even bigger challenge. Namely, traditional tracing facilities are usually provided by programming environments and are bound to a particular programming language. This is quite natural because the programming language is exactly the level of abstraction that serves for the interaction of the programmer with the computer. So tracing, too, is best performed at that level. However, if the components of the system are implemented in different programming languages, then tracing solutions that are bound to a specific programming language are not appropriate.

2.3 Solution framework

In our work, we used the following framework for tracing distributed systems (see figure 1):

- In every component of the distributed system, a new process is installed which gathers trace information. More specifically, it intercepts incoming and outgoing calls.
- There is an additional component which is responsible for collecting trace information from the other components, as well as for displaying it appropriately (denoted as `tracer`).
- Collecting trace information may either be implemented in a *push* or in a *pull* model. In any case, communication may either be arranged using the common channels of the distributed system or through dedicated channels. If it is possible to use dedicated channels for tracing, this is the best solution because this way the communication load of the system is not increased, thus enabling more realistic performance measurements. However, such dedicated channels are often not available. In such cases, the influence of tracing should be minimized; we will investigate this issue later in more detail.

From a software engineering point of view, it is vital to have a central `tracer` component. It encapsulates all details concerning the tracing mode, the level of verbosity, the output format, output device specific information *etc.* Otherwise (that is, if trace information were also output in a distributed way) the output of trace information could easily become inconsistent.

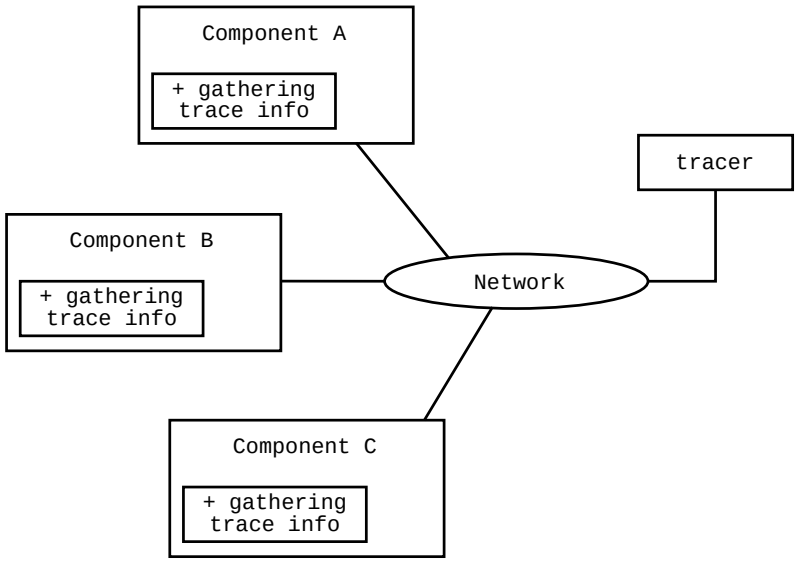

Figure 1: The distributed tracing model

From the other difficulties mentioned in section 2.2, we are going to address the problem of causality and that of global time-stamping. The problem of network and computer failures is beyond the scope of this paper.

Another vital aspect is the scope of tracing: it is logical to lay emphasis on system-level communication. The reason is that standard tracing techniques can well be used to trace execution inside a component. For that, it is still the particular programming language that was used to implement the component in question, that can provide the best tracing facility. The real challenge in tracing distributed applications is to trace the communication between the components. For this reason, we are only interested in tracing communication events. Nevertheless, it is an open problem how to combine tracing information on the operation inside the components (produced by conventional tools) with tracing information on system-level communication. Moreover, non-determinism of distributed systems makes complete traceability hard if not impossible in general. Currently, we only know about some ad-hoc solutions [8, 9], so that developing a systematic solution to this problem remains an important task for future research.

2.4 Instrumentation

One of the key issues in building a tracing system such as the one depicted in figure 1 is, how to gather trace information. The process of adding this new feature to existing code is called instrumentation.

The most widely used solution is *manual instrumentation*. This means that the programmer has to add extra pieces of code in order to notify the `tracer` about what is going on. Typically, the tracer provides specific functions for this. So the programmer will insert calls to these functions at every point in the software that is potentially critical. Usually this means that the `tracer` needs to be notified just before and just after every function call of the original code, and/or at the beginning and end of each function. (This is important so that—if necessary—the complete trace information can be displayed. Of course it is possible that only a subset of it is really interesting; the amount and verbosity of the trace output should be adjustable in the `tracer`.) The notification should include information of the call, such as the initiator and the target, parameters, return value *etc.*, since this information is an important part of the tracer output.

Assume, for instance, that the original code contains the following call (using C++ syntax):

```
result=server->do("Joe",42);
```

After instrumentation, the code becomes something like this:

```
tracer->before(this,server,"do","Joe",42);
result=server->do("Joe",42);
tracer->after(this,server,"do","Joe",42,result);
```

(Note that other events can also be traced; however, we only focus on tracing remote method invocations in this paper.)

As can be seen, manual instrumentation is tedious and error-prone. Moreover, it is necessary for manual instrumentation to possess the source code. Therefore, our goal is *automatic instrumentation*, meaning that the programmer's extra work should be minimized (ideally eliminated). In the next sections it is explored how this can be supported by the middleware.

2.5 The role of the middleware

As already mentioned, traditional tracing solutions are specific to particular programming languages, because the programming language is the very level of abstraction on which the programmer handles the computer, and so it is the programming environment itself that can provide the best tracing facilities.

When moving on to distributed systems, an additional, higher level of abstraction appears, namely that of *system-level communication*, supported by the middleware. In some cases, this also results in the introduction of a higher-level language, *e.g.* Interface Definition Language (IDL) in the case of CORBA. The same way that specific programming environments can provide the best traditional tracing solutions, it is the middleware itself that may—and should—provide the best solutions for system-level tracing.

If the whole distributed system is developed in a single programming environment, then this environment will be capable of providing tracing solutions for multiple abstraction levels, not only for system-level communication. Examples for such systems include Grade [10] and Guide [11]. However, this kind of distributed software development is not typical, largely because middleware systems have to be able to integrate legacy applications as well, which were implemented on other platforms.

The difficulties of distributed tracing (mentioned earlier) are typical tasks of the middleware. So it is again the middleware itself that can provide the best support for message delivery, time-stamping, event handling *etc.*

In the next section, this idea is illustrated on the example of CORBA: we present mechanisms that can be used to provide automatic instrumentation for CORBA applications and thus achieving tracing of system-level communication. But first, a short introduction to CORBA is given.

2.6 Introduction to CORBA

Since there is a number of good books on CORBA (see *e.g.* [12]), we only cover here those aspects that are vital for the understanding of the next sections.

CORBA enables a *client* to invoke methods of *servants* that may reside on remote computers with possibly different hardware platforms and operating systems and may even be implemented in different programming languages. For this purpose, the interface of the servant is defined in OMG IDL (Interface Definition Language). Based on the IDL description, an *IDL compiler* generates code that will handle communication between client and server transparently. The piece of generated code on the client side is called *proxy* or *stub*, whereas on the server side it is called *skeleton*.

The proxy offers the same interface as the servant, so that the client can call the methods of the proxy without knowing where and how the real servant is implemented. (See figure 2.) The proxy uses the services of the ORB (Object Request Broker) to locate the servant, to marshal the parameters of the invocation, and to send the resulting message to the servant using low-level communication primitives. On the server side, the skeleton decodes the received message,

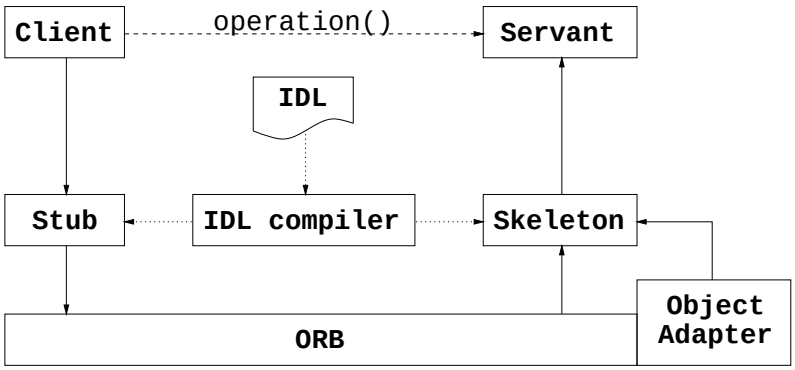

Figure 2: Remote method invocation in CORBA. The dashed line indicates the logical invocation, whereas the solid line shows the real flow of control. The dotted arrows show the process of IDL compilation

unmarshals the parameters, and calls the appropriate method of the servant with the appropriate parameters.

Server-side processing is further mediated by the *object adapter*, which is responsible for activating and deactivating servants. As object adapter, typically the Portable Object Adapter (POA) is used, which can be flexibly configured to use different policies for the mapping of object interfaces to servants. There can also be multiple POAs on the server, building a hierarchical namespace.

The protocol for the communication between different ORB instances, called General Inter-ORB Protocol (GIOP), was also standardized by the OMG. Its most widely used implementation is the Internet Inter-ORB Protocol (IIOP), which works on top of TCP/IP.

CORBA object implementations are identified using so-called Interoperable Object References (IORs). An IOR contains all information with which the ORB can uniquely identify it. In the case of IIOP, the IOR consists of the name or IP address of the machine the object resides on, the TCP port, path to a POA, the ID of the object relative to the POA, and the IIOP version, plus optionally other information pertinent to some specific services.

The current version of the CORBA specification is 3.0.2. However, at the time of writing, most available ORB implementations comply with CORBA version 2.4–2.5. ORB implementations include commercial products, such as VisiBroker from Inprise or Orbix from IONA, and also free, or open-source products, such as MICO, TAO, or JacORB. The ORB used in this work was TAO (The ACE ORB, [13]).

3 Instrumenting CORBA applications

In this section, we survey the pros and cons of several possibilities for the automatic instrumentation of CORBA applications. First, we review solutions that do not need any special support from the middleware. In the second half of the section, CORBA meta-objects will be described, which can effectively facilitate tracing. The section is concluded with a comparison of the presented methods.

3.1 Methods without middleware support

3.1.1 Preprocessing

The most obvious idea for automatic instrumentation is to try to automate the insertion of the lines of code to inform the `tracer` before and after every method invocation. This can be achieved using relatively well-known techniques: lexical and syntactic analysis to find the locations of interest in the source code as well as the pieces of information that have to be sent to the `tracer`.

On the other hand, this method also has several disadvantages. First of all, the automation method is programming language dependent. That is, a different automation tool has to be realized for every used programming language. Moreover, in order for this method to work, it is absolutely necessary to possess and to change the source code.

3.1.2 Tracing at the GIOP level

Another family of tracing approaches deals with tracing the communication between the components, and focuses on the network. Intercepting the messages on the network and decoding them based on knowledge of the inter-ORB protocol (GIOP-IIOP [14]) makes it possible to deduce the method invocations. The major advantage of this approach is that it is completely independent of the used programming languages, and the source code as well. However, it involves very low-level and tedious work, and results in a solution that is not really portable.

3.1.3 Instrumenting communication libraries

In the case of open-source middleware (such as, for instance, the TAO ORB) it is also possible to instrument the communication subsystem of the middleware instead of the application code. This way, arbitrary tracing solutions can be implemented without needing to change or even recompile the application.

On the other hand, such a solution depends heavily on the ORB. For example, if a large system makes use of multiple ORB implementations (for instance, to accommodate multiple programming languages), then the instrumentation of all ORBs has to be carried out, and this may be very different for the individual ORBs. Moreover, the efforts and costs of changing to another ORB greatly increase. Also, this approach excludes the use of non-open-source ORBs. In brief, while this method might be a quick solution in the short run, it hurts the standards-based philosophy of CORBA, and can thus become very costly in the long run.

3.2 CORBA meta-objects

Now let us examine ways how the CORBA middleware itself can facilitate tracing.

The core CORBA specification did not really support tracing in any way. It seems that although CORBA has from the beginning on provided quite a number of *reflection*-like features (such as the interface repository), tracing was not considered. Fortunately, there are some recent additions to CORBA that make tracing and automatic instrumentation possible.

The idea behind these extensions is to regard one of the objects along the standard CORBA invocation path (client – stub – network – skeleton – servant) as a *meta-object*, that can either be changed to a proprietary implementation or extended through predefined hook methods [15, 16]. In the following paragraphs, these possibilities are described in more detail.

3.2.1 Smart proxy

Smart proxies, as the name suggests, override the default proxy (stub) object generated by the IDL compiler, in order to provide additional functionality. In order to construct smart proxies, a smart proxy factory has to be created and registered as such. Of course the functionality of the default stub, namely that it sends the requests of the client to the servant (and marshals the parameters and demarshals the return value) is still needed, so the smart proxy will probably also delegate this task to the default proxy. For instance in TAO, this is achieved in the following way: the IDL compiler can be instructed to generate (besides the default proxy) a smart proxy base class that does the delegation, and a default proxy factory. After that, the smart proxy class can be extended with the necessary additional functionality. (See figure 3.)

It follows from the role of the proxy that it only participates in the communication between a particular client and a particular servant. Therefore, a smart proxy might be used to trace communication of that particular pair of objects.

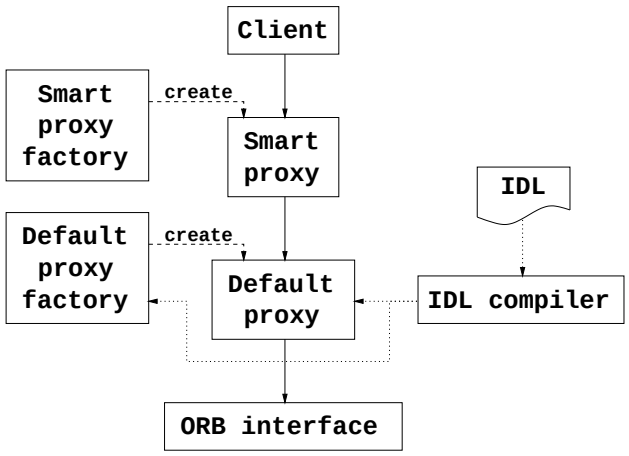

Figure 3: Usage of smart proxies

The smart proxy has relatively little freedom in modifying any details of the call. The only thing it can do is provide additional data or throw an exception. This way, it might be used for instance to provide a security framework: it can add authentication information on the client side (without a need to modify the client itself) that can be checked on the server side. Alternatively, it can check authentication of the server and throw an exception in the case of a problem, thus prohibiting insecure communication.

3.2.2 Pluggable protocol

Pluggable protocols [15] are meta-objects that make it possible to separate the component architecture and the communication protocol of the ORB. In order to achieve this, the component architecture and the high-level CORBA services access the communication protocol using the *facade* design pattern [17], which makes the communication protocol relatively easy to replace.

Unfortunately though, creating a new protocol is quite tedious and error-prone. This is exactly the kind of low-level programming task that middleware solutions usually save the programmers from. But in this case of course the middleware cannot help. Another problem is that at the level of the communication protocol it is not easy to recognize parts of the original high-level message.

3.2.3 Servant manager

Servant managers, as their name suggests, reside on the server side [16]. They receive control just before a call reaches the servant, at a moment when parameters are already demarshalled. There are two kinds of servant managers: servant activators and servant locators. The servant activator is invoked the first time a servant is accessed. The servant locator is invoked every time a request is made to the servant. Thus, the servant locator may choose dynamically a servant for processing the request. It is also possible that it actually creates the servant on the fly (see Figure 4).

This mechanism is especially useful for load balancing or fault tolerance schemes, since it enables the simultaneous existence of many parallel servants from which the servant manager may choose transparently. It can also be used for tracing, most notably if only the access to a specific server is to be traced.

3.2.4 Interceptor

Interceptors are objects implementing the `Interceptor` interface, which is—in contrast to smart proxies and pluggable protocols—defined in the CORBA specification [2]. There are three kinds of interceptors: `IORInterceptor`s, `ServerRequestInterceptor`s and `ClientRequestInterceptor`s.

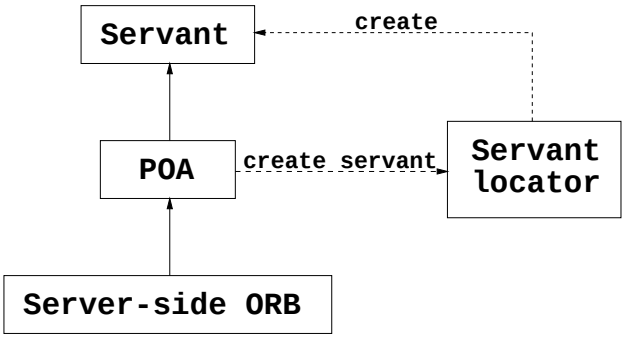

Figure 4: Usage of servant locators

Table 1: Comparison of the presented possibilities for instrumentation

Method	Language independence	ORB-independence	Source code independence	General applicability	Cost-effectiveness
1. Preprocessing	–	+	–	+	–
2. GIOP-tracing	+	+	+	+	–
3. Instrumenting comm. libraries	+	–	+	+	–
4. Smart proxy	+	–	+	–	+
5. Pluggable protocol	+	–	+	+	–
6. Servant manager	+	+	+	–	+
7. Interceptor	+	+	+	+	+

`IORInterceptor`s are invoked during the creation of IORs. They can be used to insert service-specific information into the IOR.

In the following, we will only deal with `ServerRequestInterceptor`s and `ClientRequestInterceptor`s, because they can be used for the purposes of tracing method invocations. Both interfaces define callback methods that are invoked by the ORB at specific points of a CORBA call. More specifically, the `ServerRequestInterceptor` defines the interception points `receive_request` and `send_reply`, whereas the `ClientRequestInterceptor` defines the interception points `send_request` and `receive_reply`. Using these four interception points it is possible to trace the four most important events connected to a remote method invocation: when the client sends the request, when the request reaches the server, when the server sends its reply, and when the reply reaches the client. More information of interceptors is given in the next Section.

3.3 Comparison

Table 1 presents a summary of the pros and cons of the presented methods for the instrumentation of CORBA applications. Figure 5 shows the places of the CORBA invocation that the individual instrumentation mechanisms target. The numbers in Figure 5 correspond to the numbering in Table 1.

We used interceptors for our tracing system because interceptors are much more general than smart proxies or servant managers (interceptors are not specific to a particular client or servant) and provide at the same time a much higher level of abstraction than pluggable protocols. Moreover, interceptors do not depend on the particular programming language or ORB implementation as do preprocessing or instrumentation of ORB communication libraries.

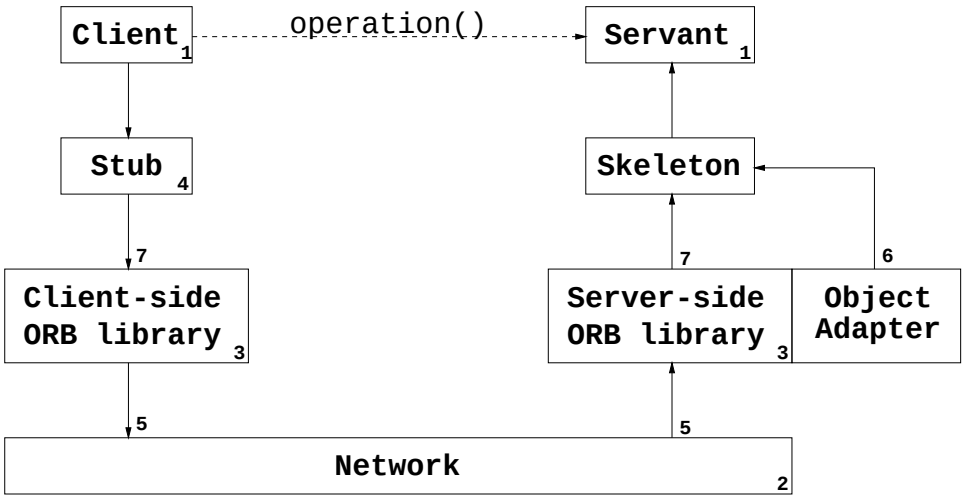

Figure 5: Instrumentation possibilities during a CORBA invocation

4 Tracing using interceptors

This section presents a tracing architecture based on interceptors. First, the operation of interceptors is described in more detail, as well as the history of the interceptor specification. Then we present in detail how interceptors can be used for tracing. In particular, we analyze the practical applicability of several possibilities for modes of operation, timestamping, and indentification mechanisms. An empirical evaluation of this tracing architecture and the particular design choices is the subject of the next section.

4.1 Interceptors in more detail

Interceptors must be registered with the ORB, that is, the interceptors are registered with a local ORB object, in a specific name space. After that, the `ServerRequestInterceptor` will intercept all incoming requests and outgoing replies, whereas the `ClientRequestInterceptor` will intercept all outgoing requests and incoming replies. This way, all inter-component communication can be traced. (See figure 6 for the flow of control.)

The interceptor interfaces define 10 hook methods, not just those four that can be seen in figure 6. The remaining six are used for very special purposes, including Time-Independent Invocations (TII, [2]) and oneway calls. The full list is given below:

- Before sending the request: `send_poll` in the case of TII, and `send_request` otherwise.
- When the request reaches the server side, but before calling the servant manager: `receive_request_service_conte` At this point, the service contexts are already available, but parameters are not demarshalled yet.
- Before the request reaches the servant: `receive_request`. At this point, all information about the invocation is available.
- Before sending the reply: `send_exception` if the servant throws an exception, `send_reply` for normal execution, and `send_other` in all other cases, *e.g.* if the call is redirected, or in the case of a oneway call.
- Before the reply reaches the client: `receive_exception`, `receive_reply`, and `receive_other`, accordingly.

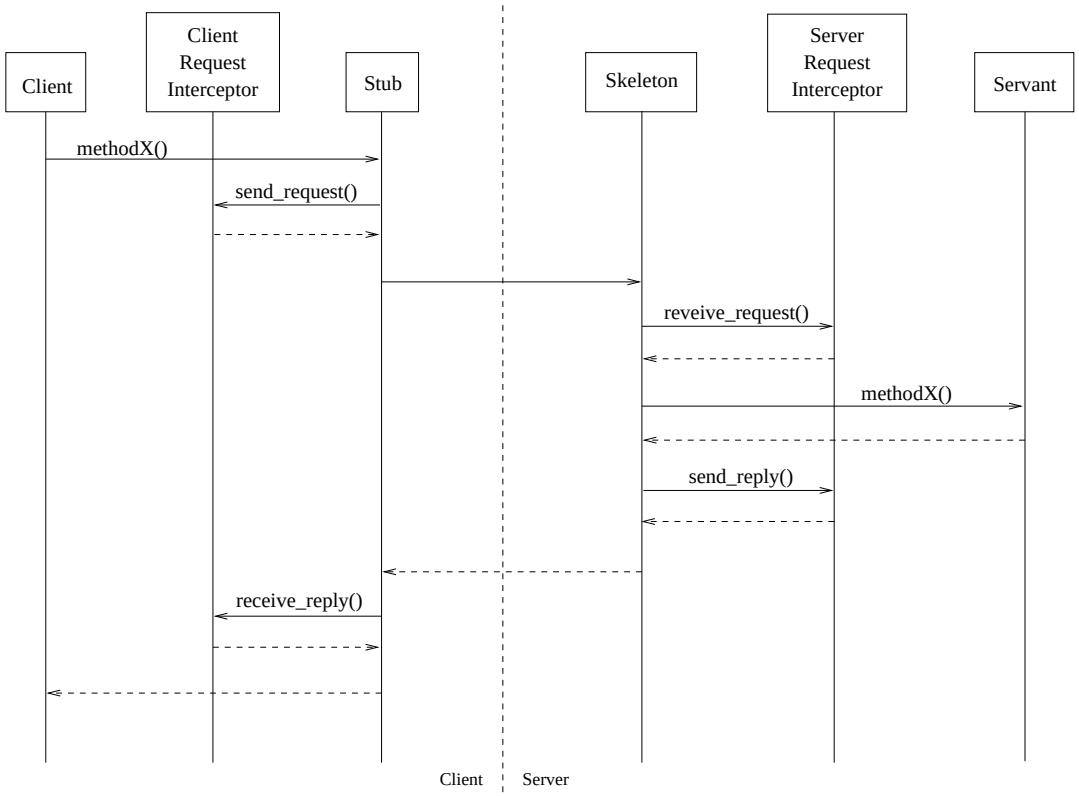

Figure 6: Interception points during a CORBA call

The interceptors obtain information concerning the current call in a `RequestInfo` object. This includes a reference to the target of the call, the name of the invoked operation, the list of parameters, the return value (if already available), a—possibly empty—list of thrown exceptions, and a list of so-called *service contexts*. Service contexts can be used for out-of-band communication between interceptors in different components: an interceptor may add extra information to a service context, which can in turn be accessed later by another interceptor.

Interceptors may not alter the information contained in the `RequestInfo` object. However, an interceptor might redirect the call by throwing a `ForwardRequest` exception. This way, load balancing or fault tolerance schemes can be integrated into an existing application, without modifying its actual source code. Also, interceptors can be used for security purposes, *e.g.* for transparent authentication [15]. These are probably the goals that interceptors were actually developed for. But, as we can see, they can also be used for tracing.

It is also possible to register more than one `ServerRequestInterceptor` and/or more than one `ClientRequestInterceptor` in the same component. In this case, CORBA provides no means with which the interceptors can communicate with each other, in fact, they do not even know about each other. Restrictions on the order in which the interceptors are executed are not specified either. However, it is specified that both kinds of interceptors are executed in a *stack-flow model*. This means for example in the case of `ClientRequestInterceptor`s that their `send_request` methods are invoked one after the other, and after having invoked the `send_request` method of an interceptor, the ORB pushes it onto a stack. When the reply arrives, the interceptors are popped off the stack, and their `receive_reply` method is invoked, but this time in the opposite order. This model also guarantees that the same interceptors intercept the request and the reply. (If an exception is thrown, then this may be only a subset of all registered interceptors.) It must be mentioned however, that the stack-flow model is only a logical model used in the specification. ORB implementations are free to use other data structures, not necessarily a stack.

4.2 History of interceptors

Interceptors were first defined in CORBA 2.2 in February 1998 [18]. However, the first definition was quite under-specified. This resulted in a number of proprietary solutions from different vendors. The problem was soon recognized by the OMG, which issued a Request For Proposals in September 1998 [19]. After some iterations of proposals and discussions, the leading vendors of the field came to an agreement, and handed in their Joint Submission in December 1999 [20]. The architecture described in this submission was considered the *de facto* standard afterwards. It has become a part of the CORBA specification with only slight modifications in CORBA 2.5 in 2001 [21]. To differentiate the new, fully standardized interceptor specification from the old one, it was given the name 'Portable Interceptors'. For simplicity however, we will refer to the portable interceptors simply as interceptors.

Since the Joint Submission is available, vendors are working on their interceptor implementation to make it conform to the specification. The ORB used in this work, TAO [13], was one of the first to introduce support for interceptors; however, also in a proprietary way:

- There were less interception points than defined by the current specification; moreover, the names and parameters of the corresponding hook methods were different.
- There was no support for multiple interceptors per component.
- The arguments and return value of the original invocation were not accessible.
- 'Cookies' were used to pass information among interceptors in different components.

There have been a number of changes of TAO since then, bringing also its interceptor support closer to the specification. At the time of writing, the latest version is 1.3.3, released in June 2003. The interceptor support of this version almost fully complies with the standard, with the following deficiencies [22]:

- The interception point `send_poll` is not implemented since it is used for intercepting TII invocations, and TII is not implemented in TAO at all.
- The standard service context slot ID allocation mechanism has not been implemented yet.
- Interceptors are not invoked for calls using the Dynamic Invocation Interface or the Dynamic Skeleton Interface (DII/DSI). Moreover, although they are invoked for Asynchronous Messaging Invocations, the arguments and the return value are not available in this case.

4.3 Tracing architecture

From the above it should be clear that although interceptors were not designed specifically for tracing purposes, they can indeed be used to trace CORBA applications. For this, only a subset of their functionality is needed, namely that they are informed of every CORBA call.

In order to trace every call, a `ServerRequestInterceptor` and a `ClientRequestInterceptor` must be registered in every component. Each interceptor sends the trace information to the central `tracer` object through the usual communication channels of the system, *i.e.* using CORBA calls. In other words, the `tracer` has to be implemented as a CORBA servant, its notification methods defined in IDL.

This way, every normal remote method invocation generates four events: (i) when the client issues a request; (ii) when the request reaches the server; (iii) when the server sends its reply; and (iv) when the reply arrives back at the client. If needed, all this information can be displayed. On the other hand, the user interface of the `tracer` should be configurable so that, say, only one event is shown for each remote method invocation. Since this is controlled centrally, the consistency of the output is guaranteed.

When interceptors themselves issue calls, care must be taken to avoid infinite loops. Namely, the interceptor will also intercept the calls it issued itself, and if it makes a call again, this results

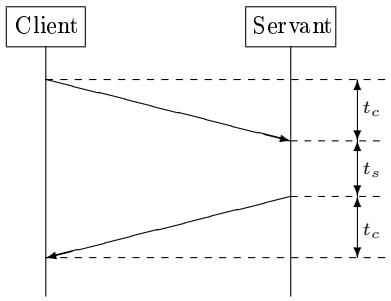

Figure 7: Duration of a remote method invocation

in an infinite recursion. To avoid this, the interceptors must check whether the target of the intercepted call is the `tracer`, and if it is, they should do nothing.

Note that local calls, *i.e.* calls within a component are not traced. This is due to the fact that interceptors are called by the ORB in the case of a CORBA call. The ORB does not even know about local calls, so consequently it cannot invoke the interceptors. On the other hand this is not a problem since the aim was to trace system-level communication between the components anyway.

4.4 Modes of operation

It has already been mentioned that the `tracer` should provide at least two modes of operation: interactive and non-interactive mode. In interactive mode, the tracer waits for user input inside the notification methods, thus blocking the system; in non-interactive mode the trace information is just displayed and the notification methods return immediately. Either way, the trace information can be directed to the display or to a file (textual or as a communication diagram). The output should be configurable, *e.g.* it can be specified whether to display all events or just certain kinds of events, if timestamps should also be displayed or not *etc.*

However, these modes of operation cause non-negligible overhead, which makes their usage for performance measurements problematic. Now our first aim is to derive an estimate on the overhead caused by the above modes, and then to elaborate other modes of operation causing less overhead.

Assuming that the participating computers are much faster than the network connecting them, the overhead can be roughly calculated as the additional time caused by additional network traffic. Let t_c denote the average time needed for a call through the network, and t_s the server-side processing time of a particular remote method invocation. (Usually, it cannot be assumed—even under the above assumption—that $t_s \approx 0$, because the server-side processing may also involve calls to other servants.) It follows that the duration of the whole invocation is $T = 2t_c + t_s$ (see figure 7).

Now consider the case in which interceptors are also present and they notify the `tracer` about every event. That is, an overhead of $2t_c + t_t$ is induced at every event—where t_t denotes the time consumed in the interceptors and the tracer—for the same reason as above and as figure 7 shows, with the only difference that the processing time is now t_t instead of t_s. Since there are four interception points, this sums up to $T' = T + 4(2t_c + t_t) = 10t_c + t_s + 4t_t$. Thus, the relative overhead is

$$r = \frac{T' - T}{T} = \frac{8t_c + 4t_t}{2t_c + t_s} \tag{1}$$

Since this is monotonously decreasing in t_s, the worst case is when $t_s = 0$, *i.e.* when server-side processing is not time-consuming compared to communication:

$$r_{wc} = \frac{8t_c + 4t_t}{2t_c} = 4 + 2\frac{t_t}{t_c} \tag{2}$$

Consequently, the relative overhead can be even more than 4 in the worst case, yielding a more than 5 times slow-down. The amount by which the relative overhead exceeds 4 depends on the amount of time spent in the interceptors and in the `tracer`. Most notably, if the `tracer` has to refresh some complicated GUI, this might take considerable time. Thus, t_t can be relatively high even in the case of non-interactive operation.

Of course, the situation is much better if the time needed for server-side processing is not negligible. It can be seen from (1) that if t_s is high enough, the overhead can become arbitrarily small.

It has to be noted that the above model can also be used if the whole 'distributed' system is actually located on the same computer. In this case, too, inter-component communication is the most time-consuming since it involves context switches which are known to be very costly on modern processors.

When tracing is used to perform performance-related measurements, this overhead can be much too large. At the architecture level, we can help by introducing new modes of operation (beside interactive and non-interactive mode):

- In *oneway mode*, the services of the `tracer` are declared as `oneway`. Thus, the messages towards the `tracer` do not block the interceptors, and this way the system does not have to wait until the notification reaches the `tracer`, which does the necessary processing, and the call returns.

- *Local mode* represents another solution: if tracing is used for time-related measurements, interceptors generate no additional network traffic by notifying the `tracer` about every event, but write out every collected information (including timestamps) to local files. Normally, it can be assumed that when it comes to performance analysis, the system is already behaving correctly. Thus, caching can be used to further decrease the overhead. Caching is usually provided by the operating system by default but of course proprietary caching policies can also be implemented. However, if the system is not reliable enough, this should be switched off (so that no events are lost in the case of a crash); this way, the trade-off between speed and reliability can be tuned. If, for some reason, the overall communication scenario of the system is needed, this may be assembled afterwards from the individual log files with the help of the timestamps and communication IDs (see later). Care has to be taken if multiple components reside on the same computer to make sure that every component writes to a separate file. Note that in local mode the responsibility of the `tracer` is limited to distributing trace-policy information (*e.g.* to inform every component that local mode is in effect) and IDs (see later) among the components.

- In *buffered mode*, the tracer is not notified directly after each communication event, but events are buffered locally in each component's name space. When the number of buffered events exceeds a given limit, or when the component is destructed, the buffered events are propagated to the `tracer` in a batch. (Oneway call is reasonable for this purpose as well.) This is advantageous because the bandwidth of the same network is usually much higher in the case of few large bursts than it is in the case of more short ones. On the other hand, large bursts can transiently corrupt performance measurements. However, if the measurements tolerate such transient errors, for instance because only the average performance is important, then it does not pose any problem. Of course this method is fastest in the case when the system to be traced works only for a limited time and the buffers can be large enough so that the `tracer` is only notified *after* the operation of the distributed system.

Of course, the overhead introduced by tracing can never be fully eliminated. But the easily implementable methods we introduced above can significantly reduce the overhead. Which of them works best, depends on the application and on the performance measures we are interested in. Some empirical results will be presented in section 5.

Note however that these modes of operation can be used only if the `tracer` does not have to return anything on the traced events. Note also that these modes of operation do not allow stopping the traced system (which is not a problem in the case of performance measurement).

4.5 Identification mechanisms

Since emphasis is laid on tracing high-level communication between components of the distributed system, the components have to be identified using unique IDs. Similarly, communication events should also be identified using IDs unique to a particular remote method invocation, so that the `tracer` can recognize events belonging to the same invocation and group them appropriately.

Note that IORs cannot be used as component IDs, because IORs identify CORBA object implementations. Hence, for instance, a client that does not implement any CORBA object, does not have an IOR either. Moreover, a component may contain several CORBA object implementations, and may consequently be associated with multiple IORs. Therefore, the IDs have to be generated.

IDs can be created either in a centralized or in a decentralized fashion. In the centralized case, there is one special component in charge of distributing the IDs. For instance, this component maintains a counter, and each time a new ID is needed, it increases the counter and returns its current value. Of course the bit length of the variable to store the counter has to be large enough so that no overflow will happen. To guarantee the uniqueness of the IDs, care has to be taken to synchronize access to this component, so that increasing the counter and returning its value are together an atomic step. Since there are common ways (semaphores, monitors, etc.) to achieve atomicity, we can conclude that implementing a centralized ID distribution scheme is fairly simple.

For the creation of component IDs, this is certainly the method of choice. Its only disadvantage is the overhead it generates, but this is negligible because it occurs only once per component. We could try to avoid or decrease this overhead by generating IDs locally, for instance based on IP addresses; however, such methods are less reliable, so it seems there is no point in using a more sophisticated generation method for component IDs.

In our case, it is reasonable to assign the role of ID distribution also to the tracer. So in practice the generation and usage of component IDs works as follows. At startup, or the first time that a communication event occurs, every component registers itself with the `tracer`, whereupon it gets an unique ID, together with some policy information. Later on, when a call is issued from this component, the request is intercepted by the corresponding `ClientRequestInterceptor`, which in turn packs the ID of the component into a service context and adds it to the call. When the request arrives at the server side, and is intercepted by the `ServerRequestInterceptor`, the ID of the caller is extracted. At this point the `ServerRequestInterceptor` knows both the ID of the caller and its own ID, which is now the ID of the callee. Similarly, at the third event, *i.e.* when the `ServerRequestInterceptor` intercepts the outgoing reply, it adds its own ID in a service context to the call, so that this information is also known at the fourth event. Note that at the first event, only the ID of the caller is known, but at all later events, both the ID of the caller and that of callee can be passed to the `tracer`.

The above way of embedding IDs in service contexts is very similar to the way that source and destination IDs are added in layered protocol stacks, such as the Open Systems Interconnect (OSI) model of the International Standards Organization (ISO).

Communication IDs, too, can be generated centrally. This means that when first notified of a remote method invocation, the `tracer` generates and returns an unique communication ID. This ID is then also added to the call in a service context and included in later notifications to the `tracer`.

However, this is not possible in oneway mode, because the first notification to the tracer returns something (the communication ID), but oneway calls cannot have return values. So either the first notification cannot be oneway, resulting in an overhead for every method invocation, or distributed ID generation is needed. Fortunately, generating unique communication IDs in a distributed way is not difficult because unique component IDs are already available. So at the first interception point, the `ClientRequestInterceptor` generates a locally unique communication ID, and combines it with the globally unique component ID of the client component to obtain a globally unique communication ID. This way, oneway tracing can be fully used, and the generation of communication IDs causes minimal overhead.

4.6 Sorting and time-stamping mechanisms

Based on communication IDs, the `tracer` can group the events belonging to the same remote method invocation. Ideally, the events should be displayed (either textually or graphically) in chronological order. However, as already noted, this is a non-trivial task in a distributed system because of the lack of a global clock and the varying and non-deterministic communication delays. This intrinsic problem of distributed systems has been known for a long time, and many solutions have been suggested (see, for instance [23, 24, 8, 11] and references therein). Here we just sketch some of them that can be applied easily in practice:

- Synchronized physical clocks. We can simply use the physical clocks of the participating computers for time-stamping events, provided that the clocks are synchronized with some external synchronization mechanism. This can be achieved for instance using the Network Time Protocol (NTP, [25]).
- CORBA Time Service. Since the problem of inconsistent local clocks arises in many distributed applications, the OMG defined the CORBA Time Service, which can be used to obtain consistent time stamps along with error estimates [26]. Unfortunately, the Time Service is implemented only in a fraction of the ORB implementations.
- Logical clock. Even if there is no global physical clock, a global logical clock can be implemented with a distributed algorithm. The clock is logical, because it has no connection with the real time; but it is guaranteed to be consistent with the partial order defined by the relation of logical precedence (also called 'happened-before' relation or causality). That is, there are no tachyons. The key of such algorithms is that every component maintains a counter (a local logical clock) and increases it at every event it participates in. Also, the current value of the counter is sent along with every message. (This can be easily implemented in our case using service contexts.) If the receiver's counter has a smaller value than the received value—which would mean a tachyon,— then the receiver increases its own counter so that it is now higher than the received value, and so the tachyon is avoided.
- Physical clocks with logical correction. A possible combination of physical and logical clocks is the following: mainly the physical clocks are used for time-stamping, but they are also sent along with every message. Just as with logical clocks, the receiver can check if a tachyon would be generated, and it can be avoided by setting the receiver's physical clock accordingly. (If the receiver does not have the permissions to change the system clock, it should just maintain a displacement, which is at the beginning 0, and update the displacement in such cases.) Although the time-stamps do not reflect the physical time precisely, today's computers are usually equipped with clocks of high precision so that in practice, time-stamps generated this way provide a good approximation of the physical time. (Actually, in this scheme the fastest clock in the system will determine the flow of time.) This approach combines the benefits of physical and logical clocks in that tachyons are avoided, and the time spent between events can also be measured (which is not the case with logical clocks). Also note that clocks never have to be set *back* which would lead to anomalous behaviour.

4.7 Startup and automation

Lastly, let us examine to what extent instrumentation can be automated using interceptors. Clearly, after the interceptors are initialized and registered, they intercept all remote method invocations, so that instrumentation is fully automatic. The only remaining issue is the initialization and registration of interceptors. Unfortunately, this mechanism is not the same for all ORB implementations, and even the standard is not programming language independent at this point. The problem is that interceptors actually become part of the ORB, so that their initialization is strongly connected to that of the ORB, which is not completely standardized.

The standard defines the `ORBInitializer` interface for this purpose. This interface has two methods: `pre_init` and `post_init`, both of them obtain an `ORBInitInfo` object as parameter.

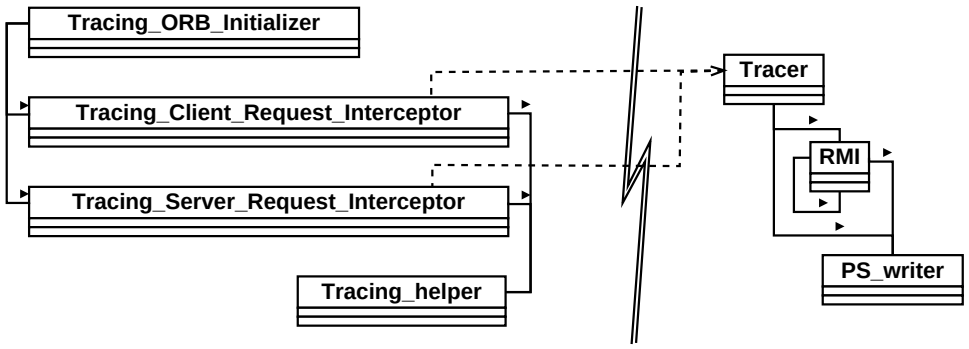

Figure 8: System architecture

The `ORBInitInfo` provides methods for the registration of interceptors. Consequently, in order to register our interceptors, we have to create a class that implements the `ORBInitializer` interface, and either in its `pre_init` or `post_init` method, we instantiate our interceptors and register them using the appropriate methods of `ORBInitInfo`. (The only difference between `pre_init` and `post_init` is that at the former not all ORB services are available.)

So far, so good. However, it remains to be clarified how our `ORBInitializer` should be registered, and this is the point where the specification is different for different programming languages. In the case of C++, the method `register_orb_initializer` is defined in the `PortableInterceptor` name space for this purpose. For Java however, no corresponding method is defined for security reasons, but the `ORBInitializer` is registered by means of Java ORB properties that can be provided for instance using command-line arguments.

Consequently, Java programs can be instrumented with no modifications whatsoever to the source code. In the case of other languages though, one line of code has to be inserted into every component to register the `ORBInitializer` (at the startup code of the component, before its call to `ORB_init`).

Inserting one line of code per component (actually, one per component *type*) is not much. However, it can pose a serious problem if the source code is not available. Fortunately, some non-Java ORBs also provide (proprietary) mechanisms to specify an `ORBInitializer` at the command-line, for instance in the form of a library. Another possible work-around is, when using an open-source ORB implementation, to slightly change the startup code of the ORB so that it loads our `ORBInitializer` automatically. (We tried this in the case of TAO, and it proved to be fairly simple.) If the application is linked against the ORB dynamically, then not even a re-link is necessary afterwards.

We believe that this part of the interceptor specification is not yet mature and it will change in the future. Note also that the automatic loading of interceptors (without any source code modification) has some security implications as well, since this way malicious code may become part of the ORB.

5 Practical evaluation of interceptor-based tracing

As a proof of concept, we have implemented the features described above in a tracing tool. The software was implemented in C++, using TAO, and has been tested on Windows NT and Linux workstations. The program currently supports three output modes: textual description of the communication events on screen and in file, as well as communication diagrams in PostScript format.

Figure 8 shows a simplified static diagram of the system. The figure consists of two parts: the right-hand part shows the classes that build the `tracer` component, whereas the left-hand part shows the classes that effectively become part of every other component. The dashed arrows between the two parts illustrate CORBA calls.

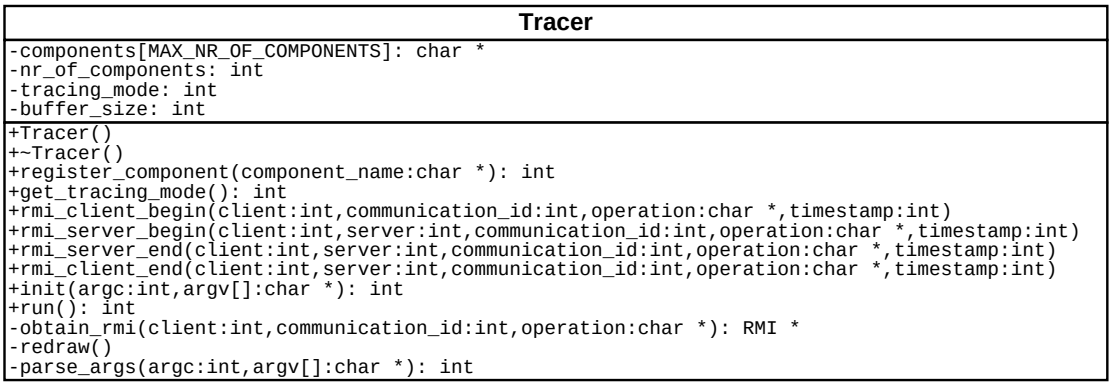

Figure 9: The `Tracer` class

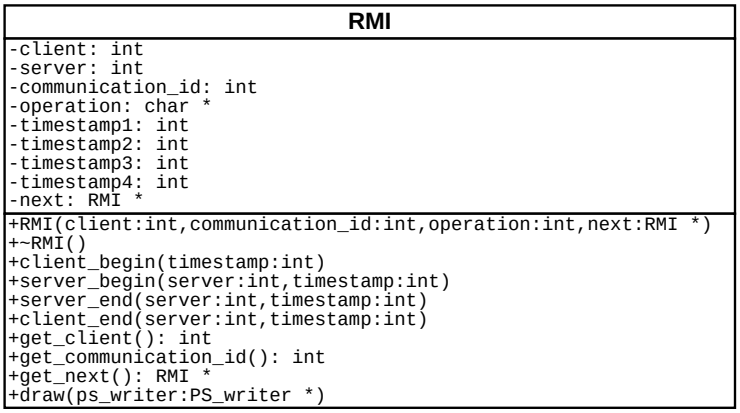

Figure 10: The `RMI` class

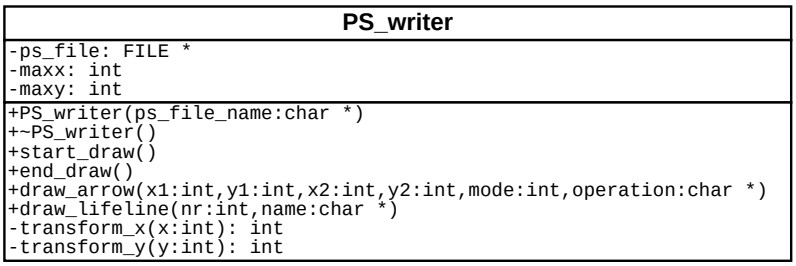

Figure 11: The `PS_writer` class

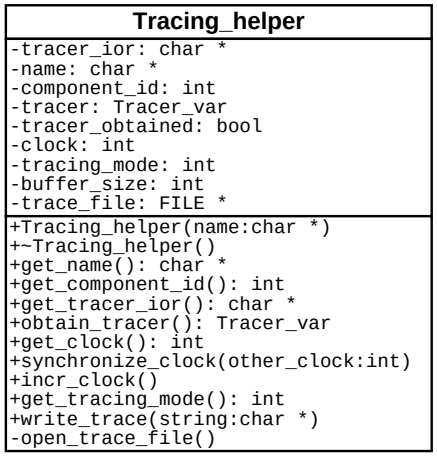

Figure 12: The `Tracing_helper` class

As can be seen, the `tracer` component consists of three main classes. The `Tracer` class (see figure 9) implements the `Tracer` interface that is defined in IDL, so that the interceptors can notify it. The information obtained from the interceptors is stored in `RMI` objects[1] (see figure 10), which are stored in a linked list. The `PS_writer` class (see figure 11) is used to produce communication diagrams in PostScript format.

The other part contains the classes that are needed for gathering trace information. The interceptors and the ORBInitializer classes exhibit the interfaces specified by the CORBA standard. The `Tracing_helper` class (see figure 12) is used to store component-specific information that is relevant for tracing. It also provides some helper functions for the communication with the `tracer`.

The implemented software performed very well in the tests, thus proving three important claims:

- The aims of tracing, as defined earlier, can all be provided for with essentially the same tracing architecture;
- The interceptor mechanism of CORBA provides a suitable framework for such a tracing architecture;
- In general, the middleware can provide powerful support for tracing in the form of meta-objects.

We evaluated the tracing tool on some small test programs and on a more complex application, which is a distributed stock information system, based on the Callback Quoter from [13]. This application consists of three components: the consumer, the notifier, and the supplier. The consumer is interested in selling stocks, provided that their price exceeds a given limit. Therefore, it registers itself with the notifier, also specifying the name of the stocks it is interested in, and the price limit. The supplier is connected to a database, and sends stock information periodically to the notifier. Every time the notifier receives a new bunch of stock information, it informs the registered consumers, provided that the information is relevant to them.

An example for a communication diagram generated by our tracing tool can be seen in figure 13. Note that—in contrast to standard UML-style communication diagrams—the arrows are not horizontal in this diagram. This way, the duration of the invocations as well as possibly concurrent activities can be better visualized.

We have also conducted some empirical measurements using the implemented tracing tool. A detailed evaluation is beyond the scope of this paper; here just the worst-case and some more

[1] RMI is just the abbreviation of remote method invocation; it does not refer to Java RMI.

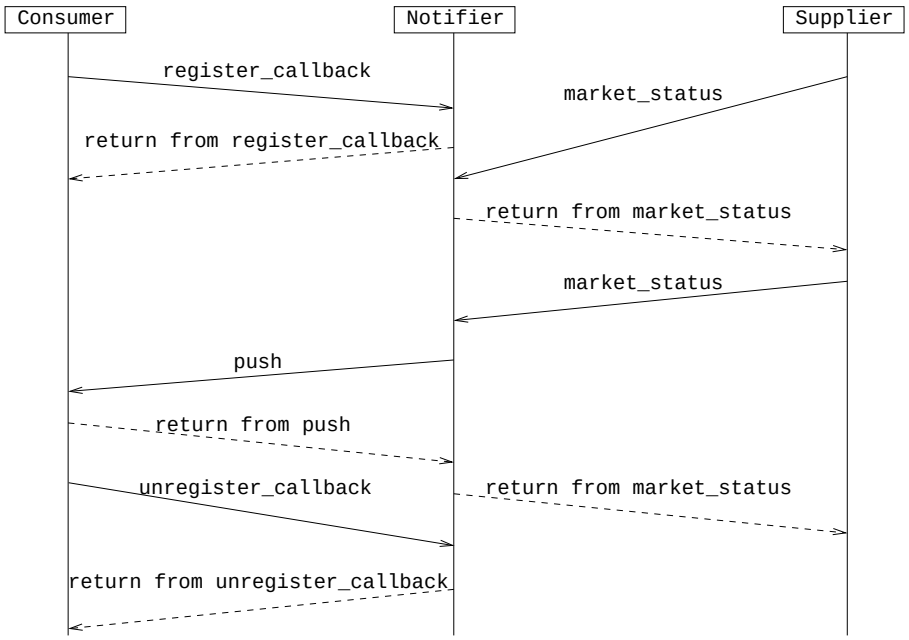

Figure 13: An example trace output

Table 2: Relative overhead in % caused by tracing

Mode	Worst-case [%]	Real application [%]
Non-interactive mode	549	7
Local mode, without caching	41	<1.5
Local mode with caching	15	<1.5
Oneway mode	37	<1.5
Buffered mode with transfer	29	<1.5
Buffered mode without transfer	13	<1.5
No tracing, just interceptors	11	<1.5

typical figures are presented for each working mode. The tests were performed on the stock information system and on a simple test application with an empty function on the server side (specifically for testing the worst-case scenario, *i.e.* when $t_s = 0$, see section 4.4), which the client called 10.000 times. Table 2 contains the relative overhead caused by the different modes of operation in the case of the two test applications. Each result is the average of 5 measurements performed on a PII/300 PC, under Microsoft Windows NT 4.0.

In the case of the stock information system, we estimated the precision of our measurements, based on the experienced deviation of the results,, to be around 1.5%. Within this error range, we found the results of all tracing modes except for non-interactive mode to be equal. Consequently, the overhead generated by them is not more than 1.5%. On the other hand, even non-interactive mode generated an overhead of just 7%. It follows that the actual overhead of the other modes is far less than 1.5%. Additional measurements would be needed to determine the overhead of the other modes precisely; however, this is beyond the scope of this paper. Nevertheless, the measured values clearly show the applicability of the method, which was our primary goal.

Concerning the worst case, the figures of table 2 clearly justify our previous estimates and show that non-interactive mode (which in the worst case generated a more than 6 times slow-down) is not usable for time-related measurements. However, both local mode with caching and buffered

mode without transfer presented a worst-case overhead of at most 15%, which is low enough to enable measurements aiming at finding performance bottlenecks, even concerning the worst case.

6 Related work

Xab [27] was one of the first tracing tools for distributed systems. Its purpose is run-time monitoring and debugging of PVM programs. Instrumentation is achieved in the form of a modified library. Consequently, instrumentation requires re-linking the software; moreover, the instrumentation process is programming language dependent. The instrumentation of C programs also requires the inclusion of additional header files, so that the source code is absolutely necessary. Xab is also different from our solution in that it does not focus on communication only, but generally on calls to the PVM library. A more recent work on the tracing of PVM applications can be found in [28, 29], which describes the adaptation of a general visualization tool (called Poet) to PVM.

Tracing of COM applications was extensively studied in [30]. That work is similar to ours concerning the aims; however, the methods are different. Since COM defines binary interoperability only, instrumentation requires low-level modifications to the COM library and/or the application. Another problem is the strict decoupling of interfaces and objects. It seems though that introducing interceptor-like meta-objects to COM would largely ease the tracing of COM applications as well.

An interesting approach for debugging RMI applications is presented in [31]. The idea is to unify server and client components to a single component for debugging. This means that each server has to provide two modes of operation: stand-alone and in-process server mode. Clearly, this is a workaround that may help in a given situation, rather than a general solution.

A similar, yet more general solution was presented in [32], and implemented in the EMUNET system. It aims at debugging TCP/IP-based distributed systems, and works by substituting the networking API with proprietary routines. This way, the distributed system becomes a uniprocessor program, in which the original components become threads communicating through shared memory. The EMUNET library is constructed in such a way that this transition can be made without modifying the original application source code, only by adding extra source code and relinking.

The MODIMOS system (Managed Object-based Distributed Monitoring System, [33]) is a framework for monitoring tools for distributed systems. It defines a layered architecture for the co-operation of local and global monitors; however, it does not address instrumentation explicitly, but assumes a suitable instrumentation mechanism. The authors also developed such an instrumentation mechanism specifically for CORBA, which is described in [34]. Their solution monitors remote method invocations using Orbix filters; however, also process and object life cycle events (creation and destruction) are monitored. On the other hand, their method has the following disadvantages: (i) it does not address automation of the instrumentation process; (ii) the source code of the application is necessary; (iii) the instrumentation mechanism is specific to a particular programming language (C++) and ORB (Orbix); (iv) the overhead is very high and quite unpredictable, in some reported cases more than 800%.

Distributed tracing is considered from the point of view of *global predicate detection* in [8]. This work aims at detecting specific states of a distributed application (e.g. a deadlock situation), which can be characterized as the conjunction of local predicates. Tracing is used to identify *consistent snapshots* of the system. Thus, communication is traced in order to uncover causal relationships between events. An implementation is also reported based on CORBA interceptors and the Java Debugging Interface. The author also reports on the difficulties of the interoperation of these two solutions. It remains an open question how this approach can be generalized to other programming languages.

A similarly general view is described in [35], which surveys possibilities for observation and control in distributed systems, and claims that debugging can be implemented based on these two. However, no practical evaluation is provided.

[36] deals with tracing the *order* of messages in a parallel program based on the message passing paradigm. The aim is to trace the minimum number of messages, using which the execution can be replayed deterministically. The method suggested by authors is based on run-time decisions on which messages to trace, and is proven to be optimal for a class of message-passing programs.

Specifically object-oriented distributed systems were considered in [11]. That work defined four different causality relations for object-oriented distributed systems: sequential program ordering, threads fork and join, synchronization, and transaction ordering. Moreover, a prototype implementation of a post-mortem analysis tool is presented based on these orderings for the Guide system.

Very similar results were reported in [9]. This work, too, defines several partial orders based on causality in a distributed object-oriented environment. Moreover, an implementation is presented for CORBA and Java. Instrumentation is achieved using a reflective language (OpenJava, which is an extension of Java). In a reflective language, meta-level classes can manipulate base-level classes, and thus can be used to insert tracing instructions.

Debugging in embedded systems is considered in [3] and in [37]. Specifically, [3] is only concerned with surveying the problems that may arise during tracing of distributed embedded systems, while [37] proposes a solution for the cyclic debugging of embedded real-time systems consisting of multiple processors. It is based on recording the events during the operation of the system, and later using this information to enable deterministic replay of the execution. This method may be combined with virtually any instrumentation technique for the recording phase.

An architecture for the supervision of CORBA applications was proposed in [38]. This architecture consists of a heavily modified ORB which is used to emit events, a modified GIOP transport layer for event multicast, and some supervision servers for collecting the events. The supervision servers build up a heavy-weight object-oriented model of the execution.

[39] presents details on the development of GLADE, the part of an ADA95 compiler that is responsible among others for tracing and replaying distributed ADA programs. In trace mode, inter-component messages are logged, which can be replayed later, in a consecutive execution of the system in replay mode. Thus, each component can be tested separately by simulating external events. Since the presented tracing facility is part of the ADA compiler, instrumentation is not an issue. Of course the solution is limited to ADA programs.

In recent years another promising research direction has emerged, which is mostly complementary to our approach: integration architectures for parallel programming tools [40, 41, 42]. The resulting interfaces enable different programming tools (*e.g.* debugger, checkpoint inserter, visualizer, profiler) to interact in order to show the programmer the most useful view on the software.

7 Conclusion

This paper has addressed the increasingly important problem of tracing distributed, heterogeneous applications. The most important contributions are:

- It has been shown that interceptors can be used to trace CORBA applications.
- The resulting tracing architecture can provide for all of the identified uses of tracing.
- A new tool for tracing CORBA applications, which was found useful in empirical evaluations.
- The overhead generated by interceptors and by tracing can be kept low enough to enable performance measurements.

Our future plans include the implementation of more advanced features in our tracing tool, as well as its integration with other, conventional tracing tools.

8 Acknowledgements

The work of Zoltán Ádám Mann was partially supported by a grant by Timber Hill LLC and by the PRCH Student Science Foundation. The work of Károly Kondorosi was sponsored by the Ministry of Education, Hungary, Application of ICT Framework – 4, No 026: 'Dynamic broker service for improving quality of database access and resource usage'.

References

[1] A. S. Tanenbaum and M. v. Steen. *Distributed Systems: Principles and Paradigms.* Prentice Hall, 2002.

[2] Object Management Group. Common object request broker architecture: Core specification, version 3.0.2. `http://www.omg.org/cgi-bin/apps/doc?formal/02-12-06.pdf`, December 2002.

[3] B. Cole. Debug and test in distributed systems. `http://www.embedded.com/story/OEG20010924S0111`, 2001.

[4] C. Ko, G. Fink, and K. Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of the 10th Annual Computer Security Applications Conference*, pages 134–144, Orlando, FL, 1994. IEEE Computer Society Press.

[5] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *1st International Conference on Aspect-Oriented Software Development (AOSD), Enschede, The Netherlands*, 2002.

[6] E. Simon. PC networking. `http://www.scit.wlv.ac.uk/~jphb/comms/esppt/esppt1/index.htm`, 1997.

[7] TechWeb. Network design manual: what is middleware? `http://www.networkcomputing.com/netdesign/cdmwdef.htm`, 1995.

[8] M. Otta. A distributed debugger framework applicable in a Java/CORBA environment. In *European Research Seminar on Advances in Distributed Systems*, 2001.

[9] L. Duchien and L. Seinturier. Reflection and debug for CORBA applications. Technical Report CNAM-CEDRIC 99-10, CNAM-Laboratoire CEDRIC, 1999.

[10] P. Kacsuk et al. GRADE: A graphical development and debugging environment for parallel programs. *Parallel Computing Journal, Elsevier*, 22(13):1747–1770, February 1997.

[11] P. Placide, G. Florin, L. Duchien, and L. Seinturier. Debugging of distributed object-oriented applications. In *Proceedings of ERSADS'95*, 1995.

[12] J. Siegel. *CORBA 3 Fundamentals and Programming.* John Wiley & Sons, 2nd edition, 2000.

[13] Real-time CORBA with TAO (The ACE ORB). `http://www.cs.wustl.edu/~schmidt/TAO.html`.

[14] F. Singleton. Help with designing or debugging CORBA applications. Linux Journal, `http://www.linuxjournal.com/print.php?sid=5453`, 2002.

[15] N. Wang, K. Parameswaran, and D. Schmidt. The design and performance of meta-programming mechanisms for object request broker middleware. In *Proceedings of the 6. USENIX conference on object-oriented technologies and systems*, 2001.

[16] N. Wang, D. Schmidt, O. Othman, and K. Parameswaran. Evaluating meta-programming mechanisms for ORB middleware. *IEEE Communications Magazine, special issue on "Evolving Communications Software: Techniques and Technologies"*, 39(10), 2001.

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

[18] Object Management Group. The common object request broker: Architecture and specification, version 2.2. `http://www.omg.org/cgi-bin/apps/doc?formal/98-07-01.pdf`, February 1998.

[19] Object Management Group. OMG portable interceptors request for proposals. `ftp://ftp.omg.org/pub/docs/orbos/98-09-11`, 1998.

[20] Portable Interceptors. Joint Revised Submission. `ftp://ftp.omg.org/pub/docs/orbos/99-12-02`, 1999.

[21] Object Management Group. The common object request broker: Architecture and specification, version 2.5. `http://www.omg.org/cgi-bin/apps/doc?formal/01-09-34.pdf`, September 2001.

[22] The ACE ORB. Portable interceptors. `http://www.cs.wustl.edu/~schmidt/ACE_wrappers/TAO/docs/interceptors.html`, 2003.

[23] A. Beguelin and E. Seligman. Causality-preserving timestamps in distributed programs. Technical Report CMU-CS-93-167, School of Computer Science, Carneghie Mellon University, Pittsburgh, PA 15213, 1993.

[24] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[25] D. L. Mills. Improved algorithms for synchronizing computer network clocks. *IEEE/ACM Transactions on Networks*, pages 245–254, June 1995.

[26] Object Management Group. Time service specification, version 1.1. `http://www.omg.org/cgi-bin/apps/doc?formal/02-05-06.pdf`, 2002.

[27] A. Beguelin. Xab: a tool for monitoring PVM programs. In *Proceedings Workshop on Heterogeneous Processing WHP'93*, pages 92–97. IEEE Computer Society Press, April 1993.

[28] T. Kunz and D. Taylor. Visualizing PVM executions. In *Proceedings of the 3rd PVM Users' Group Meeting*, 1995.

[29] T. Kunz, J. P. Black, D. J. Taylor, and T. Basten. Poet: Target-system independent visualizations of complex distributed-application executions. *The Computer Journal*, 40(8), 1997.

[30] G. Hunt and M. Scott. Intercepting and instrumenting COM applications. In *Proceedings of the 5. Conference on Object-Oriented Technologies and Systems, USENIX*, 1999.

[31] S. A. Donthy. How to eliminate debugging problems for RMI-based applications. JavaWorld Java Tip 56, `http://www.javaworld.com/javaworld/javatips/jw-javatip56_p.html`, 2001.

[32] X. Meng. EMUNET: design and implementation. A debugging aid for distributed programs in TCP/IP based networks. In *Proceedings of the IEEE International Performance, Computing, and Communications Conference*, 1997.

[33] A. Laurentowski, J. Szymaszek, A. Uszok, and K. Zielinski. A monitoring system for heterogeneous distributed environments. In *Proceedings of the EURO-PAR'95*. Lecture Notes in Computer Science, Springer-Verlag, 1995.

[34] J. Szymaszek, A. Laurentowski, and K. Zielinski. Instrumentation of CORBA-compliant applications for monitoring purposes. In *Proceedings of the European Research Seminar on Advances in Distributed Systems*, 1997.

[35] V. K. Garg. Observation and control for debugging distributed computations. In *Proceedings of the Third International Workshop on Automated Debugging*, pages 1–12, 1997.

[36] R. H. B. Netzer and B. P. Miller. Optimal tracing and replay for debugging message-passing parallel programs. In *Supercomputing '92*, 1992.

[37] H. Thane and D. Sundmark. Debugging using time machines: replay your embedded system's history. In *Proceedings of Real-Time & Embedded Computing Conference*, 2001.

[38] Ch. Gransart, Ph. Merle, and J.-M. Geib. GoodeWatch: Supervision of CORBA applications. In *Proceedings of ECOOP*, 1999.

[39] D. Neri, L. Pautet, and S. Tardieu. Debugging distributed applications with replay capabilities. In *Proceedings of Tri-Ada '97*, 1997.

[40] R. Wismüller, Th. Ludwig, A. Bode, R. Borgeest, S. Lamberts, M. Oberhuber, Ch. Röder, and G. Stellner. THE TOOL-SET project: Towards an integrated tool environment for parallel programming. `http://citeseer.nj.nec.com/46197.html`, 1997.

[41] J. M. May. *An extensible, retargetable debugger for parallel programs.* PhD thesis, University of California, San Diego, 1994.

[42] T. Ludwig, R. Wismüller, M. Oberhuber, and A. Bode. An open interface for the on-line monitoring of parallel and distributed programs. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):160–174, 1997.