Tracing system-level communication in object-oriented distributed systems

Zoltán Ádám Mann
Budapest University of Technology and Economics
Department of Control Engineering and Information Technology
H-1117 Budapest, Magyar tudósok körútja 2, Hungary
zoltan.mann@cs.bme.hu

This paper won first prize at the 2001 IEEE Hungary Section Student Paper Contest.

Abstract— **Standard 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. In this paper, the author collects the requirements that a tracing architecture is supposed to fulfill, and investigates how such a tracing architecture may be implemented in a distributed, heterogeneous and object-oriented environment. As a practical contribution, a system for tracing CORBA applications is presented, based on the *interceptor* mechanism.**

I. Introduction

As computers are more and more interconnected, the target of software development also becomes a distributed, heterogeneous system, rather than a single computer. 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 the paper focuses on the tracing of object-oriented distributed systems and particularly on CORBA (Common Object Request Broker Architecture [1]), being the most widely used middleware system. (For tracing of other middleware systems, see *e.g.* [2], [3], [4].)

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 point is the presence 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. Obviously, tracing is a vital and complicated task in such an environment.

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.

This paper addresses the problem of tracing distributed systems in the following way. First in section II, an overview is given about tracing in general and then (in section III) more specifically, tracing in a distributed environment. This is followed by a discussion on the role of the middleware in section III-D. As a concrete example, CORBA *interceptors* are investigated as a possible mean for tracing in section IV. Sections V and VI present development details and an analysis of interceptor-based tracing. Section VII concludes the paper.

II. Tracing in general

Since the term 'tracing' is rather overloaded (especially in the context of computer science), its meaning has to be clarified first (section II-A). Moreover, tracing is often used as a synonym for debugging. However, tracing is a lot more than that: section II-B presents other important aims of tracing. Section II-C covers the basics of standard tracing mechanisms.

A. The definition of tracing

As with every definition (especially those that try to define a term that has already been widely used in many different contexts), it is hard to find a perfect definition that is flexible enough to include all possible uses, and at the same time specific enough. The following definition is probably not perfect either, but it will do for the rest of this paper.

Definition: Tracing is a step-by-step execution of a program or 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 program or software system works.

Note that 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).

B. Aims of tracing

As already mentioned, debugging is not the only purpose of tracing. Indeed, as a part of this work, the following possible use cases of tracing have been identified in discussions with programmers and software engineers:

1. **Checking of correct behaviour**. 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 inner 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 of the error and in finding it.
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 of 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 large picture of how the system works (*e.g.* how existing modules co-operate and communicate), they can use tracing.
5. **Extracting documentation**. There are already tools that can extract static documentation from the source code of the 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. Typically, the goal is not a precise measurement but rather to identify bottlenecks.

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 these requirements.

However, it has to be noted that the above requirements 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 require some on-line user interaction, while the last two 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.

C. Typical solutions

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.

Usually the compiler will add extra information ('debug information') to the machine-level code so that tracing steps correspond to instructions of the source code and not to machine-level instructions. When debugging is on, an interrupt will usually be called at these instruction boundaries, resulting in a step-by-step execution of the program.

Also, debuggers typically offer more sophisticated features as well (such as Run to cursor, Breakpoints, Conditional breakpoints *etc.*) and the whole debugging-profiling-tracing functionality is available through a special user interface, which is usually integrated into the programming environment.

III. Tracing in a distributed environment

A. Difficulties in a distributed environment

As is often the case, the distributedness of the system can cause several problems.

First of all, the place where trace information is created and the place where it is needed are probably 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 object B obtained a message from object A, becomes available earlier than the information that object A sent a message to object B.

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 extremely 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, as explained in the previous section, traditional tracing tools are usually bound to a particular programming language. If the components of the system are implemented in different programming languages, this becomes infeasible.

B. Solution framework

In this, work, the following framework was used 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.

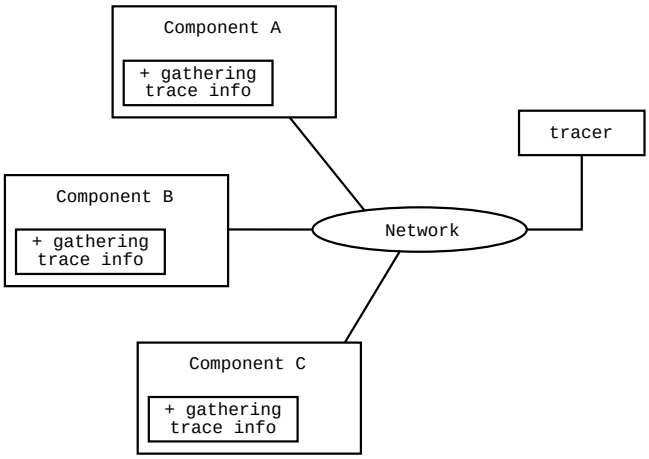

Fig. 1. The distributed tracing model

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.

Another vital aspect is the scope of tracing: it is logical to lay emphasis on large-scale communication. The reason is that standard tracing techniques may be used very well 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.

C. 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 some functions for this. So the programmer will call 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. The notification should include information such as the initiator and the target of the call, 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:

```
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);
```

As can be seen, manual instrumentation is tedious and error-prone. Moreover, it is necessary for manual instrumentation to possess the source code. Therefore, the 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.

D. 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 appearance 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 abstracion levels, not only for system-level communication. An example for such a system is GRADE [5]. However, this kind of distributed software development is not typical, largely because middleware systems have to be able to integrate legacy applications as well.)

Moreover, the problems of distributed tracing (mentioned in section III-A) 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: a mechanism is presented that can be used to provide automatic instrumentation for CORBA applications and thus achieving tracing of system-level communication.

IV. CORBA interceptors

Interceptors are objects implementing the `Interceptor` interface [6]. There are two kinds of interceptors: `ServerRequestInterceptors` and `ClientRequestInterceptors`. Both interfaces define callback methods that are invoked by the Object Request Broker (ORB) at specific points of a CORBA call. See figure 2 for the flow of control.

Both kinds of 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.

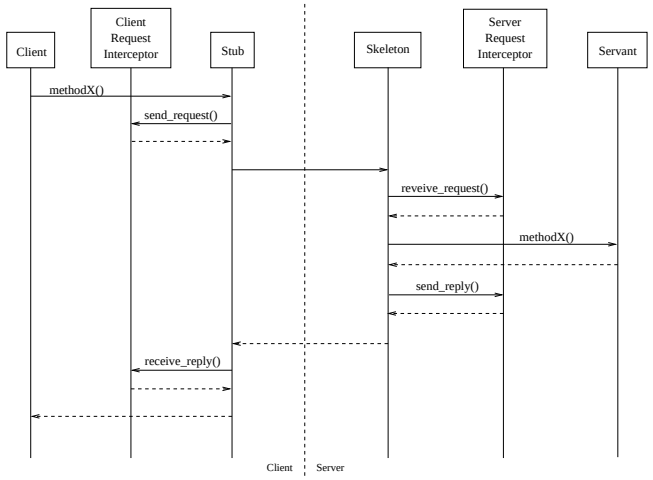

Fig. 2. Interception points during a CORBA call

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 also alter some of the information contained in the `RequestInfo` object. For instance, an interceptor might change the target of the call. This way, load balancing or fault tolerance schemes may 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. These are probably the goals that interceptors were actually developed for.

Interceptors were first defined in CORBA 2.3 [1]. However, this 9-page definition was quite under-specified. (For a comparison: the current draft interceptor specification consists of 254 pages [7].) This resulted in a number of proprietary solutions from different vendors. The problem was recognized by the OMG, which issued a Request For Proposals [8] in September 1998. 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 [9]. The architecture described in this submission seems to be the *de facto* standard since then. It has also been incorporated into the CORBA 3.0 draft.

Since the Joint Submission is available, vendors are working on their interceptor implementation in order to make it conform with the specification. The ORB used in this work, TAO [10], was one of the first to introduce support for interceptors; however, also in a proprietary way. TAO version 1.1, which is at the time of writing still the latest commercially supported version of TAO, reflects that proprietary mechanism. On the other hand, 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 1.1.14.

V. An interceptor-based 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 remote procedure call (RPC) 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 may be displayed. On the other hand, the user interface of the `tracer` may be configured so that, say, only one event is shown for each RPC. Since this is controlled centrally, the consistency of the output is guaranteed.

Since emphasis is laid on tracing high-level communication between components of the distributed system, the components involved in a particular RPC have to be identified. In order to achieve this, every component registers itself at the `tracer`, whereupon it gets an unique ID. 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 enough (namely the ID of the caller and its own ID, which is now the ID of the callee) to inform the `tracer` about the call. The first two events can be generated. 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.

Communication events are also identified using IDs, so that the `tracer` can recognize events belonging to the same RPC. Therefore, when the `tracer` is first notified of an RPC (*i.e.* when the first two events are fired), it 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` (when the third and fourth events are fired). It is then the `tracer`'s responsibility to group the events belonging to the same RPC and handle them appropriately, *e.g.* by sorting them using the CORBA Time Service.

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 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.

What exactly the `tracer` will display, depends on the mode it is used in. It has already been stated that the `tracer` must have at least two working modes: interactive and non-interactive mode. In interactive mode, the tracer waits for user input inside the notification methods, thus blocking the whole 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 in PostScript format). The output is 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.* Another mode of operation (the so-called local mode) is presented in section VI, which is useful for performance measurements.

Lastly, let us examine to what extent instrumentation can be automated. Since the used version of TAO did not yet support the standard registration mechanism of interceptors, they had to be registered manually, both with the ORB and the `tracer`. For this purpose, a new class (`InterceptorLauncher`) was created, which encapsulates in its constructor the details of the registration mechanism. So there is a single line of code that has to be inserted into the source code of every component (at the startup code of the component, before the first CORBA call that the component is involved in), creating an `InterceptorLauncher` object, which in turn automatically creates and registers the necessary interceptors, and stores the ID obtained from the `tracer`. Also, a name can be specified as an argument to the constructor of `InterceptorLauncher`, which will be used by the `tracer` when displaying events related to the component in question.

After this initialization, all tracing is done automatically; no manual instrumentation is necessary. Only the insertion of one line of code is needed for each component. But this, too, can be a problem, especially if the source code is not available. The standard registration mechanism of interceptors, as defined by the Joint Submission, will probably remedy this problem. Until this gets integrated into TAO, a slightly modified version of the TAO dynamic link library was developed, which automatically loads the interceptors. Using this library instead of the normal one, no code modifications are needed, not even a re-compile or a re-link.

VI. Evaluation of interceptor-based tracing

A. Implementation

The software was implemented in Visual C++, using the previously mentioned ORB TAO, and tested on Windows NT 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. Moreover, it provides three modes of operation:

1. Interactive mode. Tracing information is displayed on the graphical user interface of the `tracer`, and user input is needed in each step to continue execution. The extent of the displayed trace information can be fine-tuned using several options: the set of RPC events to be displayed can be specified for both synchronous and asynchronous calls. The essential use cases of this mode are: Checking of correct behaviour; Locating bugs; Better understanding of the operation of the system.
2. Non-interactive mode. The only difference from interactive mode is that the execution of the program being traced is not suspended. Essential use cases: Checking of correct behaviour; Monitoring crucial applications; Better understanding of the operation of the system; Extracting documentation.
3. Local mode. No central `tracer` is installed, but the interceptors write the locally collected information to a (local) file. This is a stripped-down variant of the tracing architecture with limited functionality, but since network communication is kept at a minimum, this is the fastest operation mode. The system automatically switches to this mode if the interceptors cannot find the `tracer`. No GUI is available, therefore the tracing options must be specified in a file named `tracer.ini`. The essential use case is: Performance analysis.

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

- The aims of tracing, as defined in section II-B, 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 with meta-objects.

The implementation also revealed some shortcomings of the interceptor mechanism of TAO, which have been fixed by the TAO team since then or are likely to be fixed in the near future.

B. Overhead analysis

Although the implemented software performed very well functionally, it is clear that the overhead generated by interceptors can be problematic in some applications, most notably if tracing is used for time-related measurements. First, a simple model for the estimation of the overhead is presented, followed by the corresponding empirical results.

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 RPC. (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 RPC is $T = 2t_c + t_s$ (see figure 3).

Now consider the case in which interceptors are also present and they notify the `tracer` about every RPC event. That is, an overhead of $2t_c + t_t$ is induced at every RPC event—where t_t denotes the time consumed in the interceptors and the tracer—summing up to $T' = T + 4(2t_c + t_t) =$

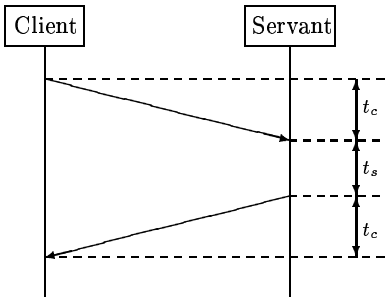

Fig. 3. Duration of a RPC

$10t_c + t_s + 4t_t$. Thus, the relative overhead is

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

The worst case is when $t_s = 0$, *i.e.* when server-side processing is not time-consuming compared to communication:

$$r_{wc} = \frac{8t_c + 4t_t}{2t_c} = 4 + 2\frac{t_t}{t_c} \quad (2)$$

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.

One possible solution to the problem of the potentially high overhead is to declare the services of the `tracer` 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. Hence, such a solution would minimize the overhead. But of course this is only possible if the methods of the `tracer` do not return any results.

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 RPC 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 (usually provided by the operating system by default) to further decrease the overhead. However, if the system is not reliable enough, this can be switched off (so that no RPC 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 might be assembled afterwards from the individual log files.

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-process communication is the most time-consuming factor since it involves cotext switches which are known to be very costly on modern processors. Care has to be taken though in local mode to make sure that every process writes to a separate file.

Finally, some empirical measurements were conducted on the implemented tracing tool. A detailed evaluation is beyond the scope of this paper; here just the worst-case figures are presented for each working mode. A test application was implemented, which also included an empty function on the server side (specifically for testing the worst-case scenario, *i.e.* when $t_s = 0$). The client called this function 10.000 times. Table I contains the average duration of this operation (averaged from 5 measurements each; all measurements were performed on a PII/300 PC, under Microsoft Windows NT 4.0).

TABLE I
EMPIRICAL RESULTS

	Duration [sec]	Relative overhead [%]
Without interceptors	29.5	-
Non-interactive mode	191.5	549
Local mode, without caching	41.5	41
Local mode with caching	33.8	15

The figures of table I clearly justify the above estimates and show that non-interactive mode (which in this case generated a more than 6 times slow-down) is not useable for time-related measurements. However, local mode with caching presented a worst-case overhead of 15%, which is low enough to enable measurements aiming at finding performance bottlenecks.

VII. Conclusion

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

- An investigation of the possible use cases of tracing.
- It has been shown that interceptors can be used to trace CORBA applications.
- The resulting tracing architecture can provide for all of the identified use cases.
- A new tool for tracing CORBA applications, which was found useful in empirical evaluations.
- A simple mathematic model for the estimation of the overhead generated by interceptors, which was justified by practical measurements.

References

[1] Object Management Group, "The common object request broker: Architecture and specification, 2.3 ed.," `http://cgi.omg.org/cgi-bin/doc?formal/98-12-01`, 1998.

[2] A. L. Beguelin, "Xab: a tool for monitoring PVM programs," in *Proceedings Workshop on Heterogeneous Processing WHP'93*. Apr. 1993, pp. 92–97, IEEE Computer Society Press.

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

[4] 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.

[5] P. Kacsuk et al, "GRADE: A graphical development and debugging environment for parallel programs," *Parallel Computing Journal, Elsevier*, vol. 22, no. 13, pp. 1747–1770, Feb. 1997.

[6] 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.

[7] "Interceptors FTF published draft of CORBA core and services chapters," `ftp://ftp.omg.org/pub/docs/ptc/00-03-03.pdf`, 2000.

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

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

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