Using Execution Trace Data to Improve Distributed Systems

Johan Moe
Ericsson Radio Systems AB
Box 1248
SE-581 12 Linköping, Sweden
+46 13-28 48 94
johan.moe@era.ericsson.se

David A. Carr
Institutionen för Systemteknik
Luleå Tekniska Universitet
SE-971 87 Luleå, Sweden
+46 920-919 65
david@sm.luth.se
Summary

One of the most challenging problems facing today’s software engineer is to understand and modify distributed systems. One reason is that in actual use systems frequently behave differently than the developer intended. In order to cope with this challenge, we have developed a three-step method to study the runtime behavior of a distributed system. First, remote procedure calls are traced using CORBA interceptors. Next, the trace data is parsed to construct RPC call-return sequences, and summary statistics are generated. Finally, a visualization tool is used to study the statistics and look for anomalous behavior. We have been using this method on a large distributed system (more than 500,000 lines of code) with data collected during both system testing and operation at a customer’s site. Despite the fact that the distributed system had been in operation for over three years, the method has uncovered system configuration and efficiency problems. Using these discoveries, the system support group has been able to improve product performance and their own product maintenance procedures.

Keywords: Tracing, distributed systems, CORBA, execution visualization

Introduction

Understanding the execution of a distributed software system is one of the most challenging problems facing today’s software engineer. During actual use, the system frequently behaves differently than the developer had intended. This may be due to unanticipated use patterns or to a continuously changing runtime environment. Difficulty in comprehension is compounded if the software engineer is not the system’s developer — a common occurrence with changing personnel and roles. We are developing a method to improve the understanding and development of distributed systems based on operational data.

Our method consists of three steps:

1. The system’s execution is traced during operation at the remote procedure-call level.
2. The trace data is post-processed in order to extract summary statistics and to reconstruct call chains.
3. The post-processed data is studied using information visualization.

We are simultaneously using and refining our method. This has lead us to use standard tools in all steps. The system uses CORBA to implement its client-server protocol, and our trace mechanism uses CORBA’s interceptor definition1 as the basis for implementation. The raw trace data is then processed and summarized with standard parsing tools (Bison). Finally, a commercial visualization tool, Spotfire.net2, is used to study trace summaries.

We are currently working with a single, large application framework and a single development group. However, we feel that the method will be generally applicable to large distributed systems and usable by other development groups. The application framework is built and maintained by Ericsson Radio Systems in Linköping, Sweden. It is part of an operation and maintenance (O & M) system for cellular telephone networks. The system consists of seven applications and more than 500,000 lines of code. Its primary
The purpose is to provide a toolbox to optimize a radio network. The system was originally deployed as a three-application framework in 1997. The system is supported by four teams. One maintenance team is responsible for released versions. Two design teams (12 people) are developing the newest version. Finally, a requirements team is handling current and future releases.

Initial use has produced promising results. Despite the fact that portions of the O & M system have been in use for years, we have discovered design flaws such as, a server raising an exception for a normal case and update requests from user interfaces that were not in use. In addition, a configuration problem was discovered that showed an abnormal number of calls to one of the system services. Performance problems and problems with many iterations over high-level interfaces have also been found.

**Background**

As a system grows larger, it has a tendency to require more attention from its administrators in order to retain performance and quality of service. Currently, administrators lack tools to help understand how a system performs during operation. As a result, system performance often degrades during maintenance — often due to the lack of knowledge about actual product use. For the O & M system, the problem of operational performance had been neglected — until recently. However, more mature customers began to demand better operational performance.

In addition, current procedures commonly used throughout the software development community have proven inadequate. Most correction and redesign work is based on trouble reports. But, trouble reports rarely include hard data. Nearly all performance and problem data comes from in-house testing. These tests focus on function, not performance. Nor, do they attempt to simulate actual use. As a result, what performance information developers receive is usually hearsay from short meetings with customers. This gives a diffuse feeling of performance and is seldom enough information for tuning, rebuilding, or other major decisions.

To solve this problem, developers need better tools for assessing system performance and behavior. These tools should help improve manual configuration and redesign. To go one step further, organizations also need tools for automatic configuration of the system, in order to minimize manual work. The maturity level of an O & M system user varies greatly. Therefore, tools should be more dynamic, fitting different users and their needs at specific moments. In order to build such tools, we need to observe a system as it runs.

Our goal is to build tools that will be used in commercial software development — first, to evaluate and improve current system performance and second, to provide input for monitoring and automatic configuration. In order to ensure that our tools solve the system developers’ problems, we are working with three different categories of developer: requirement engineers, system developers, and testers. Requirement engineers and system developers were selected because we believe that they can teach us about tuning and reconfiguration using trace information. Testers were selected in order to learn to find errors using trace information. We also expect to learn something about quality aspects such as test coverage.
Selling the Method

For a method or tool to be accepted, it has to be easily integrated into production. In addition, both developers and their managers must be convinced to use it. This means demonstrating results that are valuable to them. In our case, we also needed to convince customers to allow us to trace their O & M systems during operation. Otherwise, we would not be able to study operational system performance and much of the method’s value would be lost. This leads to the following set of technical requirements:

- The cost for retrieving trace data must be kept at a minimum, i.e., no extra work for developers and low impact on system performance.
- Automatic support for trace data analysis is necessary; manual work should be kept low.
- The visualization must be easy to use, as developers will not spend a lot of time coming up to speed on a new tool. It must also be adaptive to changing developer needs as to which data is visualized and how it is presented. Visualization must be flexible, yet focused on its purpose.

“Political” requirements were less straightforward. We had little trouble selling the method to management as Ericsson was supporting one of the authors as an industrial graduate student with the specific intent that his research would be applied within Ericsson. Even though one researcher was a member of the support team, it did not assure acceptance by other team members. This has been an ongoing process. As the method demonstrated useful results, other team members became interested and accepted that it was a useful tool for improving their product. This did not occur without qualms. For example, test engineers were disturbed when they realized the method gave a precise log of their activities. They needed to be reassured that this information would not be used to monitor them, but to improve their test procedures by comparing test and operational traces. The customer also needed more than just knowing that the impact on operations would be low. Most important was explaining how the trace would be used to improve their operations. The customer also needed the right (never used) to inspect any data leaving their operations center. Our experience is that the political aspects took much more effort than expected, but were necessary to maximize the impact of the project.

The remainder of this paper first presents a detailed description of our method and results to date. This is followed by a survey of related work and finally by our conclusions.

The Method in Detail

Our method consists of three steps: trace, summarize, and visualize. We use three steps in order to make the method more robust and adaptive to change. For example, tracing could be based on another remote procedure call system as long as it provides the same data. Similarly, the summarization can be altered and different statistics generated. In fact, we expect this to happen as we learn more about the operation of the O & M system. Finally, the visualization tool can be changed from Spotfire.net to a more specialized visualization tool.
Tracing

Much research has been invested in tracing and tracing applications, as an approach to observing a system. The emphasis has been primarily on debugging during development and in a laboratory setting. Little systematic tracing during commercial software development and customer operations has occurred. Our work explicitly uses trace information during the maintenance phase of a large commercial application.

The first problem is to determine where to add trace points in the system and what to trace. The fact that this was a commercial system imposed the following requirements on the tracing effort:

- Do not create additional work for developers,
- Have minimal impact on system performance,
- Acquire sufficient data to understand system behavior,
- Don’t acquire more data than can be analyzed.

The O & M system uses CORBA to implement an n-tier client-server architecture. As a part of the CORBA specification, interceptors are defined. Interceptors provide a hook at the remote procedure-call level that, among other things, can help create a tracing mechanism. Using them, we have created a tracing mechanism that captures the client/server communication (e.g., caller, called method, function, time-of-call, etc.). The tracing mechanism can be hooked onto any CORBA server or client during link time (C++) or runtime (Java). No extra programming is needed. Tracing mechanisms proposed by other researchers often require manual update of source or at best instrumentation, as for example³. Such implementation can give great insight from a research perspective. However, the approach is impractical for a commercial development project, where the use of commercial off-the-shelf (COTS) software makes source code modification and instrumentation problematic.

In order to minimize the impact of tracing on system operations, all data is sent to a remote log server via asynchronous calls. In addition, trace points can be turned on and off dynamically. The protocol between a trace point and the log server is built so that it helps compensate for time differences between hosts. The protocol is also built to reduce network traffic. For example, host and server data that is common for all events are sent only once — at communication set up.

In addition to providing a convenient hook, CORBA interceptors provide data at the “right” level to solve performance problems. Since the O & M system is distributed, performance problems are characterized by the delays required to perform various services. These delays are easily tracked via the call and return behavior of remote procedures. Collecting data at this level also reduces the amount of data collected when compared to collecting at the object or module level. We then extract statistics from the trace in order to reduce the amount of data that developers need to analyze.

Our design with CORBA interceptors has a couple of other advantages. The idea of interceptors allows us to use more than one implementation of CORBA. We have implemented the trace mechanism for two
vendors, Orbix from Iona, Inc. and Visibroker from Inprise, Inc. These implementations differed. With Orbix, we could not get hold of the message buffer containing parameters. Therefore, we could only get the target object and the operation sent (items explicitly handed to the interceptors). With Visibroker, this was not a problem as they provided the buffer. From this, we could calculate the amount of data sent between client and servers. Additionally, both Java and C++ were supported. For the Java implementation for Visibroker, we had to drop the notion of process as this concept does not apply to Java.

**Summarizing the Trace Data**

In the spirit of UNIX filters, three filters are successively applied to raw trace data. First, a filter sorts the raw trace data stream. Then, sequences of interactions are created whenever possible. Finally, statistics are calculated.

The raw trace data is first checked for causality violations and time differences between hosts. This could be called, building a *logical event trace* from the *measured event trace*\(^4\). The method used for time compensation is a tailored version of the algorithm proposed by Hofmann and Hilgers\(^5\).

Next, sequences are automatically built from a number of tracing events. This can be done automatically from information in the raw tracing data. This step will partition the tracing data into correct and incorrect tracing sequences.

While we can use information directly from previous sequences, we believe that the amount of information is still too high and therefore calculate summary statistics for different time frames, including by the hour, day, week, and month. For each period, the following is calculated:

- Response time: mean, variance, minimum, and maximum.
- Number of calls to a method: completed normally, completed with raised exceptions, and not completed.
- Number of calls between one client (method) and a server (method).
- Amount of data sent.

For the moment, the user needs to supply information on what interfaces each server supports. This is used to present functions not called during a trace session (e.g., to offer test coverage information).

**Visualization**

We are currently using the “Spotfire.net” visualization tool\(^2\) to analyze the trace summary statistics. Spotfire.net allows one to rapidly construct a scatter plot of the data set in either two or three dimensions. The tool allows specifying and changing the graphical encoding of parameters. If the analyst is interested in the relationship between calls ending in exceptions and those ending normally, these parameters can be chosen as the x- and y-axes. Other parameters can be configured for presentation on the graph as mark shape, color, size, etc. Spotfire.net also supports real-time filtering using widgets for specifying the ranges.
and values. This technique is called “dynamic queries”. The tool also presents a default view based on the order of parameters in the data set.

Spotfire.net is simple to use and allows the analyst to rapidly build a custom visualization. The data display itself does not contain complex visual elements, so learning time is minimal. The power is in the filtering and the real-time response to filter changes. It is simple and quick to set up a custom view to study an aspect of system behavior. We will use an early investigation into the number of exceptions thrown by the system to illustrate the ease of constructing a custom view.

- Figure 1 shows the default view generated by Spotfire.net when the trace data is imported. This view is of the number of successful calls versus the date and time.

- In order to display the number of exception calls, the analyst selects the drop-down menu for the pa-

![Figure 1](image1.png)  
**Figure 1** – Spotfire.net showing its default view after importing the trace data with the number of successful calls on the y-axis and the data-time on the x-axis.

![Figure 2](image2.png)  
**Figure 2** – Spotfire.net with the y-axis data parameter menu displayed.
parameter to be displayed on the y-axis. Figure 2 displays the menu. The menu’s selection button is just to the left of the menu.

- After selecting the number of calls ending in exceptions for the y-axis, the analyst selects by server as the x-axis (Figure 3). This shows that the event server is the only server showing many calls ending in exceptions.

- Finally, the analyst uses the parameter filters in the upper panel to the right of the scatter plot and deselects all other servers by deselecting their respective check boxes. In order to make sure that the counts are not a small percentage of the actual number of calls, the number of successful calls is selected as the x-axis parameter. (See Figure 4)

Figure 3 – Spotfire.net displaying calls ending in exception per server. Each point represents the number of such calls in an hour.

Figure 4 – Spotfire.net displaying calls ending in exception versus calls ending in success for the event server. The details of the circled point are shown in the lower panel to the left of the scatter plot. They indicate for the given hour that 197 calls failed while only 63 succeeded.
The final display shows a surprising result for the event server. Most points are above the 45° line, indicating more than 50% of all calls ended in exceptions. This indicates a potential problem warranting further investigation. (It turns out that the system was designed to respond to a “no new data” condition by throwing an exception — a questionable practice.)

It should be emphasized that this discovery was a result of exploring the data set. First, the analyst asked if there was any unusual exception behavior. After constructing a graphic display for exceptions by server, exceptions were discovered only for the event server. In order to determine the extent of the anomalous behavior, other servers were filtered out, and the graph normalized with the number of successes. The final graph shows the event server’s unusual behavior. The analyst did not a priori decide to investigate the event server, but rather to investigate the number of exceptions thrown to see if there might be an indication of a problem somewhere in the system.

As one can see from the above description, the immediate visual feedback encourages the analyst to explore the data. The dynamic queries method in effect allows the analyst to perform dozens of queries per minute by changing filters and by changing the parameters that are displayed on each axis. In addition, using a standard visualization tool is in keeping with our philosophy of keeping things as simple as possible so that developers don’t spend much time learning to use the system. We anticipate that, with time, standardized or customized views may be required. At that time, we will build them.

**Discussion**

We operated the O & M system with trace active at two sites:

- The developer’s, during the later stages of system test.
- A customer’s, during two weeks of regular system operation.

The trace data from both sites was then summarized and studied off-line. The analysis led to discoveries on two different levels. On the level of analysis, we discovered system problems, while on the level of technique we gained insight into how to use the method.

**The Importance of Different Time Frames**

One thing that we noticed immediately was that visualizing statistics gives a fast overview of the trace data. A developer can easily browse the material and get an overview of behavior and performance. One developer said that it was “like playing around with data”. This inspired him to look at the data from different views, giving a broader understanding. By selecting different time frames such as hours or days (or perhaps even longer), we seem to find different applications of the method to the development process.

By working with *large time frames* and collecting information about all possible methods, we have a tool that gives a rough estimate of coverage by showing all methods not called during a test session. While tracing during system test, it was discovered that several methods were not called. When we investigated this, we received the answer that the trace was taken during the latter stages of system test, and that the...
testing had been manually executed; therefore, complete coverage could not be expected. Still, our method provides an efficient way to verify that testing has reached a minimal level of coverage. It is also a good way to ensure that tests maintain coverage as the system is modified during maintenance.

On the other hand, tracing data from regular operation gave a good view of operational profiles. We found that the customer did not use all functions. As some of the unused functions are one of two alternative ways to accomplish a given function, a discussion started about removing the path that was not used. It was decided not to take anything away as the cost for leaving the code seemed cheaper than removal. However, no further effort will be made to improve the unused functions.

By working with small time frames, we have found problems such as unnecessary “iteration over interfaces”. Iteration over an interface is when a client repeatedly calls a remote (servant) procedure to obtain one item at a time, instead of obtaining a set of all items. The drawback is that much of the execution time is used for communication instead of data processing. (See Figure 5.) For one function, we decreased the execution time from around 30 minutes to a couple of seconds by removing one instance of iteration over an interface. Iteration is detected by identifying interfaces and methods with more than one call during a short period of time (one second in this case). Iteration over an interface appears to be a common mistake. However, it seems to take analyzing a trace of an operational system to catch it. This is an important finding since warning voices within our organization pointed out this problem in early 1999. It raises the question as to why the problem has persisted. Our method can partly compensate.

The Importance of Tracing Customer Operations

We found that many problems in the trace from customer operations did not show up in the system test traces. The iteration over interface problem illustrates this. These errors were easy to find with trace data from customer operations, but not so with data from test execution. The implication is that the system tests need to be revised, and the test data analyzed more carefully, in order to render realistic system behavior.

Before the developers accepted our method, they ignored potential problems indicated by the trace. Af-
terward at the customer site, these problems caused severe performance problems resulting in trouble reports from the customer. As a result, the maintenance team was left with a much more difficult situation than if the fault had been corrected at the first indication. The traces from the customer indicated the same behavior. This incident contributed to eventual acceptance of the method as a standard part of developer work practices.

During the time that we were tracing at the customer’s site, a new version of the O & M system was shipped to the customer. During the in-house system test, the new version’s performance seemed satisfactory. However during regular operation, the GUI responded more slowly to user input than expected. By examining the trace, we found that the event service sent events far too frequently. This slowed response time because the GUI had to regenerate much of the presented information for each event. This illustrates the importance of tracing during actual operations. While the same problem could perhaps have been detected by having many users operate at the same time during system test, observing actual operation is both more realistic and easier.

**Discovering Configuration and Design Problems**

During our initial setup, one researcher immediately spotted a strange behavior with one interface. A method had more exceptions than normal returns. This proved to be well known (by some developers) and part of the implementation. The exception was a signal for a polling mechanism to abort operation and run at a later stage. However, it was re-examined and found to consume resources when in fact the system was not in use. This behavior will be removed in a future release of the product.

We also found a server that had not been initialized properly. The server sent messages to another server with a high frequency, every five minutes instead of every 12 hours. A developer immediately reacted to this fact and found the configuration flaw. Finding such a problem in a raw trace would be difficult, as many events will have arrived at the log server during five minutes. Using the visualization, however, this became obvious.

Visualization pointed out some hard coupled objects; therefore, dynamic coupling metrics were calculated and presented to developers. The figures clearly pointed out servers and hence objects that were firmly coupled. The reaction was a common understanding that three servers and their objects could have been placed together. To the question as to why the servers were separated, we received two explanations. One theory is that it was originally believed that the objects would be distributed and replicated. This situation never arose and is now solved by other mechanisms. Another explanation was that the project was organized such that responsibility for the main servers were placed in different teams. This resulted in service divisions along team lines instead of architecturally sound lines.

A negative effect from the above exercise was that some developers started to think that the system had too many servers and proposed that adding new ones should be prohibited. However, a more balanced view prevailed. Services are added to existing servers when appropriate, and new servers are added if necessary.
Developers’ Lack of Information

When Ericsson’s developers are presented with the trace figures, it is clear that they have made incorrect assumptions. This holds for many system aspects. One, for example, is call frequencies. There is a daemon in the O & M system for handling and serving specific event notifications to one of the servers. The person responsible for implementation said that there should be something like 200 calls for a specific situation. In fact, tracing showed actual rates were one order of magnitude greater. The question was raised as to whether the higher rate, could overload the server. In the same way, it was shown that parts of the system were not often used and therefore needed no performance improvements.

Operational Profiles and Project Guidance

Musa has suggested that operation profiles can be used for work prioritization during project planning. For example, it can be important to test a frequently used function more thoroughly or earlier in a coming release. However, this has not been the case for the O & M system. The system manager and project leader stated that the “value” of the function is much more important as input for planning. High usage frequency does not by itself imply high value.

With a project leader in mind and from a risk perspective, we can say that trace information should be useful. For the O & M system, additional performance measures and prototyping started from trace information. We have also seen reconstruction of some system parts where Ericsson identified risks of low performance. Even if the project tasks are not reordered, the level of effort assigned to different tasks is changed.

Testing

With macroscopic collection and presentation, information can be used for pointing out unused functions. From that, usability engineers can investigate why, and developers can (correctly?) put less effort in maintenance of unused functions. Additionally, the trace data could establish that services in the system handling the GUI had low (or high) response time variance, which would mean that work was not (was) needed to remove this source of user irritation.

In addition, testers can use the information to improve test reality and begin to test issues such as performance as well as functionality. It seems fruitful to use data from both the operational environment and the actual design/test environment to make comparison reports. This type of information can give clues as to how well test cases simulate “real” behavior. By comparing operational traces with testing traces, we could see how relevant the test cases were and how to make improvements. Ericsson built several new test clients with better test data. They also discussed “moving” tests to customer sites, (i.e., a beta test) when they had problems creating a user model. By collecting trace data at a customer site during such tests, we would have a tool to observe the system outside the normal test environment.
Summary

The trace-summarize-visualize method has proven useful in solving a wide variety of problems. Many of these problems would be difficult to track down without the method. Within the development group, the method was initially greeted with a great deal of skepticism. Once it proved useful, it was integrated into the group’s standard practice.

Related Work

Our research draws on work in four related areas: software tracing, sequence identification, software visualization, and simulation.

Our use of CORBA is not unique. Choi and his colleagues have reported on a tracing mechanism based on CORBA interceptors. Their main concern was to understand the impact of tracing on system performance. They found it to be minimal. Similar work by Cho et al. showed an approach to clustering CORBA objects from information in a CORBA trace.

Our approach is similar to the approach used by some commercial tracing tools. Application Response Measurement (ARM) from the Computer Measurement Group is one example. ARM is an API that makes possible for applications to send trace information to a log server. The trace information can then be analyzed by commercial tools, which are compatible with the ARM specification. However, using ARM requires modification of the source code or using a remote terminal emulator that limits monitoring to user initiated transactions. For an existing system such as the O & M system, modifying the source is too costly, and using a remote terminal emulator will not provide detailed enough information. For example, iteration over an interface could not be detected with the remote terminal emulator approach.

Trace approaches can be divided into macroscopic and microscopic. Macroscopic approaches collect cumulative information while microscopic approaches concentrate on sequences of messages. No one approach is better than the other. Both can be used for comprehension, although they focus on different aspects. As an example, we often find macroscopic tools regularly running in an operational environment. With these tools, unwanted behavior can be detected. When a fault is spotted or suspected, more detail is usually desired, and a microscopic tool is employed. Our thought is that microscopic tools are necessary and can be of great use in combination with macroscopic tools. For example in our method, tracing sequences are identified. This makes it possible to combine a number of events in a sequence into one logical event. We have used simple pattern matching techniques to extract sequences. Bates has tried a similar approach.

Our method should be seen as macroscopic when it comes to visualization. It is similar to the object-oriented macroscopic visualization of R. J. Walker, et al. and W. De Pauw, et al. However, we are not focused on object-oriented systems, but on large distributed systems. We have found that as tracing sessions grow longer, one can perceive the stream of tracing data as infinite. Therefore, we present information summary statistics by time period (per hour, etc.). This is in order to build detailed operational profiles.
One uses microscopic approaches to learn how something is accomplished or to get more detail. Examples of this are seen in the work of Karam and De Pauw, et al. Karam shows execution traces on a timeline where it is possible to zoom in and out. His system provides a state-machine based interpreter for events. This, in turn, is used to construct timelines using an oscilloscope metaphor. The metaphor is further enhanced with annotation symbols that overlay the “signal trace”. De Pauw’s work focuses on displaying execution traces in an object-oriented system. They use Jacobson’s interaction diagram as a basis for visualizing execution sequences. They introduce a modification to the diagram to make recursion explicit and introduce several methods to compress the diagram. Two such methods, called “flattening” and “underlaying”, are shown in Figure 6. By contrast, our work concentrates on summary statistics of trace data over (relatively) long periods of time and uses scatter plot presentations of these statistics.

Frumkin and his colleagues report on a system that is aimed at debugging of message-passing parallel programs that includes the possibility of replaying execution. They augmented a communication library and provide two versions, with and without tracing. Their tracing must be enabled at link time by selecting the appropriate version, which is not as flexible as the run-time enabling provided by CORBA interceptors.

Renieris and Reiss have taken a different approach and automatically insert tracing calls into the program. In this way, they avoid source code modification. They provide an active display with the ability to jump from the displayed tracing data to the source code — a feature we see as a useful addition to our system. In addition, their visualization uses a spiral to depict trace data. The spiral makes better use of screen real estate by providing a denser image than linear methods. Our approach uses dynamic queries to avoid complex display compression techniques by allowing the analyst to filter out unneeded data.

Automatic fault detection has been tried by a number of researchers. For microscopic approaches, automatic loop detection has been tried in De Pauw, et al. and J. R. Larus. A totally different approach to fault detection is the use of query languages such as PARFORMAN. With this language, the authors demonstrate approaches to race detection, deadlock, and other concurrent programming problems. With query languages, the user simply asks questions about the tracing data. The problem is that one needs a good idea of what is desired. This is often not possible, especially for maintenance programmers who are unfamiliar with the system. This means that in practice they are limited to predefined queries. We believe
that a simple, graphical tool like Spotfire.net allows the user to explore the data such as in the exception call example illustrated in Figures 1-4.

In contrast to our approach of visualizing dynamic trace events, software visualization has traditionally concentrated on the source code as a basis for the visualization. Various metaphors have been used in order to increase understanding. SeeSoft\textsuperscript{21} used a source-file-based metaphor with files represented by boxes and with source code lines represented by colored lines within the boxes. The emphasis was on displaying various aspects relating source-code-modification history, although SeeSoft provided visualization of test-coverage. The call graph and class diagram have also been used as a basis for software visualization. Recent examples include C++ Lens\textsuperscript{22}, Software World\textsuperscript{23}, plus Schauer and Keller’s pattern-based tool\textsuperscript{24}. The C++ Lens displays inheritance hierarchies as trees and uses size encoding to emphasize relationships such as the number of references. Software World uses a cities-neighborhoods-buildings metaphor to represent software inheritance and containment relationships. Schauer and Keller’s tool uses a network to display interclass relationships with design patterns as an organizing theme.

Source-code-based visualizations primarily emphasize static aspects of software. Walker, et al.\textsuperscript{13} report on a trace visualization system that concentrates on dynamic aspects of object-oriented systems. Their system uses animation to convey dynamic relationships, and boxes and arrows to convey static relationships.

None of the above systems provide for real-time dynamic filtering which is available using dynamic queries\textsuperscript{6}. Some, such as SeeSoft, allow for user customization of their display.

Tracing and visualization are two ways to study systems, which already exist. However, simulation can be used before the system is implemented as well as for existing systems. A client-server system can be described in terms of clients, servers, and hardware. This description is then converted into a Layered Queuing Networks (LQN) model and either solved directly or simulated.\textsuperscript{25} This model can be use to study tradeoffs in the software architecture before implementation begins. After implementation, the model can be refined with operational data such as the data that we collected by tracing. The model can be rerun to predict the effects of architectural changes such as redistributing servers among processors. Later work in this area has developed methods to directly construct the LQN models from software architecture patterns\textsuperscript{26}, detailed modeling of database operations\textsuperscript{27}, and modeling CORBA’s RPC mechanisms\textsuperscript{28}. In all recent work, a clear trend is to derive the model directly from UML descriptions.

**Future work**

Although we have had early successes, much still needs to be done. One thing that we see is the need for more flexibility with regard to time scale. It seems clear that we need the ability to study data at both large and small time scales. The best time window depends on one’s current goal. It also seems that we need more flexibility in working with sequences. In addition to calculating statistics, we would like to examine execution details when the statistics signal anomalies. In the tester’s case, we must admit that the method’s implementation lacks some features that would make the method more useful. The figures were interpreted as coverage measures, i.e. quality measures. However, the lack of parameters in the trace
makes detailed coverage information unobtainable. For example, in the system a file database supports several queries. With our method it is only possible to see that a query was executed – not which query. However, the data domain (possible input) must also be specified so that a tool can point out those parts not used. In the situation where one uses CORBA and IDL (Interface Description Language), one does not find this kind of information. However, the information could be specified once at a small cost. Another problem is that the time interval was too coarse. The more experienced developers pointed out that they would like timestamps in microseconds – not as they currently are, in seconds. By using the method in day-to-day operations, we hope that feedback from developers will help build a robust and useful system.

From an ease of use point-of-view, we see that additional integration with source code and documentation is necessary. For example, developers might use the visualization to learn which services are the most commonly executed. A display showing frequency of execution versus time might be used. After selecting a point near the top of the scatter plot, developers would obtain the server and method names using the drill-down feature. At this point, they would need to leave the visualization in order to obtain further information. However, providing a link from the details obtained by drilling-down to related documentation and source code is better. This idea could also be used for examining trace details. To look for iteration over an interface, a developer could explore an interface’s call frequency by time period. However, all calls in a given period might not belong to the same use case. So, one might want to tie call information to use cases. With this information, the developer could drill-down to use cases and their documentation. (See Figure 7.)

We also see that the information gained in studying the system could be used to construct automatic monitoring tools for the operations staff. The trace could be monitored and compared against “normal” behavior. With a realistic model of what is “normal” system behavior, this information might be used to signal potential problems as they developed. This could lead to dynamic tuning to prevent or mitigate them. The data might also be used to plan expansion and upgrades.

![Figure 7](image.png)

**Figure 7** – An example of drill-down from object and calls to its interface to a unique sequence of events.
Conclusions

In order to provide better information, we have collected trace data from a system under test and a system in regular commercial use. This data has been parsed and then displayed using information visualization techniques so developers may obtain a better understanding of the system’s actual behavior.

With an early version of our method, we can see improvement in developers’ understanding of their system’s operational behavior. The method has been used to discover problems in a system that has been in operation for over three years. These problems include using an exception to signal a normal condition, iterative calls to an interface due to inadequate buffer size, and inadequate test coverage. Correcting some of these problems has improved system performance. Typically, these errors can be difficult to express in a trouble report. Therefore without methods like the one proposed, problems remain and annoy users for some time.

It truly seems that most developers do not completely understand a large system’s behavior. What may cause this situation is difficult to say. The sheer complexity of modern software is just one possible explanation. Lack of the opportunity to observe system operation is another. The problem can be that people simply do not see the point of observation (or have the time to do so) until they have a tool. Many of us do not see things in terms of a system when it comes to behavior and performance. For the O & M system, no one had or felt total responsibility. Therefore, the project team needed no tools, nor methods to observe system performance. This is a bit surprising, but apparent after having interviewed development-team members. At the same time, developers react to information from maintenance and their use of trace. This is often observed as developers taking further measurements to get more detailed information, with resulting actions. We have now started to explain the importance of having people grasp a more thorough picture of the system, not only their small part in it. Ericsson could appoint someone responsible for complete system behavior. However, this does not appear necessary. If the design group regularly gets performance measurements, they will do the work. It seems to be the natural behavior of a developer when such information is presented.

Taken from another perspective, we have now started to see patterns of common errors in particular repetitively calling an interface because a buffer has been inadequately sized. The method should be able to help both developers in correcting these faults and organizations in preventing common problems. For example, a line manager can start specific educational activities to prevent common mistakes.

In summary, we are confident that by tracing distributed systems at the remote procedure call level we can gain insight and understanding about how the system operates, while paying a low price. This knowledge can be used to improve the system throughout the maintenance phase of the system’s life.

Acknowledgments

This research has been in part funded by the Stiftelsen för Kunskaps- och Kompetensutveckling through their Industrial Graduate School Program. We would like to thank Kristian Sandahl for constructive
comments and Carl Rollo for proofreading. Finally, we would like to thank the anonymous referees for their contributions to improve this paper.

References


