Collecting Task Data in Event-Monitoring Systems

by

JIAJUN WU

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2004

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

Event correlation is an important analysis technique in monitoring systems. Various correlation methods have been widely applied in many systems. Recently, Sahai *et al*. proposed a correlation scheme based on the "transaction" concept in Web Services. While it has limited application because of its target dependence and scalability problems, the idea behind this solution is useful for identifying behavior patterns in distributed and parallel systems. Existing monitoring systems have no correlation method analogous to that of Sahai *et al*. We therefore wished to extract the general transaction concept and develop a correlation solution independent of the target system.

This thesis explores the task-based correlation mechanism in monitoring systems. We define a generic correlator independent of any target system. This correlator can be mapped to various concrete instances in various target systems. We develop a correlation scheme based on this correlator on top of the Partial-Order Event Tracer (POET). Our solution provides the general requirements for instrumentation as well as an algorithm to collect task-based correlation data and presents a visualization method for this correlation. We use the testbed tool and Java RMI to evaluate our solution. According to our cost analysis, our solution is efficient and has good scalability. Due to the abstract characteristic of our correlator, our solution is target-environment independent, eliminating one of the major disadvantages suffered by the system of Sahai *et al*.

Acknowledgements

Thanks are given to persons who help me kindly on this thesis. First, I would like to thank Prof. Paul Ward, my supervisor, for his supervision and guidance for the writing of this thesis. He provided lots of valuable suggestions and comments. I would also be grateful of my readers, Prof. David Taylor and Prof. Krzysztof Czarnecki, for their effort to read and comment on this thesis. My warm family gives me courage and confidence to finish this thesis successfully. There are still a number of persons who help me in my research work. Not mentioning their names does not mean that they are less valuable. On the contrary, I provide my most sincere acknowledgement here to all of the people who helped me.

Contents

List of Figures

List of Tables

Chapter 1
Introduction

A distributed system is composed of a number of loosely-coupled machines connected by some form of communication medium. In a distributed system the machines do not share system resources (memory, system clock, *etc*.). The entities in the distributed system interact by message passing. In such systems, the behavior of entities and interaction between them is not easily understood by the developer. The developer needs mechanisms to identify the behavior of the system, to enable determination of faults, and to optimize performance.

A monitoring tool is useful for a developer to track and analyze the behavior of distributed systems. It collects event data and provides functionality to analyze that data, such as visualization of events and their relationships. While such a monitoring tool is useful, the large amount of event data in distributed systems makes such analysis difficult.

Correlation is a widely used technique for event-data analysis. Event correlation is the process of determining relationships between events in order to identify patterns of events. In other words, event correlation is the process of finding related events according to some correlation criterion. It may help a developer identify behavior patterns and thus reduce the complexity of analysis. Existing tools provide various correlation mechanisms, such as "trace" in the Partial-Order Event Tracer (POET) [KBT+97], and "transaction" in the message-tracking system of Sahai *et al.* (which we will henceforth refer to as Sahai's system) [SMO+ 02]. However, these systems do not solve the problem completely since they have various limitations. For example, POET cannot capture such a relationship as "transaction." Conversely, Sahai's system can capture the transaction relationship, but only in Web Services, and does so inefficiently.

This thesis explores new ways to capture the transaction relationship efficiently and to remove the target-system dependency. Due to the target-system independence of POET, we adopt it as our base system to implement the new solution.

1.1 Motivation

For distributed and parallel systems, isolated event data captured by monitoring tools, without well-defined correlation, is of little value. Existing correlation solutions help the user identify patterns of behavior and reduce the complexity presented to the user to some degree. However, the size and complexity of existing systems is such that existing correlation solutions are insufficient. More-sophisticated correlation solutions are needed for monitoring and analyzing such systems, and more-efficient algorithms are needed for existing solutions.

Sahai *et al*. proposed a correlation scheme for message tracking in Web Services [SMO+02]. They provided a correlation solution based on the "transaction" concept. Compared with correlation mechanisms in other monitoring systems, their introduction of "transaction" solves some problems. However, it has two limitations. First, their definition and implementation are closely tied to Web Services. Second, their algorithm does not scale. We believe that "transaction" is a useful concept that can be applied in many systems beside Web Services. It can be mapped to different correlation factors in different systems. It can be used to correlate events across multiple target systems. The extension and redefinition of "transaction" will remove the target dependence in the solution of Sahai *et al*.

1.2 Limitations of Existing Systems

We studied correlation features in several systems, including POET, Log and Trace Analyzer (LTA), and Sahai's system. POET has some correlation features, such as traces, send-receive pairing, and automated abstraction. LTA [LTA Website] provides some correlation options, including URL, application ID, and time. It can correlate events based on individual or combined options of these criteria and on any user-defined criteria. The solution of Sahai *et al*. correlates SOAP messages by a tree structure in SOAP messages [SMO+02].

However, these solutions have various limitations. LTA has limited correlation options, some of which are non-deterministic. It does not define any generic correlator though one can be plugged in. In particular, it does not have a "transaction" correlator. Furthermore, it does not present partial-order information.

The scheme of Sahai *et al*. is not an efficient and scalable approach in terms of correlation data collection. It does not scale well with respect to the number of messages per transaction. Its application is also limited to the XML Web Services environment.

POET is a sophisticated monitoring system. It is independent of the target environment. However, it lacks a correlation mechanism that can provide a snapshot of a "transaction" in distributed systems, which Sahai *et al*. provided.

In summary, no existing monitoring system provides a complete and efficient solution for the transaction correlation proposed by Sahai *et al*. To build a monitoring tool with such a correlation mechanism, we must solve three major problems. They are

(1) Define a correlator which is target-system independent.
(2) Design an efficient and scalable collection mechanism for correlator information.
(3) Design a visualization mechanism for such correlation.

This thesis addresses these three problems, and provides a solution based on the existing POET system.

1.3 Contributions

This thesis has four contributions:
(1) We define a generic correlator, "task." The domain of our correlator is an abstract one which can be mapped to various concrete domains for concrete systems. Thus the correlator we define is target-system independent.

(2) A method of correlator collection is proposed to solve the problem of collecting correlation data for large and complex target systems (*e.g.*, multi-target systems). This approach includes an algorithm for instrumenting the target system and for defining the mapping between external and internal task identifiers in the event server of POET. Our approach is efficient, scalable, and flexible in terms of correlator collection, which solves the problem of the solution of Sahai *et al*. Our approach does not have significant additional cost.

(3) We implemented our solution within POET. Our tool has the functionality of collecting correlation data and performing correlation visualization. Our solution has good interoperability and backward compatibility. Any target system that works with the original POET needs no changes if correlation data is not collected and only minor changes if it is collected. The additional cost is not significant because we collect the correlator information with a small constant consumption of space and bandwidth. Our solution provides a visualization scheme for correlation at the event level. Because we adopt POET as our base visualization tool, our visualization is able to display the whole partial-order of events as well as correlation information.

(4) According to the generic algorithm in (2), we instrumented Java RMI and evaluated the results.

1.4 Organization

The remainder of this thesis is composed of four parts. Chapter 2 presents our research background and related work. In this chapter, we review three systems: LTA, Sahai's system, and POET. In particular, we discuss the correlation functionality of each of them. Chapter 3 presents the definition of our correlator and our correlation solution within POET. In this chapter, we provide general requirements and an algorithm for instrumentation as well as a method of correlation visualization in POET. In Chapter 4, we evaluate our solution. We first analyze the costs of our solution. Then, two target environments, the testbed tool and Java RMI, are used to evaluate our solution in a practical sense. Finally, we compare our solution with existing correlation techniques discussed in Chapter 2. In Chapter 5, we draw conclusions from our work and outline possible extensions.

Chapter 2
Background and Related Work

In this chapter, we give a brief introduction and review of three monitoring systems: IBM's LTA, Sahai's system, and POET. In Section 2.1, we describe the log file and Common Base Event (CBE) used in LTA. We focus on the visualization and correlation. In Section 2.2, we review the mechanisms of correlator collection and visualization of Sahai's system. In Section 2.3, the architecture of POET, its mechanism for event collection, and features of its display are described, since it forms the basis of our solution.

2.1 IBM Log and Trace Analyzer (LTA)

LTA is part of the Hyades project. It is an Eclipse-based monitoring system that monitors Java programs as well as analyzing log files generated by various systems including IBM WebSphere Application Server, IBM HTTP Server, IBM DB2 Universal Database, and Apache HTTP Server [LogTrace Website].

LTA has two sub-systems: a logging tool and a profiling tool [LTA Documents]. The profiling tool is the part that interacts with the instrumentation inside the target. The instrumentation in the target side is called the profiling agent. This agent can collect run-time data from the target process and send them to LTA for visualization and analysis. The logging tool is used to analyze various log files generated by the targets. In this way, both log data and profile data can be visualized in LTA.

2.1.1 Event Data and Collection

Two types of event format are employed in LTA. One is the format of the original log record, the other is CBE used inside LTA.

The original event data are stored in log files, and their formats and content vary from target to target. Their generation depends on the target itself. For example, log records generated by an HTTP server are different from those generated by the DB2 Database.

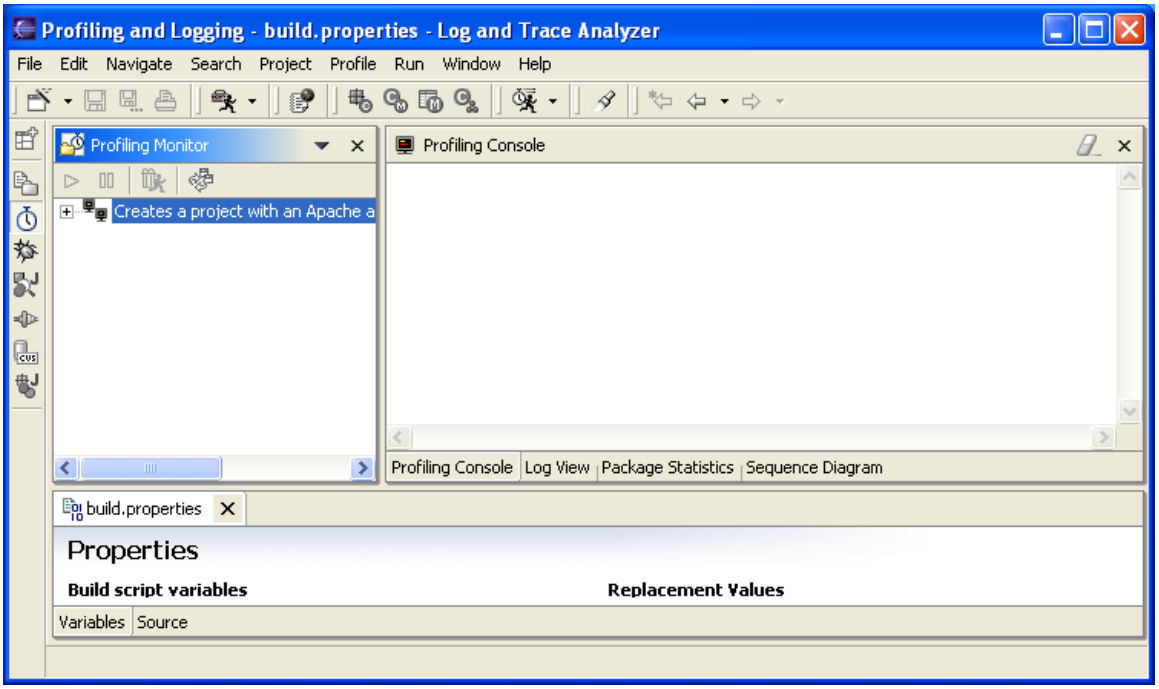

Figure 2.1: Layout of LTA

To solve the problem of the diversity of log file formats, the logging tool uses CBE to provide a consistent view for various types of event records. CBE uses an XML-based format to describe events. It defines the structure of an event in a consistent, common format [CBE Website]. Each CBE record represents an event occurring in a target. It includes the event identification, the identification of the reporting entity, the identification of the affected entity, associated message content, and related data. CBE improves the flexibility and interoperability of event data. However, its drawback is the problem of efficiency, since each CBE-format event is typically 1KB.

To convert various log formats to the CBE format, LTA needs a parser for each type of log. Such a parser is implemented as a plug-in for LTA. This structure gives LTA some degree of flexibility to deal with different event data generated by various targets. However, this parsing is another cost of using CBE.

The logging tool obtains event data from log files generated by targets. Only when LTA executes the import action is the log file containing event data read and converted to CBE format. The profiling tool obtains event data from instrumented targets.

2.1.2 Event Visualization and Correlation

LTA presents several views for users. The log view gives a tabular format. The user can view the information of any CBE event. The sequence-diagram view provides a graphic visualization for the events in the log file(s).

The correlation plug-ins correlate events based on the rules specified by the plug-in. The rule is the policy to order or group events according to the values of some property or properties of those events.

Existing correlations in LTA include correlation by time, correlation by URLs, and correlation by application IDs. They can be categorized into two types, sequence correlation and associative correlation [LTA Documents]. An example of sequence correlation is to order a set of events by time stamp. Correlating the events with same thread ID is an example of associative correlation. We will discuss these two correlations in Chapter 3.

Fragments from two log files are used to show the relationships between them. The records selected contain multiple errors so as to demonstrate the correlation found by LTA between access and error logs. The fragments are listed in Figure 2.2 and Figure 2.3. The log view and the result of correlation by time are shown in Figures 2.4 and 2.5, respectively.

2.1.3 Limitations and Restrictions

LTA has two limitations. First, large CBE records increase the probe effect for the profiling tool. This is the effect the collection of information imposes on the information being collected. Second, LTA cannot collect partial-order information at present.

```
......

9.131.0.90     -     -     [15/Jan/2003:10:41:00     -0500]     "GET
/scripts/root.exe?/c+dir HTTP/1.0" 404 289
9.131.0.90     -     -     [15/Jan/2003:10:41:01     -0500]     "GET
/MSADC/root.exe?/c+dir HTTP/1.0" 404 287
9.131.0.90     -     -     [15/Jan/2003:10:41:02     -0500]     "GET
/c/winnt/system32/cmd.exe?/c+dir HTTP/1.0" 404 297
9.131.0.90     -     -     [15/Jan/2003:10:41:02     -0500]     "GET
/d/winnt/system32/cmd.exe?/c+dir HTTP/1.0" 404 297
9.131.0.90     -     -     [15/Jan/2003:10:41:03     -0500]     "GET
/scripts/..%255c../winnt/system32/cmd.exe?/c+dir HTTP/1.0" 404 311
9.131.0.90     -     -     [15/Jan/2003:10:41:03     -0500]     "GET
/_vti_bin/..%255c../..%255c../..%255c../winnt/system32/cmd.exe?/c+dir
HTTP/1.0" 404 328

......
```

Figure 2.2: A Fragment of an Access Log Generated by an HTTP Server

```
......

[Wed Jan 15 10:41:00 2003] [error] [client 9.131.0.90] File does not
exist: c:/apache group/apache/htdocs/scripts/root.exe
[Wed Jan 15 10:41:01 2003] [error] [client 9.131.0.90] File does not
exist: c:/apache group/apache/htdocs/msadc/root.exe
[Wed Jan 15 10:41:02 2003] [error] [client 9.131.0.90] File does not
exist: c:/apache group/apache/htdocs/c/winnt/system32/cmd.exe
[Wed Jan 15 10:41:02 2003] [error] [client 9.131.0.90] File does not
exist: c:/apache group/apache/htdocs/d/winnt/system32/cmd.exe
......
```

Figure 2.3: A Fragment of an Error Log File Generated by the Same Server

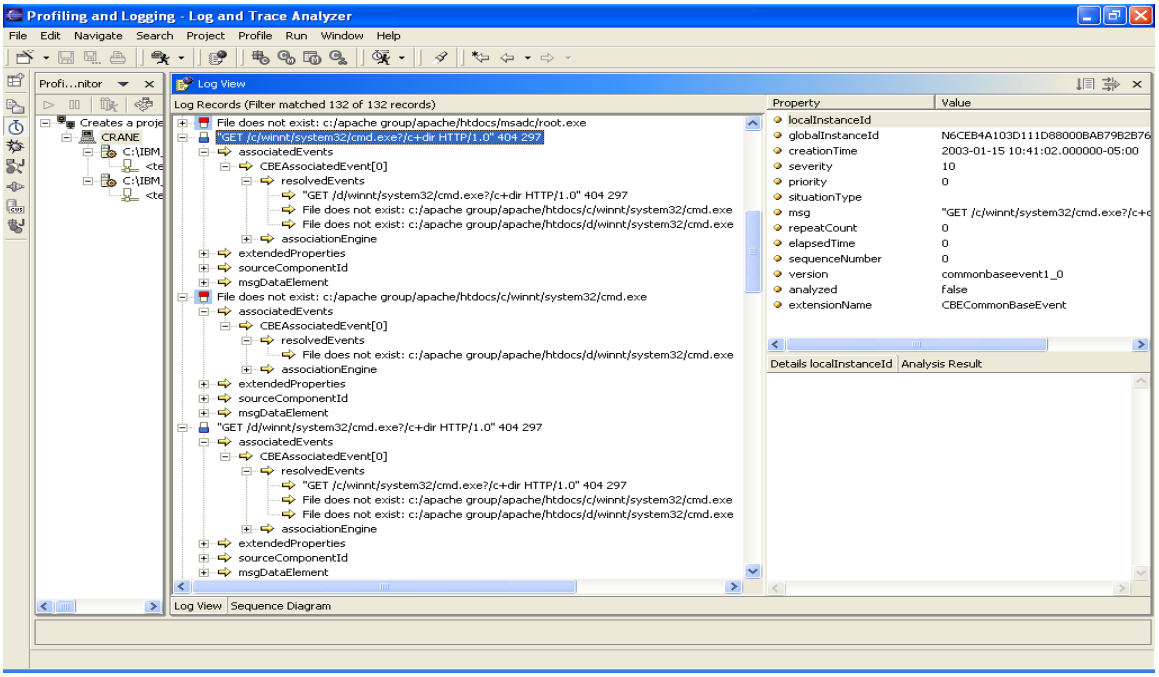

Figure 2.4: Log View

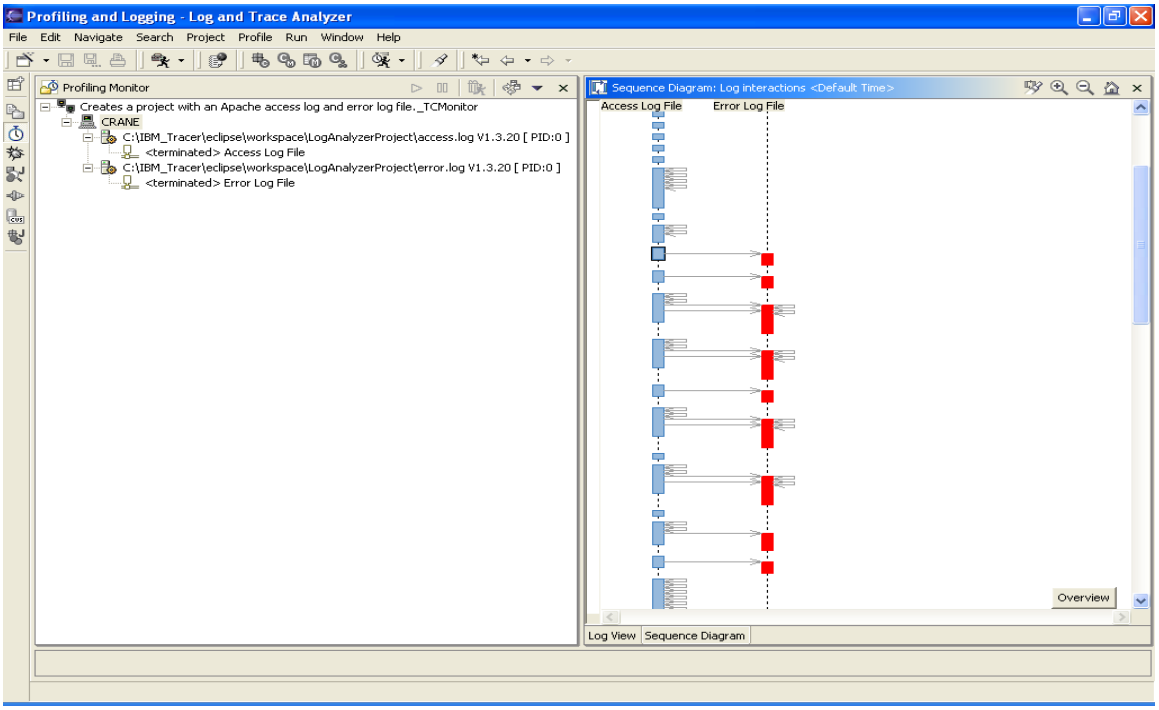

Figure 2.5: Correlation by Time

2.2 Message Tracking in Web Services

Sahai *et al*. propose a decentralized solution for message tracking in Web Services [SMO+02]. In their solution, a correlation method based on transactions is introduced. Although their correlation algorithm is specific to Web Services, the idea can be extended to a broader scope of targets.

2.2.1 Web Services and Messages

In the broad sense of the term, Web Services is a formatted message-based model for applications and web sites to be interoperable with each other. In more technical terms, it is Remote Procedure Call (RPC) where all involved parties agree on the exchange of standard-format messages, specifically using XML-based syntax. Web Services are enabled by a set of standards and technologies. They are: Simple Object Access Protocol (SOAP) [SOAP Website], Universal Description, Discovery, and Integration (UDDI) [UDDI Website], and Web Services Description Language (WSDL) [WSDL Website].

In Web Services, a complete computing service is often a composite one which comprises a set of remote invocations via SOAP across heterogeneous platforms. It is helpful for the developer to track the invocation path for a specific service.

2.2.2 SOAP Message Tracking

Sahai *et al.* use the idea of a transaction to represent a composite web service. Their concept of transaction is different from the one in database systems. Rather it is a portion of business logic with a clearly defined begin-point and end-point [SMO+02]. Their solution is to track the messages belonging to the same transaction. In contrast to LTA, event collection in this solution depends on the use of SOAP messages.

2.2.2.1 Message Data

In this solution, correlation data are collected during the interoperation of entities in various web services. Such interoperation is accomplished by extra data flowing through the entities. The extra data are in the form of a data structure called a Message Detail Record (MDR), which is shown below. The parent-MDR field represents the transaction relationship. Appendix A gives an example of a SOAP header containing an MDR [SMO+02].

```
MDR
{
    parent_mdr    :    message detail record of the parent message
    message_id    :    unique identifier of the message
    message_type  :    type of the message
    source        :    identifier of the service originating the message
    target        :    identifier of the service receiving the message
    time_sent     :    time when the message was sent by source
    time_recd     :    time when message was received by target
}
```

2.2.2.2 Message Tracking and Correlator Collection

In this solution, message tracking and correlation collection are fulfilled by building MDR trees in the header of the SOAP message. When a message is being sent, a new MDR is created and inserted into the appropriate child position of its context MDR. When a message is received, the MDR tree contained in the SOAP header is extracted and merged with the MDR-Forest stored by the receiver.

A tree structure of MDRs (for the exchanged SOAP messages shown in Figure 2.6) is shown in Figure 2.7. Each tree represents a web-services transaction. Each node in the tree represents a message in the tracking path. A child node means the message occurs in the context of its parent message. As we can see, message correlation is represented by the tree structure of the MDRs.

It is readily seen that the size of the tree grows with the length of the path of a transaction. Accordingly, the size of the header of a SOAP message is a variable determined by the length of a transaction. That is, the message-space complexity of this solution is *O(N)*, where *N* is the path length of the transaction.

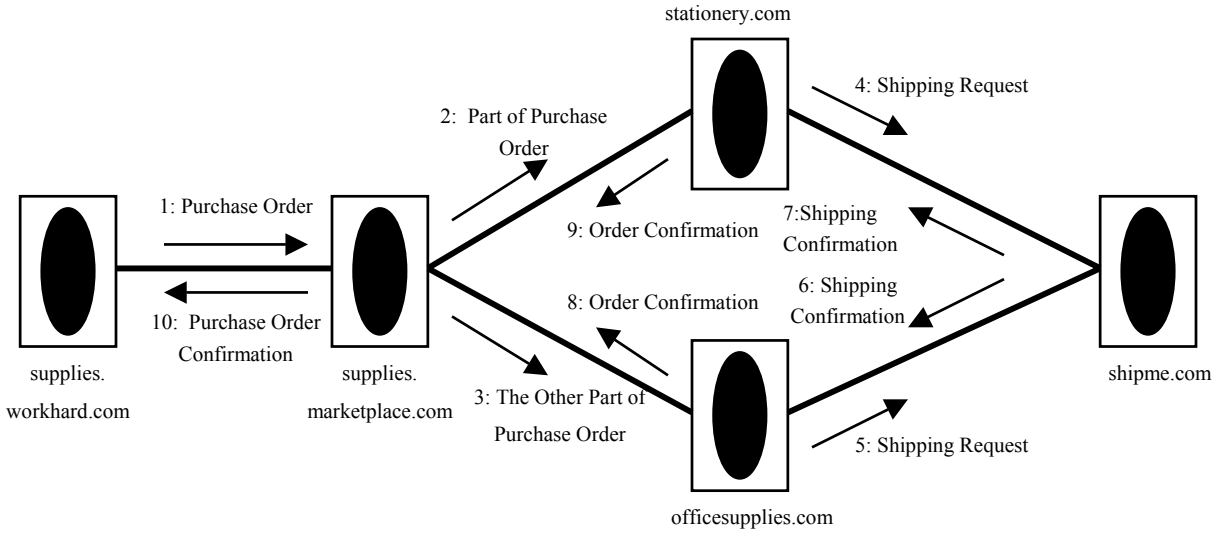

Figure 2.6: SOAP Messages Exchanged between Web Services [SMO+02]

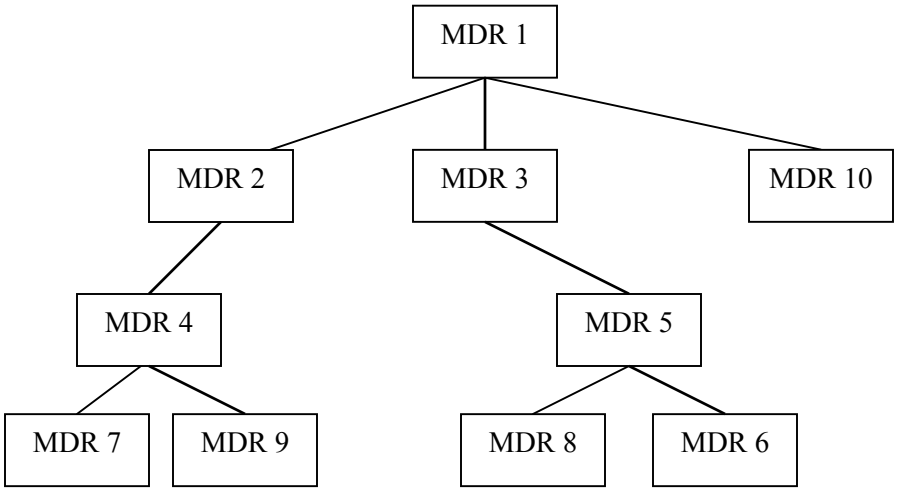

Figure 2.7: An MDR Tree Representing a Web Services Transaction [SMO+02]

2.2.3 Visualization

Each entity in the target system has its own view of the message tracking. For a specific transaction, only the initiator has a complete view of messages and their relationships. The initiator can display all of the messages and involved entities for the transaction. Figure 2.8 shows a complete visualization

view of the initiator of a transaction. Figure 2.9 shows an incomplete visualization view from the intermediate node officesupplies.com in the transaction.

2.2.4 Drawbacks and Limitations

This approach provides a deterministic correlation solution based on the "transaction" concept, but it has some limitations.

(1) It does not scale well with the growth of the number of messages per transaction. As we have discussed in the previous section, the size of the SOAP message containing the MDR tree increases through the transaction path. Its complexity is variable depending on the path of a transaction.

(2) The message tracking and collection is tightly bound to XML-formatted data. This reduces its flexibility, interoperability, and portability. It does not have good target independence.

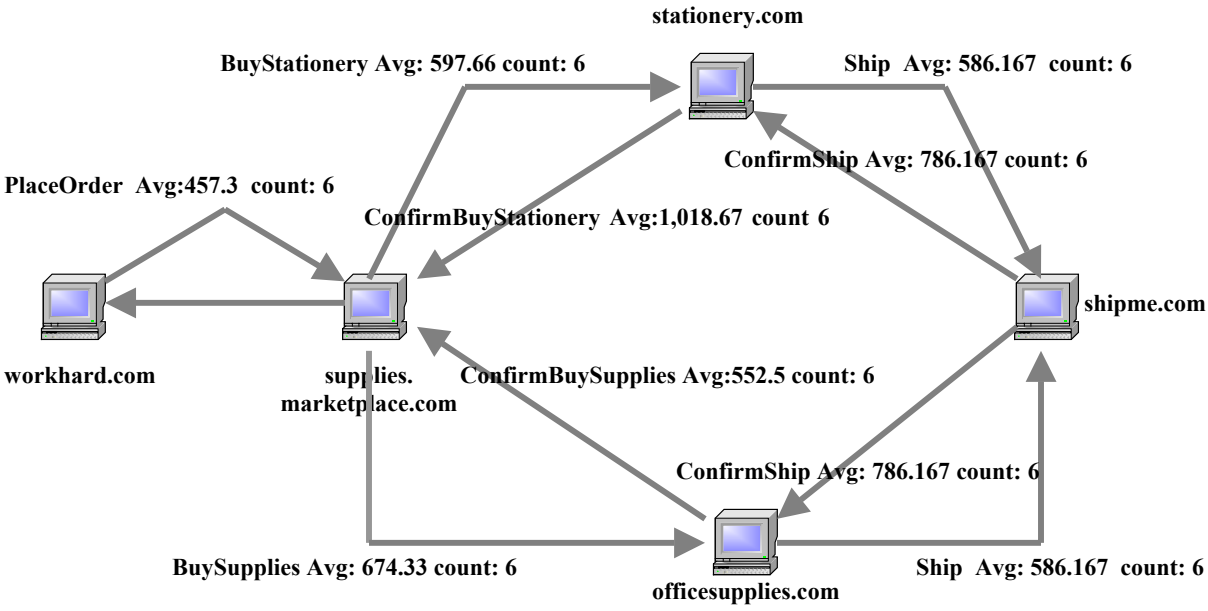

Figure 2.8: A Complete Visualization View from the Initiator [SMO+02]

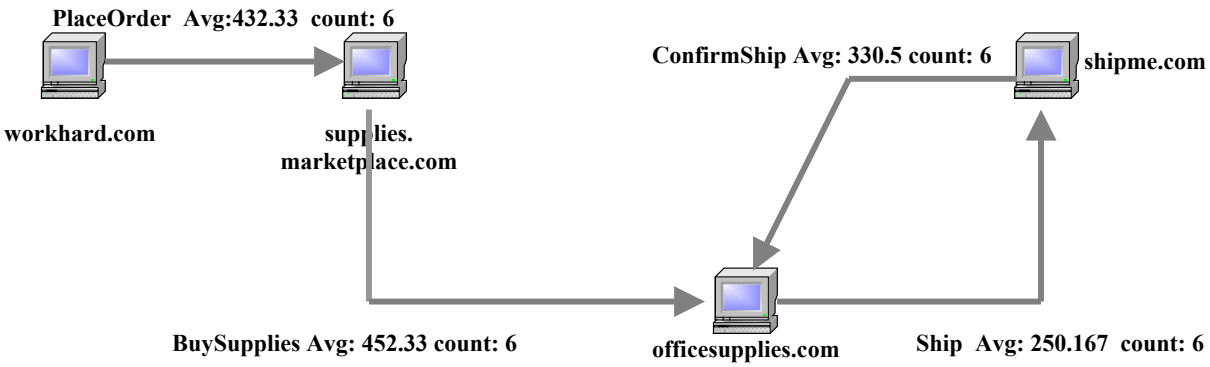

Figure 2.9: An Incomplete Visualization View from officesupplies.com [SMO+02]

2.3 POET

POET is a distributed debugging and monitoring system developed by the Shoshin lab at the University of Waterloo. POET can visualize the process-time diagram of various parallel and distributed systems, showing the partial order of events. POET is a useful tool for studying the behavior pattern of entities and the interaction patterns between entities (processes, *etc*.). It is helpful for identifying faults, anomalies, and performance problems. One of its advantages is that it is independent of any target system, and thus has been used for a variety of environments that include OSF/DCE [OSF93], Hermes [SBL+91], Concert/C [YGS+89], ABC++ [AOK+95], μC++ [BuS91], SR [And+88], PVM [GBD+94], TCP Sockets, Java, and itself (since it is implemented as a distributed system).

2.3.1 Event-based Model

The target-system independence of POET is enabled by the concepts of "event" and "trace". POET adopts the event-based approach. This approach is one of the techniques employed in formal modeling of distributed computation [War02], and was originally developed by Lamport [Lam78]. It focuses on the events which trigger state transitions rather than focusing on the state [War02].

2.3.1.1 Event

An event is a transition from one state to another. Events are "atomic," which means they take zero time to occur. The concept of event is independent of any concrete system. Events can be instantiated in a variety of concrete systems.

In monitoring systems adopting the event-based approach, the event types differ from target to target, and depend on what information the user wants to capture. For example, an RPC call can mean two events for two processes, one a send event, and the counterpart a receive event. However, an RPC may have several pairs of events between two processes if we want to capture message interactions at the TCP level.

From the point of view of the end user, the event is the unit that should visualized. However, the display of a collection of isolated events is far from enough for the user. The pattern of relationships between events is critical. The first obvious pattern is the order relationships. In distributed systems, the partial order is the ordering relationship of events. Lamport's *happened before* [Lam78] determines the partial-order relationship in distributed systems. It is denoted by "$\rightarrow$". The rules of *happened before* are

- If a and b stand for two events in the same process, and a occurs before b, then $a \rightarrow b$.
- If a is the sending point of a message and b is the receiving point of the same message by another process, then $a \rightarrow b$.
- If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.
- Events a and b are concurrent if and only if neither "$a \rightarrow b$" nor "$b \rightarrow a$" is true.

2.3.1.2 Event Collection

In a monitoring system, a critical requirement is to collect the event data generated by the target. Event data is collected by inserting small pieces of code that report to the monitoring system the necessary event information. Such instrumentation code is inserted into the operating system, run-time environment, communication library, or application code itself, as appropriate [See95]. Such instrumentation varies from system to system. The concrete instances of events are target dependent.

The event may be low level, such as local system call. It may be a higher-level one, such as an HTTP request or a SOAP action.

A problem of instrumentation is the probe effect. Instrumentation may perturb the ordering of events in a program execution so that the collection of information can actually affect the information being collected [See95]. POET minimizes this effect by collecting a minimal amount of information.

2.3.2 Architecture of POET

POET has a client/server architecture. The run-time architecture of POET is shown in Figure 2.10. For a simple configuration, POET consists of an event-server process and two client processes: the debug-session process and the checkpoint process.

The event server (also called disk server) interacts with both the monitored targets and various clients. It is responsible for receiving, processing, and storing event data from the target application, and for sending event data, on request, to its clients.

The debug-session process is the visualization part of POET. It is responsible for direct interaction with the end user [KBT+97]. It can reside remotely as well as on the same machine as the event server, which depends on the configuration of POET. Its major functionality is to obtain end-user input via the keyboard and mouse and produce an appropriate display in response. This process contains the algorithms for the debugger display, such as display scrolling, clustering, and event abstraction.

The checkpoint process is an optimization to improve system performance.

The target programs are the monitored processes that have instrumentation to interact with the event server. The instrumentation is responsible for generating event data and sending them to the event server.

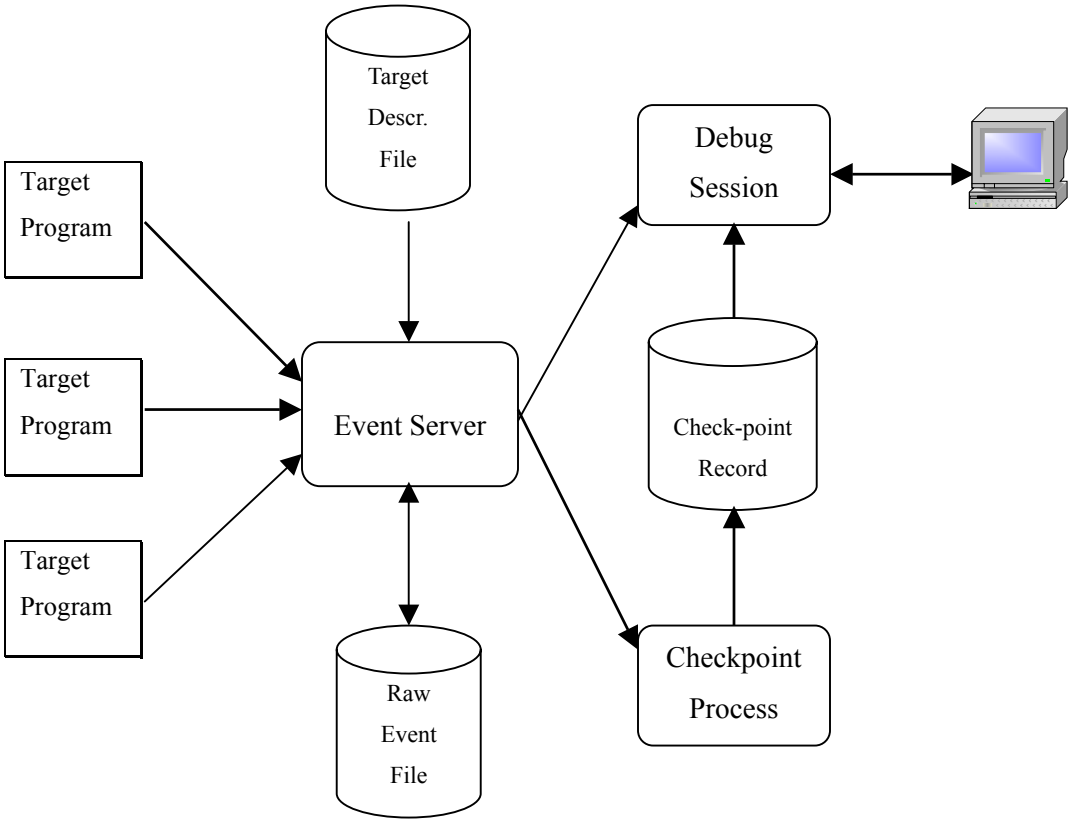

Figure 2.10: The Architecture of POET

2.3.3 Event Collection

The instrumentation inside targets sends raw event data to the POET event server over TCP/IP streams using the POET Event-Stream Protocol, as shown in Figure 2.11.

There are two types of events, normal events and text events, which are sent over this stream. A normal event contains information about the event, as well as information about its partner event, if it has one and if that information is known. The information in the event includes event type, local-trace identification, event count (*i.e.*, the event's position, starting from 0, on a trace), and real-time data. The information of the partner event includes stream identification, trace identification, and event count. A text event includes the text information of the immediately preceding normal event in the same stream.

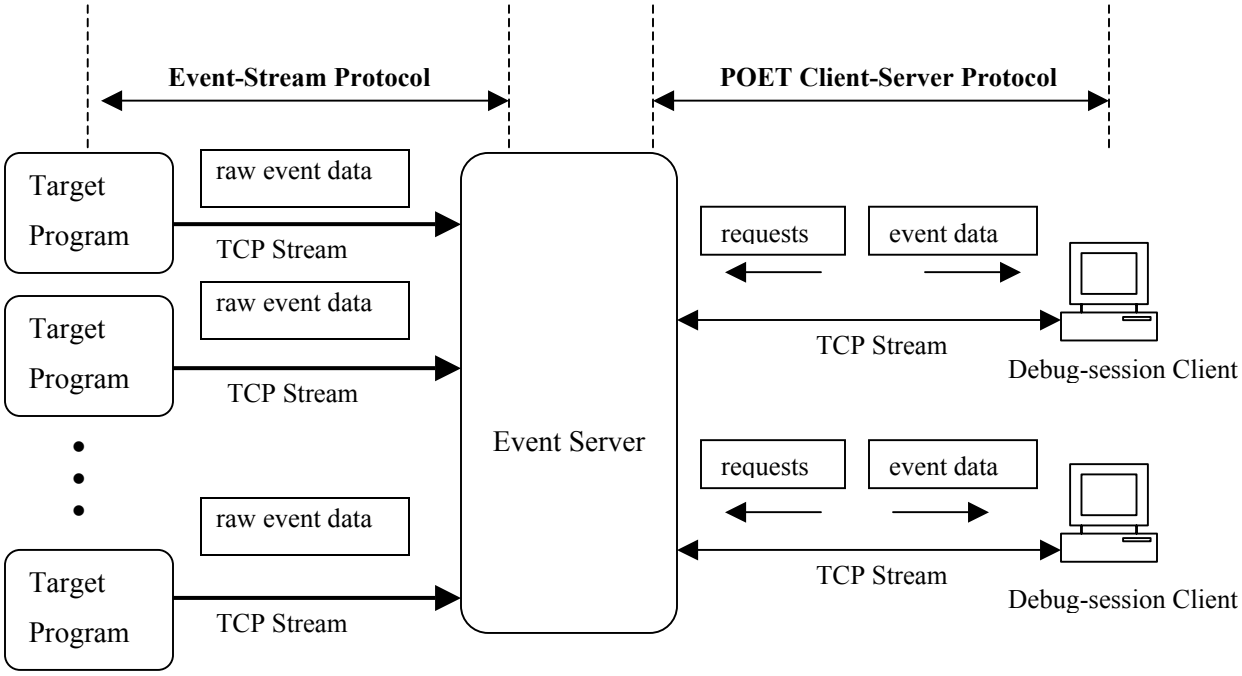

Figure 2.11: Event Streams in POET

Send-receive pairing and synchronous-event pairing are important relationships in POET. For instrumentation, the send-receive relationship of events is captured in the following way: At the time a send event occurs, the outgoing message will have data appended to it such as the stream identifier, trace identifier, and event count. When the corresponding receive event occurs, target-system instrumentation obtains these data from the incoming message. For synchronous events, the same operations are performed as described above.

2.3.3.1 UEF-Formatted File

The event server may persistently store the event data in a UEF-formatted file. A UEF-formatted file is a sequential ASCII file that is independent of different versions of POET and is platform-independent [Tay03]. A UEF-formatted file is composed of four major sections: general header, stream data, trace data, and event data. The user can reload event data stored in the UEF-formatted file. In such a case, the reload program retrieves the event data stored in the UEF-formatted file, parses the event data, and sends them to the event server through the Event-Stream Protocol. To send the event data, the reload program sets up streams to the event server as used by the original

execution. Thus from the view of the event server, the reload program operation is indistinguishable from the original execution.

2.3.3.2 Target-System Independence

POET provides target-system independence by means of a target-description file and an initial pseudo-event.

The target-description file contains relevant characteristics of a specific target environment. It is composed of a set of keywords and their values for the target and an event-description table that describes the events in detail for a target environment. The keywords include the target identifier, event-window title, and program-window title. The event-description table contains a sequence of entries each of which provides the relevant information for an event type of a specific target. The information for an event type includes index, partner-event type, and visualization characteristics [KBT+97]. New keywords and values can be added to this file if new characteristics are needed to describe a target. POET reads the target-description file and obtains the corresponding values at the time the target-environment application sends an initial pseudo-event to it, indicating the target type.

Before the target program starts to send any normal event data to the event server, it first sends a special event record, called event zero, to inform the event server of a new stream of event data. This pseudo-event contains the target identification, event parameters of the stream, *etc*. This information is used by the event server to process the event data over this stream properly. The data structure of the initial pseudo-event is as shown below:

```
typedef struct {
   int        magic_int;     /* A constant integer to indicate the byte order;*/
   char       magic_str[4];  /* A constant string to determine character code*/
   int        target_id;     /* The target identifier */
   int        stream_len;    /* The length of a stream identifier */
   int        trace_len;     /* The length of a trace identifier */
   int        text_len;      /* The length of a text string */
   unsigned   flags;         /* A flag field */
   char       stream_id[1];  /* The stream identifier, the length is
                                specified by stream_len */
} EVENT_ZERO;
```

2.3.3.3 Event Collection APIs

POET provides a set of APIs for the instrumentation to facilitate event collection. There are three commonly used API functions: *DBG_collect*, *DBG_both_collect*, and *DBG_text_collect*.

The function *DBG_collect* is used to create and transmit a single normal event without text data. Its interface is as below:

```
void DBG_collect(unsigned e_type,     /* Event type of the generated event*/
                 void*    e_trace,    /* Trace identifier of the generated
                                          event*/
                 int      e_evcnt,    /* Event count of the generated
                                          event*/
                 void*    p_stream,   /* The stream identifier of partner
                                          event*/
                 void*    p_trace,    /* The trace identifier of partner event*/
                 int      p_evcnt     /* The event count of partner event*/
                 )
```

Function *DBG_text_collect* is used to collect only text events and it has the following interface:

```
void DBG_text_collect(unsigned e_type,          /* Event type */
                      char* e_name              /* Text string */
                      )
```

Another function, *DBG_both_collect*, is used to collect event data and associated text data. It has the following interface:

```
void DBG_both_collect (unsigned  e_type,       /* Event type for normal event*/
                       void*     e_trace,      /* Trace identifier of the
                                                  generated event*/
                       int       e_evcnt,      /* Event count of the generated
                                                  event*/
                       void*     p_stream,     /* Stream identifier of the
                                                  generated event*/
                       void*     p_trace,      /* Trace identifier of partner
                                                  event*/
                       int       p_evcnt,      /* Event count of partner event*/
                       unsigned  text_e_type,  /* Event type for text event*/
                       char*     e_name        /* Text string*/
                       )
```

A call to the *DBG_both_collect* function is equivalent to a call to *DBG_collect* followed by a call to *DBG_text_collect*. The reason for using *DBG_both_collect* is to avoid interference between these two calls from a different thread in a multi-threaded environment. Specifically, the text event must immediately follow the normal event for which it provides text data, or it will either be lost, or (worse) attach its text data to the wrong event.

2.3.4 Visualization and Analysis

The POET visualization layout is composed of a number of horizontal lines, called traces, different types of symbols on the lines, arrowed lines connecting the symbols, *etc*. A horizontal line represents

a sequential entity (process, thread, *etc*.). A symbol on the line denotes an event belonging to the entity represented by that line. The symbol shapes used to represent events in a target environment are defined in the corresponding target-description file. An arrowed line connecting two symbols shows the interaction (communication) between them. Two types of arrowed line are used to represent synchronous and asynchronous communication. Figure 2.12 shows these two types of communication. For synchronous communication, two events are connected by a vertical arrowed line. For asynchronous communication, two events are connected by a sloping arrowed line.

POET also provides the functionality of displaying detailed information for an event and the partial order of events. By positioning the cursor on an event and clicking the middle mouse button, the user can see a small display field appearing beside the event. That field shows such information as the type of the event, the name of the trace the event is on, the sequence number of the event within that trace, and the text string, if it exists. In addition, the events that are predecessors of this event and the ones that are successors will be colored differently. By default, all the predecessors are colored red and all the successors are colored green. Other events (including the selected event) remain uncolored.

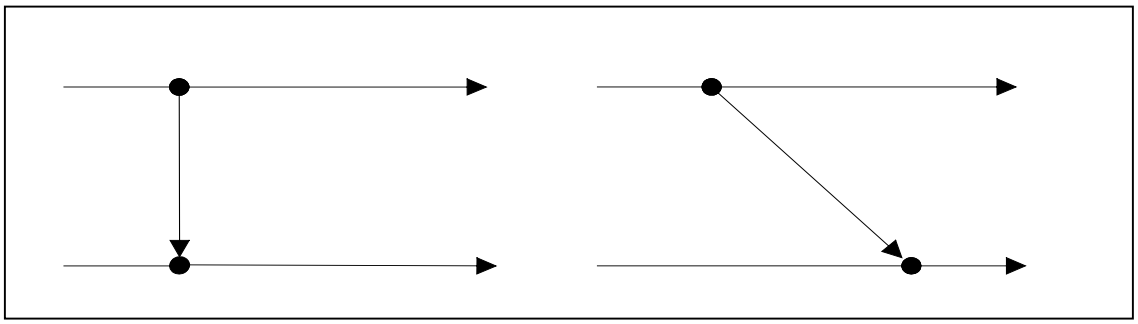

Figure 2.12: Synchronous and Asynchronous Communication

2.3.5 Correlation in POET

Various correlation mechanisms exist in POET. In particular, it correlates events in traces, and send-receive and synchronous-event pairs, as well as allowing abstraction, real-time correlation, and predicate detection.

A trace is a horizontal line in the visualization. In different target environments, it may represent different entities. It can be a process, a thread, a mutex, or any sequential entity. For an event, this trace information is a form of correlation. All of the events with the same trace identifier will be visualized on the line representing the trace.

Send-receive pairing and synchronous-event pairing are self-evident forms of correlation. They identify the send-receive and synchronous-event relationships between events, respectively. Such correlations are helpful to identify the interacting pairs of events in communication environments.

Abstraction is an important technique that reduces display complexity by skipping undesired visualization detail. In POET, there are two types of abstraction: event abstraction and trace abstraction. Event abstraction is the process of grouping multiple events into a single abstract event based on certain rules. Similarly, trace abstraction is a technique that groups a set of traces into a cluster. However, these abstractions have some restrictions. For event abstraction, the event set to be abstracted must satisfy the convexity constraint. This constraint states that there is no event outside the convex set that happens before some event in the set while some other event in the set happens before it. Convex abstract events keep the atomicity property of primitive events. However, this is obtained at the expense of plausible abstract events. Figure 2.13 illustrates a plausible, but non-convex, abstract event. The events enclosed by the dashed curve may belong to a correlated set of events. However, they cannot be grouped into an abstract event because the set does not satisfy the convexity constraint. The limitation of trace abstraction is that it cannot correlate events across parts of different traces.

Predicate detection is a search mechanism that finds the event set matching predefined constraints (predicates), especially those specifying causality relations [Xie04]. Hierarchical predicate detection requires automated event abstraction, typically requiring the event set to be convex.

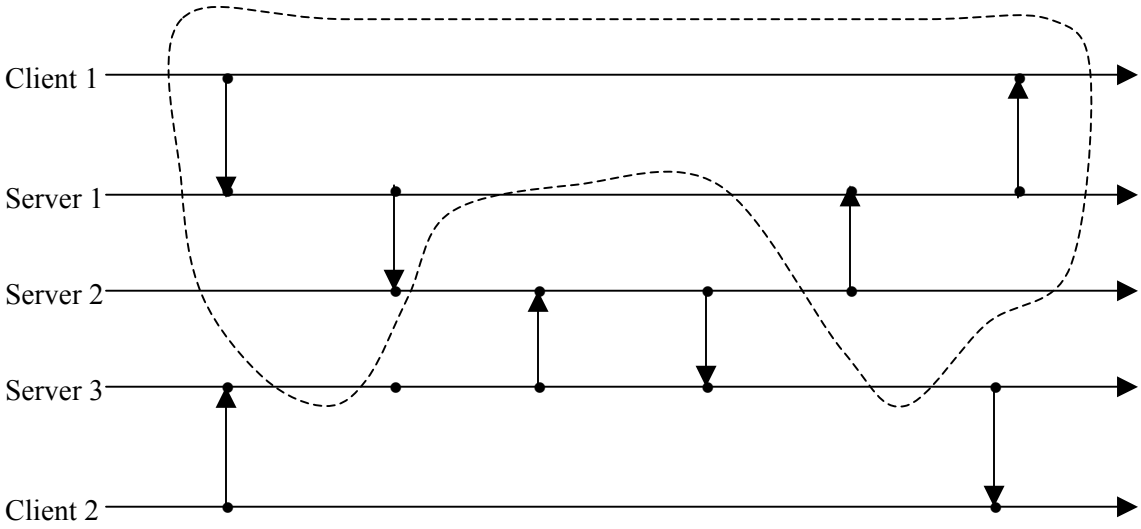

Figure 2.13: A Non-Convex Event Set

Chapter 3
Event Correlation by Task

In this chapter, we will explain the task concept and describe our correlation solution within POET. We introduce the general concepts of event correlation in Section 3.1, including correlator, domain, and categorization. In Section 3.2, we propose our correlator, "task," and give examples of mapping from it to some concrete correlators. We compare event correlation and abstraction in Section 3.3. We describe our correlation solution in detail in Section 3.4.

3.1 Event Correlation

A correlator is a function that maps events into sets. As such it must have a well-defined domain. For example, the correlator "URL" can be used in the HTTP domain. A domain may be concrete or abstract. An abstract domain is generic and can be mapped to any concrete domain. For example, in POET, the trace is a generic correlator existing in an abstract domain that can be mapped to different concrete domains, such as process, socket, or object.

In this thesis, we adopt the categorization criteria of correlation in LTA. The correlation is classified into two types: sequential and associative.

(1) Sequential correlation orders a set of events by using a specific correlator and/or rules to put them in some sequence according to the order of correlator values. The obvious example is to order events by real-time timestamp.

(2) Associative correlation clusters events by using some correlator (or correlators) and/or rules. An example is the "trace" in POET. A trace is a group that associates all events in the same sequential entity. Correlating events based on URL or application ID in LTA are other examples of such correlation.

Two factors affect the efficiency and effectiveness of a correlator. These factors are the degree of independence from the target and the cost of collection.

The degree of independence determines the adaptability of the correlation. For example, "trace" in POET is independent of any target. It can therefore be mapped to various entities (process, thread, object, TCP socket, *etc*.). By contrast, "transaction" is bound to Web Services in Sahai's system, which narrows its application for other targets. The characteristics of the correlator domain determine the degree of independence. An abstract domain enables a correlator to have a high degree of independence.

The efficiency of correlator collection determines the efficiency of correlation. For example, the correlator collection in the approach of Sahai *et al*. is not efficient.

3.2 Task Concept

An important concept in many aspects of distributed systems is "task." A task is a set of operations or actions that fulfill a specific computing purpose. It is an abstract concept that is meaningful for different distributed systems, including Web Services, distributed databases, RMI/RPC, CORBA, shared-memory systems, and parallel computing systems. In these distributed and parallel systems the computing entities may interoperate with each other to fulfill some specific computing purpose. We use "task" to refer to that purpose.

There are various instances for this concept of task. We give some examples to explain it in detail.

In web browsing, an instance of task might be viewing a web page. Such a task can be defined as the procedure of getting all of the objects (text, image, Java script, *etc*.) to display a complete web page. Thus, one web page display may contain multiple HTTP requests and responses. Figure 3.1 shows a task in web browsing. In Figure 3.1, four actions occur to complete browsing a web page (*i.e.*, web-page A).

In Web Services, the task concept can be mapped to a business transaction. In such a context, a task refers to a set of invocations based on SOAP messages to fulfill a business-computing service. For example, a user purchases an item in an online store, called E-Store.com, as shown in Figure 3.2. This

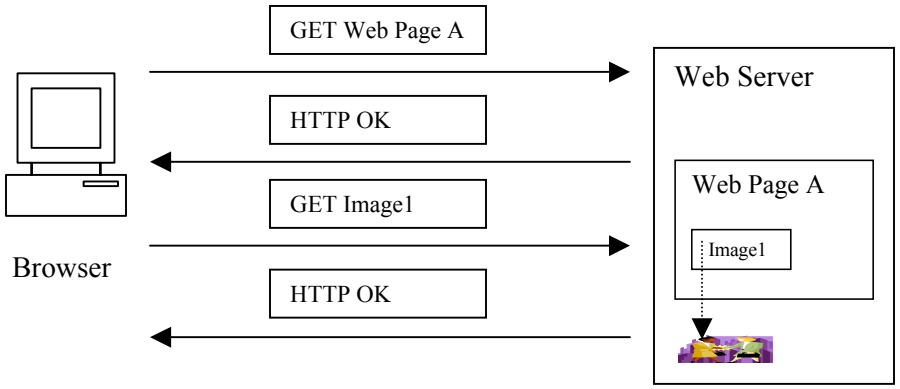

Figure 3.1: A Task in Web Browsing

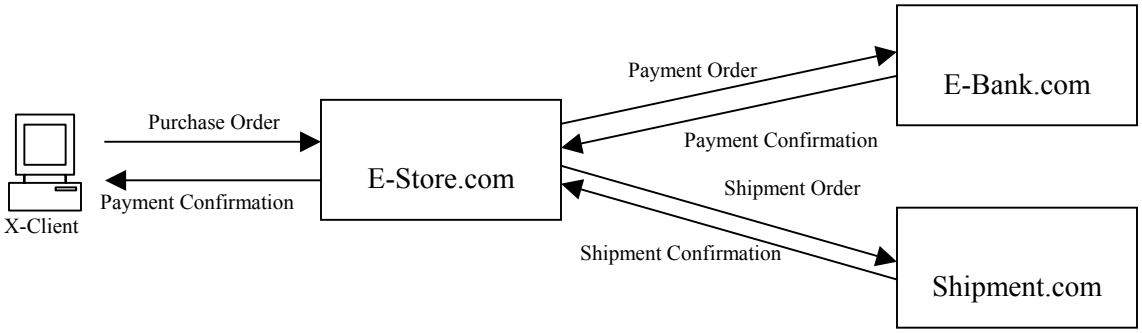

Figure 3.2: A Business Transaction in Web Services

service transaction comprises the subsequent set of SOAP-based invocations: the store sends an order to the corresponding online bank, E-Bank.com, which checks the user's bank information and gets a payment transfer from the user's account. The e-store then orders a shipping company, Shipment.com, to do the shipment. In this transaction, a set of messages is transmitted and a number of events occur in different processes. These events and messages may occur concurrently with other messages and events from other transactions. Without correlation, it is hard to identify the messages and events corresponding to this specific service for this specific user. Correlating these events and messages associated with this service can help the developer find the path of the invocation and identify any defects or bottlenecks in the whole procedure.

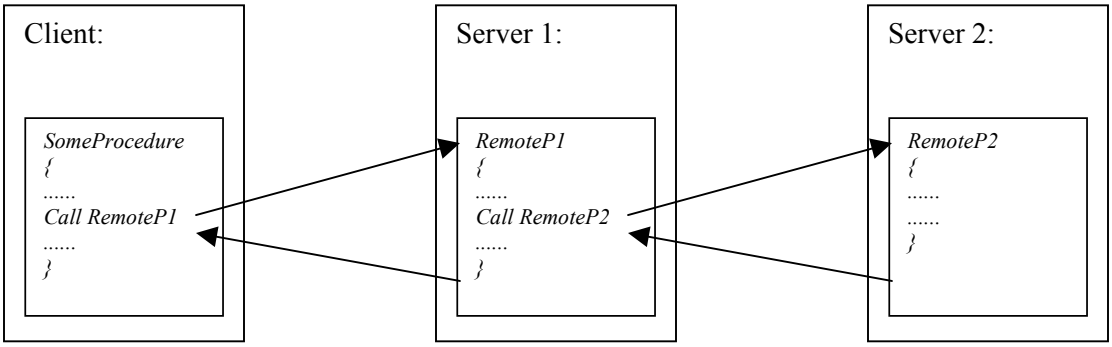

Figure 3.3: A Chain of RPC/RMI

In distributed database systems, the task concept can be mapped to an ACID transaction. A simple transaction is usually issued to the database system in SQL in this form:

> ***Begin*** *the transaction*
>
> ***Execute*** *a sequence of SQL actions (select, insert, update, delete, etc.)*
>
> ***Commit*** *the transaction.*

In RPC/RMI there may exist chains of calls or invocations. That is, a called remote procedure may call other procedures, as shown in Figure 3.3. We refer to such chains as a "composite RPC" or a "composite RMI" as appropriate. Such calls or invocations fulfill a specific computing task. Thus our task concept can be mapped to the chain of calls or invocations in RPC/RMI. Such a correlation is very helpful for the developer of RPC/RMI-based programs. It can identify all of the events involved in a RPC call or RMI invocation. Consequently it helps a developer determine whether there is a bottleneck or other problems in the complete operation of the composite RPC call or RMI invocation.

Based on the previous analysis, task is a generic and abstract concept of correlation that can be mapped to specific operations in different systems. It is independent of any target system. This characteristic makes it applicable to various target systems and it more closely reflects the natural operation of distributed systems than does the simple collection of raw events.

3.3 Nested Tasks

It can be useful to consider that there are sub-tasks or child tasks, occurring in the context of a parent task. Nested tasks are then needed to represent the relationship between task instances.

An example of nested tasks is a composite session in web browsing (*e.g.*, purchasing items on-line). Such a browsing session may comprise multiple web pages. Figure 3.4 shows a nested web-browsing session. The task is the complete session that comprises multiple web-page displays, while the sub-tasks are the display of the web pages in this session.

Another example use of nested tasks is nested transactions in a database, as shown in Figure 3.5. In this example an outer transaction contains an inner sub-transaction. The outer transaction might be viewed as a task, and the inner sub-transaction, its sub-task.

The nesting relationship between tasks can be represented using a tree structure. For each task record, a field indicates its parent task. Thus, a tree comprising the parent-child relationship can be built to represent nested tasks, as shown in Figure 3.6. This approach allows us to maintain a fixed overhead when collecting task data. Nested tasks are not investigated further in this thesis.

3.4 Event Correlation by Task

We design a correlation solution based on the "task" concept on top of POET. We choose POET as our base system because our correlation solution needs a target-independent platform. To enable our correlation to function within POET, the target, the event server, and the debug-session client must interoperate in regard to correlator data. The "task identifier," uniquely identifying any given task in a monitored environment, is the correlator data in our solution. Targets need to generate task identifiers and propagate them to each other in addition to sending them to the POET event server. The POET event server needs to process task identifiers from the target and send them to the debug-session client on request. The debug-session client needs to have a visualization method to display those task data received from the event server.

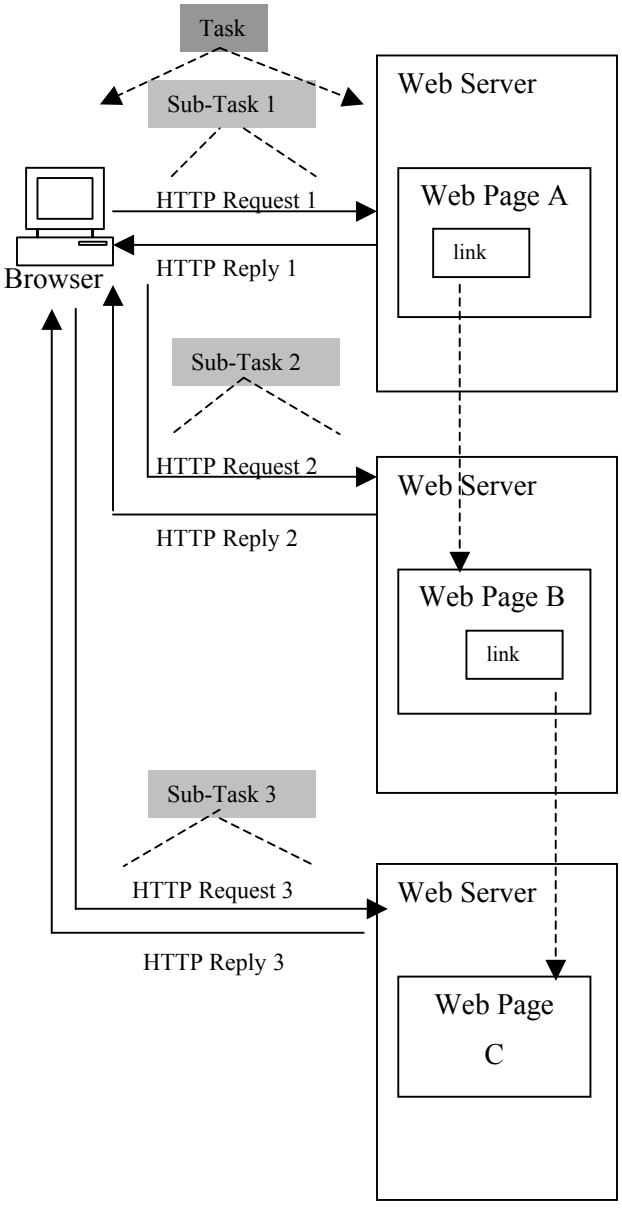

Figure 3.4: A Nested Session in Web Browsing

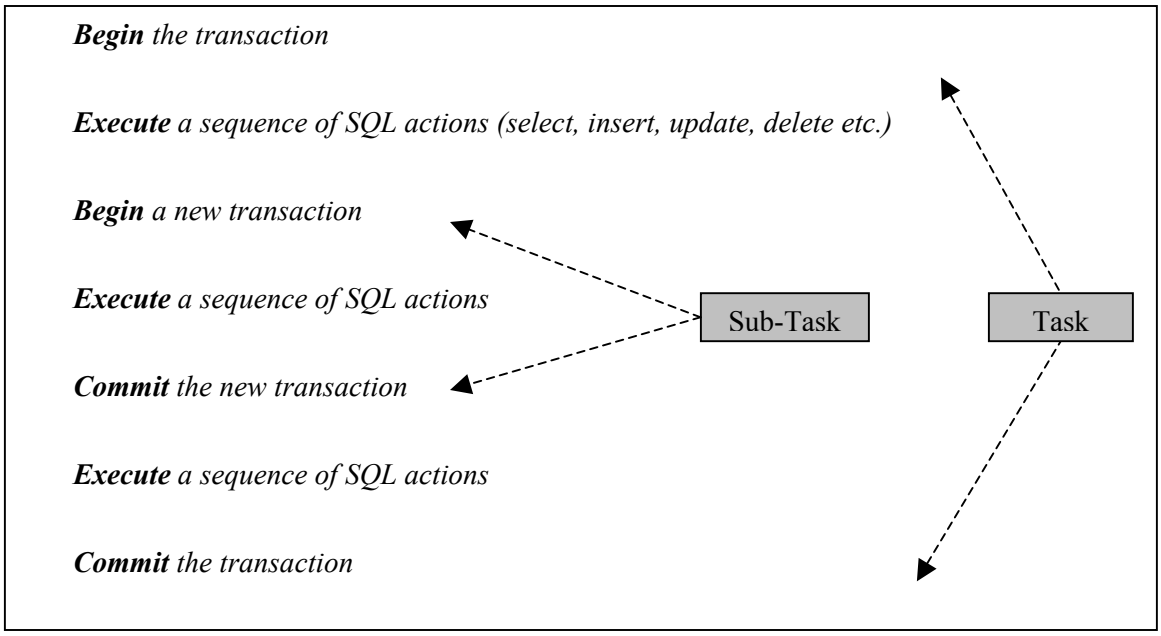

Figure 3.5: A Nested Transaction in a Database

3.4.1 Basic Consideration for Instrumentation

While the instrumentation will vary for different target environments, it should follow some generic requirements. In this section, we describe the basic requirements for multi-process and multi-thread environments communicating by message passing. In such environments, a process may handle multiple tasks concurrently. The general requirements for instrumentation then include the following.

(1) The instrumentation must clearly define its task concept. That is, the instrumentation should map the "task" concept to the desired concrete instance (ACID transaction, composite RPC/RMI, *etc.*).

(2) The task identifier needs to be globally unique across all threads and processes. Task identifiers may need to propagate across multiple processes and/or threads. To prevent conflict between task identifiers, it is necessary to keep the uniqueness of task identifier. A Universal Unique IDentifier (UUID) [OSF93] or Globally Unique IDentifier (GUID) [EdE98] may be used in some targets. While a UUID or GUID (a 128-bit number) is typically enough to guarantee the uniqueness, we do not presuppose that the correlator is always a 128-bit number.

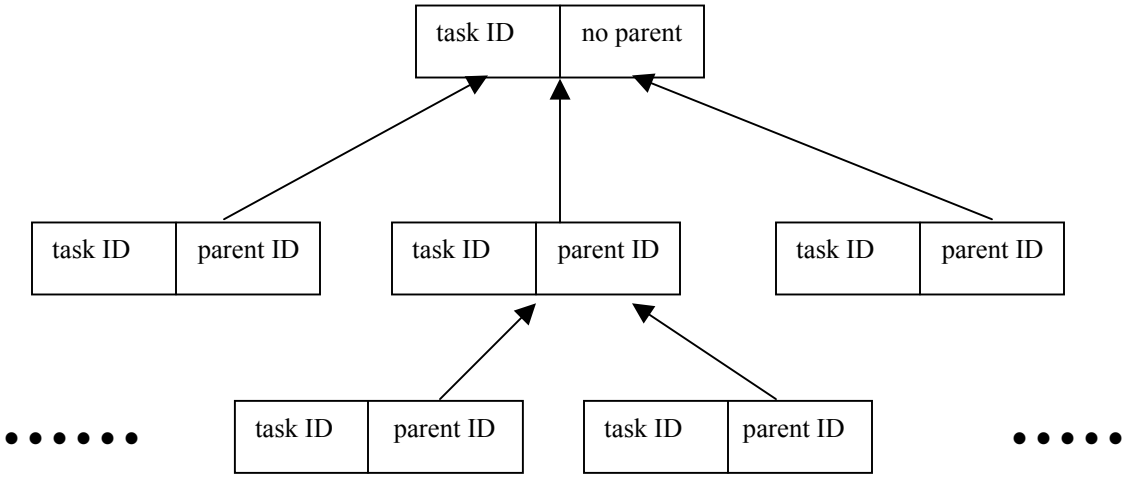

Figure 3.6: Tree Structure Representing Nested Tasks

(3) In addition to being unique, the identifier must have the same length across various target systems if tasks are to be correlated across those systems. Thus, POET and the target-system instrumentation must agree on the length of task identifiers.

(4) The task identifier needs to propagate across multiple processes that may be part of different target environments. The instrumented environment may be of multiple processes that may be part of different target environments. For example, enterprise-level web application systems have a multi-tier architecture. To capture the task data in such an environment, the task identifier should be able to be propagated across the heterogeneous platforms (web server, application server, database server, *etc.*). Currently, POET cannot handle multiple targets simultaneously.

3.4.2 Propagation of Task Identifier

The task identifier needs to be propagated to any event of the task. Consider the example of a composite RMI in Java, as discussed in Section 3.2. Such a composite RMI involves a number of JVM processes, as shown in Figure 3.7. The task identifier of this composite RMI needs to propagate among these JVM processes.

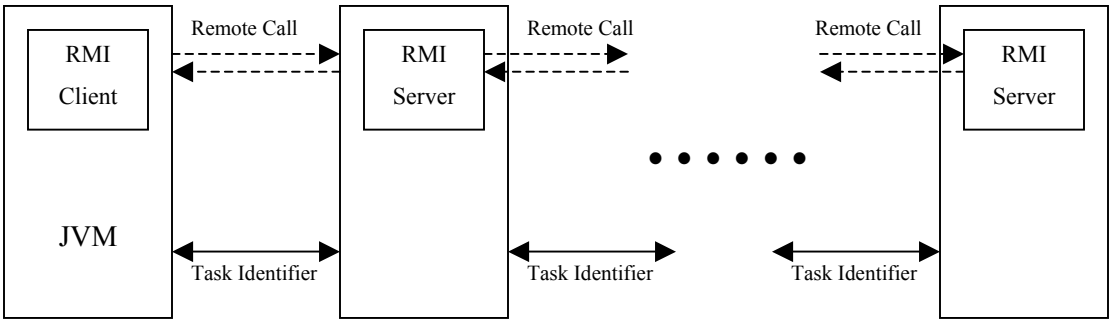

Figure 3.7: Task Identifier Propagation amongst JVM Processes

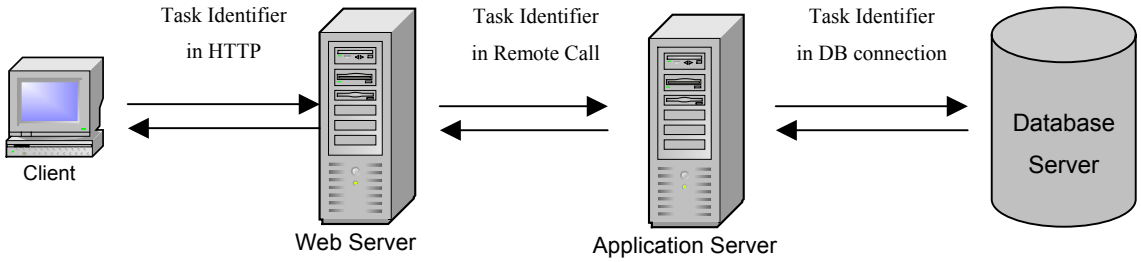

Figure 3.8: Task Flow across Multi-tier Web System

A more-complex example is a multi-tier web-application system. A task may include a chain of events occurring in a web server, application server, and database server, as shown in Figure 3.8. In this example the situation is more complex. The task identifier may be transferred by different transport mechanisms in different layers.

The instrumentation has to associate each event with its appropriate task, and must consider two aspects, inter-process communication (*i.e.*, message passing) and concurrency inside a process. The reason is that events occur in the context of both inter-process communication and concurrency. The combination of concurrency and multiple communication channels makes it difficult to associate events occurring in such an environment with appropriate tasks. To solve this problem, we introduce the "task context" concept, which stands for the current task context under which an event occurs. Figure 3.9 shows the structure of a task context.

Task Identifier	Identifier of a Computing Entity (*e.g.*, Thread)	• • • • • • •

Figure 3.9: The Structure of a Task Context

Since there may be multiple tasks simultaneously active in a target process, the instrumentation must properly switch task contexts for an expected event. This can be fulfilled by attaching a task tag, showing the task context for each thread, to each thread.

In Section 2.3.3, we introduced the method which our instrumentation uses to capture the send-receive relationship. For POET, instrumentation should solve the problem of passing the task identifier through send-receive pairing. Our algorithm is applied to four types of events: transmit, receive, unary, and synchronous. A unary event occurs without any message passing. A transmit event is a sending of a message. A receive event is a receiving of a message. A synchronous event is a pair of sending and receiving of a message in the form of synchronous communication.

In our algorithm, we use 0 as a special value to indicate "no task." Our algorithm is as follows.

(1) When a unary event occurs, the instrumentation simply obtains its task context (*i.e.*, task tag) by referencing the identifier of the thread where this event occurs.

(2) When a transmit event occurs, the instrumentation will check the task status. If the thread is dedicated to a task, the instrumentation obtains the task identifier from the task context (*i.e.*, task tag). If the thread is an initiator of a new task, the instrumentation generates a new task context with a new task identifier for this thread. The transmit event corresponds to a transmission operation. The instrumentation appends the task identifier as well as the stream identifier, the trace identifier, and the event count in the trace (the original instrumentation in POET) to the end of the outgoing message.

(3) A receive event occurs when a message arrives. Instrumentation determines whether it must initiate a new task or accept the received task identifier. In the first case, the instrumentation will take the same action as (2); otherwise, it extracts the task identifier from the message. If the task identifier

is 0, the instrumentation determines that no task data is associated with this message and processes this event in the manner of one without any task identifier. If the incoming task identifier is not 0, the instrumentation will set the task tag of the thread to the incoming task identifier and save the old one in the case that the incoming task identifier is different from the current one.

(4) When a synchronous event occurs, the operations the instrumentation performs are similar to those on a pair of transmit and receive events except that the receive end simply accepts the received task identifier since it is presumed that the two end points should have a close task relationship based on the consideration that it is not a pair of events but a single event.

3.4.3 Collection of Task Identifier

As discussed in Section 2.3.3, the instrumentation collects event data through the POET Event-Stream Protocol. We modify this protocol in our solution by adding a new field to the normal event data structure. This field contains the task identifier that uniquely identifies the task associated with that event. The type of this field is a variable of type void to be adapted to various data types of targets. In our solution, we do not specify the length of task data. Our APIs can be adapted to task identifiers of arbitrary length. The agreement on the length of task identifier between target and event server will be discussed later.

Considered together with the discussion in the previous section, the algorithm of task-identifier collection is as follows. When an event occurs,

(1) If the instrumentation determines the event is associated with an existing task or that a new task needs to be created, it puts that task identifier in the event stream and sends it to event server.

(2) If the instrumentation determines that this event does not belong to any task, it puts the special value (*i.e.*, 0) in the event stream to indicate no task data.

To facilitate the collection of task data, we modified the original *DBG_collect* and *DBG_both_collect* interfaces of POET that were introduced in Section 2.3.3.3. The new functions have very similar interfaces that can be easily used by the original users.

The first library function is used to collect a normal event with task data but without text information. Its modified interface is shown below. The only change we made is to add an argument *task_ID*, of type *void**, to the original interface. The target-system instrumenter can use this function to collect an event associated with a task identifier. If an event has no task identifier, the special value will be collected (*i.e.*, the task identifier is set to 0, or in other words a pointer to 0 is passed to the function.). This function can also be used by the original instrumenter to collect an event without a task identifier since the argument, *task_ID*, can be compiled optionally by the preprocessor. The length of the task identifier is defined in the header file, *usr_debug.h*, which is included by the file in which these functions are located.

```
void DBG_collect(unsigned e_type,   /* Event type of the generated
                                       event*/
                 void*    e_trace,  /* Trace Identifier of the
                                       generated event*/
                 int      e_evcnt,  /* Event count of the
                                       generated event*/
                 void*    p_stream, /* The stream identifier of
                                       partner event*/
                 void*    p_trace,  /* The trace identifier of
                                       partner event*/
                 int      p_evcnt   /* The event count of partner
                                       event*/
#ifdef TASK
,                void*    task_ID   /* The task identifier of the
                                       generated event*/
#endif
                 )
```

The second library function is used to collect a complete event with text information and task data. The modified interface is shown below:

```
void DBG_both_collect(unsigned   e_type,       /* Event type for normal event*/
                      void*      e_trace,      /* Trace identifier of the
                                                  generated event*/
                      int        e_evcnt,      /* Event count of the generated
                                                  event*/
                      void*      p_stream,     /* Stream identifier of the
                                                  generated event*/
                      void*      p_trace,      /* Trace identifier of partner
                                                  event*/
                      int        p_evcnt,      /* Event count of partner event*/
#ifdef TASK
                      void*      task_ID,      /* The task identifier of the
                                                  generated event */
#endif
                      unsigned   text_e_type,  /* Event type for text event*/
                      char*      e_name        /* Text string*/
                      )
```

3.4.4 Task Identifier Mapping

Inside the event server, we implement an optimization for processing task data, which transforms task identifiers from their target length to a shorter internal identifier. There are two reasons for this mapping. A task identifier from the target is long (*e.g.*, 128 bits). The display of such an identifier in the debug-session client is neither necessary nor desirable for the user. The other reason is that using such an identifier causes unnecessary time and space cost for the event server and the debug-session client.

In our solution, we use a mapping table for each POET session to transform target task identifiers to internal ones. Each time the event server receives a task identifier from the event stream, it looks through the mapping table for the internal identifier for that task identifier. If no matching task identifier is found in the table, the event server creates a new entry by assigning a new integer for this task identifier. The event server uses integer “0” to represent “no task.”

When POET persistently saves the event data to a UEF-formatted file, it stores the task identifier by using the internal format, "integer identifier." When POET restores the event data by reloading the UEF-formatted file, it simply retrieves the integer task-identifier, pads with zero bits if the length of a target task-identifier is larger than the length of an integer, and then sends it to the POET event server.

3.4.5 Agreement on Task Data between Event Server and Target

The agreement on task data between the event server and the target includes two aspects: First, the length of the task identifier is specified in a header file, "usr_debug.h," which is included by both the target-description file and the instrumentation program at compile time. Second, the event server determines the existence of the real task identifier for each event based on this value, with the special value "0" representing a void task identifier (*i.e.*, no task data). We did not specify the length of the task identifier in event zero because the version of POET we used does not use enhanced event zero.

3.5 Correlation Visualization

We adopt the POET visualization because our solution is built on top of POET. However, we use a different visualization method for task correlation from that for abstraction. In event abstraction, a set of events is clustered into one abstract event that can be viewed as an atomic event. This form of display is enabled by the convexity property of abstract events, which is not present in our solution.

Instead, we combine coloring and textual display to visualize task information. We use the functionality of the middle mouse button in POET. While keeping the basic features, we add coloration of events with the same task identifier and indicate that task identifier in the popup window.

When a popup box appears, a new field, an asterisk followed by a number (which is the mapped task identifier for the event), showing the task information will follow the trace name and event sequence number, as shown in Figure 3.11. For those events without any associated task (*i.e.*, task identifier is 0), no task identifier is shown, which is same as the original behavior of POET.

The coloring for events in different tasks maintains the original coloring scheme. Events within the same task are colored according to their precedence relationship to the event being clicked. A summary of the default coloring of our solution is shown in Table 3.1. The colors in Table 3.1 can be modified by using the POET resource file.

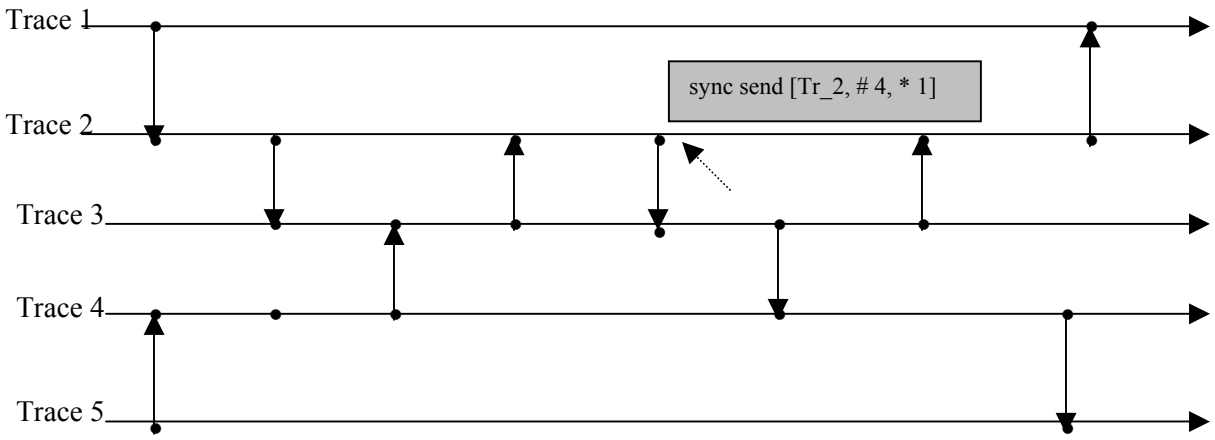

Figure 3.10: Popup Window for Selected Event

Correlation / Partial-Order Relationship	Same Task	Different Task
Predecessor	Light Green	Red
Successor	Dark Green	Green
Concurrent	Blue	No color

Table 3.1: Coloring Matrix

Chapter 4
Evaluation

In this chapter we analyze the costs of our solution, examine its use in some test environments, and compare it with existing solutions.

4.1 Cost Analysis

In our solution, extra costs are incurred since extra correlation data is collected. These costs include computing costs, communication costs, and storage costs.

On the target-system side, there will be an *O(1)* cost for each new task. For each event, if the task identifier is required, there is an *O(1)* cost when it is copied. On the event-server side, the processing cost for the transformation of task identifiers is *O(N),* where N is the number of tasks. This cost can be reduced to *O(1)* amortized by the hashing method in which each task identifier from the target is hashed to a value, which is used as the index of the internal task identifier. On the debug-session side, the cost is *O(1)* for processing the task identifier for each event.

The increased communication costs include three parts. The communication cost from the target side to the event-server side increases by V_t bits per event, where V_t is the length of a task identifier generated by the target. Similarly, the communication cost of passing task data between targets increases by V_t bits. The communication cost from the event-server side to the debug-session client side, however, is more complex to analyze, since event data is transported to the client in discrete blocks of multiple events. With task data present, there will be fewer events per block, but the block size remains the same. The effect is that for sequential access, ignoring block-header-size overhead, the cost increases by 4 bytes per event on average. For random-access, however, the communication-cost may not increase, but could double in the worst case. Specifically, when a set of consecutive events being accessed continues to fit into a single block, the cost does not change. If, on the other hand, the shift in the position of block boundaries causes a small set of events formerly in a single block to cross a block boundary, two blocks will need to be fetched rather than a single block, doubling the cost.

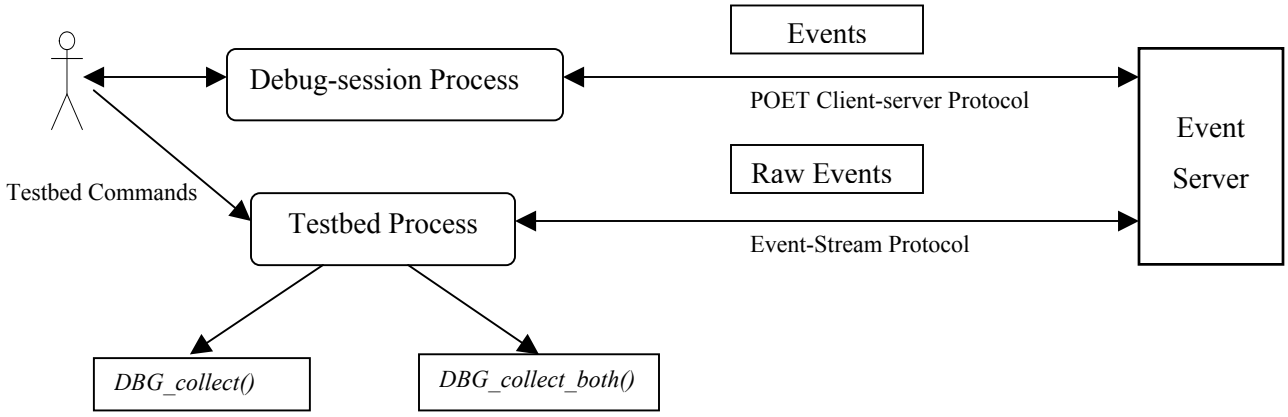

Figure 4.1: Testbed Environment of POET

As discussed in Chapter 2, POET may persistently store the event file on disk as a UEF-formatted file. With the task-identifier data, a UEF-formatted file contains an additional field for each event. Because the event server transforms the task identifier from target-specific format to an internal one, the length of the task identifier is reduced. Consequently, the extra storage cost for a UEF-formatted file is small for each event entry. For example, an integer in a UEF-formatted file occupies a small number of bytes. Compared with the length of target-generated task identifiers, typically 32 bytes, the persistent storage is reduced for each UEF-formatted file. An example of a UEF-formatted file with integer task identifiers is shown in Appendix B.

4.2 Evaluation of Task Data Collection

To test the feasibility and efficiency of our approach, we use the testbed tool to send event data to the POET server, and then display it with the debug-session process.

4.2.1 Testbed Environment

The testbed tool interacts with the end user or reads input from a script file, using the *DBG_collect* and *DBG_collect_both* API functions to send these events to the event server, as shown in Figure 4.1. By using testbed, the user can create virtually arbitrary displays for testing various facets of the debugger.

4.2.1.1 Syntax of Testbed

We made some modifications to the original testbed program, which enables it to generate task data. We implement two new commands in testbed. The new commands are shown in Figures 4.2 and 4.3.

trace_number trace_number ***t***: task_identifier

Figure 4.2: Binary Events with Task Data

trace_number ***t***: task_identifier

Figure 4.3: Unary Events with Task Data

The identifier following the "*t*:" is associated with the event(s) as the task identifier. If there is no "t:" in the line (*i.e.*, the original commands), it is presumed that the event has no associated task ID. Three sample scripts for our modified testbed program are shown in Appendix C.

4.2.1.2 Results

We use three scripts to test our solution. The first one tests simple synchronous events. The second tests simple asynchronous events. The third one tests the visualization of an event set that is not convex. In these displays, various shades of gray are used to visualize the task information of events. The events enclosed in a dashed curve belong to the same task. The dashed curve and the associated text are not parts of our visualization. They are used to enable the reader to more easily understand the diagram.

In the first test, there are two tasks. The events belonging to them form two sets. The remaining events, without task identifiers, belong to no task. Figure 4.4 shows the visualization of this script. In this visualization, we select two events to display the task information. The displays for these events are shown in Figures 4.5 and 4.6.

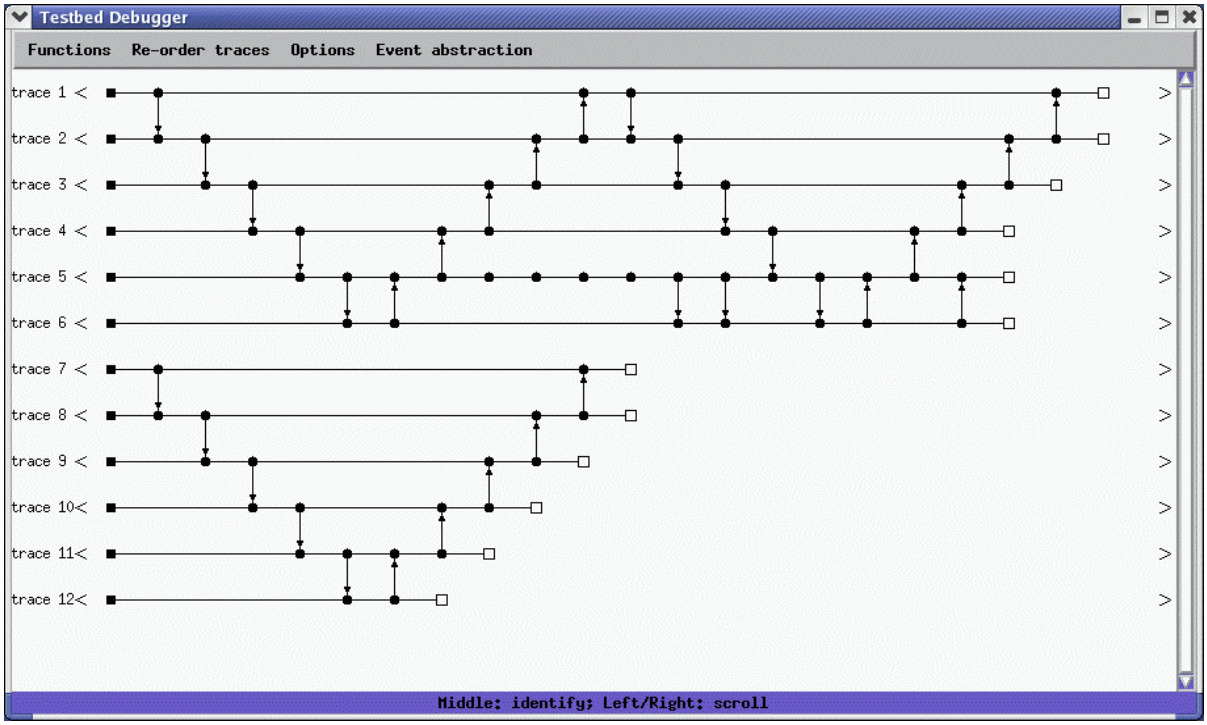

Figure 4.4: Visualization of Script 1

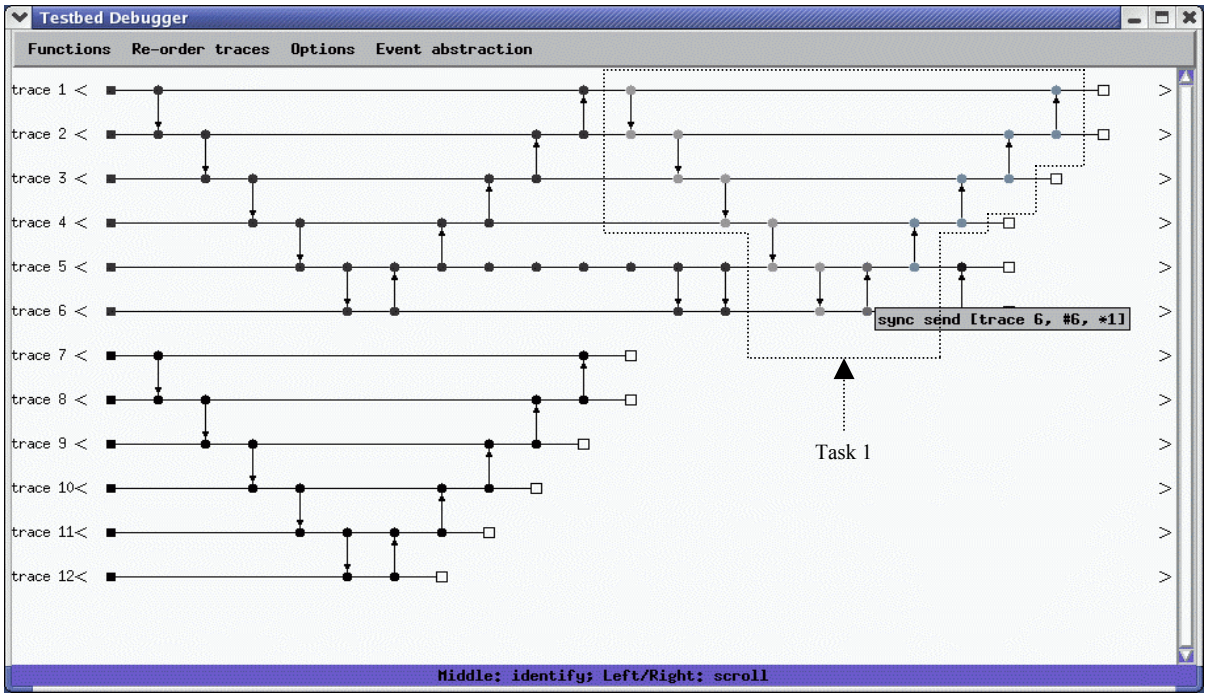

Figure 4.5: Display of an Event of the First Task

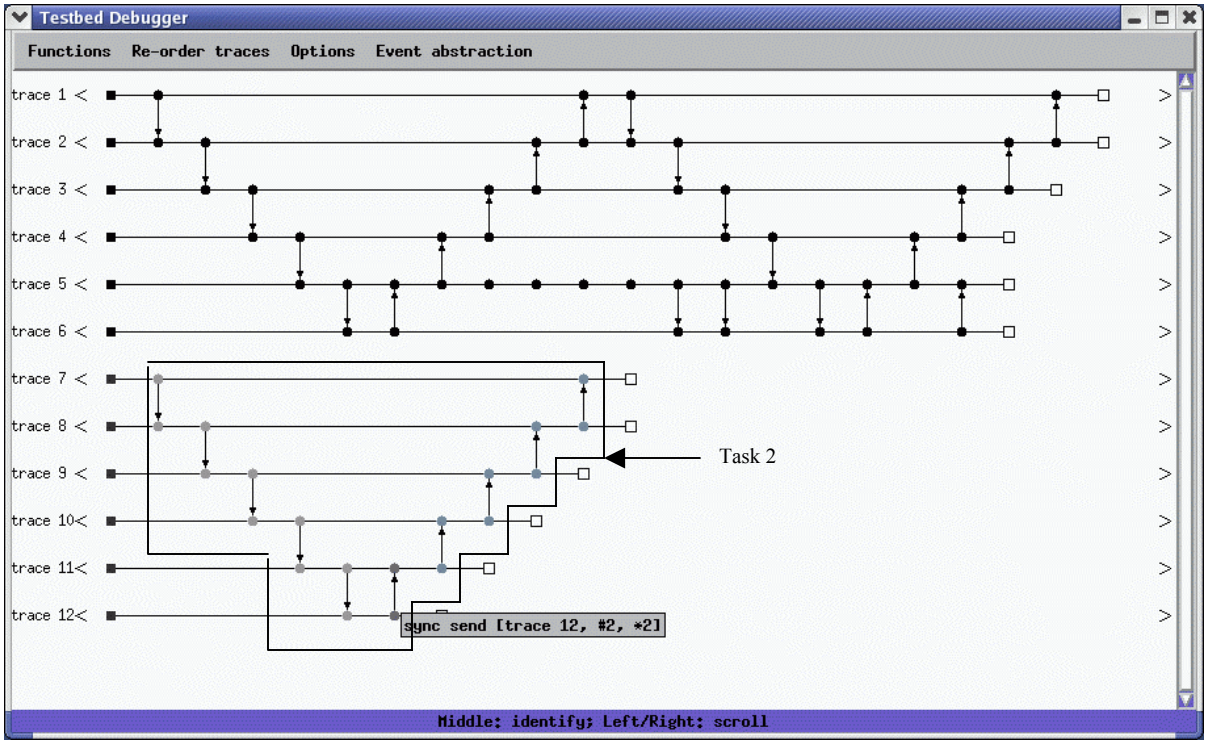

Figure 4.6: Display of an Event of the Second Task

In the second test, there are two tasks. Figure 4.7 shows the visualization of this script. Figures 4.8 and 4.9 show two events with task information.

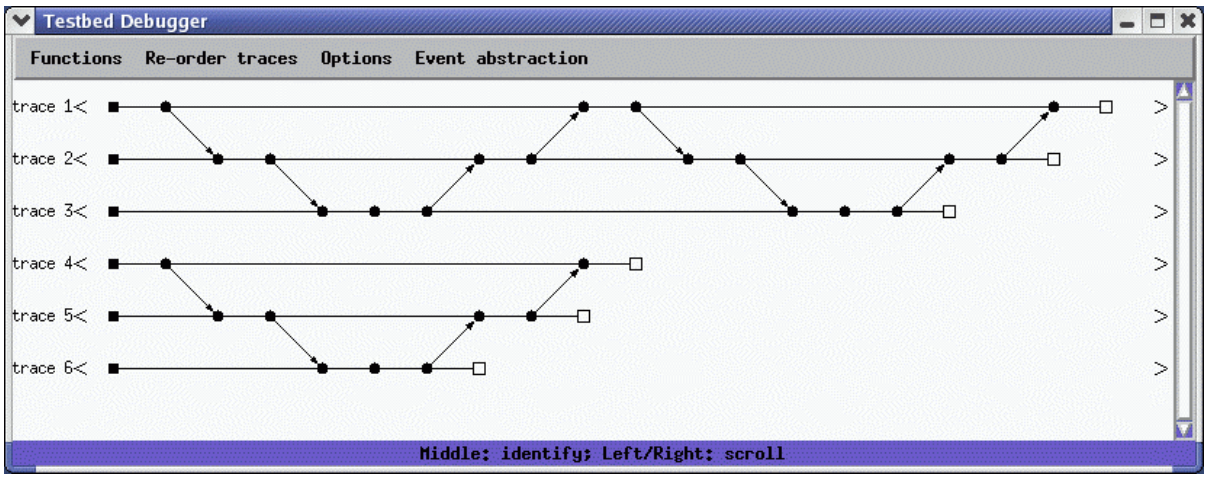

Figure 4.7: Visualization of Script 2

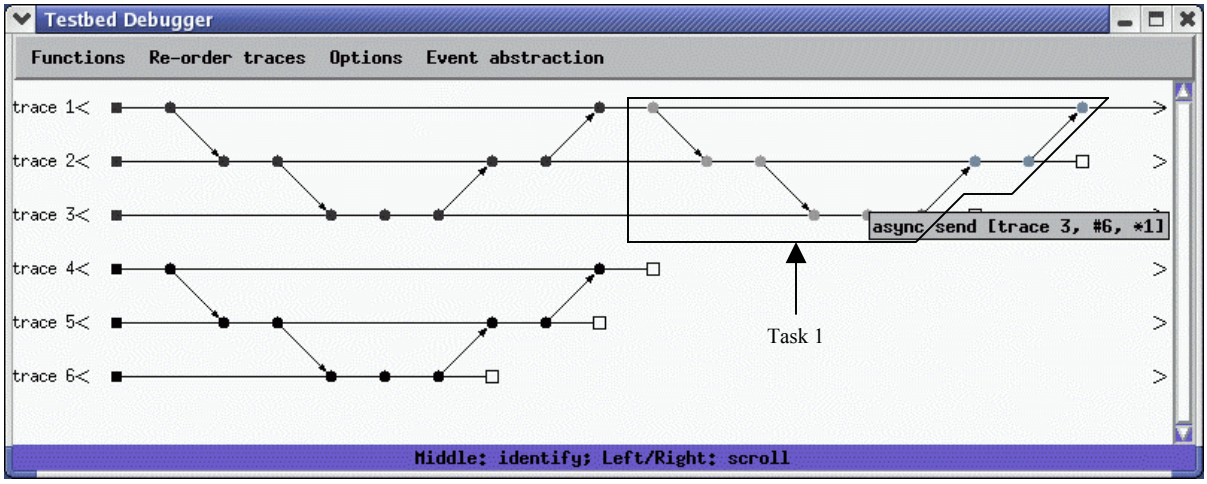

Figure 4.8: Display of an Event of the First Task

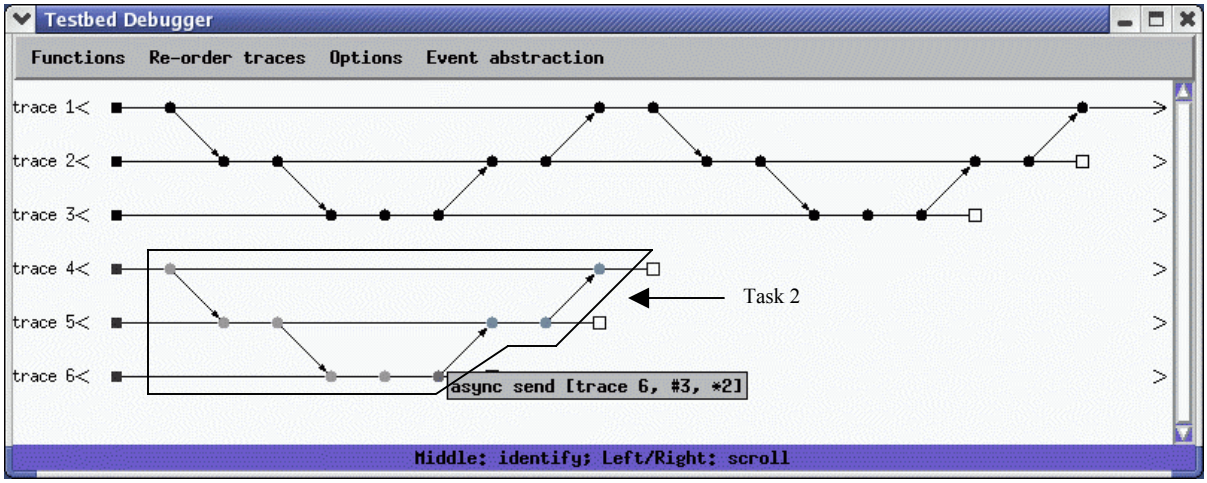

Figure 4.9: Display of an Event of the Second Task

In the third test, we see that our correlation is not constrained to be convex. Figure 4.10 is the visualization of script 3, which is listed in Appendix C. Figures 4.11 and 4.12 show two positioned events that belong to two different tasks. The events belonging to the first task cannot be abstracted because the event set is not convex.

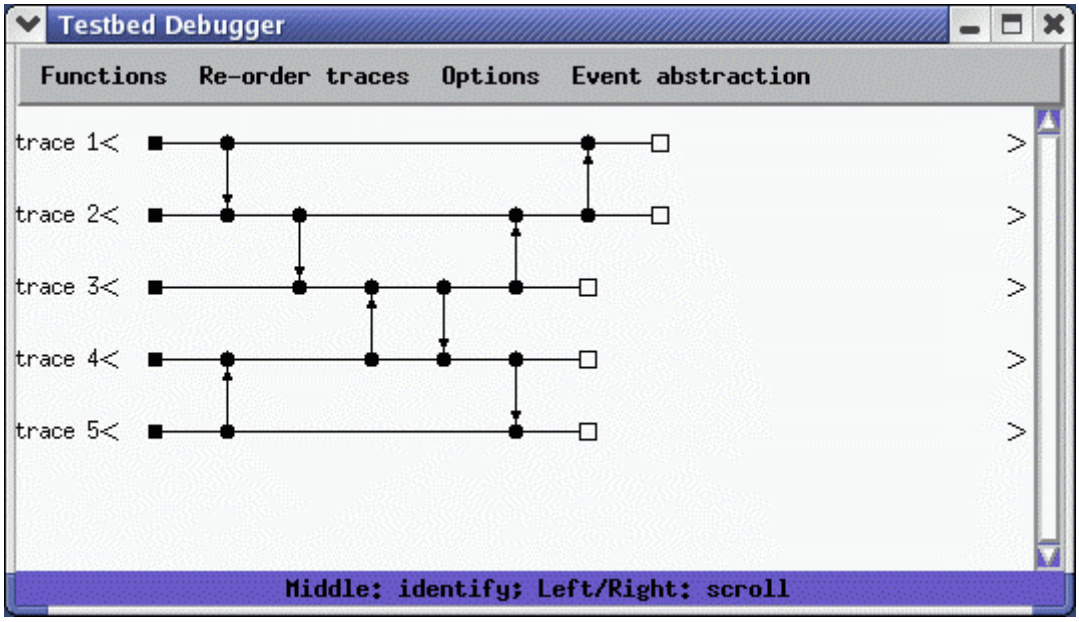

Figure 4.10: Visualization of Script 3

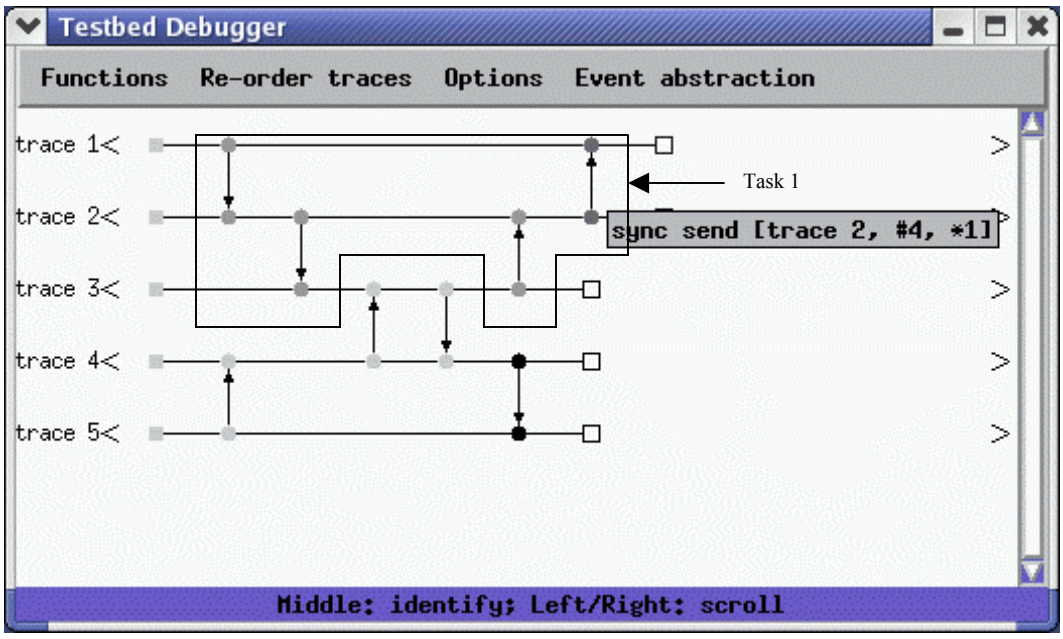

Figure 4.11: Display of an Event of the First Task

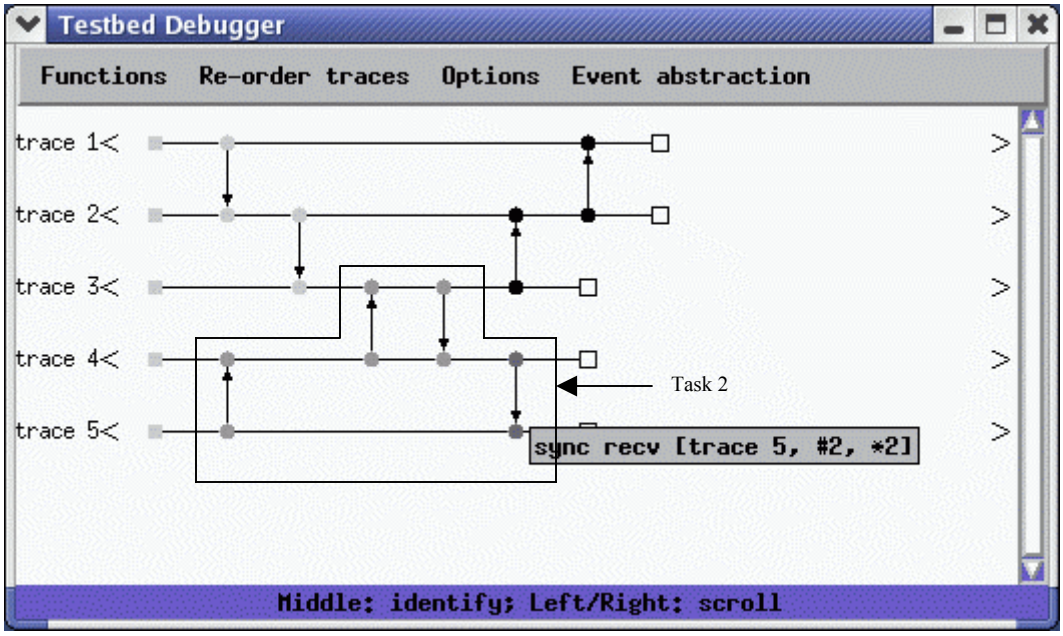

Figure 4.12: Display of an Event of the Second Task

4.3 Java RMI Environment

Java RMI enables programmers to invoke methods on remote objects residing in other Java Virtual Machines [RMI Website]. Fundamental to RMI is the object serialization to transmit parameters between the client and server. The stub and skeleton act as proxy objects that communicate with each other to transmit the parameters and return value. The architecture of RMI is shown in Figure 4.13.

In Section 3.2, we discussed the mapping of "task" in the Java RMI environment. That is, a task in RMI may refer to a chain of invocations. All of the invocations in this chain implement the original computing purpose initiated by the first RMI client. An example of a composite-RMI invocation is shown in Figure 4.14.

In a composite RMI, the intermediate RMI server acts as both a client and a server. On the one hand, it receives the invocation request from the client. On the other hand, it invokes a method on another remote object. The intermediate RMIs and execution at the final server do not occur independently, but in the context of the initial RMI. Figure 4.15 shows the infrastructure of the composite RMI.

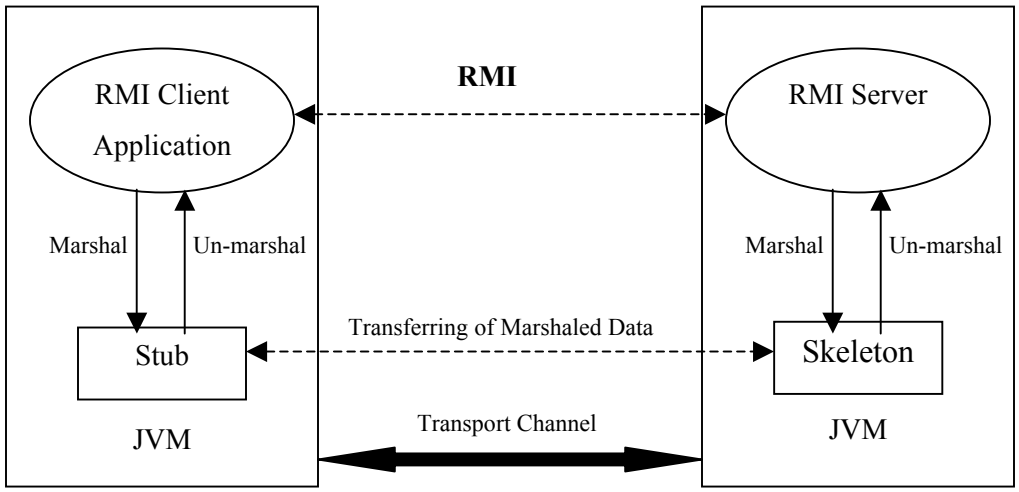

Figure 4.13: Java RMI

RMI is a multi-threaded environment. The threads in RMI can be categorized into two types: the daemon threads and the service threads. The daemon thread is responsible for receiving an incoming RMI request and dispatching it to a spawned service thread. A service thread is created by the daemon thread to fulfill the concrete computing for an RMI request. Such a thread structure is shown in Figure 4.16.

4.3.1 Instrumentation

To instrument Java RMI, we capture the events at the thread level. Therefore, a trace represents a thread in our implementation. Our instrumentation captures the following types of events:

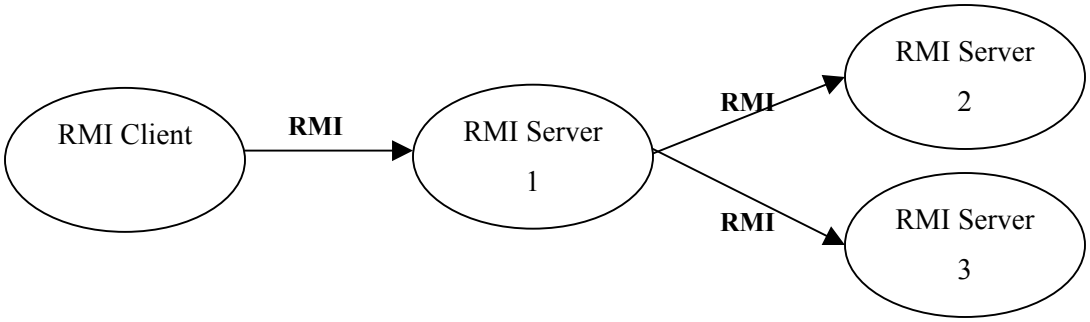

Figure 4.14: A Composite RMI

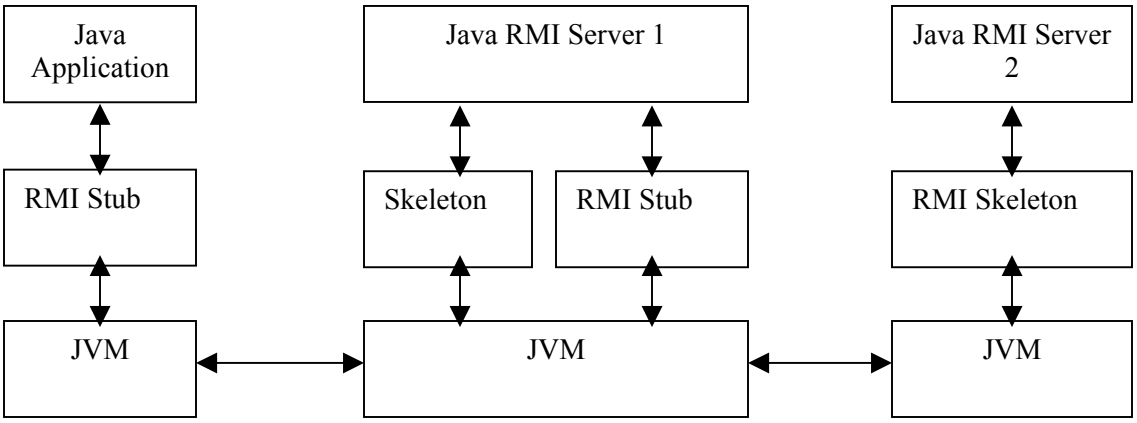

Figure 4.15: The Infrastructure of Composite RMI Invocation

• RMI Trace Start: a daemon thread or an RMI client thread starts.
• RMI Trace Create: a daemon thread spawns a service thread.
• RMI Trace Spawned: a new service thread is spawned in response to an RMI.
• RMI Invocation: an RMI client invokes a remote call.
• RMI Request: an RMI server receives an RMI call request.
• RMI Reply: an RMI server replies to an RMI call request.
• RMI Return: an RMI client receives the return values of an RMI call if the return type is not void.
• RMI Served: an RMI server finishes an invocation service without a return value (*i.e.*, the return type is void for an RMI call). There is no synchronization between the RMI client and the RMI server when the return type is void for an RMI call.

Since the "Exit" event is not in our research scope, we do not collect it. In our instrumentation, the event types are defined in a class called *EventType* that is shown below:

```
public class EventType{
    public final static int RMI_TRACE_CREATE = 1;
    public final static int RMI_TRACE_SPAWNED = 2;
    public final static int RMI_INVOKE = 3;
```

```
    public final static int RMI_REQUEST = 4;
    public final static int RMI_REPLY = 5;
    public final static int RMI_RETURN = 6;
    public final static int RMI_SERVED = 7;
    public final static int RMI_TRACE_START = 8;
}
```

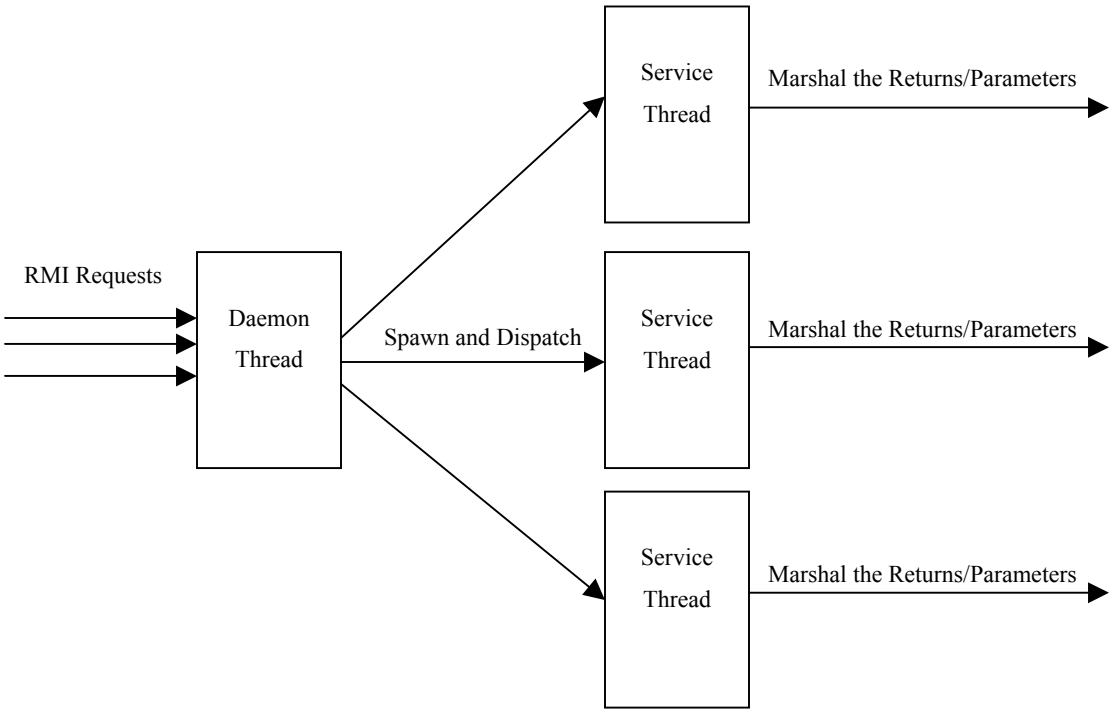

Figure 4.16: Thread Structure in Java RMI

4.3.1.1 Event Collection

Event collection has two parts: one is on the RMI server and the other is on the RMI client. On the server side, the instrumentation is inserted into the skeleton; on the client side, the instrumentation is

inserted into the stub. Both of them use a set of Java functions implemented by a class called *Collect*. The interfaces of the collection functions are the same as in the C collect functions, modified for the Java type system. The static method *collect_init()* is used to collect event zero.

```
public class Collect{
.....
    public static  int collect_init();
.....
}
```

On the RMI-client side, three types of events are collected, which are RMI Trace Start, RMI Invocation, and RMI Return. The collection algorithm is as follows:

• When an RMI client thread starts, the instrumentation collects an RMI Trace Start event. For the RMI Trace-Start event, a text event is collected following it.

• When the client stub invokes a remote call and marshals the call parameters, the instrumentation collects an RMI Invocation event.

• When it receives and un-marshals the return value, the instrumentation collects an RMI Return event.

On the RMI-server side, six types of events are collected, which are RMI Trace Start, RMI Trace Create, RMI Trace Spawned, RMI Request, RMI Reply, and RMI Served. The collection algorithm is as follow:

• When a daemon thread starts, the instrumentation collects an RMI Trace Start event. For this event, a text event is collected following it.

• Each time the server skeleton receives an RMI request, it creates a service thread for this RMI invocation. The instrumentation collects an RMI Trace Create event for the daemon thread and an RMI Trace Spawned event as the first event of the spawned thread.

• When the service thread unmarshals the parameters of the RMI, the instrumentation collects an RMI Request event.

When the RMI finishes in the service thread,

• The instrumentation collects an RMI Reply event if this RMI call has a return value;

• The instrumentation collects an RMI Served event if this RMI call has no return value. In this case, the RMI call has "void" return type.

4.3.1.2 Task Data Collection

Our instrumentation to collect task data includes three parts: generation of task identifiers, propagation of task identifiers, and collection of task identifiers.

• Generation of Task Identifier

In a composite RMI, the generator of a task identifier is the end client. The length of task identifier is 16 bytes in our instrumentation. Our instrumentation uses a class called *TaskID* to handle the generation of the task identifier. The main methods of this class are shown below:

```
public class TaskIDGenerator{
public static byte[]      getTaskID();
 public static boolean   isZero(byte[] taskID);
}
```

Since our solution is based on the thread level, the thread is the "real" generator of the task identifier. For a composite Java RMI, only the extreme-end client is the initiator of the task. The intermediate clients (also acting as servers) just propagate task identifiers. We use a flag for each thread to indicate whether it is an extreme-end RMI client. This flag of a service thread is set at the time the daemon thread spawns the thread. The operation of an RMI server is such that a service thread must be spawned for an object to accept remote requests. Thus every RMI server will have this flag set and only the extreme-end client will not have the flag set. The instrumentation then determines whether the thread is the one initiating a task by whether or not this flag is set.

• Propagation of Task Identifier

After the task identifier is generated, we need to propagate it along the RMI call path. When an RMI server invokes a remote call on other RMI servers, instrumentation has to deal with the propagation of the task identifier for these invocation calls.

In accordance with our generic algorithm of Section 3.4.2, the mapping mechanism between the task identifier and the thread is needed in the instrumentation. In our instrumentation, this mapping mechanism is implemented by attaching a task tag to each thread. A thread can retrieve the task identifier with which it is associated from the tag by using function shown as below:

```
byte[]      Thread.currentThread().gettaskID();
```

In our instrumentation, the propagation of task identifiers is fulfilled by marshaling and un-marshaling the task identifier wrapped in the RMI request, as shown in Figure 4.17. This procedure is transparent to the applications.

The propagation algorithm is as follows:

• When an RMI client marshals the parameters, the instrumentation marshals such additional data as the stream identifier (uniquely identifying an event stream), the trace identifier (uniquely identifying a thread), the event-sequence count, and the task identifier.

• When an RMI server receives an invocation request, the instrumentation un-marshals those additional data (the task identifier, *etc.*). Then, it sets the task tag of the spawned service thread to this incoming task identifier.

• Collection of Task Data

The length of task identifier is defined in a class called *TaskProperty*, which is shown as below.

```
public class TaskProperty{
public final static int taskID_Length;
}
```

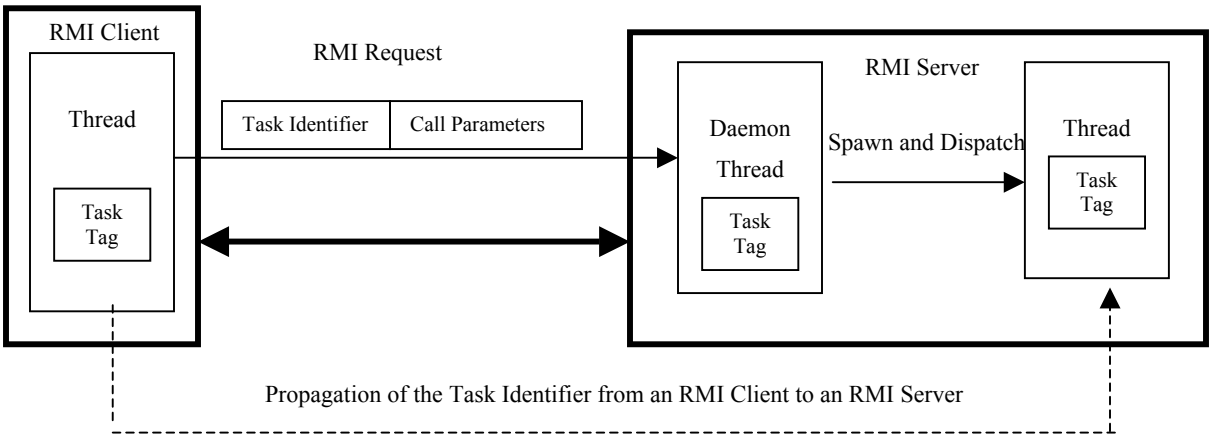

Figure 4.17: Propagation of Task Identifier

The instrumentation uses the collection methods of the *Collect* class to collect the task identifier as well as other event information (*i.e.*, event sequence count, trace identifier, and stream identifier).

4.3.2 Visualization

In our visualization, a trace represents the execution of a thread. The symbol shapes of events are defined as:

(a) The "RMI Trace Start" event is represented by the filled square.

(b) The "RMI Trace Create" and "RMI Trace Spawned" events are represented by the open circles.

(c) The other events are represented by filled circles.

All of these are specified in the target-description file.

4.3.3 Results

Our sample source code is given in Appendix D. The results of the sample are shown in Figure 4.18. Figure 4.19 shows the display, including task information, when an event is selected.

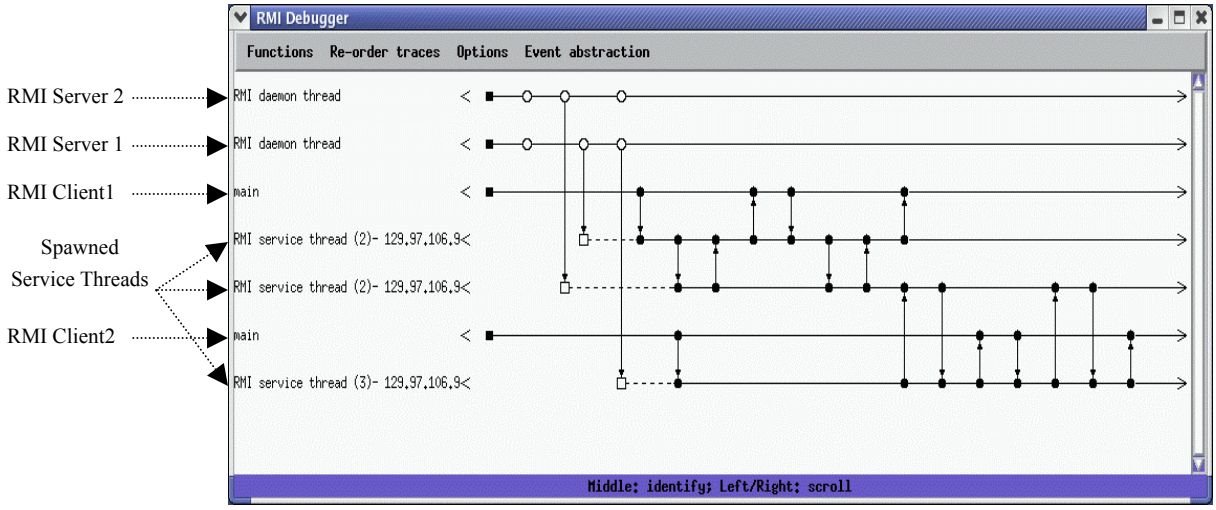

Figure 4.18: The Visualization Result for the Sample Source Code

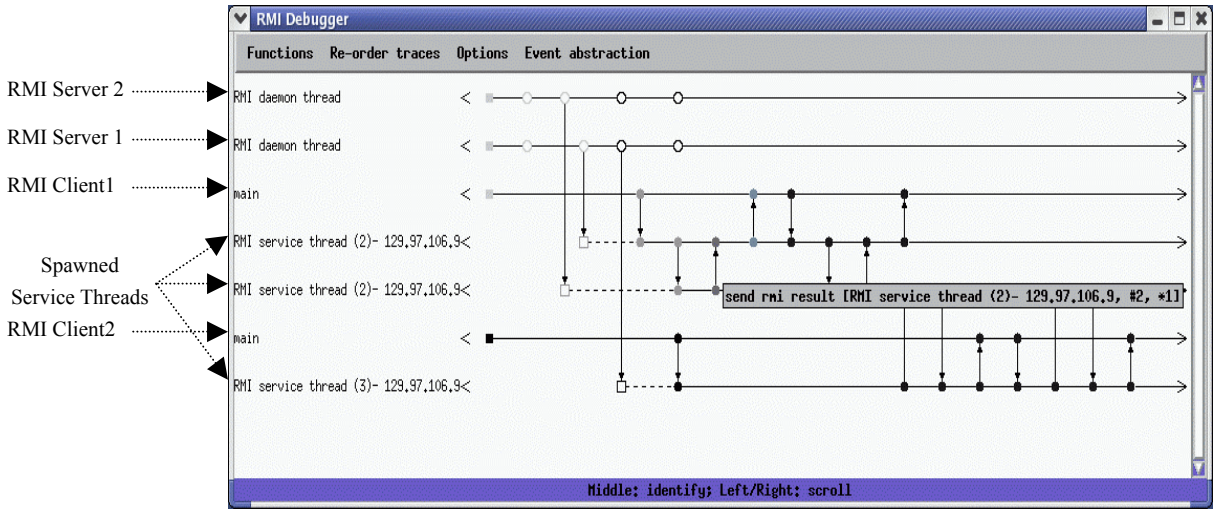

Figure 4.19: Display of a Positioned Event

4.4 Comparison with LTA

Our solution has the following advantages over LTA:

(1) LTA has no “task” concept as we define it. The correlation domains adopted by LTA currently are not abstract, whereas our “task” concept is an abstract correlation domain. It can be mapped to

various target environments (Web Services, Java RMI, *etc*.). Thus, our correlation technique has good target independence.

(2) Our solution maintains all of the features of POET. Our solution therefore can visualize partial-order as well as task information. LTA does not present partial-order information.

4.5 Comparison with the Approach of Sahai *et al.*

The idea of task comes from the approach of Sahai *et al.* However, our solution has various advantages over it.

(1) The approach of Sahai *et al.* is target-dependent. Its concept of "transaction" is bound to Web Services and its "transaction" data collection relies on XML-formatted messages and SOAP. While it is limited to a specific target, our solution is independent of any target.

(2) Our solution is more scalable and efficient than the approach of Sahai *et al.* in terms of correlation-data collection. The task data appended to each message is of constant length in our solution, while that in the approach of Sahai *et al.* is variable and frequently very large.

4.6 Comparison with POET Abstraction

Event abstraction and task correlation are not alternative but complementary techniques in POET. Both of them are useful in analyzing event data. They differ in many aspects, including the following:

(1) Our solution focuses on the identification of tasks. It does not necessarily reduce the display complexity. In some sense, our solution increases the display complexity by adding more coloring options, while event abstraction reduces the visualization complexity by clustering multiple events into a single one.

(2) They have different visualization methods. Our solution uses a coloring scheme (and pop-up text box) to visualize task correlation while event abstraction uses a clustering method.

(3) The most important difference is that task correlation is not restricted to the requirement of convexity to preserve the partial order. Such a difference enables task correlation to be applied to more event sets than event abstraction. Therefore, task correlation is more flexible, and has broader application. While it is possible to create abstract events from sets of events with the same task identifier if those sets happens to be convex, in the general case this will not be possible.

Chapter 5
Conclusions and Future Work

In this thesis, we have explored a new correlation scheme, correlation by task. From our work, we draw the following important conclusions.

First, we have eliminated the target dependence of the "transaction" concept in the approach of Sahai *et al*. by redefining the "task" concept. "Task" is a generic correlation function, and has an abstract domain that can be mapped to various concrete ones in various target systems. As discussed in previous chapters, we have seen various "task"-correlator instances mapped from our generic correlator into real target systems.

Second, we have developed a correlation solution on top of POET based on our "task" correlator. In our solution we provided a visualization method for task correlation. We proposed general requirements and an algorithm for the instrumentation of target systems. According to these requirements we instrumented Java RMI and successfully collected task data.

We used both the testbed tool and Java RMI to evaluate our solution and achieved the expected results. According to our cost analysis, our solution overcomes the scalability problem in Sahai's system. Therefore our solution is efficient and of good scalability, which is another advantage over the approach of Sahai *et al*.

From the discussion above, we conclude that our solution is a feasible, efficient, and scalable correlation solution. It is useful for the user to identify the relationships of events for various targets.

5.1 Future Work

There still exist some potential extensions for our work. They include:
(1) While we gave the basic idea of how to represent nested tasks, we did not implement nested-task collection and visualization in POET. Our coloring solution to visualize task data is not suitable for nested tasks since it will make the display complex and difficult to present task patterns.

(2) Integration of task correlation and event abstraction. Our work focused on the correlation of primitive events. We have not taken abstract events into account. Such work can be considered together with that presented here. For example, one possible approach is to use event abstraction to cluster the events in some level of a task hierarchy. The correlation can be applied in the upper level of the task hierarchy. By integrating task correlation and event abstraction, the visualization of nested tasks may be solved as well.

(3) There is much practical work in the instrumentation of other targets, especially those of heterogeneous systems, such as Web Services and multi-tier web applications. “Task” is very useful in such targets. The reason that we did not instrument such systems is that POET cannot provide multi-target functionality at this time. However, we believe that POET will provide such functionality in the near future as it is evolving fast. At the time POET is able to do that, our solution should produce more valuable information for the user to monitor multi-target systems and analyze their behavior.

Appendix A
A SOAP Message Containing MDR

(Below is from [SMO+02])

```
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV=http://schema.xmlsoap.org/soap/envelop/
   SOAP-ENV: encodingStyle: http://schema.xmlsoap.org/soap/encoding/>
<SOAP-ENV: Header>
<MDR>
<parent_mdr><parent_mdr>
<message_id>a unique message_id number</message_id>
<message_type>a type of message</message_type>
<source>a source identifier</source>
<target>target identifier</target>
<time_sent>a time record</time_sent>
<time_received>another time record</time_received>
</MDR>
</SOAP-ENV:Header>
<SOAP-ENV: Body>
    <PurchaseOrder>
      <Item count = 100> Postit sticky notes </Item>
      <Item count = 200> Stapler </Item>
     <PurchaseOrder>
</SOAP-ENV: body>
</SOAP-ENV: Envelope>
```

Appendix B

A UEF-Formatted File

```
#ADED Ascii Dump of Event Data file
#Version 1.0
#2180
#1090341567
6a6a7775
4
4
25
00000010
3
0     00000000
1     01000000
2     02000000
3
0     0     280e74ffffffef
1     1     54ffffffbc042b
2     2     03fffffaefffffefffffffdf
1     0     0     -1    -1    1
1     1     0     -1    -1    1
1     2     0     -1    -1    1
2     0     1     -1    0     1
3     1     1     0     1     1
2     1     2     -1    0     1
3     2     1     1     2     1
2     2     2     -1    0     1
3     1     3     2     2     1
```

6	*1*	*4*	*-1*	*-1*	*1*
2	*0*	*2*	*-1*	*0*	*2*
3	*1*	*5*	*0*	*2*	*2*
2	*1*	*6*	*-1*	*0*	*2*
3	*2*	*3*	*1*	*6*	*2*
2	*2*	*4*	*-1*	*0*	*2*
3	*1*	*7*	*2*	*4*	*2*
6	*1*	*8*	*-1*	*-1*	*2*

Appendix C

Sample Testbed Scripts

Script 1

```
#begin
start 0 "trace 1"
start 1 "trace 2"
start 2 "trace 3"
start 3 "trace 4"
start 4 "trace 5"
start 5 "trace 6"
start 6 "trace 7"
start 7 "trace 8"
start 8 "trace 9"
start 9 "trace 10"
start 10 "trace 11"
start 11 "trace 12"
# Do a simple synchronous RPC without task data
0 1
1 2
2 3
3 4
4 5
5 4
4 3
3 2
2 1
1 0
```

```
#Do a chain of simple synchronous RPCs with task
0 1 t: 1234567890abcdef
4
4
4
4
4 5
4 5
1 2 t: 1234567890abcdef
2 3 t: 1234567890abcdef
3 4 t: 1234567890abcdef
4 5 t: 1234567890abcdef
5 4 t: 1234567890abcdef
4 3 t: 1234567890abcdef
5 4
3 2 t: 1234567890abcdef
2 1 t: 1234567890abcdef
1 0 t: 1234567890abcdef
#Do another chain of simple synchronous RPCs with task
6 7 t: 9876543210fedcba
7 8 t: 9876543210fedcba
8 9 t: 9876543210fedcba
9 10 t: 9876543210fedcba
10 11 t: 9876543210fedcba
11 10 t: 9876543210fedcba
10 9 t: 9876543210fedcba
9 8 t: 9876543210fedcba
8 7 t: 9876543210fedcba
7 6 t: 9876543210fedcba
#end
```

Script 2

```
#start
start 0 "trace 1"
start 1 "trace 2"
start 2 "trace 3"
start 3 "trace 4"
start 4 "trace 5"
start 5 "trace 6"

async
#The first set of events without task identifier
0 1
1 2
2
2 1
1 0

#The second set of events with task identifer
0 1 t: abcdef0987654321
1 2 t: abcdef0987654321
2 t: abcdef0987654321
2 1 t: abcdef0987654321
1 0 t: abcdef0987654321

#The third set of events with task identifier
3 4 t: fedcba0987654321
4 5 t: fedcba0987654321
5 t: fedcba0987654321
5 4 t: fedcba0987654321
4 3 t: fedcba0987654321
```

Script 3

```
#start
start 0 "trace 1"
start 1 "trace 2"
start 2 "trace 3"
start 3 "trace 4"
start 4 "trace 5"

#The first set of events with task identifier
0 1 t: a1b2c3d4e5f60987
1 2 t: a1b2c3d4e5f60987
4 3 t: 9f8e7d6c5b4a0123
3 2 t: 9f8e7d6c5b4a0123
2 3 t: 9f8e7d6c5b4a0123
3 4 t: 9f8e7d6c5b4a0123
2 1 t: a1b2c3d4e5f60987
1 0 t: a1b2c3d4e5f60987

#end
```

Appendix D

Java RMI Sample Codes

1. RealTime.java

```
import java.net.*;
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;

public class RealTime extends UnicastRemoteObject
                      implements RealTimeI
{
    public RealTime() throws RemoteException {
      //        super();
    }

    public long getRealTime() throws RemoteException {
               return System.currentTimeMillis();
    }

    public static void main(String[] args) {
              try {
                   RealTime rt = new RealTime();
                   Naming.rebind("//localhost:1099/RealTime", rt);
                   System.out.println("RealTime Ready to do Time");
               } catch (Exception e) {
                  e.printStackTrace();
               }
    }
}
```

2. PerfectTime.java

```
import java.net.*;
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;

public class PerfectTime extends UnicastRemoteObject
                         implements PerfectTimeI
{
    public PerfectTime() throws RemoteException {}

    public int getPerfectTime() throws RemoteException {
       RealTimeI rt;
       long rtime;
       try {
           rt=(RealTimeI)Naming.lookup("//localhost:1099/RealTime");
           rtime =  rt.getRealTime();
       } catch (Exception e){e.printStackTrace(); return 0;}
       return 1;
    }

    public static void main(String[] args) {
       try {
            PerfectTime pt = new PerfectTime();
            Naming.rebind("//localhost:1099/PerfectTime", pt);
            System.out.println("Ready to do Time");
       } catch (Exception e) {
         e.printStackTrace(); }
    }
}
```

3. DisplayPerfectTime.java

```
import java.rmi.*;
import java.rmi.registry.*;

public class DisplayPerfectTime {

  public DisplayPerfectTime() {
                  super();
  }

  public static void main(String[] args) {
      try {
           for (int i = 0; i < 2; i++) {
           PerfectTimeI t = (PerfectTimeI)Naming.lookup
                                ("//localhost:1099/PerfectTime");
           System.out.println("PerfectTime:"+t.getPerfectTime());
           }
      } catch (Exception e) {
        e.printStackTrace();
      }
  }
}
```

References

[And+88] G. R. Andrews, *et al.* An overview of the SR language and implementation. *ACM Trans. Progr. Languages Systems*, 10, 51-86. 1988.

[AOK+95] E. Arjomandi, W. O'Farrell, I. Kalas, G. Koblents, F. C. Eigler, and G. R. Gao. ABC++: concurrency by inheritance in C++. *IBM Sys. J.*, 34, 120-137. 1995.

[BuS91] P. A. Buhr and R. A. Stroobosscher. The μSystem: Providing light-weight concurrency on shared-memory multiprocessor computers running Unix. *Software - Practice Exper.*, 20, 929-963. 1991.

[CBE Website] Many Chessell, Jason Cornpropst, John Gerken, Bill Horn, Heather Kreger, Eric Labadie, David Ogle, and Abdi Salahshour. *Specification: Common base event*. Available at http://www-106.ibm.com/developerworks/webservices/library/ws-cbe/. July, 2003.

[EdE98] Guy Eddon and Henry Eddon. *Inside Distributed COM*. Microsoft Press, February, 1998.

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA. 1994.

[GKV94] Siegfried Grabner, Dieter Kranzlmuller, and Jens Volkert. EMU - Event Monitoring Utility. Technical Report, Institute for Computer Science, Johannes Kepler University Linz, July 1994.

[GEG+01] Thomas Gschwind, Kave Eshghi, Pankaj K. Garg, and Klaus Wurster. Web Transaction Monitoring. HPL-2001-62, http://www.hpl.hp.com/techreports/2001/HPL-2001-62.html.

[LogTrace Website] IBM Corporation. Log and Trace Analyzer for Autonomic Computing. Available at http://www.alphaworks.ibm.com/tech/logandtrace.

[OLT Website] IBM Corporation. WebSphere application server, Object-Level Trace. Technical Report. Available at http://www-306.ibm.com/software/webservers/appserv/olt.html, IBM Corporation, 1998.

[LTA Documents] IBM Corporation. Log and Trace Analyzer Version 1.0.1, Help Documents, IBM Corporation, 2003.

[KGV95] Dieter Kranzlmuller, Siegfried Grabner, and Jens Volkert. Race condition detection with the MAD environment. In *Second Australasian Conference on Parallel and Real-Time Systems*, pages 160-166, September 1995.

[KGV97] Dieter Kranzlmuller, Siegfried Grabner, and Jens Volkert. Debugging with the MAD environment. *Journal of Parallel Computing*, 23(1-2):199-217, April 1997.

[Kun93] Thomas Kunz. Issues in event abstraction. In *Proceedings of PARLE '93: Parallel Architectures and Languages Europe.* Edited by Arndt Bode, Mike Reeve, and Gottfried Wolf. Published by Springer-Verlag, Munich, Germany. Lecture Notes in Computer Science. Number 694, pages 668-671, June 1993.

[Kun94] Thomas Kunz. *Abstract Behaviour of Distributed Executions with Applications to Visualization*. Technische Hochschule Darmstadt, Darmstadt, Germany. May 1994.

[KBT+97] Thomas Kunz, James P. Black, David J. Taylor, and Twan A. Basten. POET: Target-system independent visualizations of complex distributed-application executions. *The Computer Journal*, 40(8): 499–512, 1997.

[Lam78] L. Lamport. Time, clocks, and the ordering events in a distributed system. *Communications of the ACM*, 21(7): 558-565, July 1978.

[Lok95] Swee Loke. *Debugging Support for a Real-time System*. Master's thesis, Queen's University, 1995.

[E-XML website] OASIS. E-business XML. Available at http://www.ebxml.org.

[UDDI Website] OASIS. Universal Description, Discovery, and Integration. Available at http://www.uddi.org.

[OSF93] Open Software Foundation. *Introduction to OSF/DCE*. Prentice-Hall, Englewood Cliffs, NJ. 1993.

[SMO+02] A. Sahai, V. Machiraju, J. Ouyang, and K. Wurster. Message tracking in SOAP-based web services. *Network Operations and Management Symposium, 2002*. NOMS 2002. 2002 IEEE/IFIP, 15-19 April 2002, pages 33- 47.

[See95] Ilene R. Seelemann. *Visualizing Concurrent Object-oriented Programs*. Master's thesis, University of Waterloo, Waterloo, Ontario, Canada. June 1995.

[SBL+91] Robert E. Strom, David F. Bacon, Andy Lowry, Arthur P. Goldberg, Daniel M. Yellin, and Shaula Yemini. *Hermes: A Language for Distributed Computing*. Prentice-Hall, Englewood Cliffs, NJ. 1991.

[Tay95] David J. Taylor. Event display for debugging and managing distributed systems. *Proceedings of International Workshop on Network and Systems Management*, pages 112-124, August 1995.

[Tay03] David J. Taylor. File format for POET event-dump (.uef) files. Unpublished. November 18, 2003.

[Tay97] David J. Taylor. The POET Prototype: Structure and Operation. Unpublished. January 7, 1999.

[TKB95] David J. Taylor, Thomas Kunz, and James P. Black. Achieving target-system independence in event visualisation**.** In *CD-ROM Proceedings of the 1995 CAS Conference.* IBM Canada Ltd. Laboratory, Centre for Advanced Studies. Toronto, Ont., Canada. November 1995, pages 296-307.

[War02] Paul A. S. Ward. *A Scalable Partial-Order Data Structure for Distributed-System Observation*. Waterloo, Ontario, Canada, 2002.

[SOAP website] W3C. Simple Object Access Protocol. Available at http://www.w3.org/TR/soap.

[WSDL Website] W3C. Web Services Description Language. Available at http://www.w3.org/TR/wsdl.

[Xie04] Ping Xie. *Convex-Event Based Offline Event-Predicate Detection*. Master's thesis, University of Waterloo, 2004.

[YGS+89] S. A. Yemini, G. S. Goldszmidt, A. D. Stoyenko, and Y. H. Wei. CONCERT: A high-level-language approach to heterogeneous distributed systems. In *Proc. 9th Int. Conf. on Distr. Comput. Systems*, Newport Beach, CA, June, pages 162-171. IEEE Computer Society Press, Los Alamitos, CA. 1989.