Joint Conference of 1996 International Computer Symposium
December 19~21, Kaohsiung, Taiwan, R.O.C.

A Tool for Layered Analysing and Debugging of Distributed Programs

Wanlei Zhou

School of Computing and Mathematics
Deakin University
Geelong, VIC 3217, Australia
wanlei@deakin.edu.au

Abstract

Analysing and debugging distributed programs is much more difficult than analysing and debugging sequential programs. One of the reasons is the communication among programs (processes) which may happen concurrently and nondeterministically. To be able to analyse such communication events is therefore an essential task for any distributed program analyser/debugger. This paper describes the design and implementation of a tool for layered analysing and debugging of distributed programs. In the highest level, the tool displays the communication relationships between programs (eg, server and client programs). In the second level, the communication between processes is displayed. In the third level, the communication between events is displayed. The forth level is the lowest level: it uses a text editor to show the relevant statements that carry out the communication. The tool has been regularly used in analysing and debugging student assignments on distributed programming for three years.

1 Introduction

A distributed program can be viewed as a group of *program parts* (PPs, each PP can be a process or even a program. An example is the client and server program parts of a distributed program) that work together on a single task, and the concurrency and communication among these parts are the main reasons that make debugging of distributed programs difficult [4, 14]. To be able to analyse such concurrent and communicatiing events is therefore an essential task for any distributed program debugger/analyser.

This paper describes the design and implementation of a tool for layered analysing and debugging of distributed programs. The tool has been implemented on networks consisting of DEC/HP/SUN workstations and it has been regularly used in teaching of courses related to distributed systems since 1993 (mainly used in analysing and debugging student assignments on distributed programming). The tool helps a user to analyse and debug a distributed program in a top-down fashion. In the top level, the tool displays the communication relationships between program parts (eg, server and client programs). The top level gives a user the overall function of all program parts involved in the distributed computing. In the second level, the communication between processes is displayed. Because a program part can split into several processes and these processes can run concurrently with processes from other program parts, the second level therefore gives a user a clear picture of co-operations among all (or a selected group of) processes. In the third level, events related to some particular communication and a partial ordering among these events are displayed. The forth level is the lowest level: it displays the relevant statements that carry out some particular communication. The first three levels help the localisation of bugs and the forth level helps the analysis/fixing of bugs.

The reminder of this paper is organised as follows: Section 2 presents an example to illustrate our layered analysing and debugging of distributed programs. Section 3 describes the design issues of the tool. Section 4 describes issues related to the tool's implementation. Section 5 presents some related work, Finally, Section 6 summaries the paper.

2 An Example

We use a simple debugging example to illustrate the layered analysing and debugging process. When teaching "Distributed Computing" course, a student asked me to find out why his exercise program did not work properly. The program was a "Send-and-forward" system. The server (named as `Server`) acted like a message storage. A user used the client program (named as `Client`) to send a message (with a receiver's name) to the server. The server kept the message until the addressed receiver (also a client program) asked the server to forward messages.

We started the server part of the student program first. Then a client part was used to send a message to the server. The normal message exchange should be as follows:

1. The client sends a request to the server.

2. The server processes the request and sends an acknowledgement back to the client.

3. Upon receiving the acknowledgement, the client sends the message to the server.

4. The server sends an acknowledgement back to the client after receiving the message.

When the student's program was executed, both the server and client program parts were hung up. It was difficult to guess what was happening inside these two program parts. We then used our tool to record the events of the program and obtained the top-level diagram of Figure 1.

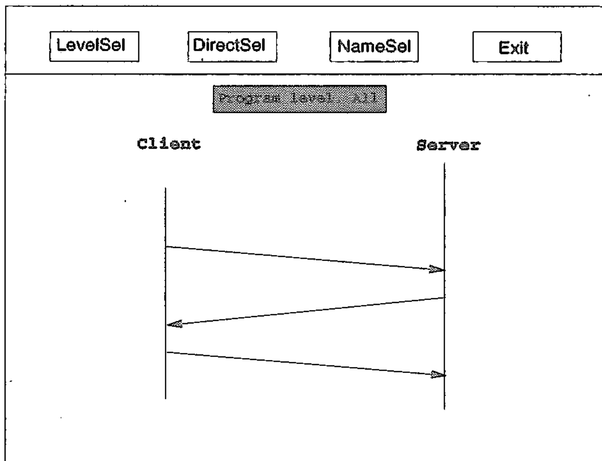

Figure 1: Debugging: The Level 1 Picture

From this diagram we know that the client sent the request, the server acknowledged and then the client sent the message. We further used the second level to locate the processes that were responsible for the failed communication. Figure 2 is the process level diagram.

Three processes were involved. For better understanding of the communication we asked the tool to display event tables for all these three processes. Table 1, Table 2 and Table 3 are these tables (where E represents *Event Symbol*). The second column of these tables lists the event symbols. The correspondence of these event symbols and their detailed event names (see Section 3.1) is given in Table 4.

From the event tables we knew that process 6917 (client process) sent a request to the server process 6911. Then process 6911 forked a child process (6922) to manage the communication with the client. Process 6922 then acknowledged to process 6917. After that process 6917 sent the message to process 6922. It seemed that process 6911 worked normal. What we needed to know more was the communication details between process 6917 and process 6922. So the third level diagram of Figure 3 was used to show the event relations of these two processes.

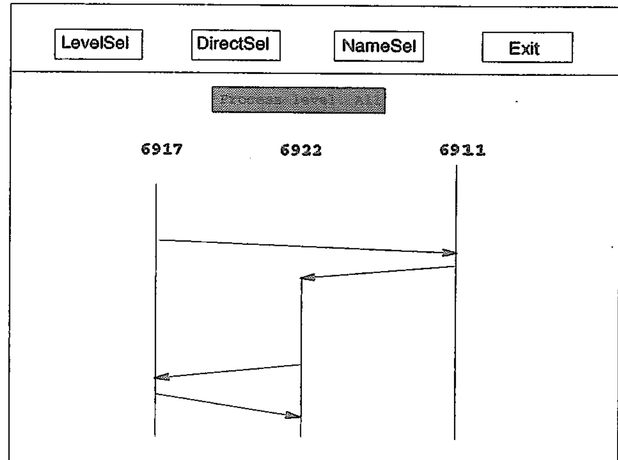

Figure 2: Debugging: The Level 2 Picture

No.	*E*	*Meaning*	*Pred*	*Succ*
1	a_1	Server begins	-	-
2	a_2	Create a socket	-	-
3	a_3	Bind to a name	-	-
4	a_4	Receive request	6917.4	-
5	a_5	Fork a child process	-	6922.1
6	a_6	Forced exit	-	-

Table 1: Event Table of Process 6911

From the level 3 diagram, we knew that the possible error locations were:

1. The program section between events 6 and 7 of process 6922 (the final acknowledgement was not sent properly).

2. The program section between events 6 and 7 of process 6917 (the acknowledgement was not received properly).

We analysed the program section of possibility 1 using a text editor and found out that inside the message storing function, some value of the client socket address was mistakenly re-assigned. That caused the `sendto()` call of the server child process to send the acknowledgement to an unknown address. After fixed the mistake, the program worked correctly.

No.	*E*	*Meaning*	*Pred*	*Succ*
1	b_1	Client begins	-	-
2	b_2	Create a socket	-	-
3	b_3	Bind to a name	-	-
4	b_4	Send request	-	6911.4
5	b_5	Receive ACK	6922.5	-
6	b_6	Send MSG	-	6922.6
7	b_7	Forced exit	-	-

Table 2: Event Table of Process 6917

Joint Conference of 1996 International Computer Symposium
December 19~21, Kaohsiung, Taiwan, R.O.C.

No.	*E*	*Meaning*	*Pred*	*Succ*
1	c_1	Child process begins	6911.5	-
2	c_2	Close parent's socket	-	-
3	c_3	Create a socket	-	-
4	c_4	Bind to a name	-	-
5	c_5	Send ACK	-	6917.5
6	c_6	Receive MSG	6917.6	-
7	c_7	Forced exit	-	-

Table 3: Event Table of Process 6922

Event symbol	*Event name*
a_1	5f712182b3c1.00.0282c22615000000
a_2	5f712182b3c7.01.0282c22615000000
a_3	5f712182b3d3.02.0282c22615000000
a_4	5f712182b3e1.03.0282c22615000000
a_5	5f712182f1f1.04.0282c22615000000
a_6	5f7127aa23c5.0c.0282c22615000000
b_1	5f7126079344.00.0282c2e0d2000000
b_2	5f712607a109.01.0282c2e0d2000000
b_3	5f712607a10d.02.0282c2e0d2000000
b_4	5f712607a114.03.0282c2e0d2000000
b_5	5f71260df164.04.0282c2e0d2000000
b_6	5f71260df13a.05.0282c2e0d2000000
b_7	5f71290a9843.06.0282c2e0d2000000
c_1	5f712188af41.05.0282c22615000000
c_2	5f712188b0a7.06.0282c22615000000
c_3	5f712188b322.07.0282c22615000000
c_4	5f712188b346.08.0282c22615000000
c_5	5f712188b395.09.0282c22615000000
c_6	5f71218ef563.0a.0282c22615000000
c_7	5f7127aa01d2.0b.0282c22615000000

Table 4: Event symbols and event names

3 Design Issues

3.1 Definitions

Before describing the architecture of the tool, we present several definitions which are important in our discussion.

In order to monitor events we have to assign each event a name. Definition 1 is the method which can uniquely name any event.

> **Definition 1.** The *name* of an event (*event name*) is a "unique ID" (UID), defined as a string of characters and an optional user attached affix. The format is `t.r.h[.affix]` where `t` is the timestamp, which is stamped by the local host; `r` is a sequential number; `h` is the host ID; and `affix` is an optional affix name (a character string) which is defined by a user.

If *en* is an event name, we use *en.t* to denote the timestamp, *en.r* to denote the sequential number, *en.h* to denote the host, and *en.a* to denote the affix of event name *en*, respectively. *En.h* can be used to identify on which host the event happened, and *en.t* denotes the occurrence time (relative to the local host)

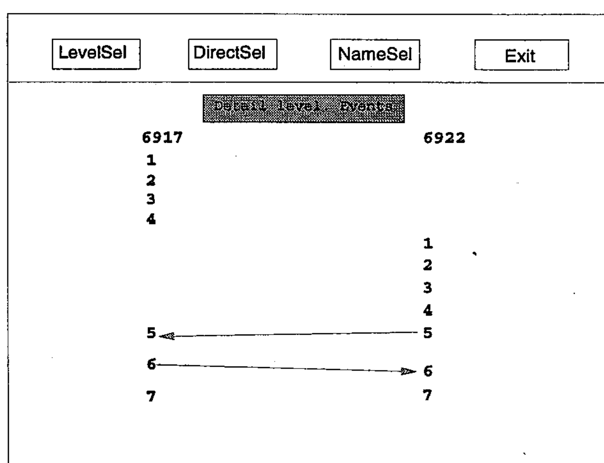

Figure 3: Debugging: The Level 3 Picture

of the event. In the case that n events happened simultaneously at the same host, *en.r* is used to differentiate these n events. It is not difficult to insure that no n (n is a hexadecimal number and $n \leq 255$) adjacent sequential numbers generated are the same. For example, we can have a sequential number generator (which is accessed sequentially by the event name generator) that generates numbers from 0 to 255 each time it is accessed. If the number reaches 255, it goes back to 0 and the circle begins again. So all events of a distributed program can be uniquely identified by using their event names if the maximum number of concurrent events in any host is $n \leq 255$. The assignment and usage of the affix will be described in Section 4.4.

> **Definition 2.** A *preliminary event* e is defined as a pair (f, m). Where f is called a *fact* and m a *message*. A fact is a thing which happens during a program's execution. Notice that not all facts of preliminary events are interesting to a programmer, but they may happen during the program's execution. A fact can be, for example, the creation and destruction of a process, the issuing of a message sending call, or an the issuing of a message receiving call, and so on. A message is the information attached to the fact, such as the parameter values of a message sending call.

The basic relation between preliminary events is the *happened before* relation introduced by Lamport [7]. This relation can be easily extended to cover process creation and termination as well as message-passing events [5]. Definition 3 defines the relationship between preliminary events in our system.

> **Definition 3.** Let $E = \{e_i\}$ be the set of all events of a distributed program. If

event e_1 causes the occurrence of event e_2, or e_2 immediately follows the occurrence of e_1 within the same process, we say that e_1 is a *predecessor* of e_2, and e_2 a *successor* of e_1. Especially, if e_1 and e_2 are in different processes, we call e_1 a *remote predecessor* of e_2 and e_2 a *remote successor* of e_1. This is denoted as $e_1 \leq e_2$. If e_1 is a predecessor of e_2 and e_2 is a predecessor of e_3, then we say e_1 is also a predecessor of e_3.

For example, if e_1 is the event "issuing a request sending call" of a client program, and e_2 is the event "receiving a remote request" of the server program. If e_2 happens because of e_1's happening, then e_1 is a predecessor (remote predecessor) of e_2 and e_2 is a successor (remote successor) of e_1. If e_3 is the event "receiving acknowledgement" of the same client and happens immediately after e_1, then e_1 is a predecessor of e_3 and e_2 is a remote predecessor of e_3. Also, e_3 is a successor of e_1 and a remote successor of e_2. It is easy to know that $(E, \leq)$ is a *partially ordered set.*

Sometimes a user may be interested in the combination of several events. For example, if a server has two remote procedures that will access an object, it is interesting to see if these two procedures are all called during the execution, or to know the execution order of them. We give the following definitions.

Definition 4. Let e_1, e_2 : E and $e_1 = (f_1, m_1)$, $e_2 = (f_2, m_2)$. By $f_1 * f_2$ we mean that the happening of e_2 follows the happening of e_1, that is, $e_1 \leq e_2$ holds. By $f_1 + f_2$ we mean that we cannot tell which of e_1 and e_2 happened first, that is, there is no predecessor relation exists between these two events. We can view "*" as sequential and "+" as concurrent. By $f_1 \cap f_2$ we mean that both e_1 and e_2 occur. By $f_1 \cup f_2$ we mean that either or both e_1 and e_2 occur (if e_i does not occur, we denote that as $\neg e_i$). So we have

$$f_1 \cap f_2 = \begin{cases} f_1 * f_2 & \text{if } e_1 \leq e_2 \\ f_2 * f_1 & \text{if } e_2 \leq e_1 \\ f_1 + f_2 & \text{otherwise} \end{cases}$$

and $f_1 \cup f_2 = f_1 \cap f_2$ *or* f_1 *or* f_2. Similarly, by $m_1 * m_2$ we mean that message m_1 is followed by m_2 and by $m_1 + m_2$ we mean that two messages are independent each other.

Definition 5. If $e_1 = (f_1, m_1)$ and $e_2 = (f_2, m_2)$ are events, then

$$\begin{aligned} e_1 \cap e_2 &= (f_1 \cap f_2, m_a), \\ e_1 \cup e_2 &= (f_1 \cup f_2, m_b), \\ e_1 * e_2 &= (f_1 * f_2, m_1 * m_2), \\ e_1 + e_2 &= (f_1 + f_2, m_1 + m_2) \end{aligned}$$

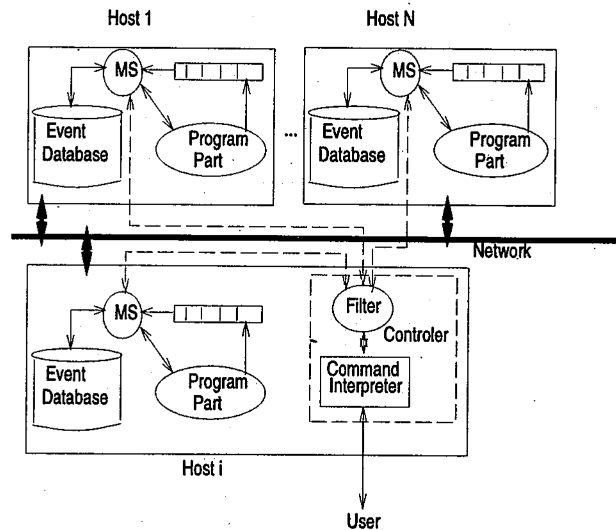

Figure 4: The Monitor Structure

are also events. Where

$$m_a = \begin{cases} m_1 * m_2 & \text{if } e_1 \leq e_2 \\ m_2 * m_1 & \text{if } e_2 \leq e_1 \\ m_1 + m_2 & \text{otherwise} \end{cases}$$

and

$$m_b = \begin{cases} m_a & \text{if } e_1 \cap e_2 \\ m_1 & \text{if } e_1 \textit{ and } \neg e_2 \\ m_2 & \text{if } e_2 \textit{ and } \neg e_1 \end{cases}$$

We call these new events *combined events.* Combined events are usually defined by programmers.

The priority of the above operators is, from high to low, $\cap$, $\cup$, $*$, $+$. So the expression $e_1 * e_2 \cap e_3 \cup e_4 + e_5$ is actually $(e_1 * ((e_2 \cap e_3) \cup e_4)) + e_5$. It can be proved that if E is the set of all (preliminary and combined) events of a distributed program, then $(E, \leq)$ is still a partial order set.

3.2 The Structure

The tool consists of a controller and a group of managing servers. The controller has two main parts: a user interface (including a command interpreter and I/O) and a filter, while a managing server (MS) consists of a server and an event database. Each host which has program parts being analysed/debugged on it has a managing server. The controller can be invoked at any host. By communicating with each related MS, the controller can present the monitored results to the user. Of course, several controllers can be invoked by several users simultaneously. Figure 4 illustrates the structure of the tool.

Three steps are taken during analysing and debugging. At the *monitoring* step, all events that happened on one host are monitored by the local MS and

recorded into the local event database. All preliminary events including communication-oriented calls, process forks and exits in both client and server program parts are monitored, and a user can also define some combined events through the Event Definition File (EDF) and let the monitor to record them. After all events are recorded, the programmer uses the *ordering* step to order events. At that time, each MS exchanges remote predecessor / successor information through the controller and has all remote relationship ordered. Then, local predecessor / successor relationship is established by each MS over its local event database respectively. The last step is *debugging*. By combining the results on all related event databases, the filter can present the execution trace of the distributed program in several levels.

4 Implementation Issues

4.1 Debugging Library

No doubt an effective distributed debugger has to be deeply embedded into the operating system or even has the help of dedicated hardware components for achieving sufficient speed and transparency. Because of the difficulty of modifying operating systems and obtaining hardware support, most debugger and monitor researchers use software techniques as a substitute. This makes the implementation much easier, but the performance is not very good, especially for real time systems (may be even completely not suitable for real time programs). At this stage, we provided a *debugging library* which has to be linked with a program being debugged. This library provides replacements for the following BSD4.3 UNIX operating system calls:

```
fork()       : create a child process
signal()     : signal a process
kill()       : signal a process
socket()     : create a socket
bind()       : bind a name to a socket
listen()     : listen for a connection
accept()     : accept a connection request
connect()    : initiate a connect
close()      : close a socket
write()      : send out a message
read()       : receive a message
send()       : send out a message
recv()       : receive a message
sendto()     : send out a message
recvfrom()   : receive a message
sendmsg()    : send out a message
recvmsg()    : receive a message
```

These replacements first perform some work required by the tool, such as reporting to the local MS of the event's happening, and then do the normal work of the original calls. After the programmer thinks the program has been debugged, the program can be re-linked with ordinary libraries.

A utility program is used to change all the above system calls of a program into their corresponding debugging library calls (the replacements). This is done by converting the name of a call into a string of capital letters. For example, a `fork()` call is converted into a `FORK()` call, which is a debugging library call. After the pre-processing, the program can be compiled normally and linked with the debugging library. Another utility program is used to change all the debugging library calls back to normal system calls after the debugging.

4.2 Managing Server

On each host, there is a managing server (MS) which consists of a server and an event database. Each entry of the event database stores an event and includes the following fields:

name	event name. A UID
pid	process ID
pgm_name	program part name
p_name	predecessor event names. A UID array
s_name	successor event names. A UID array
fact_info	fact information
message	message part of the event

The fact information of an event is a character string that provides readable information of the fact. For preliminary events, if the user does not provide fact information, such information will be assigned by the debugging library. For example, they may be "begin sendto() call" or "fork new process", etc. Otherwise the user provided character string is used. For combined events, they are assigned by programmers. The message part of an event is stored as a string of bytes and type information. The filter uses the type information to illustrate the byte string and displays the result to the user through the command interpreter.

An MS has the following functions:

1. Database management. Responsible for the management of the local event database. The database is protected by the MS and any access of it must go through the MS.
2. Event logging. When an event occurs, it is logged by the local MS into the event database. Because the events may happen very fast (or even concurrently), while the logging of an event requires some amount of time, we use an event queue to queue up all events waiting to be inserted into the database. All events are queued into a event queue after their happening, and the local MS looks at the queue and puts the events into the event database.
3. Communicating with the controller. All communications among MSs are conducted by the controller. A lot of commands are issued by the controller and performed by MSs. This is the only way a user can access the event database.

Storing all events of a distributed program execution may cost too much memory, especially for large programs. Facilities are provided to allow a user to select and store events that may be interesting. Also a user can let the monitor store only the fact parts of some events and whole information of other events.

4.3 Controller

The controller consists of a command interpreter and a filter. The command interpreter accepts and analyses commands from a user and controls the filter to perform the required functions.

The filter has three main functions. Firstly, it maintains the communication between the command interpreter and MSs. After the command interpreter accepted and interpreted a command, it is passed to the filter to have the appropriate MSs to execute the command. Then the results of the execution are interpreted and passed to the command interpreter through the filter. secondly, the filter maintains the communication between the MSs. In that case the programming of an MS is much simpler. The last function of the filter is to interpret the message part of an event. When the user requests to view the message part during debugging step, the filter will find the appropriate message and use the type information to illustrate the byte string, and then pass the result to the command interpreter.

4.4 Event Definition File

All preliminary events are automatically logged by the tool if the program being analysed/debugged is linked with the debugging library. Sometimes a user may find it is more convenient to define some new events during debugging. The *Event definition file* (EDF) is used for that purpose. The following is a very simple EDF which defines a combined event `BothCreated` as the intersection of two preliminary events `CreateSock1` and `CreateSock2`. That is, if both preliminary events happened, then the combined event also happened.

```
BEGIN
  preliminary Event:
    CreatedSock1, CreatedSock2;
  Combined Event:
    (BothCreated, "Both sockets are created");
  BothCreated = CreatedSock1 ∩ CreatedSock2;
END
```

Several steps are needed to use an EDF. At first, the user inserts into the program parts being analysed/debugged an *affix definition function* before each preliminary event which is to be used in the EDF. The format of the affix definition function (defined in the debugging library) is `affix_define(affix)`, where `affix` can be any character string. This string will be appended to the preliminary event name when the event happens. Secondly, the combined events are built by using these `affix` names, and the EDF is read and evaluated by the controller. When any of these affixed preliminary events happens, they are sent to the controller by the local MSs (of course, the local MSs also record them as usual). The controller then evaluates the combined events expressions and records them into the event database if any of them is true. The predecessors of a combined event are all the events (preliminary and / or combined events) in the right hand side of the event expression.

4.5 Ordering Events

As we have known, the time system of each host in an LAN is not synchronised, so we can not use the timestamp in definition 1 to order all events. But the timestamp can certainly be used to order events happened in one process because they always remain within the same host. That is, events within each process of a distributed program can be fully ordered. But it is impossible to fully order events of different processes. As mentioned earlier, there do exist some partial ordering relationship among these communication and process fork events. The following steps are used to establish the partial ordering among all events:

1. Communication related predecessors. When a communication related event happens (for example, the issuing of a `sendto()` or `recvfrom()` call), it will cause the happening of an event which belongs to another process (and also possibly, on another host). In that case, the first event is changed (by the debugging library) to carry not only the original information, but also the event's name. On the other hand, the second event is also changed (by the debugging library) to not only receive the original information, but also the first event's name, and this name is stored by the local MS as the predecessor of the second event.

2. Process fork related predecessors. When a process fork event happens, it will cause a new process to be setup and executed. This event is changed (by the debugging library) to carry the name of the event, and the first event of the new process will use the carried name as its predecessor event.

3. Combined event related predecessors. When an event with an affix definition happens, it will cause the controller to evaluate the related combined event expressions. So, the name of the event is sent to the controller and stored as one of the predecessors of the related combined events.

4. Form all remote successors. In (1), (2), and (3), all remote predecessors will be established after the termination (normal or forced) of the programs being debugged. The remote successors are built by each involved MS and the controller at this moment. Each MS checks all events in its local database. If its event e has a remote predecessor named f, then the MS will be responsible for storing e as f's successor. It is easy if e and f are in the same database (for example, the `fork`

events). Otherwise $f.h$ is used to locate the MS it belongs to and f's successor will be stored by the communication of these two MSs through the filter.

5. Form all other successors and predecessors. All the events within a process are ordered by their timestamps and their predecessors / successors are stored by using this order. In one process, the predecessor of event e is event d if $d.t$ is immediately less than $e.t$. And the successor of e is event f if $f.t$ is immediately greater than $e.t$. This ordering is performed by each MS concurrently.

By using the above method, all communication events can be partially ordered by predecessor / successor relations. For the events within a single process, we can fully order them by their timestamps. Combining these two relations together, we can have some partial ordering over all events of a distributed program.

4.6 Layered Debugging

After the monitoring and ordering steps, the user goes into the third step, the debugging. As we mentioned before, the first thing of debugging is to locate the bugs. A top-down view of the program is a suitable way of localising bugs.

At the top level (program level), communication between program parts is displayed. A distributed program may have many program parts. The tool provides a facility to let the user select program parts for display. Because in each event entry, there is a field pgm_name specifies the name of the program part (Section 4.2), we use this information and the event ordering to draw the communication diagram between program parts. At this level, no event detail is displayed. The user can have a nice top-view of the program communication.

From the top level, the user may have some idea that which part of the program probably has a bug. So some relevant processes can be selected and the second level (process level) will display the communication between these selected processes. If nothing can be found at the top level, the user can ask the tool to display all the processes in the second level. We also use a field (pid, see Section 4.2) in every event entry and the event ordering to draw the diagram.

Usually from the second level, some processes can be selected for further investigation. The third level (detailed level) displays the events and communications between selected processes (of course, it can also display all the events relations of the program). In this level, the event numbers are used and the user can consult these numbers with the event tables (see below).

From the third level, the bug locations will be found and the relevant program segments will be displayed using a text editor in the forth level (text level). The user can analyse the program segment and fix the bug here.

During the displays, the user can view event details in two levels. At the *table level*, the events of a selected process is displayed in a table form (*event table*). Then the user can select an event within an event table and ask the tool to display its details (*detail level*).

5 Related Work

A lot of techniques have been derived for programmers to improve the debugging process. There are two major approaches: *debugging with repeated execution* of the program (or *cyclic debugging*) and *debugging with trace* of program execution [9]. In cyclic debugging, a user executes the program in a controlled manner until an error is detected. The program can be re-executed to produce the same execution behaviour. This is a very convenient way for small programs or programs that have little communication among their concurrent parts. But for a larger distributed program, executing the entire computation several times while repeatedly setting breakpoints may be very costly. Also, sometimes the re-execution of a distributed program may not result in the same behaviour because of the nondeterministic characteristics. In debugging with trace, no reproducibly behaviour is needed. The generated trace is no longer nondeterministic and can be analysed in any controlled ways that a programmer prefers. But the generating of the trace may be very costly in time and space, and also the events in the trace are still not fully ordered. Some techniques are needed to analyse the program trace. In this paper, the latter method is used.

Event-Based distributed debugging has been widely discussed. Bates and Wileden [2] used a method called *behavioural abstraction* to hierarchically define higher level events in terms of sequences of primitive events (such as process creation, page fault, and message exchanges). In his latter work [1], Bates described a system (EBBA) for debugging on heterogeneous distributed systems based on his behaviour abstraction.

In a distributed system, sometimes the re-execution of a long program is very costly. *Replay* is a technique that allows a user to examine the course of an erroneous execution *without* re-executing the program [12]. LeBlanc and Robbins [8] provided some degree of replay in their debugger. After the collection of all events, the events are displayed sequentially. Both single step and continuous display are supported.

Debugging a distributed program can be divided into two phases. At the first phase, called *localisation*, we need to locate which part of the program has a bug. Then at the second phase, called *analysis/fixing*, we analyse the code that may cause the bug and fix it. Unfortunately almost all existing distributed debug-

gers only provide communication events occurred in lower level. They assist in fixing a bug after having obtained a rough idea about the bug's localisation. This only provides information for the second debugging phase. To dig out the possible bug locations using the existing debuggers is a difficult job as there are usually many events involved.

A project of high-level debugger for parallel programs is described by Caerts *et al* [3]. They use several *abstraction levels* (from the coarse-grain interacting processes or threads to textual representation of the program) in their debugger. A top-down method following the abstraction levels is used to locate a possible bug. But the debugger is limited to message-passing on shared memory systems.

There have been many efforts in visualising distributed and parallel program execution to facilitate the understanding of these programs [13] [11]. However, most of these tools are too complex to use and therefore are not useful for novices. These existing tools usually do not allow easy study and experimentation with programs. Some existing tools provide graphical pictures of algorithms or data structures in action [10] [6], but do not display the structures and source code of programs.

6 Summary

The design and a preliminary implementation of a tool for analysing and debugging distributed programs is described in this paper. The tool has several managing servers which record the events of program parts of their hosts into their local event databases. By using an ordering scheme, all events of a distributed program can be partially ordered, and the event graphs in different levels and the relevant event tables can be built. These event graphs and tables are then used to locate the possible bug positions in a top-down manner. Facilities are also provided to define combined events and to view the details of the events. The tool has been regularly used in analysing and debugging student assignments on distributed programming since 1993.

References

[1] P. Bates. Debugging heterogeneous distributed systems using event-based models of behaviour. *ACM Transactions on Computer Systems*, 13(1), February 1995.

[2] P. Bates and J. Wileden. An approach to high level debugging of distributed systems. *SIGPLAN Notices, Proceedings of SIGSOFT/SIGPLAN Symposium on High Level Debugging*, 18(8), August 1983.

[3] C. Caerts, R. Lauwereins, and J. A. Peperstraete. A powerful high-level debugger for parallel programs. In *Lecture Notes in Computer Science*, volume 591, pages 54-64. Springer-Verlag, Germany, 1992.

[4] W. H. Cheung, J. P. Black, and E. Manning. A framework for distributed debugging. *IEEE Software*, pages 106–115, January 1990.

[5] C. Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, August 1991.

[6] M. T. Heath, A. D. Malony, and D. T. Rover. The visual display of parallel performance data. *Computer*, 28(11):21–28, November 1995.

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

[8] R. LeBlanc and A. Robbins. Event-driven monitor of distributed programs. In *IEEE 5th Distributed Computing Systems*, Colorado, May 1985.

[9] B. P. Miller and J. Choi. A mechanism for efficient debugging of parallel programs. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, Atlanta, Georgia, June 1988.

[10] T. L. Naps. An object-oriented approach to algorithm visulisation – easy, extensible and dynamic. *SIGCSE Bulletin*, 26(1):46–50, March 1994.

[11] R. Samtaney, D. Silver, N. Zabusky, and J. Cao. Visulisation features and tracking their evolution. *Computer*, 27(7):20–27, July 1994.

[12] R. S. Side and G. C. Shoja. A debugger for distributed programs. *Software – Practice and Experience*, 24(5), May 1994.

[13] J. P. Singh, A. Gupta, and M. Levoy. Parallel visulisation algorithms: Performance and architectural implications. *Computer*, 27(7):45–55, July 1994.

[14] W. Zhou. Prototyping and debugging remote procedure call programs. In *Proceedings of the Australian Software Engineering Conference '90*, pages 333–338, Sydney, Australia, May 1990.