A DEBUGGING TOOL FOR DISTRIBUTED SYSTEMS

Hans Scholten and John Posthuma
University of Twente, Dept. of Computer Science
Enschede, the Netherlands

ABSTRACT

This paper describes parts of the design of a debugger for a distributed real-time multimedia system. Emphasis lies on the distributed aspect of debugging, which means that attention is paid to the external behaviour of the processes. This type of debugging is useful to find communication or synchronization errors. However, experience learns that this is not enough: the debugger must also provide hooks for the user to use traditional sequential debuggers. This type of debugging focuses on the internal behaviour - or: internal logic - of processes. For the sequential debugging part a normal debugger like GDB can be taken. Three key elements of the debugger are events, filters and recognizers. By definition events are the lowest level of system activity that may be observed by the event debugger. Filters are applied to remove events from the stream of events produced by the debuggee that are of no interest for the programmer. Recognizers are used to recognize behaviour -right or wrong- of the system. By combining events, different levels of abstraction are introduced, thus alleviating the task of the programmer.

KEYWORDS

Distributed systems, debugger, event debugging, behavioural abstraction, communication and synchronization errors

INTRODUCTION

Debugging is an essential step in developing a software system, since every nontrivial system contains bugs. However, no precise or elegant method has been developed for the debugging process; debugging is generally considered an art rather than a science. The introduction of a distributed environment makes the situation even worse, since this both complicates the debugging process and gives rise to new types of bugs. In recent years, researchers have developed some helpful debugging techniques, like event based debugging, for distributed environments.
With respect to debugging the following points are characteristic for distributed systems:

- It is difficult (if not impossible) to get a global state at one point in time.
- A distributed system has the tendency to be large and to have a large state space, which raises the problem of manipulating large quantities of state data for debugging at execution time, and the analysis of it.
- Distributed asynchronous systems are nondeterministic: two executions of the same system may produce different, but nevertheless valid, orderings of executions.
- Monitoring a distributed system alters its behaviour.
- Error latency is a problem in distributed debugging. Usually, there is a lag between the occurrence of an error and its discovery. This lag exists in sequential programs, but it is even worse in distributed ones.
- Interactions between the system and the system developer can be complex. The programmer must analyse the information generated by all the interacting processes. To make this possible a good (graphical) user interface must be provided.

EVENTS

The choice of events determines the lowest level of system activity that is possible to observe and implies a particular viewpoint on the system. According to Bates [Bates89] the rule of thumb is that each call or operation on implementation routines and structures is a candidate for a primitive event. For example, system application programmers would require a set of primitive events resulting from invocation of basic system services (system calls), e.g. task_create or file_open. However, there is a problem when only using system calls. The programmer cannot specify other types of events and is restricted to the lowest type of event in the system. A more flexible solution is to allow the programmer to define his own events. An example of such events are function calls operating on a shared structure.
In DEBUT, the debugger that will be discussed, events are generated whenever a call to the operating system or to user defined functions is made. This is accomplished by means of probes that substitute the original call by a call to a special debug library:

```
send_message(destination, data)
```

```
-> probe(send_message, destination, data);
```

An event instance records information when some event occurs. An event instance must first of all contain a field indicating what type of event it is. It must also contain information about the location and the identification of the process that generated it. This means the identification of the node, and also the task and thread identification (and maybe the location in the source code, if it does not cause too much overhead).
Another important aspect is the time at which the events occur, which is necessary to order the events. A timestamp will be used to order events. Finally, the event instance must also contain event specific information. An event will often be a function call. These calls have in, out or in/out parameters and each call results in an result value. These parameters are different for each function call and therefore cause event instances of variable length. The parameters will be stored together in one event. But the difference between in and out parameters indicate that a call consists of two parts: the request and reply. In some cases it can be useful to record these two parts as separate events. This is for example the case when a call blocks for a long time. The first event is already available, but the user does not see it, because the debugger waits for the result. In case of a deadlock, the user never sees the call that caused it. However, this will increase the number of events with 100 percent, increasing the needed processing time and thus the probe effect. That is why only one event per call will be generated.

This means that with one event two timestamps will be associated: one at the beginning of the processing and one at the end. This can be used to see how long the call lasted (or to see how long a process was blocked by the scheduler as a result of the call). Another reason for generating only one event per function call is that a call has only effect after it has been executed. The formats of the event instance is described below.

```
event = <
    event_type,
    begin_time, end_time,
    node_id, task_id, thread_id, code_location,
    <all in and in/out parameters of the call>,
    <all out and in/out parameters of the call>,
    system_call_result
```

Events are generated by probes that execute in user space and are added to the user program. This requires re-compilation [Vliet92].
A probe works as follows. When a function call (or system call) is made, not the real function is called, but a substitute. In this substitute function, important information about the function call is saved, like the type of call, the time and the parameters. After this is done, the actual function is called. When it returns to the substitute function, the time and the return values are saved. Now the event occurrence is complete. It will be sent to the local debugger, which resides on every node, where it will be processed.

FILTERS AND RECOGNIZERS

Filtering is applied in almost every event-driven debugger. The debugger produces a stream of events generated by the debuggee (the program that is being debugged). The purpose of filtering is to remove events from the stream that are of no interest to the programmer. It is of course possible to use no filters at all so that all generated events will be available. But filtering has three major advantages:
- It becomes easier to analyse the event information (reducing the state space).
- Less storage is required. The user is usually interested in a small part of all possible events.
- Less information has to be kept or transmitted. This is advantageous since this reduces the intrusion on the debuggee.

The debugger supports user-defined filter definitions, which are based on the event format. Only events that satisfy some filter definition during execution are passed.
Filter definitions consist of three parts:
- Selection part. This part contains the type of event the user is interested in.
- Condition part. In this part the user can define restrictions for all the fields of an events instance. A field can be tested against a constant value using one of the compare operators >, <, =, !=, >= and <=. Fields may also be compared with each other.
- Exclusion part. In this parts some of the fields can be excluded from the event instance. This is to reduce the influence on the debuggee and the storage requirements. It can for example be useful when only the fact that a message is sent is of importance and not the actual data.

The filtering technique is used to make the events generated by the debugger more manageable. But even when filtering is used, the data can be too voluminous for the programmer. That is why another technique will be used to help the programmer: recognizers. A recognizer recognizes the behaviour of distributed programs and copes with the complexity and concurrency of distributed systems. It is used to analyse the interaction between processes and thus to alleviate the task of the programmer. This interaction can be seen as the behaviour of the system.
Behaviour of systems can be described in numerous ways. In papers written on the subject of distributed debugging, a number of methods are proposed for describing behaviour of systems. Examples of these methods are: behaviour specifications [BFMSV83], interval logic [HHK85] and behavioural abstraction [BW82] [BW83] [Bates87] [Bates89]. The recognizer in this framework will have similarities with behavioural abstraction. Behavioural abstraction allows the behaviour of two or more processes to be combined. (However, in most cases it will suffice to specify the behaviour on a per process basis.)
Recognizers may be used in two ways. The first one is just to observe the behaviour of the system, which is performed by building high-level events from lower-level events: behaviour gets summarized.

The second task of recognizers is to specify important points in the execution of the system. If, for example, the programmer wants to know the value of variable x after the event sequence a;b;c he can use a recognizer. The recognizer could return control to the programmer, so he can retrieve the value himself. But to ease the task of the programmer, actions can be specified in advance: each recognizer can have one or more actions connected to it.

```
SEQUENCE a ; ( b | c )
DO action1 ; action2 ; action3
```

With the recognizer as in the example above, it is only possible to specify correct behaviour of the system. Take for instance a recognizer that has to recognize the sequence a;b;c, and suppose that event d is not allowed during this sequence. If the recognizer has seen event a, it will wait until it sees event b. If now event d takes place the recognizer will stay in the same state. The user cannot specify this extra condition (unless by specifying all possible bad behaviours). An extra part is added to the recognizer definition, which is called the verify part, where events are specified that must or must not occur between the recognition of the first and last event of a sequence. The verify part has its own actions related to it:

```
SEQUENCE a $begin_verify 1 ; b ; $end_verify 1 c
DO action1 ; action2 ; action3
VERIFY 1 d
DO action4 ; action5 ; action6
```

The verify conditions described above only concern the presence or absence of event occurrences. This however, is not the only information that can be used to verify that the system behaves correctly. Information provided by sequential debuggers can be used like the values of variables or expressions. This possibility is given by the MuTEAM debugger in [BFMSV83].

OVERVIEW OF THE FRAMEWORK

In figure 1 a global overview is given of the framework for the debugger. The different parts of the debugger will only be described briefly, due to limitations in the available space here. Of these parts only the LDBs and the probes reside on the same nodes as the debuggee. To reduce the probe effect the other parts are placed on free nodes.
The Graphical User Interface (GUI) is an interface between the user and the actual debugger. Its task is to offer the user easy control over the debugger and provide some useful functions. This interface is described in [Berendsen92].
The GUI interacts with the controller. This controller takes care of the interaction between the GUI and the other parts of the debugger: the global debugger (GLDB) and the database. But other tools could be plugged in to the controller like the preprocessor for adding probes to the application, a normal command tool or a window to display/edit source code.
The main task of the database is to store the events generated by the application program. These events can be used for a replay so that the user can inspect the system behaviour at his own speed. The events in the database can also be used to answer queries about the system performance.
The GLDB is the main part of the distributed debugger. It receives commands from the controller. It must process these commands and send commands to the local debuggers (LDBs). The GLDB collects the resulting replies and send them back to the controller. The GLDB thus is a distributor that communicates with the LDBs.
The LDBs are installed on every node where processes run that have to be debugged. This means that only the LDBs (and the probes) run on the same nodes as the debuggee. The LDBs perform the commands of the GLDB and send back the results.
The probes are part of the user process. They provide the events on which the debugger is based.
Sequential debuggers are added to this framework for two reasons. First, the event-based approach is not enough to locate a bug. It is a technique to narrow down the area of the bug. After unexpected events are found, the programmer must interact with the sequential debugger to exactly locate the error. By integrating the sequential debugger in the design, the user can simply request to connect a such a debugger to one or more tasks. The second reason is that it is allowed to specify sequential debugger commands as actions of the recognizers. For this, the sequential debugger must be connected to the LDB. Possible actions, including

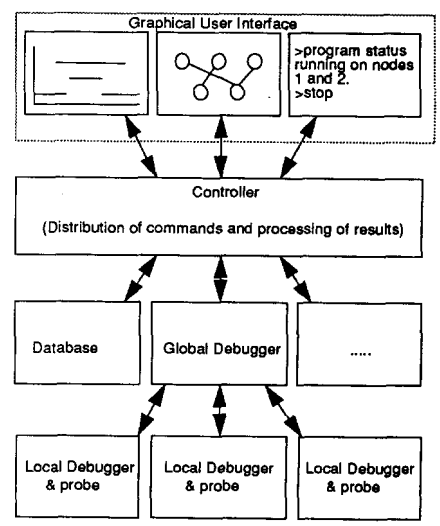

FIGURE 1: Overview of the framework

sequential debug actions, are:
- Add new filter definition.
- Remove existing filter definition.
- Add new recognizer definition.

- Remove existing recognizer definition.
- Return task (thread) status.
- Start/stop executable.
- Stop/resume/delay task (thread).
- Quit LDB.
- Other commands for the sequential debugger.

CONCLUSION

In this paper a debugger for a distributed (real-time) system is presented. The project started in 1991, but because the debugger is based on an earlier one, designed for a parallel system [Scholten90] [Sauer90], already some results can be shown.

Main features of the debugger are modularity with pluggable units, the programmable recognizers and filters. Filters are used to restrict the number of events that reach the debugger. Recognizers recognize the behaviour of a (distributed) program and cope with the complexity and concurrency of distributed systems. Interaction between processes can be analysed and acted upon.

Some parts are implemented, these are the recognizers and filters and parts of the user interface. The probes are being programmed and tuned at the moment.

The sequential debugger is adapted in such a way that it fits under the graphical user interface. Furthermore a prototype of a "back-trace" mechanism is added to the sequential debugger, which enables the programmer to undo instructions that are executed.

Future work will include the implementation of the database and research in the field of presentation and analysis of event histories.

REFERENCES

[Bates87] Peter C. Bates. *Shuffle Automata: A Formal Model for Behaviour Recognition in Distributed Systems*. COINS Technical Report 87-27, University of Massachusetts, January 1987.

[Bates89] Peter Bates. Debugging heterogeneous distributed systems using event-based models of behaviour. *Proceedings of the ACM SIGPLAN/SIGSOFT Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):11-22, January 1989.

[Berendsen92] Eric Berendsen. *A Graphical User Interface for the Huygens Debugger*. M.Sc. Thesis, University of Twente, Department of Computer Science, August 1992.

[BFMSV83] F. Baiardi, N. de Francesco, E. Matteoli, S. Stefanini, and G. Vaglini. Development of a debugger for a concurrent language. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, published in ACM SIGPLAN Notices*, 18(8):98-106, August 1983.

[BW82] Peter C. Bates and Jack C. Wileden. EDL: a basis for distributed system debugging tools. In *Proceedings of the Fifteenth Hawaii International Conference on System Sciences*, pages 86-93, January 1982.

[BW83] Peter Bates and Jack C. Wileden. An approach to high-level debugging of distributed systems. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, published in ACM SIGPLAN Notices*, 18(8):107-111, August 1983.

[Elshoff89] I.J.P. Elshoff. A distributed debugger for Amoeba. *Proceedings of the ACM SIGPLAN/SIGSOFT Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):1-10, January 1989.

[HHK85] Paul K. Harter, Jr., Dennis M. Heimbigner, and Roger King. IDD: an interactive distributed debugger. In *Proceedings of the Fifth International Conference on Distributed Computing Systems*, pages 498-506, Denver CO, May 1985.

[Lamport78] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7), July 1978.

[Oracle89] Oracle Corporation. *Pro*C User's Guide, version 1.1*. 1989

[Sauer90] Frank Sauer. *Design and Implementation of a Debugging Tool for TUMULT-64*. M.Sc. Thesis, University of Twente, Department of Computer Science, August 1990.

[Scholten90] J, Scholten. On debugging in parallel systems. In *Proceedings of Tencon'90*, pages 264-268, Hong Kong, September 1990.

[Shaw80] Alan C. Shaw. Software specification language based on regular expressions. *Software Development Tools*, pages 148-175, Springer Verlag, New York, 1980.

[Snodgrass84] Richard Snodgrass. Monitoring in a software development environment: a relational approach. *Proceeding of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, published in ACM SIGPLAN Notices*, 19(5):124-131, 1984.

[Stevens90] W. Richard Stevens. *UNIX network programming*. Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[Vliet92] Edwin van Vliet. *Probes for the Huygens Debugger*. Thesis, University of Twente, Department of Computer Science, June 1992.