Monitoring Multimedia Systems

J.Scholten and J. Posthuma

University of Twente, Dept. of Computer Science
P.O.B. 217, 7500 AE Enschede, the Netherlands

Abstract

As part of a multimedia project research is being done in the area of monitoring and debugging. The monitor to be discussed in this paper is intended for use in distributed systems, in particular in a network of multimedia workstations. The monitor is the first phase in the development of a multimedia debugger. It taps (selected) information streams, tasks, communications etc. The monitor then displays assorted information on a graphics display and/or stores it in a database for later display and analysis.
The debugger is not concerned with 'low level' entities in the system, like values of variables, or contents of registers. This kind of information is handled by a source level debugger which is executed from within the 'high level' debugger. This approach ensures a consistent userinterface for the debug system.

Introduction

In a complex distributed environment 'normal' debuggers offering low-level facilities often fail to work due to certain proporties of the system:

- the absence of a global clock makes it difficult to construct ordered traces without introducing substantial overhead,
- the system is nondeterministic. Reruns of a program, necessary to locate a bug, may give different behaviour,
- a huge amount of information is retrieved by monitors. Suitable abstractions must be used in order to make it comprehensible to the programmer.

Additionally, in distributed systems classical debuggers do not present important data a programmer might be interested in, like process information or information on communication between processors.
Another problem is the tapping of information out of the system. This will influence the behaviour of the processor because a (software) tap takes processor cycles. This is called the probe effect [1]. A bug that occurs during normal execution of a program might never be reproduced while under the debugger's control [2]. Perhaps a solution to this problem, be it an expensive one, is described in [3], where extra hardware is used to implement a non-invasive monitor for a realtime distributed system. A major disadvantage is that not all events may show on the bus. Circuit integration and use of caches reduces bustraffic and thus the number of events on the bus. The use of manufacturer provided processor hooks circumvent these problems. This approach is taken in the Flight Recorder [4] where the coprocessor protocols are used for event tracing.

Clocks

Events are collected per processor and then sent to a central point where they are ordered in time. To get a proper ordering of events a global system time is needed. Fidge [5] shows that the logical clock described by Lamport [6] is not suitable for a distributed system, because it will implicitly order the events totally and causality will be lost. The partial ordering by the logical clocks of Fidge could be a solution, but are not very suited for our system because, since processes are created and deleted dynamically, the number of processes is not constant. Therefore the vector used for timestamping all communication is variable in length and may become quite long. This will result in a relatively large communication overhead. It will even get worse when the system is scaled to thousands of workstations.
To minimise communication overhead the solution of a global clock is proposed, in which case all events are times-

0-8186-2755-7/92 $03.00 © 1992 IEEE

tamped with a time that is equal in all workstations connected to the system.

The mechanism to accomplish the global clock is quite simple. The accurate time is transmitted by the Physikalish-Technische Bundesanstalt Braunschweig in Germany. Broadcaststation DCF-77 has a range of over 1000 kilometers and may be received everywhere in Europe. The time, even corrected for daylight saving time, is transmitted in coded form. All workstations receive this code and once a minute the local clock is synchronised with the transmitted time, thus ensuring one global time systemwide (at least within Europe). The solution is quite cheap: hardware already exists for personal computers to control the system clock by DCF-77.

Events

The debugger is based on an earlier one, built for a multicomputer [7] It is event-based, where an event may be any interesting action that changes the state of the system. Events are described as:

```
event = <
  event_type,
  begin_time, end_time,
  node_id, task_id, thread_id,
  in, in/out parameters,
  out, in/out parameters,
  result
>
```

`Event_type` specifies the type of event. Event_types of interest are: communication, task and thread actions, scheduling activities, operations on memory objects and processor assignments. `Begin_time` and `end_time` specify the time, or period, the event takes place. `Node_id`, `task_id` and `thread_id` specify the place and identification the event occured. The rest of the definition specifies the data associated with the event.

Communication is done using ports. A port is a simplex communication channel, implemented as a message queue managed and protected by the kernel. A port is also the basic object reference mechanism. Ports are used to refer to objects, operations on objects are requested by sending messages to the ports which represent them. A port set is a group of ports, implemented as a queue combining the message queues of the consistent ports. A thread may use a port set to receive a message sent to any of several ports. A message is a typed collection of data objects used in communication between threads. Messages may be of any size and may contain inline data, pointers to data, and capabilities for ports.

Since ports are used to refer to objects (even memory objects), events should be gathered at the same point where messages are handled. Port sets can be used to collect events of a group of objects, for example, all the events of the file-system.

A task is an execution environment and is the basic unit of resource allocation. A thread is the basic unit of execution. The conventional notion of a process is represented by a task with a single thread of control.

A memory object is a storage object that is mapped into a task's virtual memory. Memory objects are commonly files managed by a file server, but a memory object may be implemented by any object that can handle requests to read and write data.

Threads are the basic unit of scheduling. On a multiprocessor host, multiple threads from one task may be executing simultaneously (within the task's one address space). A thread may be in a suspended state (prevented from running), or in a runnable state (may be running or be scheduled to run). Tasks may be suspended or resumed as a whole. A thread may only execute if both thread and its task are runnable. Resuming a task does not cause all component threads to begin executing, but only those threads which are not suspended [8].

Events are stored as objects in a database and during debug sessions de debugger retrieves data by means of a query language. Snodgrass asserts that the relational model is an appropriate formalism for structuring the information generated by a distributed monitoring system [9].

Most events have associated data. This data can be rather simple, like the processor name or number. But sometimes this data is unstructured and complex. In the context of multimedia systems video frames and audio samples are examples of such data. In the last stage of the project the debugger will be integrated with a 'complex object server' which is subject of another part of the project. During the initial phases the monitor uses a database implemented with Oracle [10]. Unstructured and complex data is stored in separate files with pointers stored in the database.

Filters

The amount of collected events may be far to much, so before storing them in the database for later analysis and replay, they must be filtered. Filtering is done in two stages. A first crude filter is associated with each tap in the system. These filters determine which events are sent to the central monitor. They are kept as simple as possible to minimise the probe effect. It is not necessary to install the filter on the node it belongs to; it's placement may be on a free node. A trade-off has to be made: running the filter on the local node takes cpu time, but has the advantage that associated actions are executed immediately. Running the filter on a free node has the disadvantage that all events have to be transmitted to the filter node.
An example of a filter instruction is:

```
<
  On node_1
  Select message_primitives
  With (begin_time < 1000)
       ^ (task_number = 2)
  Exclude message_data
  Do record status node_1
>
```

The second stage of filtering is done with recognizers. It's goal is to find sequences of events. A first application is to replace a sequence of primitive events by one 'high-level' event. This enables monitoring at different levels of abstraction, according to the user's wishes. A second application is where the user wants the recognizer to perform a certain action at a specified sequence of events. A third application is scanning the stream of events for illegal event sequences (verification).
An example of a recognizer instruction is:

```
<
  Sequence a;b
  Where a = <msg_send, task_1>
        b = <msg_receive, task_2>
  Do record system_status
>
```

A query language for the database may be used as a filter for the events that are stored in the database. Queries are controlled by the userinterface or more directly by the programmer.

Userinterface

A graphical userinterface, offering different levels of abstraction, displays the information during the debug session [11]. Special care must be taken to show complex data belonging to events, i.e. video frames or samples of voice transmissions. Furthermore, time-object diagrams and animation of program execution will be basis for the userinterface.
The userinterface controls all processes, not only the monitor, but also the database and a source level debugger. Because of this coordinating role of the userinterface the user doesn't need to be aware of these underlying processes. After finding, for instance, a certain sequence of events (monitor) the user can ask for the value of a variable (source level debugger). This query may be realtime, during monitoring the system, but it could also be a query on the database.
The userinterface has a number of events, divided roughly into four groups: status information, filter settings, setting of breakpoints and debugger control functions.

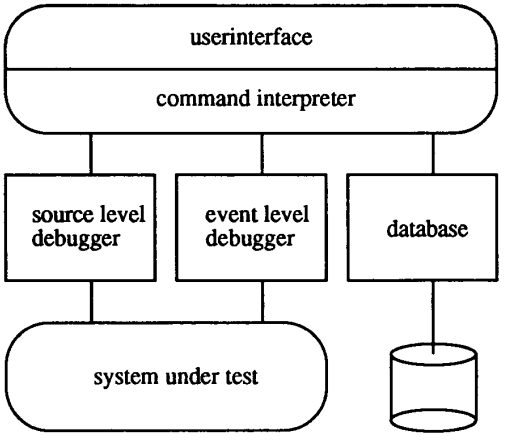

Figure 1: Global system view

Conclusion

The monitor is part of a multimedia project that started in 1991. It is only partly implemented and major parts of it are still in the design phase. The source level debugger is ready and the associated part of the userinterface is implemented. A prototype of a new feature, the backtrace, in the source level debugger is ready, but still needs a lot of testing. Backtracing enables the user to 'unexecute' instructions. If an error occurs it is now possible to go back in the program to find the cause of the error. It is simular to the execution backtracking approach described in [12].
The event taps in the system are being implemented now, but may need a redesign. Taps are now placed at entry points (traps) of the kernel. To do so one must alter the kernel to be used for debugging. A disadvantage is that it depends on the processor one uses. The next design incorporates a preprocessor that places debug incantations in the source code. The debugger is more portable and no modifications of the kernel are needed.

Bibliography

[1] Gait J.: *"A Probe Effect in Concurrent Programs"*, Software - Practice and Experience, Volume 16 No. 3, March 1986

[2] Cooper R.: *"Pilgrim, a debugger for distributed systems"*, Proc. of the 7th International Conference on Distributed Computing Systems, Page 458-465, 1987

[3] Tsai J.J.P., Fang K. and Chen H.: *"A Noninvasive Architecture to Monitor Real-Time Distributed Systems"*, IEEE Computer, March 1990, IEEE Computer Society Press.

[4] Michael M. Gorlick: *"The Flight Recorder: An Architectural Aid for System Monitoring"*, Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, May 1991, Santa Cruz, California, Page 175-183

[5] Fidge C.J.: *"Partial Orders for Parallel Debugging"*, ACM SIGPLAN Notices, Volume 24 No. 1, Page 183-194, Jan. 1989

[6] L. Lamport: *"Time, clocks, and the ordering of events in a distributed system"*, Comm. ACM, Volume 21 No. 7, Page 558-565, July 1987

[7] J. Scholten and P.G. Jansen: *"Distributed Debugging and Tumult"*, Proceedings Second IEEE Workshop on Future Trends on Distributed Computing Systems, Cairo, Page 172-178, Sept. 1990

[8] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian and M. Young: *"Mach: A new kernel foundation for Unix development"*, Summer Usenix Conference Proceedings, Page 93-112, 1986

[9] Snodgrass R.: *"Monitoring in a software development environment, a relational approach"*, ACM SIGPLAN Notices, Volume 19 No. 5, Page 124-131, 1984

[10] Oracle Corporation: *"Pro*C User's Guide Version 1.1"*, 1989

[11] Heath M.T. and Etherbridge J.A.: *"Visualizing the Performance of Parallel Programs"*, IEEE Software, Volume 8 No. 5, Page 29-39, Sept. 1991

[12] Agrawal H., DeMillo R.A. and Spafford E.: *"An Execution-Backtracking Approach to Debugging"*, IEEE Software, Volume 8 No. 3, Page 21-26, 1991