

## On Debugging in a Parallel System

J. Scholten and F. Sauer  
University of Twente  
Department of Informatics  
P.O.B. 217  
7500 AE Enschede  
the Netherlands

### Abstract

In this paper a description is given of a partly implemented parallel debugger for the Twente University Multicomputer (TUMULT). The system's basic method for exchange of data is message passing. Experience has learned that most programming errors in application software are made in calls to the kernel and the interprocess communication. The debugger is intended to be used for locating bugs at this level in the application software. It is assumed that basic blocks of the debuggee can be debugged using a traditional sequential sourcelevel debugger.

### Introduction

Despite the application of techniques to limit the number of errors in software, it is impossible to prevent bugs in any realistic software system. This is especially true for complex parallel and distributed software. Debugging therefore is an essential phase in the software development process. This is proven valid in the TUMULT environment.

TUMULT is a project of the University of Twente in cooperation with Dr. Neher Laboratories (PTT) and Océ Nederland B.V. [7]. The current system is of the MIMD type. [4].

The project focusses on the design of a distributed computer, its realtime operating system, together with a hardware architecture that supports the operating system, and software tools, such as a preprocessor and a parallel highlevel debugger. Centre of the system is the communication network developed to fit the message passing communication. It provides for a very fast communication link between nodes.

Programming such a system is difficult and errorprone. Not only the usual problems related to designing software are encountered, but the programmer also has to deal with synchronization and communication between processes and processors. All these problems make debugging tools for parallel systems an interesting, but essential and complex issue. In [14] an extensive bibliography can be found, containing about 300 publications related to the topic.

The next chapters give an introduction to parallel debugging and related problems, a short survey of TUMULT and a description of the TUMULT debugger.

### Distributed Debugging

A distributed or parallel system consists of a collection of processes working together to accomplish a task. Each process

is a deterministic program, able to execute separately from, and concurrent with other processes. Because of this complexity debugging such a system is far more difficult than debugging a sequential program:

- normal sequential debugging techniques, like tracing and setting breakpoints, are mostly based on a program counter and a process state. For a parallel debugger these concepts need to be extended, since parallel systems contain concurrent processes and thus have more than one program counter and process state,
- it is difficult to determine a system's global state at a certain time because of communication delays between nodes,
- asynchronous distributed or parallel systems are nondeterministic. Two or more executions of the same program may yield different, but valid orderings of events. Therefore, if an error occurs in the system it may be difficult to reproduce it at a later time,
- in a sequential program the behaviour is not altered by changing the elapsed time between two successive program instructions. Monitoring processes in a distributed or parallel system alters its behaviour if a process is stopped or slowed down. If a debugger causes atypical computations (a typical computation is one that could reasonably occur in absence of the debugger), bugs which occur in normal execution of a program might never be able to be reproduced while under the debugger's control [2]. This is called the probe effect [6],
- in a parallel system much more information is available that needs to be presented to the user. Therefore the userinterface between system and programmer is more complex.

Many debuggers described in the literature are event based: the debugger monitors the system's behaviour by detecting certain events, eg. message-send, message-recv, system-call, etc. The main differences lay in the implementations, in the ordering of events and the userinterface.

An interesting approach is behavioural abstraction [1], where the programmer has to describe the system in terms of events. The actual events during execution then are compared with the description. Events are modelled using hierarchical abstractions. Filtering and clustering are techniques to do so.

The monitor is the basis of event based debuggers. It consists of a process that detects and collects events and sends them to a global debugger. In Pilgrim [2] the monitor is called the agent, in the Jade system [10] observer. Other examples of event based systems are Amoeba [3] and Bugnet [9].

Monitors may be used in two ways. One way is during execution of a program while debugging it. The other approach is that events are recorded during execution for a later replay. An example is Instant-Replay [12]. A major problem with replay concerns the ordering of events. Some sort of global clock must be implemented to define a consistent ordering of events in separate nodes in the system, because the debugger must be able to replay the events in the right sequence.

### The TUMULT Multicomputer

The objective of the TUMULT project is research in the field of parallel architectures in general and in the field of multicomputers in particular. A small family of modular extendible systems has been designed, of which two are operational now: TUMULT-6 and TUMULT-15. A third prototype, TUMULT-64, is still under development [8]. TUMULT-15 is a multiprocessor in which up to 15 nodes are connected through an interprocessor communication network (IPCN). The nodes in the system do not share one global memory [20]. The basic method for the exchange of data is message passing. Major characteristics of the IPCN are:

- it has a modular extendible ring structure,
- total capacity of the ring is 20 MBytes/sec. (worst case), 40 MBytes/sec. (typical),
- all nodes may send or receive simultaneously, provided the sum of all messages does not exceed the total ring capacity. Most networks allow only one communication at the same time, resulting in a rather high overhead in time,
- when a link is established information is transferred at a high rate by means of dma (typically 4 MBytes/sec.).

Each node consists of a processing element (PE) belonging to the Motorola M680X0 family, with local memory and a network interface, and may be extended with other processors, a floating-point processor, memory or i/o. Because a standard type local bus is used (VME bus) extensions are realized with commercially available standard boards. Communication between PE's in one node is based on the same primitives as those used for communication between nodes. Memory connected to the bus can be used as global memory for that specific node. This can be dangerous, because communication via global memory is not controlled by the IPC and may interfere with normal communication. However, some applications require global (shared) memory and TUMULT's architecture does not forbid such utilisation. A PE configured with more than one processor and with global memory may be seen as a small TUMULT system (and is indeed in use as such by some groups in our department).

The IPCN is completely transparent for the application: it doesn't matter which IPCN is used; the communication primitives offered by the system are the same for all IPCNs, be it the fast ring or the slower VME-bus.

The hardware is controlled by a distributed realtime operating system, the TUMULT executive, where each processor has its own multitasking subsystem [13]. The TUMULT distributed realtime operating system is written in Modula-2 [22] and has a layered structure. See [Figure 1].

The most basic operations and objects are provided by the kernel: scheduler, interrupt handling mechanism, threads, etc. On top of the kernel is the IPC, the interprocess communication [17]. It offers the functionality of mailboxes, send and receive primitives and a primitive remote procedure call. The filesystem layer offers a

transparent distributed filesystem. Each node has a local manager (LOM). It receives, interprets and executes commands from the global manager (GLOM) concerning the loading, starting and execution of application processes, as well as the collection of status information [21]. The GLOM, residing on exactly one node, distributes processes over the nodes and orders the LOM to execute them. Finally, the operating system is separated from the application program by the application programmer's interface (API). This API provides for a language independent interface, so applications written in several languages can make calls to the executive. For the debugger the API is an important layer, because here the debugger plugs into the system.

The IPC can transfer variable length records from any sending thread to any receiving thread (a thread is a light-weight process), for which mailboxes are used. Mailboxes are created dynamically by processes. Communicating threads do not need to know the physical location of each other, nor do they need to know the location of the mailbox, all is taken care of by the mailbox manager. Each mailbox has a unique dynamically chosen global name and type of message that may be transferred via this mailbox. Creation and disposal of a mailbox are independent actions and may be performed by any thread. Usually a mailbox will be created by a thread that intends to communicate via this mailbox. Mailboxes can be disposed of at any time [15].

Mailboxes contain an arbitrary number of buffers to store messages temporarily. This allows sending threads ahead of receiving threads until all buffers are filled. Communication via a mailbox that contains buffers is asynchronous. If a mailbox contains no buffers (zero buffers) the communication via this mailbox is synchronous, i.e. a sending thread is delayed until another thread performs a corresponding receive.

Before communication via a mailbox can take place the mailbox has to be created. Once the mailbox exist, threads that want to send messages to this mailbox (senders) and threads that want to receive messages from this mailbox (receivers) establish a communication channel by specifying an entrance to the mailbox. In case of a sender this entrance is called a sender port and in case of a receiver a receiver port.

Two types of communication can be distinguished, the unidirectional and the bi-directional communication. In the unidirectional communication, communicating threads are sender and receiver threads. Sender threads can send messages (to a mailbox) and receiver threads can receive messages (from a mailbox).

In the bi-directional communication, communicating threads are client and server threads. Client threads can send a request message (to a mailbox) and wait (block) until the corresponding reply message is received. Server threads can receive a request message (from a mailbox) and, after having served the request, return a reply directly to the client.

To support both types of communication the IPC offers a number of primitives which are divided into the following five categories:

#### **1. installation of a mailbox**

```
InstallMailbox(  
    name, mailboxType,  
    typeSize1, typeSize2,  
    numberOfBuffers  
);  
(* 'name' defines the unique global name  
   that identifies the mailbox.
```

'mailboxType' defines the type of the mailbox (uni-directional or bi-directional).  
 'typeSize1' and 'typeSize2' define the types of the messages that can be transferred using this mailbox. In case of a bi-directional mailbox 'typeSize1' defines the message size and 'typeSize2' defines the size of the reply message.  
 'numberOfBuffers' defines the number of messages that can be buffered by the mailbox.  
 \*)

## 2. connection to a mailbox

```
Connect (
    mailboxName, portVar
);
```

(\* 'mailboxName' defines the name of the installed mailbox.  
 'portVar' is the port variable that has to be connected.  
 The type of 'portVar' must be one of the predefined types 'SENDERPORT', 'CLIENTPORT', 'RECEIVEPORT' or 'SERVERPORT'  
 The type of 'portVar' determines the type of connection.  
 \*)

## 3. actual communication

```
Send(
    sendPort, message
);
```

```
Receive(
    receivePort, messageVar
);
```

```
RequestReply(
    clientPort, request, replyVar
);
```

```
SendReply(
    serverPort, id, reply
);
```

## 4. disconnection from a mailbox;

```
Disconnect (
    portVar
);
```

## 5. disposal of a mailbox.

```
DisposeMailbox(
    mailboxName
);
```

## The TUMULT Debugger

This chapter will give an outline for the TUMULT debugger. First of all the constraints for the debugger are mentioned. Most are required to minimize the probe effect.

- Debugging must not require recompilation of the target program,
- the debugger must be independent from the kernel, so there will be one kernel for both normal use and running programs under the debugger's control,

- if possible, the monitor should be a part of the kernel. It must be as fast as possible and give a minimum of overhead.

Heart of the debugger is the record and replay section. In order to get a proper ordering of events a global system time is needed. [5] shows that logical clocks described by [11] are not suitable because they will implicitly order the events totally, in which process information may get lost. The partial ordering by the logical clocks of Fidge could be a solution, but they are not very suited for the TUMULT system: since processes are created and deleted dynamically, the number of processes is not constant. Therefore the vector used for timestamping all communication, in which each process has its own element, is variable in length and can become quite long. Beside, every time a process is created or deleted, this event must be broadcasted so that every process can modify the length of its own vector. This will give a relative large communication overhead.

A solution to the mentioned problems is the introduction of a physical shared global clock. Now events are timestamped with a time that is equal in all PEs of the system. The ordering of events is firm in that case. Timestamping can be done by the monitor instead of by each process separately. Luckily, implementing a global clock in TUMULT is not very complex. The network is synchronous and already uses a global clock running on a frequency of 10 MHz [16]. By adding an addressable counter to the network interface the shared global clock can be implemented.

Two kinds of monitor will handle the stream of events in the system: the local event manager (LEM) and the global event manager (GLEM). See [Figure 2]. The naming is consistent with the naming conventions in TUMULT: LOM and GLOM (local and global manager), and LMM and GLMM (local and global mailbox manager).

The LEM intercepts all calls to the application programmers interface (API), and produces a raw stream of events. In this context an event is any interesting action in a process that can have an effect outside that process. In this context an event is any interesting action in a process that can have an effect outside that process. Because the GLEM may not be interested in all events, or only be interested in certain sequences of events, the events are filtered. The filter is programmable by regular expressions, so the eventstream can be completely customized depending on the needs of the user doing the debug session. Events are recorded into event records, which have the general form:

*<category, location, time, attributes>*

The operating system is divided into modules, each consisting of data, procedures and functions. The category of the event is defined by the procedure or function that is called and the module this procedure or function resides. The attributes consist of the parameters of the called procedure or function and the function result. The number of the processing element and the thread identification form the location. Since a call to the operating system may cause the scheduler to block the current process and to resume a waiting process, two timestamps are recorded: one when entering the API and one when returning. As a side effect these timestamps can be used for performance measurements of the system. The complete

definition for the event record is:

```
<  ModuleNumber,ProcedureNumber,      (category)
   NodeNumber,ThreadId,                (location)
   BeforeTimeStamp,AfterTimeStamp,     (time)
   .
   .
   parameters,                         (attributes)
   .
   .
   result
>
```

The GLEM functions as an interface between the TUMULT system and a SUN workstation. It maintains a mailbox to which all LEMs send their eventstreams. A SCSI interface provides a very fast bi-directional communication link between TUMULT and the workstation. The GLEM is situated on the same node as the GLOM and GLMM to have access to all system administration, in particular the names of threads and mailboxes.

The main debugger is situated at the workstation. A major part consists of routines to store, retrieve and manipulate recorded events. [18] asserts that the relational model is an appropriate formalism for structuring the information generated by a distributed monitoring system. In TUMULT the embedded database will be implemented by using existing libraries. This database is part of the TUMULT environment and used non-exclusively by the debugger.

Presenting the data to the user is not trivial. In the process of debugging the amount of recorded data might become extremely voluminous and not very comprehensive to the user. Two main alternatives exist to this problem: presenting data in textual views or in graphical views. The first prototypes of the debugger did employ line oriented terminals to display events. Now a choice is made in favor for the graphical views, using grouping as a means to reduce the amount of information presented to the user. In Mona [10] grouping is used as an abstraction mechanism for graphical views. A group combines a set of processes into a single icon, hiding all interaction inside the group.

### Conclusion

In this paper a distributed system (TUMULT) and its debugger are presented. Main features of the debugger are: it is event based, using a monitor for intercepting these events; record and replay are the main debugging techniques; preprocessing of events is done by programmable filters; the userinterface is graphical, using grouping as the main abstraction mechanism. Some notes have to be made on the application of monitors in the system. The detection of events is an active process that consumes time. This gives rise to serious implementation constraints when used in realtime systems, as TUMULT. Capturing events must not cause a decrease in the system's realtime capabilities. For this reason only system calls are considered to be events and no effort has been made to detect interrupts. A solution to this drawback may be the noninvasive architecture as described by [19]. Extra hardware is used to detect events, without interfering in the system itself. Research in the area of parallel debugging started in 1989. Parts of the debugger are implemented now, other parts are still topic of research. By now initial versions of the global and

local event managers are implemented. A slow serial link between the frontend processor and the TUMULT system is replaced by a fast SCSI communication link. The userinterface is partly textual, partly graphical. The debugger is part of the TUMULT environment and fully integrated in the system. It uses the system's filesystem, database and userinterface.

The languages used to implement the debugger are Modula-2 and C. X Window System and OSF/Motif are used for the graphical userinterface.

### Literature

- [1] Bates P.C., Wileden J.C.: "High level debugging of distributed systems; The behavioural abstraction approach", Journal of systems and software 3(4), 255-264, (1983).
- [2] Cooper R.: "Pilgrim: a debugger for distributed systems", Proc. of the 7th International Conference on Distributed Computing Systems, 458-465, (1987).
- [3] Elshoff I.J.P.: "A distributed debugger for Amoeba", ACM SIGPLAN Notices 24(1), 1-10, Jan. (1989).
- [4] Feng T.: "A survey of interconnection networks", Computer, Dec. 1981, IEEE Computer Society Press.
- [5] Fidge C.J.: "Partial Orders for Parallel Debugging", ACM SIGPLAN Notices, 24(1), 183-194, Jan. (1989).
- [6] Gait J.: "A Probe Effect in Concurrent Programs", Software - Practice and Experience, Vol. 16, No. 3, March (1986).
- [7] Jansen P.G., Smit G.J.M., Scholten J.: "A survey of TUMULT, a real-time multi-processor system", Dept. of Computer Science, Twente University of Technology, the Netherlands, int. rep. no. INF-86-2, (1986).
- [8] Jansen P.G. and Smit G.J.M.: "TUMULT-64, a realtime multiprocessor system", Dept. of Computer Science, Twente University of Technology, the Netherlands, int. rep. no. INF-89-58, (1989).
- [9] Jones S.H. et al: "BugNet, a real time distributed debugging system", Proc. of the 6th symposium on reliability in distributed software and database systems, pp 56-65, March (1987).
- [10] Joyce J. et al: "Monitoring distributed systems", ACM Transactions on Computer Systems, 5(2): 121-150, May (1987).
- [11] Lamport L.: "Time, clocks and the ordering of events in a distributed system", Communications of the ACM, Vol. 21, pp 558-565, July 1978.
- [12] LeBlanc T.J. and Mellor-Crummey J.M.: "Debugging parallel programs with Instant Replay", IEEE Transactions on Computers, C-36(4): 471-482, April 1987.
- [13] Luttmer M.L.M., Jansen P.G.: "The TUMULT kernel", Dept. of Computer Science, Twente University of Technology, the Netherlands, int. rep. no. INF-87-15, March 1987.
- [14] Pancake C.M. and Utter S.: "A bibliography of Parallel Debuggers", SIGPLAN Notices, Vol. 24, No. 11, pp 29-42, 1989.
- [15] Ribbers H., Jansen P.G.: "Communication facilities of Tumult, a tutorial" Dept. of Computer Science, Twente University, The Netherlands, mem. inf-87-10, Feb. 1987.

- [16] Scholten J., Jansen P.G.: "TUMULT the Twente University Multiprocessor", Proceedings of the IEEE workshop on future trend of distributed computing systems in the 90's, pp. 111-118, Hong Kong, Sept. 1988.
- [17] Sijbers A.: "A distributed File System for TUMULT", M.Sc. Thesis, Dept. of Computer Science, Twente University of Technology, the Netherlands, 1987.
- [18] Snodgrass R.: "Monitoring in a software development environment: a relational approach", ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Development Environments, published in ACM SIGPLAN Notices 19(5): 124-131, 1984.
- [19] Tsai J.J.P., Fang K. and Chen H.: "A Noninvasive Architecture to Monitor Real-Time Distributed Systems", Computer, March 1990, IEEE Computer Society Press.
- [20] Vuurboom R.: "Communication in a loosely-coupled multiprocessor system: its architecture and implementation", M.Sc. Thesis, Dept. of Computer Science, University of Twente, the Netherlands, 1984.
- [21] Wijnstra F.J.: "Design and implementation of a command processor for TUMULT", M.Sc. Thesis, Dept. of Computer Science, University of Twente, the Netherlands, 1990.
- [22] Wirth N.: "Programming in Modula-2", Third corrected edition, Springer Verlag, 1985.

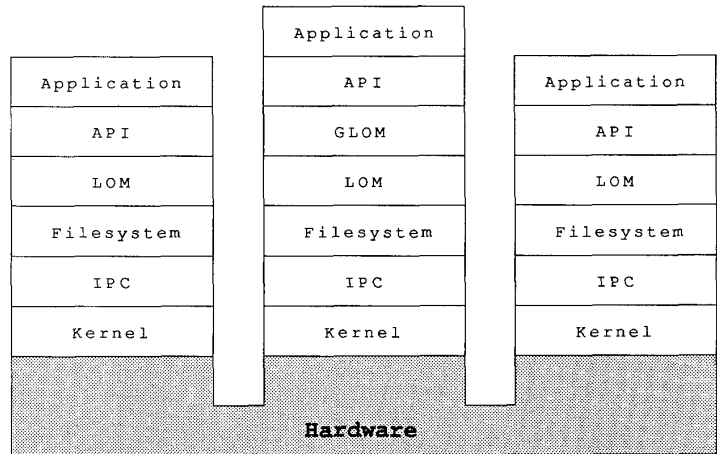


Figure 1: The TUMULT Executive

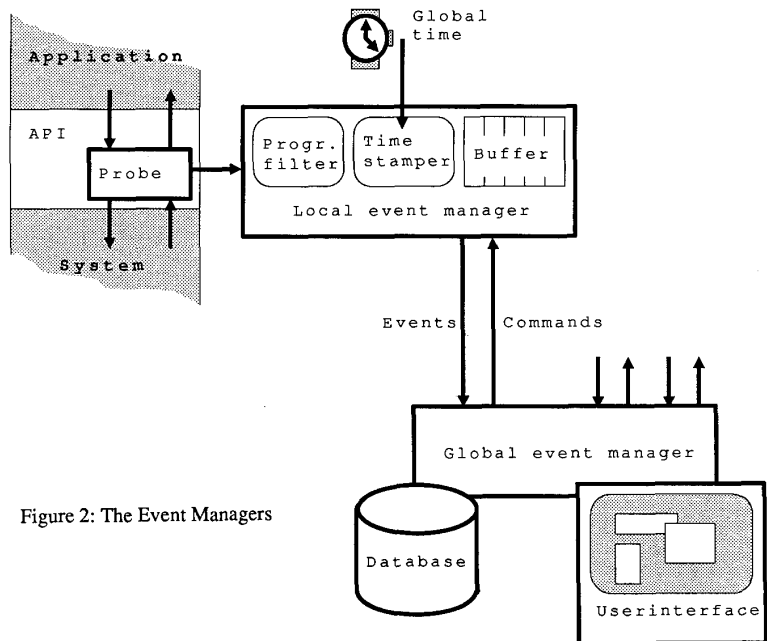


Figure 2: The Event Managers