Open Systems Laboratory for Distributed Operating Systems

Ishwar Rattan
Central Michigan University

Abstract

The recent advances in microprocessor technology and local area networks have made it easy to put together computer systems with a large number of machines connected by a high speed network. These systems need radically different software. In particular, the required operating systems have to deal with new ideas such as fault tolerance, load balancing, incremental growth, computational speed up, and transparency not found in traditional centralized operating systems. To integrate the concepts of distributed operating systems (DOS) in out undergraduate curriculum, a new course has been designed. It uses DOS laboratory which contains fourteen PC/ATs running under MINIX 1.5 (with networking kernel). This paper describes the course, the laboratory set-up, and the experiences in using the laboratory.

Introduction

Since the mid 1980s, two major advances in computer technology have been evident. First, powerful microprocessors (16, 32 and even 64 bit) with computing power of earlier mainframes are abundant. Secondly, a larger number of these can be connected together through high speed networks which allow data transfer at 10 to 100 million bits per second.

These have lead to increasing use of distributed systems - a large number of CPUs each in a separate machine communicating via a high speed network that appears to the users as a single computer (in marked contrast to the traditional centralized, single CPU systems).

An efficient use of these distributed systems involves use of radically different software, in particular the operating system. It has to address the issues of transparency (single system view to users) and performance (load balancing and computational speed up) while providing reliability (system availability and fault tolerance) and flexibility (incremental growth)[1,2,3].

The time has come to integrate these concepts and issues involved in these Distributed Operating Systems in the Computer Science curriculum at the Undergraduate level. To this end we have introduced a new course on DOS supported with a strong laboratory component. The course addresses the basic issues in the design and implementation of DOS with specific examples from case studies. The laboratory is used to explore and crystallize the concepts in a realistic programming environment. This paper presents our first experiences in this endeavor.

The Course

An undergraduate course on DOS has been introduced starting Fall 1994. It is a three credit offering under ``Special Topics in Computer Science" to seniors and is also available to Masters students. A more detailed DOS course has been introduced in the MS curriculum.. The course uses the laboratory established through an NSF/ILI grant (DUE: 9350654).

The DOS course has the regular senior/MS level Operating Systems (OS) course as its prerequisite. The prerequisite covers the standard OS topics (file system, process and memory management, I/O subsystem etc.) involved in the design and implementation of standard, centralized systems. It, too, has a strong laboratory component associated with it. Until the DOS laboratory, we used an OS laboratory based on PC/XTs running under MINIX OS[4], established in 1989 through an earlier NSF/ILI grant (USE: 8851239).

The new DOS course builds on this background. Particular emphasis is placed on the issues of transparency, reliability, performance and flexibility. A summary of the topics covered in the course is given below.

þ Review of centralized systems.

þ Distributed systems - terminology, multiprocessor and multicomputer systems, network OS, network file systems, design issues: flexibility, performance, reliability, and transparency[1] (case study: SunOS NFS[5]).

þ Communication in distributed systems - layered protocols, client/server model, remote procedure call mechanism, group communications and message ordering (case studies: Amoeba[1], SunOS RPC).

þ Synchronization in distributed systems - clocks, mutual exclusion, transaction model, distributed deadlocks and related algorithms.

þ Threads in distributed systems - thread as opposed to a process, design issues in user and kernel level threads (case studies: kernel-level[1,6], user-level[7]).

þ Distributed file systems - file servers, stateless file servers, file service, directory service, interface models, file sharing, file replication(case studies: Amoeba[1], Andrew file system[8]).

The Laboratory Set-up

The laboratory consists of fourteen PC/ATs. There are four 486DX2/66s (16Mb RAM), six 486DX/33s (8Mb RAM), and four 486DLC/33s (4Mb RAM) machines. Each machine has a 520Mb IDE hard disk, a 1.4Mb floppy drive, a 101 key keyboard, a WD8003 ethernet card, a hercules monochrome video card and monitor. As a graphical interface is not critical in the study of operating systems, no investment was made in SVGA controller cards and monitors. The machines are on a separate LAN segment, isolated from the rest of the machines in the department.

The machines run under MINIX 1.5 networking kernel OS. We started with MINIX for the following reasons. It is used in the prerequisite OS course so students are already familiar with its interface and utilities. In addition, it is an open system -- the source code is available and can be modified if needed (see experiment 5). The networking kernel uses a networking protocol based on the remote procedure (RPC) call mechanism[9] (see appendix for details). This protocol is compatible with the form of RPC used in the distributed OS Amoeba[10]. This makes students relate their projects directly to the case study in the classroom.

MINIX[4] is an open OS designed by Dr. A. S. Tanenbaum. It is a multiuser/multitasking OS similar to UNIX. It is system call compatible with version V7 of UNIX. It has over 170 utilities and more than 200 library functions. It was primarily designed to teach OS and related courses. It is available with full source (in C) and easy-to-read installation and reference manual. It also has a Kernighan and Ritchie C compatible C compiler i.e. it is a complete programming environment.

MINIX 1.5 is distributed by Prentice-Hall. It comes on eleven 3.5" disks which include the source files. Using MINIX directly from floppies as distributed is cumbersome. We have installed it on a 32Mb hard disk partition[11]. A backup copy is kept on another hard disk partition of same size. A student only needs a boot disk to use the system. The students are not allowed to save their work on the hard disk, they use their own floppies.

Based on our experience we make the following observations on setting up the laboratory.

1. Use PC/ATs with 4Mb or more RAM to accommodate a larger size kernel and leave enough memory for user processes (MINIX does not support virtual memory or swapping).

2. If a PC has 16Mb or more RAM, the kernel has to be patched to work correctly. One has to get the patch from the MINIX interest group and then use the patch program to apply it to the kernel.

3. WD8003 (8-bit) ethernet card is the only one supported under Amoeba networking protocol.

4. MINIX can only recognize primary MS-DOS partitions on hard disk. It can not access a partition larger than 32Mb.

5. The cluster of machines should be on a separate LAN segment as Amoeba RPC tries to use the full bandwidth of the ethernet which may interfere with other systems on the LAN.

6. Installation and maintenance requires some knowledge and system administration experience of UNIX.

7. Most of the information needed is available in MINIX 1.5 reference manual. Generally, Dr. Tanenbaum is willing and prompt in answering questions, but he is not always available. Some help can be obtained on the MINIX interest group comp.os.minix on the Internet.

Details of Experiments

The experiments for the laboratory work are designed to clarify the basic concepts of flexibility (system scales easily to accommodate increase in number of machines with corresponding increase in performance), performance (running an application should not be appreciably worse than running it on a single CPU system), reliability (system should be available and functional in presence of failures), and transparency (system should provide a single system image) as applied to DOS. They explore the following ideas.

þ Load distribution - the system should try to balance and distribute the work load on a number of machines (experiment 3 and 8).

þ Fault tolerance - the system should continue to provide service in presence of failures (experiment 4).

þ Computational speed-up - the system should use faster remote CPUs for compute intensive applications where computational time is larger than network overheads (experiment 5).

þ Access transparency - the system should provide same mechanism to access local or remote resources (experiment 6).

þ Performance transparency - there should be no appreciable reduction in performance due to communication overheads (experiment 5 and 9).

Experiment 1 is designed to (re)acquaint the students to MINIX, and to see the `basic' structure of a `client' and a `server' program using the Amoeba RPC primitives (see appendix for details). The students are given predesigned client and server programs. These programs provide the basic framework for the remaining experiments. The students study, compile, and execute them on the system (the client and server run on separate machines).

Experiment 2 involves writing a simple client/server application using the RPC primitives studied in the previous experiment. The client reads a message from the keyboard, sends it to the server, and displays the reply received on the screen. The server echoes back whatever it got from the client.

Experiment 3 allows the study of incremental growth. The client sends a 16Kb message to the server several times, and measures the time taken. The server keeps echoing back the message received. The scenario involves two identical clients. First there is only one server, and client completion time is measured. Next the situation is repeated with two identical servers. Now the service time for the clients is less, due to load sharing by the two servers.

Experiment 4 explores the idea of fault tolerance. The client used in experiment 3 is modified to not quit on failure of a transaction. If the server fails to respond, the client waits (sleeps for 10 seconds) and retries the transaction. This allows ample time for simulating a server crash. The server machine is powered down and a new server started on a another machine which starts providing service when the client retries the failed transaction.

Experiment 5 allows the study of nature of local and remote computations. The idea is to compare the cost (time taken) of local and remote computations for the same computing problem (of exponential complexity). Programs are written to find (recursively) the nth Fibonacci number locally (on the same machine) and as a remote computation (on a remote but faster CPU -- using client/server model). The client runs on a 486DX/33 and the server on a 486DX2/66 machine. A graph between n and time taken to compute it is plotted. After some value of n, the remote computations start giving better performance. This happens when computation time is higher than communication overhead. To see the cross over point from local to remote computations, system clock resolution had to be enhanced by modifying the CLOCK_TASK in the kernel.

Experiment 6 permits the study of access transparency. A client stub `rfib()' (for recursive calculation of nth Fibonacci number) is written and placed in the system library. A client/server application is written in which the client can call rfib(). The computation scenario involves two cases. First, when the server runs on the same machine as the client. Second, when the server runs on a separate machine. In either case, the client has no knowledge of the server's location.

Experiment 7 is useful in studying the concept of a stateless server. A stateless server does not keep track of the state of transactions with the clients, and hence, makes recovery from a crash much easier. The goal is to write a simple stateless file server/client application. The server provides read and write services on files. The client program requests four operations on files: **ropen** -- open a file on server for read or write operation, **rclose** -- close an already open file, **rread** -- read a specified number of bytes at the given offset from a file already opened for reading, and **rwrite** -- write a specified number of bytes at the given offset to an already opened file for writing. The client maintains all the information needed on the state of its transactions with the server. If a server fails and comes back up, it does not have to know what it was doing when it crashed.

Experiment 8 allows one to explore the concept of a centralized name server (maps a service to the server providing it). Such a server permits other servers to register their services with it, and provides server identification to clients desiring a specific service. A name server to keep track of the services of experiments 2 and 6 is written. Modifications are made to the servers (of experiments 2 and 6) so that they register their services before accepting any client requests. The clients on the other hand, are modified to ask the name server to identify the specific server before transacting with it.

Experiment 9 allows the study of workload distribution and parallel nature of computations. Suppose that it is desired to find the sum of each row of a 10x10 matrix of integers. The problem needs one client and ten identical servers. The client distributes the rows to ten servers (one per row), waits for results to come back and displays them on screen. Each server sums up the row received and sends the results back.

Conclusion and Future Plans

In the first offering of the course there were sixteen students in the class. They worked in groups of twos, and completed all the experiments. A majority of the students found it a novel experience. Over all, the students gained a better understanding of the design and implementation issues in distributed OS as a result of the work done in the laboratory.

MINIX 1.5 networking kernel only provides a network OS environment. It is not suitable for a complete exploration of the some distributed OS concepts.

þ Data transparency -- ability to access data (from a file) in a manner transparent to an application. It can only be achieved if the system supports a remote file server.

þ Migration transparency -- ability of resources and processes to move around and still be accessible to an application under the same name.

þ Concurrency transparency -- ability for a number of applications to share resources automatically.

þ Parallelism transparency -- ability of activities associated with an application to happen in parallel in a transparent manner.

To explore the above issues to some extent, a real distributed OS is needed in the laboratory. We plan to use Amoeba 5.2. Amoeba[10] is a general-purpose operating system designed to take a collection of machines and make them act together as a single integrated system. It has a `microkernel' based architecture. The kernel supports basic process, communication, and object primitives. Everything else is built on top of this with user-space `server' processes. Amoeba uses a novel idea of a pool of processors. The processors are dynamically allocated to processes as needed. We plan to run all server processes on a dual processor Sparc-Station 20 and use the 486DXs as processors in the pool. Some of features of Amoeba that allow to enhance the laboratory experiments are:

þ Mutli-threaded processes.

þ File replication facility.

þ Group communication with complete ordering of messages.

þ Fast local internet protocol.

þ Parallel programming language Orca.

As of now, we have Amoeba installed and running on a cluster of a Sparc-Station 20 and 486DX2/66s. The system is being tested and is expected to be used in the next offering of the course.

Acknowledgment
The project was partially funded by the NSF/ILI grant (DUE: 9350654). Matching funds were provided by Central Michigan University.

Appendix

RPC allows a program to call procedures located on other machines. When a process on machine A calls a procedure on machine B the calling process is blocked and invocation of the procedure on B takes
place. Information can be transported from caller to callee in procedure parameters and can come back as procedure results. No message passing or I/O is visible to the programmer. This transparency is achieved via stub modules - special procedures that marshall and unmarshall the parameters in a message and invoke kernel primitives for sending the message. These stubs are placed in the library. On client side, a client stub and on the server side a server stub (procedural interface that a client can call) is used. In a program, invocation of the server stub looks like a local procedure call. In order for a client to do RPC with a server, it must know the server's address. This is done by allowing a server to choose a unique port, used as the address for the messages sent to it. Amoeba networking protocol is a stop and wait protocol that

guarantees at most once delivery of a message. It has four kernel primitives which provide the basic interface between two transacting processes. These are:

1. *getreq()*: used by a server process to get a request from a client. It indicates a server's willingness to listen to a port.

2. *putrep()*: used by a server to send a reply to a client.

3. *trans()*: used by a client to do a transaction with a server. The client is blocked till the reply comes back.

4. *timeout()*: sets the time limit at which a trans() gives up - relates to locating the sever process.

The typical structures for a server and client under Amoeba RPC are given below.

```
Server:
  Ignore signals.
  Repeat
    Have the server listen to a specific port.
    Wait for a request to come in for that port.
    If no errors occurred then
         Carry out the work.
         Send back the reply.
    Fi
  Forever

Client:
  Initialize the server port.
  send a message to the server listening at that port.
  If no errors occurred then
        Transaction was successful.
  Else
        Transaction failed.
  Fi
```

References

1. A. S. Tanenbaum, *Modern Operating Systems*, Prentice-Hall (1992).
2. A. Goscinski, *Distributed Operating Systems: The Logical Design*, Addison-Wesley (1991).

3. G. Coulouris and J. Dollimore, *Distributed Systems Concepts and Design*, 2nd Ed., Addison-Wesley (1994).
4. A. S. Tanenbaum, *Operating Systems: Design and Implementation,* Prentice-Hall (1987).
5. Sun Microsystems, Inc. *NFS: Network File System Protocol Specification,* RFC 1094, NIC (1989).
6. NeXT Computer, Inc. *NeXT Operating System Software*, Chapter 4, Addison-Wesley (1991).
7. F. Mueller, *A Library Implementation of POSIX Threads,* Summer USENIX Conference (1993).
8. M. Satyanarayanan, *A Survey of Distributed File Systems*, Review of Computer Science, Vol. 4, pp. 73-104 (1990).
9. A. Birell and B. Nelson, *Implementation of Remote Procedure Calls*, ACM Trans. of Computer Systems, Vol. 2, pp. 33-59 (1984).
10. S. Mullender et al., *Amoeba: An Operating System for 1990s*, IEEE Computer, Vol. 23, pp. 44-53 (1990).
11. I. Rattan and L. P. Singh, *An Operating Systems Laboratory Using Microcomputer Based Workstations*, 1990 ASEE Annual Conference Proceedings, pp. 594-598 (1990).
12. A. S. Tanenbaum et al., *MINIX 1.5 Reference Manual*, Prentice-Hall (1991).

Biographical Information

ISHWAR RATTAN is an Associate Professor of Computer Science at Central Michigan University. His research interests include operating systems, distributed operating systems, and computer networks.