1-1-1986

Using Low-Cost Workstations to Investigate Computer Networks and Distributed Systems

Mark Sherman
Dartmouth College

Ann Marks
Dartmouth College

USING LOW-COST WORKSTATIONS TO INVESTIGATE COMPUTER NETWORKS AND DISTRIBUTED SYSTEMS

Mark Sherman, Ann Marks

Technical Report PCS-TR86-126

Using Low-Cost Workstations to Investigate Computer Networks and Distributed Systems

Mark Sherman
Department of Mathematics and Computer Science

Ann Marks
Thayer School of Engineering

Dartmouth College
Hanover, NH 03755

March 1986

1. Introduction

A quality education in contemporary computer science requires that students gain experience with realistic systems. Many efforts at bringing empirical computer science to undergraduates focus on rather old technologies, for example, building a compiler or simulating a disk scheduler. Although efforts are being made to use some newer technologies, such as the advanced graphics abilities of the workstations in Brown's Balsa system[1,2,3,4] and MIT's Project Athena,[5] the efforts are concentrating on teaching traditional material in a new medium. However, the medium itself -- networked workstations in a server environment -- is worthy of exploration by undergraduate students in a laboratory setting. At Dartmouth, we developed a Computer Network Laboratory to let students experiment with computer networks, protocols and distributed systems. Through this article, we wish to share our experiences in the design of the laboratory and give an example of how the laboratory was used in a computer network course.

2. Requirements for the facilities

The requirements for a computer network or distributed systems laboratory are quite different from the facilities needed for a typical computer science course in algorithms, data structures or compilers. When we decided to offer a network course emphasizing the experimental aspects of computer networks, we examined our campus-wide, time-sharing facilities and found them inadequate. We decided that our laboratory facilities required:

- Many Machines
- Reconfigurable Facilities
- Unprotected, Open Systems
- Economy and Uniformity

Each criterion is explained below.

Many Machines. To investigate how a distributed system worked, we needed to physically distribute the system among several machines. Therefore each student would require some collection of machines for each assignment. As we describe in "Development Systems for Students", we allocated five machines per student. With so many machines, we also found that we needed extra machines to cover the inevitable breakdowns during the course.

Reconfigurable Facilities. Because networks can use a variety of topologies, we needed to provide a way to reorganize the machines to create networks. Some experiments examined contention on a bus

which required many hosts to generate traffic; others measured a gateway between two networks; still another considered only point-to-point communication without interference. Each experiment required a different configuration of machines. Further, our laboratory uses distributed development and debugging facilities. For example, a gateway experiment may require network monitors on each network in contrast to a contention experiment that needs only a single monitor. Therefore, it is essential that we be able to organize our facilities in many different ways without extensive hardware construction.

Unprotected, Open Systems. The access provided by an unprotected, open system is needed to support a broad range of activities. Because many of our experiments need to replace or monitor very low-level features of the operating system, we required that students have full access to any part of the machine. For example, some of our experiments were designed to measure the overhead expense of using general operating system facilities in a distributed system vs making a special purpose operating system. Some facilities, such as a network monitor, require writing very low-level parts of an operating system that directly control the network hardware.

Economy and Uniformity. We have found that if students can use a set of equipment for more than one course, and if they can use the equipment after the course has finished, there is a greater incentive to learn the details of the systems and development aids. With a greater mastery of the basic system, the students are able to concentrate their attentions on the course material rather than on the details of the particular system they are using. One way to provide the uniformity is to encourage the students to buy a machine of their own. Obviously, students' funds are quite limited, so the machine must be economical. Thus the need for economy is not merely so that we can afford the many machines required for the course, but so that individual students can also afford to purchase equipment for their own use. A similar effect is achieved if uniform equipment is used in a curriculum: having learned, for example, Unix®, in one course, a student brings his or her knowledge of how to use the editor and mail system to the next course using Unix.

3. Inadequacy of Current Systems

Unfortunately, available systems did not meet our criteria. Historically, educational institutions have relied on a campus computer utility for teaching the practical aspects of computer science. The utility is usually a centralized time-sharing system. But neither a utility nor a centralized, time-sharing system provide adequate facilities for experimenting with computer networks or distributed systems. The nature of a shared resource for use by students usually implies:

- Other people rely on the utility;
- Relatively few systems are available;
- Systems cannot be reconfigured;
- Nondeterministic and asynchronous aspects of systems are inaccessible for study.

Because other people rely on the computer utility, the computation center (or other similar authority) must ensure that the utility is available, reliable, robust and secure. None of these aspects can be assured if a large number of inexperienced students start making kernel changes in the running system. Therefore, students using a shared facility have limited access to a system's resources and underlying hardware features. The limitations were designed to protect the system from a user and to protect one user from another. For example, a nonprivileged user can not write and measure a true elevator scheduling algorithm for a disk drive -- doing so would jeopardize other users who store files on the disk. Instead, a user is typically allowed to use synchronous calls to file system primitives which provide a high-level byte-stream abstraction. For the student, a simulator is employed to experiment with a disk scheduling algorithm.

Unfortunately, such simulators usually beg the issue of what is to be demonstrated, since there is no way for most simulator writers to compare their simulators against the actual system -- the simulator writers are also nonprivileged users!

Even if students were allowed to change a utility system, a typical computer utility is inadequate for a network or distributed systems laboratory. First, the students' experiments could not be adequately controlled, that is, other users of the system would perturb the results and interfere with the student's work. Second, most utilities are wired into a campus-wide network that has a predetermined configuration. The hardware and software for reconfiguring the network is generally unavailable, even if the utility's operators were inclined to let a class reconfigure a system.

4. Using Collections of Workstations

Because the centralized, time-sharing utility could not meet our needs, we were drawn towards the use of workstations as the basis of our laboratory equipment. A typical workstation provides many of the same environments available on a shared system. However, a user may change arbitrarily the underlying configuration of a workstation's software or hardware in order to experiment with alternative techniques or to demonstrate certain properties. Some features that can be investigated include new disk configurations and algorithms, network protocols, interrupt structures, processor scheduling algorithms and memory allocation algorithms. Another feature of workstations is their economy: one can purchase several workstations in order to demonstrate distributed algorithms, to investigate system architectures that improve reliability and to experiment with network services.

While we were designing our laboratory, Dartmouth College became an Apple University Consortium member which allowed us to purchase Macintosh® equipment[6] at a substantial discount. Therefore we could obtain the necessary equipment at an acceptable cost. Further, we believed that students would purchase large numbers of the same machine for their own use. At the same time, a decision was made at Dartmouth to use the network architecture being proposed by Apple (now called AppleTalk®[7]) to connect together all of the Macintoshes on the campus. Therefore we could expect to find local expertise on the construction of networks using Macintoshes. Thus we made a decision to run our laboratory on Macintoshes with AppleTalk.

Using Macintoshes. Although the Macintosh might be familiar to the students and relatively inexpensive, we required substantial hardware and software facilities to use the Macintoshes in our laboratory. We were generally satisfied with the computational power of the system. The original Macintosh is a Motorola 68000 based system using between 128K and 2M of RAM. It contains a sound driver, an internal disk drive (approximately 400K bytes), a bit mapped display (minimally configured with 512 by 342 pixels) and a mouse. A 64K ROM provides some predefined operations for graphics, fonts, menus, windows, various device drivers, and other system utilities. Although useful, all of the predefined operations may be replaced by routines in RAM or ignored by a program entirely. Full access to the underlying hardware is supported.

We also required a reasonable collection of network software and hardware for connecting Macintoshes together. We used materials from four sources: Apple, Kiewit (Dartmouth's Computation Center), Tanenbaum's text on computer networks[8] and some materials we developed ourselves.

From Apple, we acquired the basic AppleTalk software and documentation. The AppleTalk network architecture can be used to connect up to 32 hosts on a broadcast network. Multiple networks may be

connected through bridges. A host uses a carrier-sense, multiple-access with collision-avoidance scheme to place packets on the network. The lowest level of the architecture uses an unreliable datagram protocol called LAP (link access protocol). The current implementation runs at 230.4 kilobits/second over a twisted pair of wires. The next higher layer, Datagram Delivery Protocol (DDP), provides unreliable datagram delivery across multiple AppleTalk networks. A name binding protocol, a routing table maintenance protocol, a zone information protocol and a transaction protocol all reside at the next higher layer. From Kiewit, we acquired the KSP (Kiewit Stream Protocol) libraries that implement a reliable byte-stream protocol.[9] Because we were using a Pascal development system, we were able to implement directly several Pascal algorithms from Tanenbaum's book. Finally, we developed a set of libraries and programs that implemented a substantial fraction of DoD's Internet protocols (TCP/IP).[10,11]

When the course was run, our laboratory contained approximately 20 Macintoshes and 4 Lisas® (Macintosh XLs®) for use in the course -- sometimes more Macintoshes and Lisas were available when students or faculty left their personal machines in the laboratory, sometimes fewer were available because of broken power supplies, mice and keyboards. Students wrote programs in Lisa Pascal and Motorola 68000 assembler (as appropriate), and cross-compiled them on the Lisa for use on the Macintosh. Most Macintoshes were minimally configured, that is, had only 128K of memory and no extra peripherals, but we had several machines with 512K of memory, several external 400 kilobyte disk drives and one 10 megabyte internal disk. The Lisas could also be configured to act like Macintoshes with a megabyte of memory. We built cabling, terminators and related hardware to support seven experimental networks which could be wired in many fashions. Students could produce hard copy output by attaching one of several Imagewriter dot-matrix printers to a Lisa or Macintosh, or by uploading the program (or bitmap image) to one of the time-shared computers and using a shared printer.

Students were permitted to use the equipment in any reasonable fashion. They were given 24 hour/day access to the materials. In addition, many students (and their roommates) owned Macintoshes that were connected to the campus AppleTalk network. Therefore they could use the equipment in their dormitories for some aspects of the course.

5. Our Experiences

The Computer Network Laboratory class was designed to be a joint course between the Mathematics and Computer Science Department (COSC 88) and the Thayer School of Engineering (ENGG 128). The students taking this course would have completed prerequisites at the intermediate level in large system design and in computer architecture.

We offered the computer network laboratory as an elective during the 1985 winter quarter (about 10 weeks). We had 20 students, most of whom were junior or senior computer science majors. Most of the students had a reading knowledge of Pascal and had used the Macintosh for turn-key applications, such as word processing and figure drawing, but only one had any experience writing a stand-alone Macintosh program. We spent about two weeks teaching students how to write simple Macintosh applications and how to use the network software packages.

The students had to produce one program a week for the first six weeks. Most programs required the students to investigate some network feature. For example, a Maze game illustrated a broadcast protocol, a sliding-window algorithm illustrated a byte-stream protocol and a time-server program provided experience with soft layering of protocols. For each assignment, we provided a shell of a Macintosh source program for handling most of the operating system (Toolbox) overhead, a working sample solution

(runnable application only) that could be run in a Macintosh and a "broken" sample solution (runnable application only) that violated the protocol in some unspecified way. Our syllabus is reproduced in the appendix "A Network Course"; a more detailed description of the laboratory assignments can be found in the apendix "Laboratory Assignments".

During the last four weeks of the term, the students worked on their own projects (usually two people to a project). Some of the projects were:

- 3-D Maze Game;
- 2-D Maze Game that allows maze walls to be added or removed during play;
- AppleTalk-Internet gateway with dynamic routing for load balancing;
- Multiconversation "CB" system that allows arbitrary subsets of participants to converse without interference from others (multiuser Unix Talk);
- Name and File Servers;
- Telnet and TCP packages;
- DoD Internet gateway for moving IP packets between AppleTalk and a serial connection.

6. Some Unique Student Experiences

For the most part, the projects were of the student's own design and construction. They applied the theories and techniques taught in class to construct a working network system. In doing so, they became aware of several issues to which they had no previous exposure. Three of these issues, consistency, security and reliability, are discussed below.

Consistency. Because students were working with several versions of a program on more than one machine, they were faced with real problems of consistency. Not only did students have to ensure that each machine was running a working version of a program, but the program had to guard against some incompatible, older version of the program somehow remaining on a network. In contrast, a new release of a program on the central machine was installed over the old version and everyone used the new release. On the network, there was no way to recall old versions of programs. This was especially true for early releases of game programs that were distributed over the campus network to friends. Hence programs had to be written to recognize and take appropriate action when faced with inconsistent data generated by older versions of a program.

Security. A related problem concerned security. Many programs made assumptions that only well behaved clients would use a protocol. Although gross errors in the protocol could be handled, a malicious program could cause inaccurate data to be introduced into many clients and the general strategy of a protocol subverted. A simple example came up in the context of game programs. As long as all programs accurately reported the actions taken by players, all players could be competitive. However, students soon learned how to write programs that could send data that would make a player appear invincible: players could move just before bullets came; scores could be adjusted arbitrarily during a consistency check; other players could be ignored. Indeed, these kinds of "malicious" programs were frequently designed along with the actual application as a way to monitor performance, test certain protocol features and do other maintenance operations. Within our environment, the distribution of the maintenance program usually accompanied the distribution of the application, thus the security of most systems built in class could be easily compromised. The availability of similar workstations outside of our laboratory allowed students who distributed their programs outside of class to circumvent the little protection provided by the network.

Reliability. The students also had to deal with issues of reliability. Machines on the network, and even

the network itself, could be disconnected or refuse to participate in a protocol. Programs had to validate information in a way that students never experienced: there was no guarantee that "reading" a remote location that had been "written" previously would yield the same value. Noise on the network could corrupt data; messages could be duplicated, arrive out of order or never arrive. Tradeoffs between tardy, reliable data and recent, unreliable data had to be accommodated. These issues of reliability were never explored on the time-sharing utility because the operating system took great care to make the user's interface look reliable. The ability to use a personal workstation allowed students to experience unreliability and experiment with ways to control it.

7. An Optimistic Future for Laboratories in Distributed Systems

Our experiences with using workstations in a computer network laboratory were quite positive. We feel that the students learned a great deal from the environment; according to the class evaluation forms, the students agreed. Much to our surprise, the ratings given by the students for the value of the course and for the material learned were the highest that either of us have seen at Dartmouth. Perhaps the best estimator of our success was the number of anecdotes told to us by students who performed experiments of their own design that had no direct relation to any assignment -- they just wanted to satisfy their own curiosity.

The students' and our feelings seem to match the expectations of instructors from other computer science departments: the use of workstations in this context was novel and exciting; few believed that we could run the course as we did. Colleagues at Brown, Cornell, Stanford, UCLA and Carnegie-Mellon tell us of the many ways of using personal computers as replacements for their time-sharing systems, but few gave applications that let students exploit the ability to change the operating system, to alter the network capabilities or to write distributed applications. From our conversations, we found an increasing use of workstations' bit-mapped graphics in courses. However, all of the systems provided by the instructors were intended to be used as computer utilities, much like current computation center facilities. In contrast, the systems we wanted our students to build were to be highly experimental and usually unstable. We did not use workstations as a replacement for central time-sharing computers or as advanced graphics terminals.

Our focus on letting students have full access to a workstation's facilities also qualitatively changed the nature of the students' experiences. In general, students do not write privileged or distributed programs on computer utilities. Therefore they faced neither the technical issues of consistency, security and reliability, nor the social and ethical implications of their systems. Because we encouraged our students to explore the interactions and performance of real systems, we ran into one controversy: protection of privacy vs needs of maintenance programs. As we mentioned above, students learned the value of maintenance programs, and wrote them as necessary. Since the campus uses the same kind of network as we provided in our laboratory, the maintenance and monitor programs written by students for the class's experimental networks function equally well on Dartmouth's public networks. The knowledge of how to build and use these maintenance programs became a minor controversy on campus; the local student paper wrote editorials questioning the wisdom of allowing students to have monitoring programs that could be used on networks that carry private and possibly confidential information.[12,13] Although issues concerning privacy and ethics have always been included in computer science courses at Dartmouth, the issues now have a tangible feel which we hope students can appreciate better.

Distributed systems and networks are becoming an outstanding feature of the computer science landscape. Therefore we should be prepared to provide substantial laboratory materials to aid students in learning the principles, features and limitations of this technology. We found the construction and use of a

computer network laboratory to be feasible and effective. Macintoshes are sufficiently inexpensive that many can be purchased for use as a computer network laboratory, yet they are powerful and sophisticated enough to permit explorations of technical issues that can not be pursued on central time-sharing computers or even workstations used as computer utilities. Because students can gain access to the most secure levels of the operating system in a workstation, they can affect the availability of resources and violate the privacy of fellow students. Thus students who participate in a course like ours are forced to consider the security and ethical implications of their work.

We are sufficiently encouraged by our use of workstations that we are preparing another course, this time in software engineering, that uses the capabilities of workstations. We already use the same equipment for our computer architecture course and the Physics Department offers a laboratory-equipment interface course using similar equipment with special controllers, amplifiers and sensors. All three courses recognize the importance of distributed systems and networks, and each course discusses how that technology affects the subject. Through this article, we wish to encourage other universities to view workstations as a new, flexible resource that can be used to enrich courses well beyond a simple replacement of time-sharing machines for service courses.

8. Acknowledgements

Two students who took the Computer Network Laboratory course, Joel Margolese and Jeff Chase, reviewed this article and offered helpful suggestions.

Part of our equipment was provided by a grant to Dartmouth College by Apple Computer. Our IP packages were based on the PC IP implementation for the IBM PC developed at MIT[14]; our development of the Macintosh IP materials was supported in part by a grant from Apple Computer to Carnegie-Mellon University.[10] Other course development was supported in part by a grant from the Mellon Foundation and in part by a grant from the General Electric Foundation.

9. References

1. Marc H. Brown and Robert Sedgewick, "Progress Report: Brown University Instructional Computing Laboratory," *ACM SIGCSE 15th Annual Technical Symposium on Computer Science Education,* Philadelphia, PA., 1984.

2. Marc H. Brown and Robert Sedgewick, "A System for Algorithm Animation," *SIGGRAPH '84 Conference Proceedings,* Minneapolis, MN., July, 1984.

3. H. Kocak, M. Merzbacher and M. Strickman, *Dynamical Systems with Computer Experiments at the Brown University Instructional Computing Laboratory*, Technical Report CS-84-14, Department of Computer Science, Brown University, Providence, RI. 02912, June 15, 1984.

4. Karen E. Smith and Elisabeth A. Waymire, *Brown University's Computerized Classroom: An Experiment in the Principles of Courseware Design*, Technical Report CS-84-18, Department of Computer Science, Brown University, Providence, RI. 02912, October 2, 1984.

5. Edward Balkovich, Steven Lerman and Richard P. Parmelee, "Computing in Higher Education: The Athena Experience," *Communications of the ACM*, Vol. 28, No. 11, November 1985, p. 1214-1224.

6. Apple Computer Inc., *Inside Macintosh*, 20525 Mariani Ave., Cupertino, CA. 95014,1985.

7. Apple Computer Inc., *Applebus Developer's Handbook*, 20525 Mariani Ave., Cupertino, CA. 95014, 1985. (Sometimes referred to as *Inside AppleTalk*.)

8. Andrew S. Tanenbaum, *Computer Networks*, Prentice-Hall Inc., Englewood Cliffs, NJ., 1981.

9. Richard Brown, The *Kiewit Network Stream Protocol (KSP)*, Technical Memo, Kiewit Computation Center, Dartmouth College, Hanover, NH. 03755, December 18, 1984.

10. Mark Sherman, *A Network Package for the Macintosh using the DoD Internet Protocols*, Technical Memo, Department of Mathematics and Computer Science, Dartmouth College, Hanover, NH. 03755, 1984.

11. Stanford Research Institute, *Internet Protocol Transition Workbook*, Network Information Center, SRI International, Menlo Park, CA. 94025, March, 1982.

12. Esther Schrader, "'Peek disk' raises Computer Science ethics doubts: used for 'Networking,'" The Dartmouth, HB 6175, Dartmouth College, Hanover, NH. 03755, January 8, 1985, p. 1-2.

13. Editorial, The Dartmouth, HB 6175, Dartmouth College, Hanover, NH. 03755, February 12, 1985, p. 10.

14. John L. Romkey, *IBM PC Network Programmer's Manual*, Technical Memo, Computer Systems Research Group, Laboratory for Computer Science, MIT, Cambridge, MA. 02139, 1984.

15. Gursharan S. Sidhu and Alan B. Oppenheimer, *AppleTalk Printer Access Protocol*, Technical Memo, Apple Computer, 20525 Mariani Ave., Cupertino, CA. 95014, February 15, 1985. (Also published as part of *Inside LaserWriter* which is available from Apple.)

16. Samuel J. Leffler, William N. Joy and Robert S. Fabry, *4.2BSD Networking Implementation Notes - Revised July, 1983,* Technical Memo, Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA. 94720, 1983.

17. Danny Cohen and Jon Postel, *The ISO Reference Model and Other Protocol Architectures*, Technical Report, Information Sciences Institute, University of Southern California, 4676 Admiralty Way, Marina del Rey, CA. 90291, 1983.

18. J. H. Saltzer, D. P. Reed and D.D. Clark, "End-To-End Arguments in System Design," *ACM Transactions on Computer Systems*, Vol. 2, No. 4, November, 1984, p. 277-288.

19. Bob Lyon, Gary Sager, J.M. Chang, D. Goldberg, S. Kleiman, T.Lyon, R. Sandberg, D. Walsh and P. Weiss, *Overview of the Sun Network File System*, Technical Memo, Sun Microsystems, 2550 Garcia

Avenue, Mountain View, CA. 94043, January, 1985.

20. Anthony West, *The Sun Network File System, NFS -- Business Overview*, Technical Memo, Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA. 94043.

Appendix I: Development Systems for Students

I.1 Development Tools

A successful computer laboratory needs sufficient tools to allow students to perform their work effectively and efficiently. Especially among manufacturers who offer reduced or restricted systems for "educational use", there is a widespread belief that student facilities need only minimal support. In fact, students need the most advanced tools to which they can gain access. All of the usual needs to improve productivity that are asserted for commercial projects apply equally well to students -- those alone should justify the tools. Because of the academic calendar, students face some unusual situations that demand sophisticated tools for their projects: exceptional learning curves, productivity requirements and strict deadlines.

Exceptional Learning Curves. A typical 10 week quarter will have about eight assignments, one per week. Since academic regulations generally prohibit assignments due-dates during the last two weeks of a term (to prevent finals being surreptitiously given during the regular term), students are required to produce some relevant piece of software during their first week in class. The students must therefore learn the editor, decode compiler diagnostic messages, understand any provided packages and interpret the documentation before they can start on the intellectual content of the first assignment. Unlike an industrial environment, students are expected to work *without* collaboration of their coworkers. Therefore each student must learn a great deal of incidental material before any real work can actually be performed. A good set of tools can greatly minimize this effort. As a simple example, a structure editor can save time by making sure that a program is syntactically correct. By the time a student gets to an advanced course, he or she understands the concept of a block and should not waste time learning that PL/1 uses the predefined identifier **do** and requires a semicolon while Pascal uses the keyword **begin** and has no semicolon. Good tools can improve the learning curve of a system and permit a student to spend time on relevant aspects of a course.

Productivity Gains. A second reason for good tools is to minimize the unproductive work that a student performs. A typical student programming environment requires the student to perform source control, backup and even the basic edit-compile-link cycle in a primitive or manual fashion. Even when the student is familiar the system software, a student can waste enormous amounts of time verifying a particular linker switch or breaking a program into small enough units so that a compiler's symbol table does not overflow. Although this time might be considered useful if one wants to experience "real world" problems, the time is wasted in that less effort is spent learning compilers, operating systems or computer networks.

Unfortunately, the problem here is not only with unavailable tools, but also with poorly written assignments. In examining the materials used in a variety of courses at a number of schools, we saw exercises of such breadth that realistic scaffolding for the problem was as large as the final program. Although learning how to build scaffolding is useful in general, it is irrelevant to the objectives of most assignments. Therefore instructors must take the responsibility of ensuring that sufficient supporting materials are available for assignments. Together with good tools, students can concentrate on the content of the assignments and not merely on the form required by the systems programs.

Hard Deadlines. Like industry, students have real deadlines. Although a grade penalty can lure some students into meeting the weekly deadlines, it is the end-of-term deadline that causes the most problems for a student. The problems we are concerned with are not whether a particular program is written or not, but whether the student has had an adequate opportunity to study the material in the course. In industry, one can usually slip "one more week". When a term ends at a university, there are no more weeks to slip;

the instructor cannot cover another lecture, extend another assignment, or require another reading. Therefore students must be able to finish their work expeditiously. The end-of-term deadline is immutable; missing that deadline usually means misunderstanding some course material. Therefore an instructor should try to provide good enough tools that expedite the completion of assignments.

I.2 Some Requirements for an Educational Environment

A reasonable environment for software laboratories in computer science should include the following kinds of tools:

Quality editors — At the very minimum, a good screen editor should be provided. Better would be a programmable editor, such as Emacs. Still better would be a structure editor for the languages being used. Ordinary text editors with a good string matching and replacement facility, batch editors and file-difference locators are also valuable aids.

Translators — A fast compiler with good diagnostics is needed. For large projects, a good, fast separate compilation facility is needed. For intricate algorithms, a good interpreter for the same language can help immensely. The translators should either provide or be integrated with symbolic (source level) debuggers and performance analysis tools.

Source control — Large projects are built out of many parts and will require tools to build and manage libraries. Students need a way to organize their materials and coordinate efforts among many people. A source control tool turns out to be critical when some members of a student project live on campus and some live off campus. Such people cannot effectively leave notes for one another and rely on technological aids to keep order in their chaotic system.

Mail systems — A good mail system is needed to record notes on design decisions, communicate about meetings between team members, ask for extensions from the instructor and coordinate people with vastly different schedules who cannot attend face-to-face meetings.

Printers — Students need fast, convenient, good-quality hard copy for their work. When a computation center piles all listings for an entire day for all users on a table in random order, the students waste both their own time and computer time resubmitting print requests for lost (or unfound) listings. 3-hole punched, notebook-sized paper is far more convenient than large, fanfolded 14-inch paper. As students start to use bit mapped displays for design documents, the printing facilities for these graphics must be provided as a matter of course, not as a special device for which students are charged $0.50 a page.

Backups — Although students should take some responsibility for backing up their work, there should be some straightforward facility to assist them. For example, a student should be able to upload an entire disk of materials to the campus utility for safe-keeping.

Application dependent support tools — Special projects may require special tools. Some projects should have access to a parser generator, lexical analyzer or load generator. For example, we provided two such

> tools for our network class. The first was a network spy that could record all traffic on the network. Students used this tool to observe how their programs communicated over the network. The second was a packet injector, which could place an arbitrary kind of packet on the network. This allowed students to feed erroneous packets to their system for testing purposes.

Our list is not intended to be exhaustive, but we feel that the current state at most universities is far below even the suggestions above.

Appendix II: A Network Course

Our Computer Network Laboratory course was designed as an elective course for students to gain experience with network protocols and distributed systems. Through experiments, we wanted students to observe the effects of unreliable communication, unstable network routing algorithms, window size on stream protocols and malicious network agents. We also wanted students to construct broadcast-based applications, to use soft layering of protocols, to build a server and to build a client. After students completed our laboratory exercises, we wanted them to design and implement a network project of their own choosing. We believe we met our objectives using the course format described below.

The course met twice a week. Each class meeting lasted 105 minutes with a 5-10 minute break between 50-minute halves. One class meeting a week was devoted to general network theory while the other was devoted to a discussion of the week's laboratory assignment or a case study. Below we give a week-by-week description of the material that was covered in each lecture.

Week	Theoretical Part of the Course
1	Overview of Networks: layering, ISO-OSI model description and evaluation
2	Network Design Issues: topology and its effect on performance; design of network topologies, review of graph theory
3	Physical Link Level: basic communications techniques
4	Data Link Level: flow control, error recovery
5	Network Level: network-wide control, queueing theory results
6	Routing, stability, multiple networks, bridges, gateways
7	Local area networks: contention, token passing
8	Transport Level: communications services for user processes, remote procedure calls, message passing operating systems, virtual sockets
9	Distributed computing systems: synchronization, two phase commit, load sharing
10	Future of Computer Networks: technology trends, privacy, legal aspects, international networks

Week	Laboratory Part of the Course
1	Overview of Macintosh Operating System and Toolbox, Lisa Development System
2	Overview of AppleTalk architecture and implementation
3	Overview of DoD Internet Protocols, MacIP package, implementing soft layering of protocols
4	Broadcast and distributed protocols
5	Connection initiation, flow control, security issues
6	Design of gateways, bridges, addressing, connecting heterogeneous networks
7	Guest Lecture/Case Study: SUN's Network File System; Project ideas
8	Guest Lecture/Case Study: Apple's Printer Access Protocol
9	Final project review
10	Guest Lecture/Case Study: Practical issues of wiring dorms with AppleTalk, remote evaluation of network performance, maintenance problems, building codes.

The readings for the course included documentation on the Macintosh's Toolbox,[6] documentation on AppleTalk protocols,[7,15] documentation on various DoD Internet protocols,[10,11,14,16] documentation on locally developed network protocols,[9] critiques of the ISO-OSI model[17] and protocol design in general,[18]

overviews of Suns' NFS protocol[19,20] and Tanenbaum's textbook on computer networks.[8]

Appendix III: Laboratory Assignments

The laboratory assignments were intended to provide experience with the design and performance of several network architectures. In particular, the students had to become comfortable with the available development facilities, the general organization of the communications software and a variety of protocols.

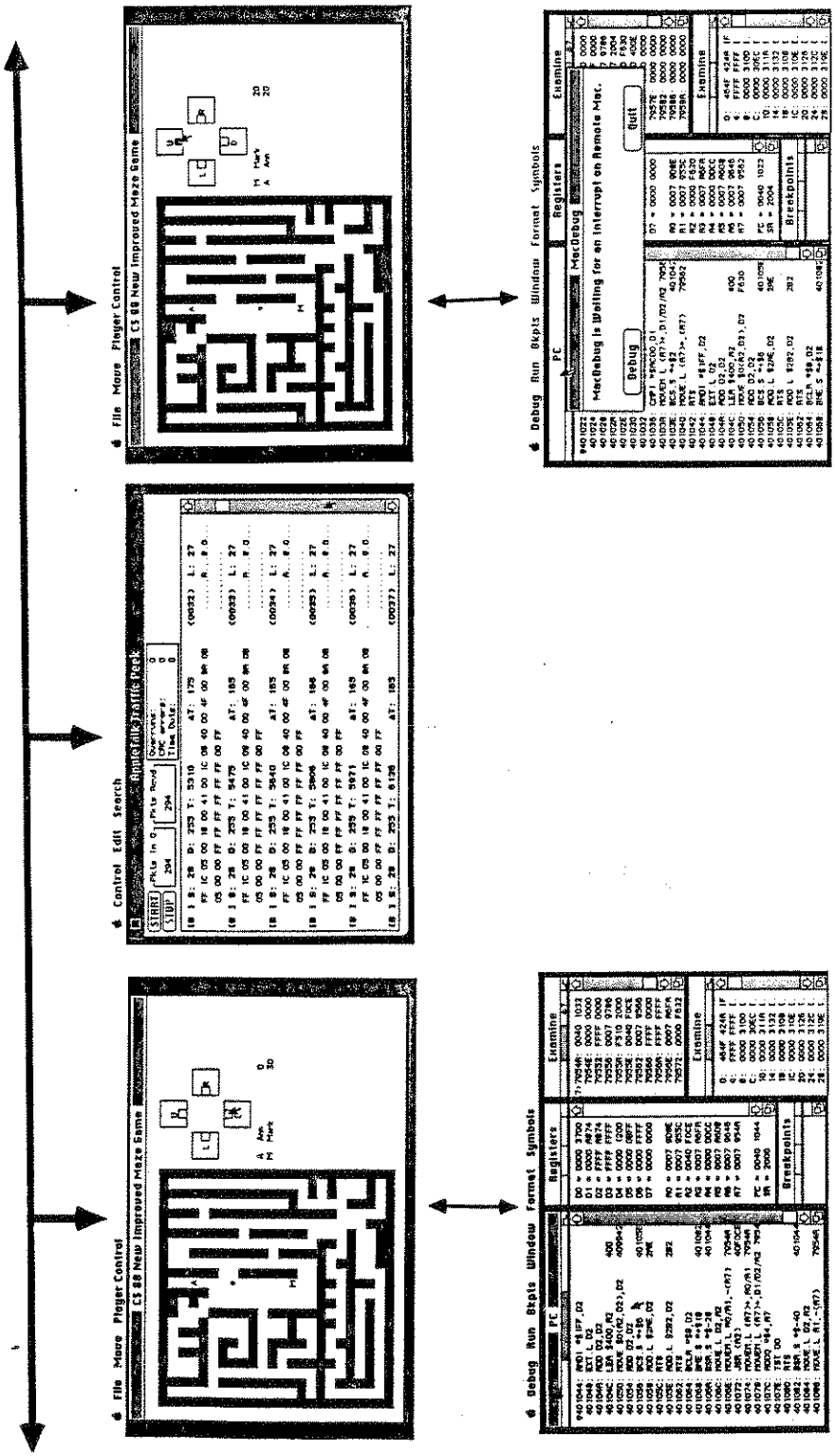
Our plans originally included a hardware laboratory where students would experiment with a network interface and observe attenuation, echoes and other phenomena. Unfortunately, we could not assemble the necessary facilities to provide this experiment. If we had had our hardware laboratory equiped before the course started, we would have assigned the experiment between the second and third laboratories listed below.

In addition to the hardware and software facilities of the Macintosh, students were also given accounts on a Unix machine where they could use the mail system, print files and backup disks.

For each assignment, the students were given a handout that described where to find appropriate documentation (essential given the sheer mass of documentation provided), a working sample solution (runnable object code only), source for the sample solution with the relevant parts of the assignment removed and a broken sample solution (runnable object code only). We gave the partial source for the sample solution to minimize the busywork for students. Most programs for the Macintosh follow a standard form that initializes various facilities, dispatches events to appropriate routines and maintains the visual coherence of the screen. We wanted the students to concentrate on the network issues and not on Macintosh details. Therefore we provided the sources for correct interactions with the operating system and toolbox, and let the students add the network facilities. For example, the provided source for the "talk" program in laboratory 5 created the two windows and provided procedures for writing and maintaining the text in the windows. The students wrote the procedures for timeouts, retransmitting packets, initial connection protocol, and so on.

We also provided broken programs, that is, programs that superficially seemed to work. However, the programs would violate the requirements of the assignment in various ways, for example, the broken program for laboratory 5 would send packets in the wrong order, duplicate packets, ignore received packets and sporadically refuse acknowledgements. Students used these programs to aid their debugging, and we used the programs for evaluating the assignments. (As a practical matter, we combined both correct and broken solutions in one program that had a switch for controlling its mode of operation.)

We tried to give the student a reasonable debugging environment for each assignment. A typical debugging configuration used five minimally-configured Macintoshes as shown in Figure 1 on thenext page.



Three Macintoshes -- the ones across the top of the diagram -- are connected to an AppleTalk network. Two of the Macintoshes run the student's application program which communicates via whatever protocol is being studied. The figure shows the sample solution for laboratory 4, the Maze Game, running on the machines at the ends of the figure. Underneath each machine running the Maze Game is another Macintosh acting as a debugger. Debugging machines are connected to application machines by a serial link and control the Macintosh running the application program. A programmer may set breakpoints, examine and change memory, trace instructions and perform the usual debugging operations on the application machine through a multiple-window, mouse-oriented program on the debugging machine. Between the the two application machines is a "spy" -- a promiscuous listener on the network. It can display all network traffic in a packet format. The spy machine is used to observe how the two application machines are communicating. Although typical, the exact configuration in Figure 1 is not essential. Debugging machines may be removed, application machines may be added, multiple networks can be bridged (possibly using multiple spies), and a "packet injector" program can be run on a Macintosh connected to the network. These and other configurations were constructed by students during the course. Indeed, an essential element of the course requires reconfiguration of equipment for investigating different protocol issues.

The students were given 7-10 days for each laboratory listed below. Although our environment does not completely meet our set of criteria for development tools, our students did not complain about a lack of materials, an undue amount of busywork or the total time to complete an assignment. We did receive several complaints that some assignments did not include references to documentation that were, in fact, required for completion of the assignment.

Assignments

1. Practice using the Lisa to program the Macintosh: write a Pascal program that can create a window, write text to that window, present dialogs to read strings and numbers, and manipulate various dialog controls (radio buttons, push buttons, check boxes); learn how to move files between Macintoshes, Lisas and Unix.

2. Practice using the Macintosh Protocol Package and its tools: wire together several networks, write a Pascal program that can send and receive DDP packets, distinguish broadcast packets from single-destination packets, accommodate packets of many different types, maintain several socket connections, validate packets, inject packets into the network via the Poke program, trace packets via the Peek program.

3. Write a Time Server and a Timer User: write a program that implements the DoD Internet time server protocol. The program should use UDP for packet delivery (UDP was provided). Multiple, simultaneous requests should be handled correctly. Write another program that asks the time from the server. The UDP package is implemented with soft layering, so the time server must be fit into the same paradigm.

4. Implement a distributed Maze game: write a program that uses a Maze-Game protocol for playing a version of Maze War. The protocol describes how players report their positions and interactions with other players. Students must also implement features of the protocol for players entering the game and leaving the game. No "master" host can be used -- the protocol is egalitarian.

5. Implement a reliable byte stream: Use the P5 (full-duplex, sliding-window, byte-stream protocol with

retransmission on timeout and piggy-backed acknowledgements) protocol given in Tanenbaum (P6 for extra credit -- P6 uses a more sophisticated acknowledgement technique). Write a program that implements the Unix "talk" program on the Macintosh, i.e., divide the screen into two windows where the left window displays all characters typed on a local machine and the right window displays all characters typed on a remote machine. The program should use AppleTalk's Name Binding Protocol for connection setup.

6. Measure file transfer protocols: Write a program that uses a half-duplex protocol (packet/acknowledge) for transferring a file using DDP. Compare the performance of the program with a (provided) program that implements the DoD Internet Trivial File Transfer Protocol (also a half-duplex protocol but with two extra protocol layers: UDP and IP).