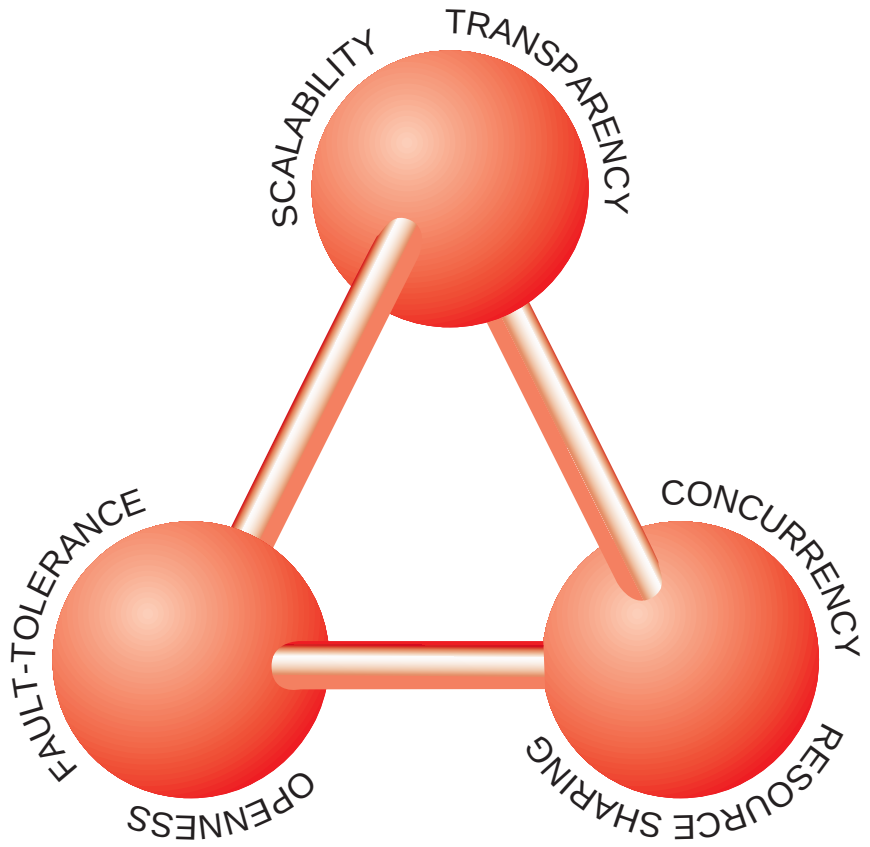
Distributed Systems

COSC 5000

Mark Zieg



Dr. Mike Moody

July 21, 1997

1.10 Discuss the concept of *shared resources* in a distributed system. Give examples of resources that might be shared. Say what the implications of resource sharing are for:

(a) the types of application that can be supported;

(b) hardware costs in a distributed system.

Shared resources are among the primary benefits of any networked system, including distributed systems. Both hardware and software resources can be shared, including printers, databases, and internet connections. By sharing resources, companies can save money while providing a richer variety of higher-quality resources to users. Shared resources also make possible entirely new categories of applications, including many extremely successful "killer apps" which would have been impossible to implement without access to shared resources.

Printers were some of the first and most popular resources to be shared across a network. In the school where I last worked as Network Manager, we had about 200 client nodes (Macs and PCs) on our network, each of which required access to a printer. Even by purchasing the cheapest dot-matrix or inkjet models available, we would have easily incurred $40,000 in capital expenditures and encumbered an immense burden of consumable costs had we tried to outfit each individual workstation with a local printer. Instead we networked ten color inkjets, five high-speed black-and-white laser printers, and one high-speed color laser printer, all for less than $20,000. As a result, each user received reasonably convenient access to a variety of printing devices, all of which provided greater output quality and/or speed than an economy-priced local printer could have.

Software applications and files can also be shared across a network. For example, also at my school, I installed a CD-ROM server that provided LAN-wide access to multimedia encyclopedias, which allowed concurrent users to query and retrieve a variety of data formats. These were "read-only" shared databases which extended functionality to older workstations which weren't equipped with CD-ROM drives. I also developed a "computer work order" database that teachers could use to electronically submit computer service requests. That was a read/write shared database which allowed concurrent add/modify/delete transactions. Both of these were software resources being shared across a network.

Of the two, the work order system was the most significant, because it represented a distributed system which could not have been implemented without resource sharing (the multimedia encyclopedias, in contrast, could theoretically have been implemented without resource sharing, at a much higher cost, by purchasing individual CD-ROM drives and encyclopedia programs for each potential user). A genuine multiuser transaction system can be implemented only through common access to a single shared database (whether physically distributed or centrally consolidated).

A number of so-called "killer apps" have evolved in the marketplace to take advantage of this functionality. One of the most common is the nationwide Automatic Teller Machine network which allows access to shared bank databases from remote terminals. Even more common are the credit-card authentication systems used by Visa and Mastercard to provide up-to-the-minute cardholder account information to merchandisers across the country. One of my favorite commercially successful distributed systems is the Amazon.com virtual bookstore, which is completely dependent upon its

shared database of titles and reviews. Another oft-cited example in business trade journals is the Wal-mart inventory system, which uses a distributed client-server approach to greatly increase the efficiency of their distribution and inventory procedures.

1.13 What is openness?

Openness is the characteristic of a system describing the ease with which it may be interfaced by other systems, configured to meet specific needs, and extended with new capabilities and functionality. In a commercial context, the trait usually represents the ease with which one vendor may connect to or augment another vendor's product. There are three main ways to make a system open. The first is to design the system in a modular, structured fashion with consistent module interfaces and APIs, typically with some system of hooks, interrupts, or events to which future modules may be linked. The second is to publish those interfaces so that other programmers may write modules which abide by them. The third is to avoid hard-coding configuration details, instead allowing users and administrators to conveniently modify key system variables which affect the system's suitability for a particular application or environment.

Unix is an essentially open operating system because it is composed of a great many independent modules which generally conform to a standardized calling syntax and share a common understanding of input, output, and error devices, communication pipes, etc. Any one of those modules (like the gcc compiler) can be replaced or enhanced without necessarily disrupting the rest of the OS. The system is also readily configurable through straightforward editing of the various ".conf" and ".cfg" files scanned at boot time. Moreover, since much of the source code for Unix is in the public domain (at least Linux still is and, I think, FreeBSD), the interfaces are eminently accessible to programmers. Likewise, many of the applications and protocols frequently associated with Unix, such as vi/emacs, NCSA httpd, and TCP/IP are open in the sense that their interfaces are well-known and easily customized or interfaced at very low levels.

Besides its convenience for programmers and system administrators, openness is becoming a requisite for commercial success in consumer software as well. Two very popular examples of applications which owe much of their success to their open and extensible nature are Adobe Photoshop and Netscape Navigator. Both support proprietary, but well-documented and published, frameworks for extension through additional modules written by third-party vendors. Popular Photoshop filters include Kai's Power Tools and Alien Skin's Eye Candy, both of which vastly increase artist productivity. Similarly, much of Netscape's popularity is due to the ease with which users can download and install third-party plug-ins which add support for video streaming, vector graphic formats, and Acrobat files. Other software products which support this kind of plug-in architecture, and have thus spawned entire industries of third-party extensions, include the applications QuarkXPress, PageMaker, and Fractal Painter, and of course operating systems like Windows95 and the MacOS.

However, openness is not limited merely to software systems. Hardware systems can be made open as well by publishing and abiding by standard physical connections, pin-outs, and power requirements. The massive PC clone industry was made possible not only by Microsoft's popular (but essentially closed) MS-DOS operating system, but also by the availability of cheap, interchangeable modular components which could be mated to industry-standard EISA or PCI bus designs. Video cards, hard drive controllers, and NICs from different vendors could be plugged into a universal Intel-based motherboard and be expected to co-exist with a minimal amount of playing (and praying). The popular Centronics parallel print interface, RS-232 serial interface, and SCSI system peripheral interface are other examples of successful consumer-oriented open hardware standards.

1.16 Define concurrency and parallelism. What opportunities for parallelism arise in distributed client-server systems (for example a distributed UNIX system)?

Concurrency refers to multiple processes or operations that logically occur at the same time or that appear to be simultaneous to the user(s), but whose instructions may or may not literally be executed at precisely the same time. Non-parallel concurrency may be achieved through interleaving, time-sharing, or very fast throughput of queued requests. Parallelism refers to processes or operations which literally do occur simultaneously and therefore require multiple CPUs.

True parallelism remains a rarity on consumer- or business-oriented PCs because of the high cost of multiple CPUs and due to the cost and complexity of operating system and application software which can efficiently utilize multiprocessing architectures. On the other hand, parallelism has been popular in mainframe environments for years because the cost-benefit ratio of adding multiple processors to a large host system, which may support dozens or hundreds of users, can be very favorable. This is in contrast to concurrency, since process interleaving and multithreaded operation have become common features in most popular PC operating systems.

Distributed systems, on the other hand, encourage many kinds of parallel operation by their very nature. Since distributed systems tend to spread their workload of services and resources across multiple computers, even low-end single-CPU computers can partake in the fun of parallel processing. The advantages of such parallel computation are manifold. Non-locking database operations (read requests, for instance) can be serviced simultaneously by the resource managers of replicated database files on different servers. Floating-point intensive rendering operations can be processed in parallel by sending different objects to different client workstations for local rasterizing.

When I was a sophomore at UCF, a friend of mine who was the current captain of our ACM "A" Programming Team engaged in a particularly unique form of distributed parallel competition. There is an annual international Othello contest in which contestants write Othello programs which compete against one another. My friend's program, written in Turbo C and 8086 Assembler, used SPX/IPX calls to distribute its game tree search and evaluation (based on Shannon's algorithm) across 92 Intel 80386 PCs networked over ethernet. As a result, it was able to search very deeply and

very quickly. On the day of the Othello contest, he and I had to be at Florida International University for the Southeast regional ACM programming tournament (another unrelated contest), so he had a friend execute and monitor his Othello program and phone hourly reports down to us in Miami. Therefore, he himself was competing in parallel in two different contests, one of which involved a massively parallel distributed system. In the end, he placed first in the Southeast regionals and placed second in the world with his Othello program, which lost only to an anonymous computer from the north known simply as "Harvey". A great day for my friend.

1.17 How can the design of a distributed system ensure that it will be scalable?

The essence of scalability is that any resource utilized or accessed by a system may be assumed to increase or decrease in quantity arbitrarily. The connected Internet is a example of a system which has scaled up gracefully and with relative ease. When Microsoft was developing WindowsNT, they obviously gave some thought to scalability because they were able to later produce not only the Wolfpack server clustering system but also the scaled-down and minimalist WindowsCE with comparative ease. Novell's NDS, introduced in Netware 4.0 and made usable in 4.1, is also an extremely scalable directory system.

The primary design method for ensuring scalability is to assume that any resource could exist in quantity (specifically, matching the regexp /x+/, indicating one or more) and provide a mechanism for selecting one instance of the desired resource from a pool. Such quantities should theoretically be allowed to grow without limit, but for the same of efficiency and performance can usually be assumed to cap-out at about 10-1000 times the current projected usage and availability (in other words, leave 3 to 10 address bits free for future expansion). That means that typical resources such as file servers, printers, and even display consoles should not be hard-coded in as single entities, but programmatically selected from a list of functionally similar resources, even if the list begins as a set of only one element. Furthermore, when coding the name spaces for available printers, servers, users, etc, hash tables composed of linked lists would be preferable to static arrays (or at any rate, the arrays should be capable of dynamically resizing when they are almost full or empty).

(Actually, to also meet the companion requirement for fault tolerance, systems should be coded to deal with /x*/ instances of each resource—ie, 0 or more—so that they will be able to calmly and predictably prompt users or system administrators when a needed resource cannot be acquired, rather than simply crashing with an unhelpful core dump.)

The Internet's phenomenal growth and popularity with commercial and consumer users over the past two years is directly attributable to the availability of plentiful IP addresses for corporate LANs and for ISPs serving home dial-up users. There are so many addresses available because the protocol was initially designed with a 32-bit address field, even though projected usage in the 1970s and 1980s could have been easily been met with a 16-bit field at most. By "wasting" a little bandwidth and adding a little to

network latency time, the designers left room for the massive hordes of online users we see today. However, even this generous name space is beginning to seem cramped as hardware vendors and operating systems begin to assign IP addresses not only to CPUs and user sessions, but to printers, monitors, and even telephones. As a result, there are already plans to introduce IP version 6, which will replace current 4-byte IP addresses with 16-byte versions, providing plenty of IP addresses for every television, VCR, and toaster oven in the world—at least as far as we can see today ☺

1.18 Define fault tolerance.

In our household, fault tolerance can be best summarized as "Jonathan-proofing", which is a reference to our two-year-old little boy. Anything made of linoleum, vinyl, or formica is fault-tolerant; most things made of glass, fabric, and paper are not ☺

In the computer world, fault tolerance goes by other names: robustness, crashworthiness, and idiot-proofing are common idioms. Each of these refers to a program's (and by inference, programmer's) ability to deal well with the unexpected. Typical faults which a well-written program must tolerate gracefully run the gamut from the mundane (printers running out of paper, floppies needing to be unlocked, e-mail addresses to be spell-checked) to the catastrophic (construction workers cutting through network cable, custodians tipping over file servers, leaky roofs in wiring cabinets—all of which I've experienced first-hand). In fact, while teaching middle school, my computer labs had to deal with students shoving bubble gum into disk drives (followed by a disk and a forced reboot); kids sticking bent paper clips into wall electrical outlets and blowing entire circuits; Pepsi poured into keyboards; well-meaning teachers plugging DB-25 serial devices into SCSI-1 ports; and network cables stuck into digital phone jacks, which were then dialed and rung (producing a surge of electricity which blew out NICs and motherboards).

So long as the program in question is not actually running on one of these abused systems, but merely depending on them as available resources, the program should be able to detect the fact if not the type of the error and enter some sort of fallback state. Besides the typical log to STDERR and some sort of visual notification to the user, a program might also need to trigger a rollback if the fault has caused data to enter an undefined or unpredictable state. Also, if the system was designed in accordance with the creeds of scalability as outlined above, the program may then be able to select a comparable resource from its internal tables or external name and binding services and continue operations on the newly acquired replacement resource.

For instance, the internet firewalls at Tribune corporation, the parent company of The Orlando Sentinel and the source for our frame-relay cloud, uses four BSD Unix stations for our firewall. As it turned out, three of them blew up last week, which slowed our internet connection to a crawl—but it did keep working, because although the system was intended to be scaled and load-balanced over four servers, it was sufficiently fault-tolerant to keep running on only one. Best of all, the entire problem was completely

transparent to the users, who noticed a perceptible slowdown but otherwise did not have to reconfigure their machines or deal with any error messages.

Some of the ways programs may be designed to be fault-tolerant include automatic data validation and integrity checking, redundant mirrored backups, RAID data storage, event-based error handlers, support for transactional rollbacks, or inclusion of a remote notification system such as built into the fabulously robust Compaq Proliant servers (which can page an administrator in the middle of the night if a resource is suspected of nearing or entering a fault state).

1.22 A department has a multi-user computer, a few workstations and a number of terminals and is planning to extend its computing resources. Make the case for installing a distributed system rather than another multi-user computer and terminals.

Having faced such a decision before, I can attest that the issue is a good bit more complicated than the question suggests. In fact, I've already seen one mainframe-terminal system replaced with a poorly designed and badly supported distributed system, and the department in question is currently in the process of moving back to a centralized system. Before I could recommend such a migration in good faith, there are a number of preliminary questions which must be addressed.

1) Assessed computing needs

What does the department use these computers for (which begs the question of what kind of department is it)? The truth is, many typical departmental applications can be served perfectly well from time-shared terminals. What kind of work are these terminals performing? What are the bottlenecks? Are there kinds of data the terminal users need to access but cannot? Has reliability ever been a problem? How important is fault-tolerance to the business unit? Has the number of terminals concurrently supported by the host ever presented a limiting factor? Are there new features or functions which users would like added to the system but which have been difficult or impossible to incorporate into the current system? Has there been a problem interfacing with outside systems?

Depending on the answers to these questions, a distributed system may or may not be of advantage to the department. For instance, if the department has difficulty interfacing with a new corporate SQL server, is running mission-critical apps that can never be allowed to fail for more than a few minutes without direct fiscal repercussions, and the current host has few expansion options to allow it to support new users, resources, and functionality, then a distributed system could be introduced to solve many of these problems. On the other hand, if the nature of the work is primarily text-based data entry screens, a backup host has already been arranged through an affordable third-party vendor, data interchange with other systems can be processed handily with a handful of perl scripts, and users are content with the current feature set and response time, then maybe a better alternative would be to upgrade the current host with more RAM, secondary storage, and terminals.

2) Upgrade options

Before buying (or certainly developing) any new system, at least cursory attention should be given to opportunities for expanding existing system investments. If the current host is proving to be a bottleneck due to a heavily utilized CPU, consider whether additional (or more powerful) processors can be installed into the main box. Likewise, if frequent thrashing occurs due to overrun cache buffers, explore the availability of RAM upgrades. If the impetus to move toward a distributed solution is to gain additional client stations, check whether new terminal modules can be added to permit multiplexing of additional terminals, especially across remote links. If software connectivity is a driving issue, contact vendors to see if a gateway is available to convert data to a standard interchange format.

3) Cost of porting software applications

This is a step I learned the hard way: porting applications is never as easy as expected, and every day spent re-coding an application that was already working (until you decided to move it to a new platform) is money essentially thrown away unless the port results in measurable gains in profitability (the only benchmark most managers and bean-counters are interested in—promised gains in productivity and efficiency can be surprising ephemeral and difficult to verify). Although it would be hard to arrive at a fixed figure, I estimate the cost of porting an application between significantly different platforms (such as COBOL to C++, or one multi-user host to distributed servers) to be at least a third the cost of developing the app in the first place—sometimes considerably more, because in its own way porting can be more difficult than rebuilding from scratch, especially when crossing paradigms.

4) Training and empowerment

Training is one of my least-favorite "hidden costs," because it is one I seem incapable of remembering when doing my own migration budgets. Technologists look at new WYSIWYG GUIs and graphical front-ends like Access 97 and immediately think how much friendlier and easier to use they are than green or amber 80-col CRTs. However, from the perspective of many users, it's just so many commands to re-learn. Figure a minimum of 20 to 60 hours of re-training time per employee, just to get them up to speed at the same level of competancy as they had on the old system—again, money in the furnace.

Graphical operating systems also tend to have many more ways for users to get themselves into trouble. The old DASI terminals I used to support almost never needed adjustment because they only had a few CMOS configuration screens and you needed a password to get into those. Not so Windows95! Every user wants to click that "Start" button and see "where they want to go today"—which is often not where you wanted them to go. Just something to think about now, because you'll certainly be reminiscing about it in the future when every user discovers the joy of visiting the Dilbert Zone when they should be entering monthly sales tallies.

5) Future expansion plans

I believe that this is often one of the best reasons to migrate to a distributed system, yet it is one of the least-often put forth on budget justification because it lacks the sense of immediate gratification. The fact is, information system technology is advancing so quickly that managers intuitively know that they can't afford to remain stuck with obsolete technology and communications structures forever—eventually they'll need to connect to a modern system which just refuses to talk EBCDIC, or add a custom application which can't be compiled under CTSS. Managers know this. They also know that part of playing the "technology game" is sometimes blowing some money just to stay dealed in, because if you ever fold just to save some short-term dollars, you're forever cut out of future winnings. Up to a point, staying current is itself an end worth pursuing.

That's where expansion plans come in. Even if you are running fine today, what will your needs be tomorrow? Classic host-terminal systems are notoriously cranky about expansion, especially when users start demanding bizarre features like gaining access to raw data so they can create their own Excel crosstabs and bar charts. If your current system is less than open about communicating with outside systems, accepting third-party extensions, or dispersing processor and storage loads across multiple systems, then it might be worth migrating just on the likely bet that someday soon you'll be needing those features, even if today you don't.

Reasons to take before your resident bean-counter include the following (if the cost of migration is too big to sneak into a weekly expense account under a description like "renew subscriptions to trade journals ... $5,000,000"):

Scalability: describe the massive expansion you'll need to undertake when next year's revenues go through the roof and the company has to ramp up production and services. Nobody is willing to nay-say a prediction like that or they'll look like a poor team player.

Cost Savings: show how much money you'll be able to save by sharing expensive resources like color laser printers between dozens of users (you can fail to mention that any laser printers at all will be added expenses, since everyone currently crowds around the two giant line printers in the mainframe room).

Actually, those are the only reasons I'd mention. Most time-sharing systems already appear to be concurrent, so that would look like a dead end, and the potential performance gains from a genuinely parallel solution would only suggest that current employees are spending time dawdling between screen updates—time when they needn't be paid. Likewise, if you went on about the increased fault tolerance of distributed systems, you'd only be suggesting that your current disaster-recovery plan is inadequate—and that would be your fault as system administrator. Finally, concepts like openness and transparency are simply too obtuse for the average non-technical person to grasp.

In conclusion, never commit a department or business unit of any size to migrate to a new platform merely because news magazines promote a new

bandwagon (or because a question in a textbook prompts you to). Distributed systems represent a powerful and flexible new paradigm of information system design, but they are not necessarily appropriate for every situation, and the benefits they provide do not always justify the cost and complexity of a departmental migration. On the other hand, if you perceive that your legacy system will one day become a corporate burden rather than a competitive advantage, starting the move now to a distributed system will give you the flexibility meet a variety of future computational demands. In short, competent department directors should approach new system paradigms with the same envious-but-critical gaze as a general contractor eyeing a shiny new Ford F-350 when he's got a rusty but otherwise perfectly servicable S-10 in the garage.

1.23 Discuss some of the consequences that might arise when one computer in a distributed system fails and the rest continue to run:

(i) if the failed computer is a workstation;

(ii) if the failed computer is a file server.

If a workstation fails in a distributed environment, you've got one user who's likely in a foul mood but a whole department full of peers who continue to happily process away. In fact, unless the workstation in question managed some shared resource (such as providing the physical link between the network and a popular office printer, or locally storing a spreadsheet file) the rest of the stations should be able to chug along without even noticing their comrade's plight.

(Two minor exceptions: if the operating system allowed processor sharing between nodes, as built into NEXTstep for instance, then other users might notice a slight slowdown of their systems; on the other hand, if the workstation crashed because the user was constantly downloading large image files from the naughtier corners of the internet, they might actually experience a surge of released network bandwidth.)

However, if the failed computer is a file server, then every user on the system could experience difficulties, depending on the degree of fault-tolerance that was implemented. If the resources managed by the server were replicated across multiple servers, then users could continue working, albeit more slowly, from the reduced server pool (cf 1.18). In fact, if the system was designed with a high degree of transparency, users might not even notice the fault.

(An exception: since the failed computer was specifically identified as a file server, it is likely that one or more users could experience loss of any changes to open files since the last replicated update. That could represent anywhere from ten minutes' to a week's worth of lost work. Also, any blocking transactions which depend on the server would probably be forced into an indeterminate state and trigger rollbacks, which would result in a bit more lost work—but typically not very much.)

1.24 A student laboratory contains a number of networked workstations for use by various groups of students. Any particular student

Having taught in a variety of networked computer labs, I've got very specific ideas on how to deal with this. In fact, one such lab of Macintosh LC III computers had exactly 80MB of internal storage in each unit, and another lab of Apple IIgs computers had floppy drives but no hard disks.

comes to the laboratory from time to time and does not necessarily always expect to use the same workstation. How do you suggest that the students' files should be stored:

(i) if the workstation has a small hard disk (for example 80MB);

(ii) if the workstation has a floppy disk drive, but no hard disk?

First of all, I'm a strong believer in server-based storage for any student lab for several reasons. First (and second, and third), *students will "lose" floppy disks!* They will be forgotten in lockers, at home, bent in textbooks, folded in notebooks, crammed into pockets then dirtied on playing fields, run over with bicycles, used as frisbees (a very popular sport, as the sharp corners of 5.25" disks can actually be lodged into drop-ceiling tiles), etc, etc, etc. In short, if you expect students to bring their work in each day on floppy disks, expect fully 50% of those disks to be unusable by the end of the project—leaving you at a loss at how to grade the students, since they probably did do at least some of the work, but you have no way to retrieve the data.

The solution is easy: don't let any files leave the classroom! Some teachers compromise by keeping locked disk boxes in the classroom so that students may simply leave their disks with the teacher, but I've found that to be inadequate in most cases (and an unnecessary strain on the teacher to be accountable for such a collection of disks). I much prefer to keep everything on the file server. File servers are relatively cheap when compared to the total cost of outfitting a lab—it's hard to equip a 25-station lab with hardware, software, and peripherals for less than $40,000, and a decent file-and-print server can be added for only $2,000 to $5,000. Moreover, since a single server can easily support a half-dozen labs without strain, the system would be a good investment for the entire school.

Besides guaranteeing availability of student work, there are several other advantages to a server-based solution. For instance, it makes it much easier to grade projects, since the teacher can simply sit at his or her desk and sift through the various class directories. Also, the instructor/network manager can include the student files in automated backups, further reducing students' excuses for not having their work done. On a fully networked campus (mine was), this also allows students to access their individual files from stations outside the classroom, such as a general-access lab in the school media center.

However, there are a few problems associated with server-based storage. The biggest problem I had was with students accessing other students' work and copying their projects in whole or part. That can be solved with user-based security methods (I tried group-based security tied to individual classes, but they just copied and deleted their classmates' work instead). However, the administrative overhead involved in configuring and managing individual security clearances for 120 students each semester can get to be a bit of a headache. My solution was to have a student assistant generate and configure the individual student accounts and then lump them all into a single group to provide general trustee rights, leaving each of them a single private login directory with a storage limit of 1MB or so. Issuing passwords can also be a pain, but that pain can be eased by starting each password as the student's first name, then configuring their account to require a new password (of 5+ characters) every 3 logins or so. Many will forget their own passwords, but that's easily overriden from an administrative station at the teacher's desk.

You'll notice that in all this discussion I haven't distinguished between stations that are equipped with hard disks and those without. That's because,

as far as data file storage is concerned, I would (and did) treat them the same, for the same reasons. The only advantage of hard drives on student stations is it allows a graphical operating system and a handful of application programs to be stored locally, which speeds boot time and application response time considerably. However, I would feel perfectly justified in totally—*totally*—locking the local hard drives of every student station so that they couldn't so much as save the time of day without generating an error message.

Fortunately, there are a variety of commercial security packages on the market to allow you to do just that, most priced very attractively for academia. Note that on Intel machines you'll need to configure the CMOS to use a C:,A: boot sequence instead of the default order, and then you'll need to password-protect the CMOS to keep enterprising students from simply booting off their own DOS disk and wrecking havok with your system. Even then, I've had clever 12-year-olds bring in screwdrivers to open the CPU case, remove the battery for a few minutes until the CMOS resets, and then hack themselves in just as pretty as you please. Those kids are typically the ones I chose as my teaching assistants, to keep them out of trouble as much as anything else.

2.5 Distinguish between flat and hierarchic naming schemes. Under what circumstances is each most useful?

Flat naming schemes keep all of the identifiers from a particular namespace in a single logical set; there is no perceived nesting of objects within other objects, and any ordering and filtering is generally performed only at the reporting level. As a result, duplicate identifiers are typically not allowed (or are permitted only for items of differing types), and there is a relatively low useful boundary on the size of the table. The biggest advantages of flat naming conventions is that they are very easy to implement and are extremely fast for small name spaces. The biggest problem is that they don't scale well when table sizes grow much beyond a hundred entries or so.

Hierarchic schemes, on the other hand, encourage a tiering or stratification of leaf objects within group objects. This allows similar items to be grouped logically with other items possessing similar attributes. It also permits duplicate naming of leaf nodes within different groups, since local context sufficies to uniquely identify each duplicate. The biggest advantage of hierarchic naming conventions is that they are typically scalable, with very little modification or extra effort, to thousands or hundreds of thousands of identifiers. The only real problem with hierarchic structures is that they require extra effort on the part of the original system designer to develop. They can also be less efficient than flat name spaces for extremely small name spaces.

To be honest, it's getting harder and harder to find examples of flat name spaces. I first started using MS-DOS at version 2.11, which I believe had just begun to support directories and subdirectories. I did use Apple DOS 3.3 for a while, which supported only a single flat directory, but as soon as the first Apple hard drive came out, ProDOS came with it, supporting a contextual hierarchy of directories. It's worth noting that the driving force behind both transitions occured when hard drives became commercially viable. Flat name spaces worked just fine on floppy disks, which rarely held more than a

hundred files. However, hard drives promising storage of thousands of files demanded a more structured approach to maintain response time and provide a convient organizational metaphor to users. (Happily for some, NetWare 3.x remains a champion of flat name spaces to store users, printers and servers, and looks like it plans to live forever in spite of it's deeply hierarchic 4.1 cousin.)

It's important to distinguish between the name space as experienced by the user and the underlying data structures used by the programmers. Both flat and hierarchic name spaces can be easily implemented in a B-tree for fast searching and updating. For that matter, a hierarchic structure could be implemented in a flat data structure if anyone felt like doing so. Also, some nominally hierarchic name spaces aren't as structured as they could be. For instance, the ".com", ".edu", and ".org" structure of the internet DNS service remains a remarkably flat organism. The number of ".com" domains alone is well into the hundreds of thousands, which can be treated as a bloated flat table which happens to fall within a specific hierarchical group.

The choice of name space structures is usually up to the system designer, although more and more customers are becoming saavy enough to request easily-scaled and -managed hierarchies. On a recent web-based vacancy database I designed for the Osceola County school system, for instance, I choose to use a flat name space to hold the table of school entities. I based that decision on the small number of schools involved (26), the projected growth rate (1 school per year), and the small amount they were paying me. In contrast, had I provided the same system to the Orange County school district, I probably would have taken the trouble to make the name space hierarchic in recognition of their much larger system (130+ schools).

2.9 Explain the difference between monolithic and open operating systems.

Monolithic operating systems were built around the assumption that the mainframe was to be the primary, and often sole provider of computational power for the business unit or academic department. Since all computing requests went to the mainframe, it made sense to design the mainframe operating system to perform every function. For the sake of security, efficiency, and error-proofing most of these functions were bundled into the system kernal where they could not be inadvertantly modified by devious or bungling programmers. This model worked for a time, and it was a good time, until...

...the dawn of personal computing, when a couple of hackers named in a garage brought a useful degree of power to the desktop, and a bright young programmer with the soul of a lawyer sold a watered-down operating system to the boys at Big Blue. Henceforth, a trend emerged which steadily reduced the stream of requests flowing into the glass cages of corporate IS, and instead users began finding their own solutions to computational problems. Suddenly, or so it now seems, the workflow model went from "user → mainframe → user" to "user [→ mainframe] → user", with the host sometimes being cut out of the loop completely.

This brought a change in the design of host operating systems. No longer could system designers assume that they would be sole source for specific functions or applications. Even services traditionally buried deep into OS kernals, such as file access and name resolution, could now be provided off-host, and with greater accessibility and fault-tolerance to boot. Customers also began to insist on greater customizability and flexibility than could safely be delivered from within a kernal.

As a result, kernals took their cue from their old friend CPUs and "shrunk their die size." By removing everything not directly related to basic resource management (memory, processes, basic communication and peripheral calls) to outside the kernal, system integrators became free to selectively install just those services they needed, and configure them as they chose. Services outside the kernel could also be designed in a much more open manner, permitting levels of inter-system communication and collaboration never before available.

It was that ability to share service information between hosts which really made distributed systems viable. When key services remain locked inside a kernal, it is very difficult to implement such key distributed features as resource sharing, openness, and transparency. On the other hand, services implemented outside a kernal can all be designed to support distributed systems by incorporating the rudiments of distributed system design.

Unix is an oft-cited example of a monolithic OS, since it packs a large degree of the system's total functionality into its megabyte kernal. Recent outgrowths of the Unix family line, such as Mach, have retained many of the classic Unix features but compartmentalized them and moved them outside of the kernal to create a much more open environment.

2.11 Discuss the efficiency of use of the processor and memory resources in a distributed system based on a simple workstation-server architecture. What techniques can be used to improve it?

In the simplest workstation-server distributed system there will almost always be instances of inefficiently applied resources. In particular, key physical resources such as processing power, RAM, hard drive space, and network bandwidth tend to be non-uniformly distributed during periods of high load. That occurs because these resources are necessarily physically tied to individual nodes (workstations) and servers. Although system designers can provide mechanisms to share access to these resources across a network, such access will always be constrained by the latency of the network connection and will be almost certainly slower than a direct physical bus connection.

For instance, an engineering team may have five workstations on their lab floor. Three of those stations may be in use by rendering or circuit-routing software, which involve very high levels of computational power and are well-suited to multiprocessing. One of the other systems could be in use by a technical writer producing flowcharts of the next project phase (relatively low computational load) and the other could be displaying an idle-mode screen saver while its operator carries on a telephone conversation. As a result, three of the stations will be maxed-out on computing power and leave their users in an effective blocking state until they have completed their tasks. Meanwhile,

two other high-power workstations will be sitting by running at perhaps 5% and 1% of their potential utilization. This represents an inefficient use of resources because the collected workstations are running at less than 70% utilization in spite of the fact that three operators are in blocking states.

The most likely design improvement to deal with such situations is to allow each unit to share a portion of its processing power—ie, CPU and RAM—with other stations, so long as such sharing does not inordinately crimp its own processes. This is analogous to the peer-to-peer file sharing which has existed on Macintosh computers for about seven years, and which has recently appeared in Windows 95. In each case, the user can configure how much of his or her hard disk to share across a network, and with whom. The NextSTEP operating system, based on the Mach kernel, takes this approach one step further by allowing the user to specify whether he or she wishes to share the station's processing power as well, with whom, and how much of a slowdown the user is willing to tolerate.

By incorporating such technologies into a distributed system, processing bottlenecks are reduced by allowing CPU-intensive processes to take advantage of parallel processing opportunities on other available processors—in effect, emulating a processor pool architecture "on the honor system." In the example provided above, the three CPU-intensive tasks could have been completed up to 40% faster without unduly inconveniencing the other users. This not only provides a faster, more responsive, and more productive environment for the users, but also saves money by allowing existing capital to be utilized fully before requiring additional upgrades or stations.

2.12 Give reasons for and against equipping workstations with local disks.

There are two main kinds of local disks which may installed on workstations, each of which provides its own set of advantages and complications. The earliest such decisions dealt with the inclusion of floppy drives (removable media), which were the only kind of local disk drive affordable to many users. More recently, a new round of questions has arisen about the pros and cons of local hard drives (fixed media).

Floppy drives are the oldest example of random-access digital local storage (I think), and still remain popular and surprisingly unchanged in spite of massive research and product development in the storage industry. The main advantages of floppy disks are their portable nature (leading to the first affordable local-area networks, "sneakernet"), their comparative cheap price (making backups affordable and easily accessible for most user-generated documents), and the removable-media aspect (which allows new software to be easily distributed, sold, and installed via diskette). All of these reasons appeal immensely to consumers and individual users, who like the ability to share files, load new programs, and backup valuable documents. Network administers appreciate a fourth quality of floppy disks, which is that it is very convenient to boot a system from a "clean" floppy diskette while troubleshooting, especially when data on the fixed drive is suspected of corruption.

However, it is those same network managers who express the biggest concerns about local floppy drives. Namely, the ability to share files implies the ability to illegally copy software off the system or install new applications which aren't licensed or even introduce conflicts in key production applications. Furthermore, the days of "free computer love" are over; every user must practice safe computing or risk exposure to one of the thousands of viruses circulating on various platforms. Networks are particularly susceptible to such contagions, as an entire LAN can be brought down by a single infected floppy data disk. Finally, security-conscious companies (or schools) may find that local floppy drives can often be used to circumvent a number of surface security levels, at least on the client side.

There are different issues involved in adding or removing local hard drives. Once upon a time, in the early days of computing, there were things called "dumb terminals" which featured little or no local storage—not even RAM beyond that required for a video display and small I/O buffers. These terminals lived and died at the whim of their mainframe host, which stored everything deep within its centralized banks. This could be described as the "Epoch of IBM," as their mainframes and operating systems played central roles in many terminal-host environments. In time, that computing model was replaced by the current environment of powerful PC and workstation clients storing much of their own data, and only sending requests to a central server when they need something outside their local cache. This time may best be described as the "Microsoft Era," since their operating systems were dominant factors in devolving power out to the desktop.

Now it seems that the pendulum is swinging back the other way, and a return to "thin clients" may be emerging. The principal proponent of this model, Larry Ellison of Oracle, hopes to usher in a new "Ellison Eon" in which the prototypical client will be a diskless "NC" (network computer). The NC, in turn, will keep all of its applications (written in Sun's Java language, supposedly) on a central server (which could well be distributed, one supposes). So surprisingly, after almost two decades of adding resources and peripherals to the desktop client, we're once again contemplating their removal. This time around, as before, the argument is that network bandwidth has again caught up with application needs and is once more capable of conveying OS and application code to the client as well as data. Well, almost—it turns out that Microsoft Office 97 actually takes a few minutes to launch even over today's high speed networks, so it was decided to re-code every application in the minimalist Java language, which will in some mystical manner optimize the bloated code that has become the norm for commercial productivity software. Without removing any features, of course. Oh, and it will also turn the trick of making a web browser seem like a perfectly natural environment in which to modify spreadsheet files and design 3-D circuit schematics.

The fruit of all of this technical wizardry (or is it pedagogery?) is that applications will be easier to upgrade, since only one copy will be stored on the central server, and that client workstations will be cheaper, not only for

the raw physical unit but also the extended TCO (Total Cost of Ownership). This is the sort of thing that makes accountants and network managers very happy, because it means that they have to do less work. It remains to be seen what the user response will be (which will primarily depend on what the network response will be, which has yet to be proven in field studies).

I personally suspect, however, that it will be games like "Doom" that prove to be the real killer-app of the NC phenomenon—and I do mean "killer" app, in that it will kill the idea of the NC for thousands of users. Why? Because with applications being stored on the server, it will be up to LAN administers to decide what applications they're going to allow. And when the network bandwidth becomes scarce (and it always does), it'll be games like "Doom" which are the first deleted from the system—if they're ever allowed there in the first place. And that's when the users will stage an anti-NC revolt, because once they get used to the personal conveniences of local storage—like texture-mapped games to play during lunch and "brainstorming" sessions (read "deathmatch"), they won't want to give them up. And as users, they'll be in the best position of anyone to sabotage the whole NC movement, by constantly complaining about access speeds, resource availability, etc.

The funny thing about technology is that it's stuck in forward gear—you can't throw it into reverse, and that's why I think local storage—at least as far as fixed media is concerned—is here to stay.

2.20 Distinguish between buffering and caching.

Buffering is temporarily storing data in RAM (or some high-speed medium) for quick and convenient access. Caching is a special instance of buffering in which the data being buffered is retained specifically because subsequent read or write operations are expected to be performed on that particular data block.

Besides caching, buffering is used for a variety of tasks. In the most general sense, virtually all data and code held in RAM can be considered to be buffered—in fact, in the C language, all memory blocks, arrays, and structures are explicitly referred to as buffers. Input and output buffers are frequently applied to concatenate many small I/O operations into one large block operation to avoid overhead costs (ie, disk seek time, network latency, etc). I/O buffers also allow different processes and devices to share data at electronic speeds, even when they process that data at different rates (ie, a fast CPU sending 25 pages of data to a printer buffer, which will slowly feed it to a 4ppm print engine). Finally, buffers are used as general-purpose holding areas for data which is not meant to be processed but merely passed on (for instance, ethernet NICs have small on-board buffers to hold incoming packets just long enough to run a checksum and determine if the packet is to be passed to the host CPU, routed to another network, or dumped).

Caches are special-purpose buffers which hold local copies of remote data to avoid communication costs (which may range anywhere from a microsecond load operation, in the case of a processor cache, to a 10-second internetwork server request in the case of a distributed database). Data is held in the cache for as long as feasible in the hope that future requests may be

satisfied from in-cache rather than necessitating the comparatively expensive option of retrieving a fresh copy from the remote source. So-called “write caches” are actually little different from the output buffers mentioned above.

2.24 Why are distributed systems intrinsically less secure than centralized computer systems?

There are many reasons why distributed systems are much more difficult to secure than centralized systems. The physically distributed nature of the systems provides a variety of opportunities for unauthorized access to data channels. Also, the software design requirements of a distributed system leave room for digital forgeries aplenty.

The network provides the single biggest obstacle to fully securing distributed systems. In order to achieve many of the desired benefits of distributing storage and workload—concurrency, transparency, fault tolerance, scalability, etc—data must pass routinely pass across a variety of circuits, routers, and so forth. In fact, in the case of network protocols such as token ring and ethernet, data is constantly being broadcast to and through computers which are never intended to read it. In those cases, all that keeps unauthorized users from accessing the data directly is the configuration of the network interface. Even if a potential snooper does not have access to a network node, or the network does not rely on broadcast or peer-to-peer routing, mere physical access to the data cables can allow an interloper to copy, analyze, or even modify packets as they go by.

There are a variety of ways to protect a distributed system at the network level. One of the most basic is to limit access to the physical network to whatever degree is possible. In the event that physical security cannot be assured—as is the case with any system running over the connected internet, for instance—data encryption is always a viable option. By using one of the many available RSA or PGP encryption algorithms, data may be exchanged across the most public networks with reasonable confidence that it cannot be read, and that any attempts at modification would result in checksum errors (note that data packets could still be blocked, but that’s more of a fault-tolerance issue than security). One of the earliest examples of encryption being used to protect a distributed system (of sorts) being promogulated across public communication channels is the ENIGMA system used by Nazi Germany in World War II.

Another large issue that security-conscious distributed systems designers must face is that of user and resource authentication. An integral concept in distributed systems is having servers manage resources, clients passing requests, and making sure the responses get back to the same client who requested the service. That whole system requires a reliable naming system which makes sure clients and servers are correctly and consistently identified. Without making a serious effort to authenticate that both clients and servers are who they say they are, a hacker could falsely identify himself as a client and make requests of services to which he should not have access. Alternately, he could position himself to receive server replies properly addressed to other legitimate clients. Finally, he could falsely identify himself as a server and accept requests for services he cannot or should not provide.

Again, the best solution to this problem is by employing a system such as Kerberos to authenticate that all participants in a client-server conversation are exactly who they claim to be. Indeed, thanks to the bulging prospect of e-commerce on the world-wide web, the internet has seen a deluge of new digital-certificate technologies emerge over the past two years.

3.2 What is the task of an Internet router? What tables must it maintain?

A router is responsible for passing packets between networks. Internet routers in particular are used to route IP traffic (although general-purpose routers can be configured to route a variety of protocols at once, such as IPX and AppleTalk). Routers are needed when a process on a computer on one network needs to communicate with a process on another computer on another network. The first process sends its message by passing it down through the layers of its protocol stack until it reaches the network layer. At that point the NIC attached to the first computer sends out the message across the physical network (using a broadcast ethernet message, or by grabbing the next available token, or whatever). Neither the process nor the NIC, however, are responsible for knowing whether the recipient computer is on the same network or a different one; that's the job of the router.

The router, being connected to the source network, receives the packet as it is being circulated through the network via whatever protocol reigns there. The router then analyzes the packet header and determines whether the target computer is on the original network or another network. If the packet is targeted for a computer on the source network, the router need do nothing and simply ignores the packet, assuming that the proper recipient will pick it up in due course. On the other hand, if the header bytes suggest that the recipient will not be found on the local network, the router must decide where to send the packet so that it will eventually find its way to the correct addressee.

To decide where to send the packet, the router must consult internal tables containing the addresses of other routers with which it can communicate, and information about the network to which those routers are attached. Hopefully the router will already know of at least one route between the source network and the target network. However, due to the massive size of the connected internet and to its dynamic nature (in which new LANs and WANs are added every day), it is predictable that the router will not always possess foreknowledge of the target.

There are two solutions I can think of to deal with this situation, and I'm not clear on which is actually employed by standard internet routers. First, the source router could simply make a "best guess" and send the message on to another router, hoping that the router at the next "hop" will know about the target network, or perhaps the hop after that. A better solution may be for the router to send a special routing request packet off to a bunch of different routers asking if any of them know where to find the recipient network. If any of them know, they could reply with a route to the target. If they don't know, then they could in turn pass on similar requests to other routers they know about. This would ultimately provide a "breadth-first-search" of the entire Internet until the target was found. To prevent a surplus of unnecessary

routing packets, some sort of "time-to-life" value could be encoded, perhaps informing every fifth router along the chain to directly query the source router and ask if the search had yet been concluded or if it should be continued.

Finally, the router attached to the recipient network needs to maintain a table of the addresses (IP addresses, in this case, or more likely just the subnet addresses) of the client nodes connected to its network.

3.3 What is the task of an Ethernet bridge? What tables must it maintain?

Bridges are like routers except that they only work with a single protocol. Ethernet bridges connect two or more ethernet networks. Ethernet bridges come in two main flavors: dumb and smart (AKA cheap and expensive, AKA chatty and efficient). The simplest sort are little more than repeating hubs and simply send all traffic from one network to every other network the bridge talks to. This results in an awful lot of needless traffic and henceforth collisions, although it does allow nodes on different networks to communicate.

A better solution, but one requiring more processing on the part of the bridge (and therefore RAM and circuitry, both of which cost money), is for the bridge to maintain a table containing the ethernet addresses of every computer on each network to which it is attached. That way it can choose to only bridge packets which are actually directed to nodes on different networks.

One complication would be the instance in which one bridge was used to connect LANs A and B, and another bridge connected LANs B and C. For a message to successfully pass from LAN A to C, one of these methods must be implemented: either each bridge must maintain comprehensive tables of every ethernet address of every network to which it is attached AND every LAN to which those networks may be bridged (and so on infinitum); or the bridge can simply broadcast packets addressed to unknown nodes to every attached network, figuring that some bridge will recognize them somewhere. (Or the bridge could try to interrogate other bridges to ask if they recognize the address, but that's really more of a routing function and beyond the demesnes of mere bridges, although the line between routers, switches, and bridges has grown blurred in recent years).

3.4 Describe the work done by the software in each protocol layer when the ISO Reference Model is implemented over an Ethernet.

The first five layers will neither know nor care whether the local network is ethernet or not. The topmost application layer provides protocols customized for specific applications and services. The ones I know best are all from the internet: FTP, telnet, SMPT/POP, and presumably http. I don't know what would constitute valid examples of application-layer protocols in Netware or Macintosh environments, but I suspect that NDS would be up there somewhere.

The presentation layer is charged with translating data into a portable format using standards such as XDR and Courier. I assume that this would translate EBCDIC into ASCII if necessary, for instance, but I don't know whether other forms of data conversion like uuencoding are carried out there or, more likely, outside of any protocol and manually performed by user processes. At least, I've always had to do my own uudecoding in the programs I write, so I'm not sure what the presentation layer was doing for

me. Likewise, this layer is nominally responsible for protocol-based encryption, but every implementation of encryption I've dealt with has been performed by user applications, so I'm not sure sure how this relates to modern office environments.

The session layer establishes and maintains connections between processes. This is an extremely important role in internet operations, which rely almost exclusively on the connection-oriented TCP protocol. The session layer in the computer of the source process communicates with the computer of the target process by passing a message down through the transport, network, data link and physical layers, across the network, then back up through the analogous protocol layers on the target computer.

Once a session has been established, the transport layer can send the message (or stream) from the source process to the recipient process. Different transport protocols subdivide messages differently, but all are responsible for segmenting long messages into chunks (ie TCP packets) of either fixed or variable length, tagging them with sequence identifiers of some sort, then passing them on to the network layer. The transport layer on the receiving end is similarly responsible for reconstituting the message from its component parts and in the correct order. Some transport protocols require acknowledgements of each chunk while others depend on resubmission requests from the recipient.

The network layer is first charged with determining a path from the source computer to the target computer, possibly across networks or internetworks. Note that for stacks based on connection-oriented transport protocols, the network layer is first called upon to determine a route by the session layer; by the time the first chunk of the content-message is ready to be passed, a route has already been established and the network layer shouldn't need to do any further route-detection. All that's left for it to do is keep passing message chunks (or stream packets) down to the data link layer with route information, usually in the form of a datagram with sender and recipient addresses encoded in the header.

The data link and physical layers are the only layers that have any interest in ethernet *per se*. The data link layer is duty-bound to accept each chunk (ie, data) from the network layer and pass it (ie, link) to the physical layer (hence the name, data link). Before doing this, the data link layer must determine the ethernet address of the receiving computer (at least for the first "hop," which is often to the nearest router).

Finally, the physical layer is comprised of the physical cables which stretch between nodes, servers, hubs and routers. In ethernet, this used to be almost exclusively "thicknet" or "thinnet" co-ax with locking BNC connectors, which lent themselves readily to bus or ring topology. More recently, CAT-3 to -5 UTP (unshielded twisted pair) and 10BASE-T has become the norm, with most new installations adopting so-called "star" or "homerun" topologies.

3.11 How can we be sure that no two computers in the Internet have the same addresses?

First of all, there are many pairs of computers on the Internet with duplicate addresses; I wouldn't be surprised to find several thousand duplicate IP values on any given day. This is partly due to rare but predictable errors on the part of otherwise careful network administrators, but more often due to the bungling of well-meaning but ignorant PC users who feel compelled to fiddle with their computer configurations just to see what happens (typically in a misguided attempt at self-instruction, which is usually a good thing but not when it comes to bringing a network down). Fortunately, having duplicate IP addresses on your network generally shouldn't affect you, unless (1) your address is one of the duplicated values, or (2) you rely on a resource such as a server or router which holds a duplicate address.

Usually I don't worry about duplicate addresses, because if it ever happens I usually get a phone call pretty soon from one of the affected users and I can resolve the problem at that point. I haven't tried this, but I imagine that you could ping an address and check the replies that come back to look for duplicates. Also, if I knew more about programming routers, I'll bet you could query the router directly for a list of all the IP addresses mapped to ethernet addresses on the local segments. At that point all you'd have to do is figure out which user maps to which ethernet address, which the part I usually have trouble with. I'm not sure if Novell lets you search user information with NDIR based on ethernet address or not, but I do know that you have to buy a third-party network management tool to pull that trick on a Mac network (I just did for $8,000). (Of course, under Unix you could probably just finger an IP address to get the contact info for whatever yutz has been playing with *ifconfig*, but I don't manage Unix networks.)

One way to circumvent this problem is to use DHCP or BootP servers to automatically assign dynamic IP addresses to clients. Besides guaranteeing that no two users will be issued the same address (at least not at the same time), this also allows a network manager to leverage a small number of IP addresses—say, a single class "C" license—into a satisfactory connection serving several hundred users (assuming that not everyone decided to check ESPNet at the same time). Another way to eliminate duplicate IP addresses is to not issue them at all, and instead use a gateway (like NetwareIP, MacIP, or one of the many SOCKS implementations) to route an entire LAN's internet traffic through a single IP host.

3.12 Explain the relationship between domain names and Internet addresses (IP addresses) in the Internet.

A domain name is an alphanumeric label mapped to a particular IP address for the convenience of weenie humans to whom tags like "163.193.177.64" prove too cumbersome. The mapping is one-to-many, meaning that a single IP address can have a number of different domain names, but one domain name can only point to a single IP address (this does not prevent server clustering, as a request to the "home" server can easily be handed off to a secondary server for load-balanced scalability). Also, a single computer can have more than one IP address, which has led to an entire new industry in virtual hosting.

Domain names are resolved by issuing calls to a DNS (Domain Name Service) process. Many networks simplify management by issuing "local" domain names which are not distributed out to the internet. Intranets in particular enjoy complete freedom to issue as many domain names as they want for internal use. Internet-accessible domain names take time to be replicated across the world's many DNS servers, however—figure eight hours before local changes are fully reflected worldwide.

Domain names are issued by the InterNIC registry service, and possibly by several new commercial registries—the issue seems to be in flux at this writing. There were originally only a few types of addresses: .edu, .com, .net, .gov, .org, and .mil covered most of the gamut. With the internationalization of the internet, however, domain names have been restructured to match the geopolitical location of the host: names like .co.uk, .ac.jp, and .k12.fl.us now proliferate. Personally I think this is an unfortunate trend, because it compromises a lot of the transparency that once made internet access so ambiguous and fun (which country am I talking to right now? I can't tell!)

At one time most addresses were free, but now all commercial addresses cost $50 per year with a 2-year minimum. Academic addresses remain free, fortunately. I believe I read that the domain name "business.com" was recently re-sold for a record-setting $100,000. One of the earliest and rudest ways people made money on the web was by registering dozens of likely commercial names like "windows95.com" and then re-selling them to the rightful trademark holders; I think that this practice has now been banned.

4.1 Compare sending a message with sending data over a stream.

The distinction between sending a message and sending data over a stream can perhaps be compared to the difference between voice mail and a telephone conversation. With voice mail, a static message of fixed length is sent in one complete body, and the recipient doesn't hear the beginning of the message until the sender has reached the ending and hung up. A telephone conversation, in contrast, allows the recipient to receive each "sound byte" as it is issued, permitting recieving operations (like taking notes or doodling as the case may be) to occur almost in parallel with the data transmission. Furthermore, the stream remains "open," even during the dreaded seven-minute lulls, until one party breaks the connection.

Unfortunately, that analogy breaks down in that a telephone conversation is a two-way stream while BSD streams are one-way (although two may be established to support bidirectional communication); analogies with e-mail vs. chat have the same problem. One could make a more accurate analogy involving live television news reports versus recorded VHS tapes, but I think the point's been sufficiently made and the horse is quite dead.

In any case, streaming technologies are becoming quite "the thing" on the internet. Almost no one bothers posting .au sound files on the net anymore, which require several minutes of silent downloading before the music starts (at which point you have an unused internet connection while you listen to the sound file play). Instead, the multimedia world is moving to the new ".ra" RealAudio files, which allow you to hear the sound as it is being streamed,

thus providing a theoretical doubling of user productivity (assuming that downloading sound files from *The Simpsons* can be considered productive...). Likewise with byte-served Acrobat files, RealVideo, etc, etc.

4.2 One way of managing the conversion of data types is for computers always to convert data into a standard form before transmission. Explain why this may be inefficient, and describe an alternative. Which of the alternatives should be used?

Universally translating every message into a common data format is inefficient for many reasons. For one thing, the vast majority of communication is already between similar computers, making most conversions unnecessary; it would seem to make much more sense to have the session and presentation layers collaborate to negotiate a common format when a connection is first established. In many cases that would at most necessitate encoding 8-bit binary files into 7-bit text versions (using BinHex and its kindred).

Furthermore, if a single "standard form" were adopted, it would almost certainly be drawn from the lowest common denominator and feature such infinitely-compatible but horrendously-inefficient methods as sending numeric values as alphanumeric digits (ie, using a whole byte of information to store a 10-state digital value). Besides wasting bandwidth by the bucketful, that adds tremendously to pre- and post-processing time.

If a universal data format were to be adopted, I would recommend that it include a type-descriptor signalling that the following packet utilized one of the following four formats:

- A straightforward format optimized for simple binary and textual data in which integers and floating point values are stored in a binary representation and other information was passed in ASCII (possibly with optional support for the new two-byte character sets), provided for easy and fast communication over high-speed networks.
- A totally flattened format in which all data was passed as 7-bit text, provided as a contingency-of-last-resort format should client-server negotiation fail to find common ground elsewhere in the protocol.
- A lossless compressed format based on an algorithm comparable to LZW (only without the licensing issues), provided for optimized transmission of specific binary data over low-bandwidth connections.
- A lossy compressed format using something like wavelet or JPEG compression for very fast transmission of content-uncritical binary data over low-bandwidth connections (like real-time multimedia data).

4.9 Explain why a process might need to have several ports.

When dealing with interprocess communication, it is often convenient for a single process to hold several ports. Heavily loaded server processes can maintain several "receiving" ports to expedite communication with multiple clients. Likewise, client processes which are communicating with several servers concurrently might like to have multiple ports waiting for responses. Also, any one process might appear as a client to one computer and a server to another, so it would vastly simply things if the process maintained separate input and output ports.

I imagine that streamed services like telnet and FTP (when used over TCP), in which an established connection might be held for hours at a time,

would require a unique server port for each concurrent client connection (suggesting hundreds of such ports for popular university servers). I haven't configured a web, FTP, or POP server in a while, but I remember that they asked the administrator to set the max number of "listen" ports, which I imagine is for concurrent connections. I also remember reading that modern web browsers can attempt to make extra connections in order to recieve multiple page elements (usually binary graphic files) in parallel.

4.22 Give a list of the main design issues and options for the design of group communication.

Group communication is a key tool in the design of distributed systems. Many key attributes of distributed systems such as fault tolerance, scalability, concurrency, transparency, and resource sharing can be implemented most efficiently using group communication (they can be implemented without group communication, but with far more difficulty and message overhead). Typical uses for group communication include replicated services, multicast queries, and multiple updates of redundant data.

In order to implement these operations, system designers must decide what guarantees must be made about data communications. For instance, if data are to be replicated across many servers, the data at all servers must be required to be at exactly the same state at all times, or else client read operations will return unpredicatable results, which is rarely acceptable. Furthermore, in the case of multiple clients and multiple servers, communications may or may not be required to be processed by every server in the same order. Solutions to these problems exist, but they vary in terms of efficiency, reliability, and cost.

One of the most basic problems is guaranteeing that each member of a receiving group receives every broadcast communication. This is known as atomicity, and is basically a plural version of the common "transaction" operation. Fully atomic communications are those which are known to have been successfully received by every member of a group and can therefore be trusted and acted upon. Although fully atomic communications are feasible, they are relatively expensive and therefore avoided unless strictly neccessary. Alternatives to atomic communication include reliable multicast, which guarantees that at least one member of the group receives the communication, and unreliable broadcast, which makes no guarantees whatsoever.

For some applications, even atomic communication is not enough, because the order of communications can be as significant as their reliable transmissions. There are four basic kinds of ordering which can be applied: unordered, totally ordered, causually ordered, and sync-ordered. Total ordering guarantees that messages will be processed in the same order by every recipient, but does not make any claims about what that order will be. Causual ordering seeks to determine a happened-before relationship between messages, and forces messages to be processed in the same sequence in which they were created and/or sent. Sync-ordering allows a series of messages to be synchronized at specific benchmarks to maintain data integrity. Unordered message processing is a basic FIFO unprioritized queue.

For critical applications which require the maximum reliability of fully-

ordered atomic broadcast, efficiency becomes an issue. As seems to be the rule in programming, the most easily coded solutions are usually the least efficient. Therefore, for time-critical applications it is worth the extra effort to develop or utilize group communication routines which take advantage of broadcast capabilities of local-area networks such as ethernet. Designers should also seek to minimize the number of messages passed between processes and thereby cut down on overhead traffic.

One common trick to do this is make acknowledgement messages the exception rather than the rule, by replacing common "success" acknowledgements with much more rare "failure" receipts. Known as "negative acknowledgement," this can be accomplished by attaching a small, serialized message count to the header of each transmitted message. Another technique to reduce network traffic is called "hold-back." In this method, undeliverable messages are left (or "held back") at network communication junctions until they can be delivered, usually as indicated by monitor or sequencer processes responsible for ordering and atomicity. This is generally quicker than sending a message all the way from a remote source.

4.23 Give an example in which the ordering of multicasts sent by two clients is not important. Give an example in which it is important.

Consider a replicated financial database which stores payroll data. Assume that a manager makes a change to reflect an employee's promotion and subsequent raise from $6 to $7.50 per hour, at the same time as the local United Way campaign representative sends a weekly command to deduct 0.5% of each employee's paycheck. If the commands were received at one server in the order (promotion, United Way), then the resulting deduction would be recorded as $1.50 ($7.50 x 40hrs x .5%). However, if the other server received the commands in the order (United Way, promotion), then the deduction would be recorded as only $1.20—producing a 30-cent discrepancy. Although this is a relatively insignificant amount, it points to a potentially catastrophic flaw in the system design which could rapidly lead to escalation and faults (or worse, audits).

Part of the danger with the flaw is that it could take some time to be discovered through actual use (this is why system testing needs to include a conceptual dataflow to isolate trivial cases which may not come up frequently in typical data sets). For instance, had the United Way campaign withdrawn a flat amount from each paycheck rather than a percentage, no discrepancy would have been observed ([pay + raise] - deduction == [pay - deduction] + raise).

4.25 Give an example to show the importance of multicast atomicity. Explain why a reliable multicast might fail to be atomic.

If data is being written or updated to a distributed database, then anything less than fully atomic multicast produces a chance—indeed, with many systems a likelihood—that data will exist in different states on different servers. That means that subsequent read operations may return data from before the update or after (transaction procedure handling usually prevents the special case of reading data while it is being updated). Sometimes that's a big deal; other times it's not. For instance, the distributed DNS database is in a constant state of flux and it takes several hours for an update made at one

node to percolate throughout the rest of the system. That means that a user who subscribes to one DNS server may get the "new" IP address for a particular domain name while another user who subscribes to a second DNS service may get the "old" IP address—or none at all. On the Internet, that's an accepted way of life and people have learned to deal with it. Likewise, NDS updates take a while to be replicated across a NetWare 4.1 WAN, and cc:Mail post office changes can take their own sweet time making it to every server. No big deal.

However, not all systems are as flexible. Consider the ignition systems controlling the launch of a space vehicle such as the shuttle. There are a lot of little motors, engines, ignitors, hydraulics, and pneumatics that have to move through a tightly choreographed dance in order to get that payload off the pad. Moreover, every space fan (and critic) knows that launches have a remarkable tendency to be cancelled or postponed with mere seconds left on the countdown clock. Imagine the disaster that would occur if a broadcast message was sent to the launch systems to cancel the ignition sequence, but only 95% of the effected systems received the "no-go" message. That would leave 5% of the systems believing that they were still supposed to fire, which could result in all kinds of unfortunate behavior (such as a single solid rocket booster igniting—and once one of those gets lit, you *don't* shut it off!). In such a case, it would be almost definitely preferrable for none of the systems to obey the shutdown message and instead continue with the lift-off in spite of whatever last-minute problem had emerged (such as learning that one of the astronauts had forgotten their toothbrush).

For this kind of system, which isn't all that far-fetched given the massive defense and aerospace industries in central Florida (imagine a bomb in which the ignition timer recieved the "go ahead" message but the release-and-drop mechanism did not), merely "reliable" broadcast doesn't quite cut it. That's because reliable transmission only guarantees that one or more group members received the message, based on the fact that a reply was sent back to the sender. "One or more" isn't good enough when you need total collaboration between every member of the team. For that level of commitment you need a fully atomic transmission, which means that every member needs to be guaranteed of receiving each and every message, because sometimes "almost perfect" is worse than "none-at-all".

5.1 Which of the parameters of these two procedures are *input* and which are *output* parameters?

Both of the parameters to the *Vote* function are input parameters and both of the parameters to the *Result* function are output parameters. The *Vote* parameters have to be providing input to the remote server, which would have no other way of recording who had voted or for which candidate. The *Result* parameters have to be providing output from the server, because the client process would have no other way of knowing which candidate won or how many votes he or she received.

5.5 **In an implementation of the *Election* service (Exercise 5.1) over UNIX, there is no need to supply users with "voter's numbers" because the users' UIDs can be used for this purpose. Explain how a *user package* can provide a simpler interface to *Vote* than the RPC interface.**

By building the *Vote* and *Result* functions into a user interface, the system designer can automate many tasks and insulate them from the user. In this case, the *Vote* function interface which the user sees need only have one parameter: the candidate for whom the user wishes to vote. After being called, the "stub" function within the user package can add a second parameter—the user's UID number—before issuing the RPC call. Although I'm not familiar with Unix system programming, the function could be implemented something like this:

```
#include <system.h>
#include <rpc/rpc.h>

#define DELIMITER '\t'

void Vote( char *candidate )
{  unsigned int thisUID;
   char *serverName = "vote.webster.edu";
   Data data;

   thisUID = system( "getUID" );

         /* however you do this */

   if( !( clientHandle = clientCreate( serverName,
         VOTESYSTEM, VERSION, "tcp" ) )
      {  clientPcreateerror( serverName );
         exit( 1 );
      }
   data.length = sprintf( data.buffer, "%s%c%u",
      candidate, DELIMITER, thisUID );
   rpcCall( clientHandle, "write", &data );
   clientDestroy( clientHandle );
}
```

6.1 **Discuss each of the tasks of encapsulation, concurrent processing, protection, name resolution, communication of parameters and results, and scheduling in the case of the UNIX file service (or that of another kernal that is familiar to you).**

The Sun NFS 4.0 file system meets, to varying degrees, each of the requirements of a resource manager in a distributed file system: encapsulation (consistent interface and hidden implementation details), concurrent processing (ability to handle multiple clients in a transparent fashion), and protection (access security). Furthermore, the public-domain NFS interface supports each of the major functions required in an invocation: name resolution (locating a server), communcation (passing requests and results), and scheduling (sequencing concurrent client requests for processing).

The NFS services are tightly encapsulated and hidden from the user. In fact, NFS is a model poster child for interface transparency, since the calling syntax, once mounted, is identical to the access method for local files. Processing concurency, however, is not quite as advanced: structural design choices severely limit scalability to less than 100 concurrent clients, and add little to Unix's built-in locking semantics. (Version 4.0 now provides advisory record locking, but still has much room for improvement.) However, the system is reasonably well protected, utilizing authenticated logins and including a new (v4) optional DES encryption method to enhance the default access control lists. Each of these traits has proven essential when rolling out

new commercial and experimental distributed systems.

Although encapsulation contributes significantly to a system's modularity and security, it leaves the users dependent on the suite of included interface calls. To be useful, these calls must include methods for name resolution, communication, and scheduling. The NFS system provides a richly endowed name resolution system which is managed by its *mount service*. Filesystems can be resolved and bound at boot time, after user login, or delayed until a given file is called upon and then Automounted completely in the background. This provides a high degree of location and migration transparency, and load-balancing properties built into Automounter add to the system's concurrency transparency.

Communication in NFS is essentially undifferentiated from standard file-level communication from the user's and programmer's perspective, although a lot more message passing goes on "under the hood." A typical request from a client process is directed to the Unix kernel, where it is intercepted by a "virtual file system" (VFS) layer responsible for distinguishing between local and remote file calls. The VFS layer passes the request to the NFS client module, which goes out over the network to a server's NFS server module, back through the server's respective VFS layer, and ultimately ending up at the Unix file system on the server computer. The request is processed and the response (positive, negative, or indifferent) is channeled back to the originating process along the same route.

Closely tied to communication is scheduling, which impacts how queries are queued and processed for response. A very important distinction between Unix's native file system and the NFS distributed extension is that NFS goes to great pains to be *stateless*. That means that most operations are idempotent and can be repeated one or more times without changing the result; this is an important characteristic in a networked environment when some messages end up being sent or received more than once.

At the Orlando Sentinel, we use NFS modules on key Unix and Netware servers to provide a common shared file address space to our Windows and Macintosh content producers. The transparent file access is a major benefit which significantly lowers the learning curve of new employees in our rapidly expanding online service divisions, and reduces custom programming modifications to various Perl, TCL, and Java scripts which drive key content-repurposing applications. Happily, we have not yet encountered the concurrency limits in NFS; so far our departments have not pressed the system with too many users or requests, but it is foreseeable that such thresholds may some day be of concern. Finally, the protection procedures provided under NFS aren't of extreme importance to our users, since our multiple firewall layers circumvent outside fiddling and basic access control prevents mistaken file modifications in-house.

6.2 Explain what is security policy and what are the corresponding mechanisms in the case of a multi-user operating system such as UNIX.

Security policy is a set of general rules describing who is allowed what forms of access to which data. These rules are defined to restrict unauthorized access to sensitive resources while permitting legitimate principals (users and groups) to utilize needed resources. Security policies are implemented by security mechanisms, which are the actual methods employed in code, hardware, and protocols to enact the desired policies.

There are a great many threats which comprehensive security policies must guard against. These include leakage (illegal read access), tampering (illegal write access), resource stealing (illegal resource utilization), and vandalism (destructive or malicious mischief). Each of these access violations requires some means of sneaking into a system with feigned authorization. Typical hacker methods include eavesdropping (packet snatching), masquerading (pretending to be another principal), message tampering (altering message contents while in-transit), and replaying (repeating encrypted transmissions, particularly destructive in non-idempotent operations). Prior to launching such an attack, there are several ways to infiltrate a system available to those with nefarious intent, including virus programs, worms, and trojan horses. Another kind of attack, known as a "quality-of-service" attack, neither accesses nor modifies system data, but simply aims to make system resources unavailable to principals, using such tools as the infamous "ping of death."

A system can be tested for weaknesses to many of these attacks with the much-maligned SATAN program, which allows a system administrator to easily search for common security holes needing to be patched. Another increasingly popular way to test for security holes is to actively solicit attacks, with some sort of incentive to successful (but non-destructive) attacks. A "Crack-the-Mac" contest sponsored recently in Scandanavia attracted a fair amount of attention when hundreds of thousands of attacks, spread over several months, failed to tamper with a Macintosh-hosted web server, even though a $15,000 reward was offered. Following the "success" of that contest, many more public security tests have been promoted by other commercial server vendors. Although these kinds of "spectacle" security demonstrations are popular with consumers, most system administrators demand a more thorough and scientific demonstration of a system's security, such as rigorous mathematical proofs.

In order to successfully blockade a system against attacks, capable security mechanisms must be implemented. Standard Unix installations include a number of mature and effective blocks against casual snooping. One of the most important Unix security mechanisms is the access control list, which prescribes access to files using a user/group/universe concept. Access to files is thus filtered using an encrypted password file to authenticate users and "capabilities" and "permissions" stored with each file. Another security mechanism popular with Unix systems is the Kerberos authentication protocol, which uses short-lived tickets to denote priviledged access between a client and a particular server.

6.3 Explain the program linkage requirements that must be met if a server is to be dynamically loaded into the kernel's address space, and how these differ from the case of executing a server at user-level.

The Chorus distributed operating system provides an interesting optimization which allows servers to be dynamically loaded into either typical user-mode address space or the kernel's protected address space. Keeping a server inside the kernel allows the system to operate with fewer mode-switches, but at the cost of somewhat lessened system security. In order to transparently take advantage of this capability, interprocess communication is generalized to the point where calling (client) processes use the same message passing interface whether the receiving server is inside or outside the kernel; in fact, clients typically have no way of knowing whether a particular server is in the kernel or not. Likewise, servers inside the kernel may act as clients of other servers also inside the kernel, neither being aware that the other is just a short memory-hop away.

One foreseeable problem with this arrangement, which I did not find addressed in the text, is how the system should handle dynamically linked runtime libraries which are shared between kernel-level processes and user-level processes. In general, such common libraries are loaded into memory only once and mapped to each executing process which relies on the code. This greatly improves memory conservation by eliminating duplicate code blocks. However, if a single library were being shared by a kernel-process and a user-process, then the library would need to be loaded in either the kernel space or user (unprivileged) space. In the first instance, user-level processes accessing library routines would have implicit access to modules running in supervisor mode, which is a security violation of the first order. In the second case, kernal code would be continually calling user-level libraries and incurring overhead costs with each context swap. The most likely solution, since the provision of kernel-space servers already suggests a willingness to put system response speed ahead of other concerns, is to allow any given library to be loaded both in the kernel's space and in user-space and automatically direct program calls to the appropriate module.

6.4 How could an interrupt be communicated to a user-level server?

System interrupts occur and are reflected in kernel (protected) memory space and are thus generally invisible to user-level processes. When a user process (typically a server or monitor process) needs to be able to view or respond to these events, portions of system memory can be mapped to the user memory space. Special system calls are necessary to implement such mapping, and they obviously require the kernel's cooperation.

7.1 The file service model treats the management of file directories as a separate service. What information is stored in directories? What are the advantages and drawbacks of the separation?

The model file service described in the textbook separates directory management from the file service. Each service has its own distinct interface, and different file metadata is maintained by each module. By dividing directory and file management tasks, several advantages are created over a unified file/directory service. There are also a few disadvantages to the division of services, but on the whole it provides a powerful and flexible system of managing file information.

The directory service is responsible for storing as much of the file metadata as possible without severely impacting efficiency. In particular, the directory

keeps track of the textual filenames perferred by humans, and maps those strings to the integer UFID identifiers preferred by digital computers. Directories also store useful information about a file, such as the access control list defining who is allowed to read, write, and execute each file; file types, for operating systems which care about such things; and file ownership for tracking and auditing. Finally, directories store information about other directories and thus make possible the deeply nested hierarchic structures to which we have become accustomed.

There are several advantages to this schema. First of all, the separation of services allows multiple directory systems to be mapped to a single file service. In the heterogenous network environment common to modern academic and corporate installations, this allows a single file system to transparently service clients from multiple operating systems and greatly facilitate cross-platform collaboration. Novell Netware servers provide this functionality by allowing DOS, Macintosh, and NFS name spaces to be loaded for each mounted volume. Separation of services also contributes greatly to a system's openness, by allowing one service to be upgraded or replaced without affecting other system functions (as long as calling interfaces are left unaffected). Another advantage of special interest in distributed systems is that directory systems are not necessarily tied to the same physical computers or networks as the file service, allowing systems to be distributed and apportioned as necessary for individual applications.

There are a few drawbacks to such a system, but for the most part they are inconclusive and easily sidestepped. For instance, each added software layer brings with it a degree of added overhead and latency, potentially degredating total system performance. Another problem related to separation is that failure of one service—of the directory service, for example—makes access to the remaining file service problematic at best.

7.2 Why are the file attributes stored with files and not in directories?

(Hint: several directory entries can refer to the same file.)

There are at least four good reasons to keep key file attributes local to the files rather than in a separate directory service. First, multiple directory systems can point to a single file, and it would not be proper to base file access on the calling directory system rather than file ownership or the defined access control list. Second, even within a single directory service, multiple entries can point to a single UFID in the case of aliases, symlinks, shortcuts, etc. Third, it would be inefficient to have to refer back to the directory service every single time a client tried to modify a file; it is much quicker and easier to be able to authenticate such calls within a single server. Fourth, even separate file systems typically allow direct access calls which bypass the directory system, and it would not do to allow such precocious clients to gain unauthenticated access to an entire file system.

9.1 Describe the names (including identifiers) and attributes used in a distributed file service such as NFS (see Chapter 8).

A distributed file service must maintain a variety of data tables listing the many names, identifiers, and attributes associated with each file. These labels are not only for internal use in resolving client requests but also to provide a flexible and human-useful calling interface to other programs. Listed

herewith are some of the many descriptors which a service such as NFS must maintain:

Directory Name, which holds the textual name of the enclosing directory;
File Name, which holds the textual name of a particular file entry;
UFID, which holds the computer-readable bitmap uniquely identifying a single file entity;
File Attributes, many held in bit-flag arrays, connoting such information as the access control list, file ownership, the file length in bytes, and time/datestamps; and
Filesystem Status Variables, which report data like the current volume block size and the amount of unallocated storage space.

Some would also consider elements of the calling interface to be exported identifiers, including function names like *readdir(), symlink(),* and *setattr()*. In a similar vein, file services must track both their own receiving port identifiers and the reply-to ports of their clients, plus the names and addresses of other services on which it depends, such as name and synchronization servers.

9.7 How does caching help a name service's availability?

Although no subsystem in a distributed environment is exactly frivolous, few are as essential to moment-to-moment operations as the name services which allow different units to contact one another across a network. For that reason, name services are generally designed with extremely high levels of availability. One of the main methods system designers employ to guarantee high uptime and accessiblity is extensive use of caching.

By caching recently resolved name/address pairs, systems can communicate with each other without bothering the name servers before every transaction. This is a very good thing, for otherwise the name servers would be totally swamped with redundant traffic asking the same questions over and over, forgetting the answer after every reply.

Part of caching involves the use of secondary servers which act as shared caches for whole networks of computers. While individual hosts may be expected to efficiently cache dozens or hundreds of recent name/address pairs, secondary name servers can effectively maintain many thousands of records for rapid resolution. In the occasional instance when neither a host nor a secondary server can resolve a name, a primary server can handle the request from its mammoth distributed tables. In fact, by caching (*namePrefix, serverName*) pairs, an unknown name may be resolved even in the absence or failure of a root server by directly contacting the secondary servers responsible for that branch of the name tree.

As an excellent example of how this works, late last week (Friday, July 17, 1997), the following message was broadcast to clients of the BBN network:

> Last night, the primary interNIC Root Server, the machines that all Internet backbone providers use to identify domains on other

> networks, was corrupted by a possible power problem at the location where it is housed. This has caused most root servers on the Internet to have incorrect data as well. The interNIC has corrected their primary root server, but are awaiting secondary servers across the various networks to upload the corrected information.
>
> Until this has occurred, all customers on all Internet providers will have difficulty reaching addresses in the .com and .net domains. We do not currently have a definitive time frame for when this will occur. We will continue to update this message regularly and as we receive more information.
>
> *(For a complete history of this ticket, do "finger ticket-128295@ tickets.bbnplanet.com")*

Thanks to local and secondary server caching of previously resolved addresses, many users were able to carry on normal internet activity without even noticing the outage or corrupted tables. Only in the case of “new” addresses—or addresses so old as to have been rotated out of the cache—did users have difficulty reaching their target destination.