Principles of Computer System Design
An Introduction

Answers to Exercises and Solutions for Problem Sets

Jerome H. Saltzer

M. Frans Kaashoek

with contributions by many others

Version 5.0

TABLE OF CONTENTS

Answers to Exercises

1. Answers to the Exercises of Chapter 1

Ex. 1.1 **A** is false. Modularity has no direct impact on incommensurate scaling. Its impact on complexity comes from reducing the number of interconnections. **B** is true. By eliminating unnecessary interconnections, modularity helps control unexpected propagation of effects.

Ex. 1.2 **A** is false. Hierarchy has no direct impact on module size. Instead, it provides structure for the interconnections among modules. Its primary impact on complexity comes from constructing stable subassemblies and thus reducing the number of modules that interact at each assembly point, so **B**, **C**, and **D** all are correct.

Ex. 1.3 Almost certainly false. Although there would be clusters of interconnections, the existence of mutual friends would provide many non-hierarchical cross-connections.

Ex. 1.4

A. Yes. Digital hardware, which forms the basis for today's computing, has for several decades improved dramatically with time.

B. No. A complex system is hard to describe easily, and usually no single person understands it all.

C. Yes. Interacting requirements and features are usually observed in complex systems.

D. No. Complex systems often exhibit emergent properties, but more often than not they come as an unpleasant surprise and it is more typical for them to be the cause of worse performance than expected.

Ex. 1.5a *16 modules × 100 lines per module = 1600 lines.*

Ex. 1.5b Ambiguity: If A calls B and B calls A, we can count that as either one or two interconnections. Assume two interconnections; the alternative produces numbers half as large. Each of the 16 modules contains at least 15 intermodule calls (at least one to each of the other 15 modules), so the number of calls is at least 16 x 15 = 240.

Ex. 1.5c

16 submodules × 100 lines per submodule = 1600 lines
+ 4 main modules × 100 lines per main module = 400 lines
gives a total of 2000 lines

Ex. 1.5d Module-to-module: Each of the 4 main modules contains at least 3 intermodule calls (at least 1 to each of the other 3 main modules), so the number of inter-main-module calls is at least 4 × 3 = 12.

Submodule-to-submodule: In any main module, each of the 4 submodules contains at least 3 intersubmodule calls (at least 1 to each of the other 3 submodules), so the number of intersubmodule calls is at least 4 main modules × 4 submodules per main module × 3 calls per submodule = 48.

Main module-to-submodule: There must be at least 4, from each main module to at least one of its submodules; but we were told to count module-to-module and submodule-to-submodule interconnections, not module-to-submodule interconnections, so we omit these.

The total is then at least 12 + 48 = 60 interconnections.

Ex. 1.5e Ben has reduced the minimum number of interconnections by a factor of four, at the cost of increasing the code size by 25%.

Cons: The 25% larger code will probably have 25% more bugs. If the application isn't suited to the new hierarchical organization, flows of information and control may be more complex than they used to be (and the actual number of interconnections may be proportionally more above the minimum than before). Finally, when code grows in size, the added instructions require processor attention, so if processor speed is a performance limitation, performance might go down.

Pros: On the other hand, the drastic reduction in the minimum number of interconnections probably reduces complexity, and if subroutine calls are expensive the smaller number of calls might increase performance. Because of the complexity reduction, maintenance should be simpler, since a change in a submodule is likely to affect only the three other submodules that call it. Similarly, an error in a submodule is likely to produce disruptive results only for the three submodules that call it, so tracking down bugs may be easier.

We need more information to know for sure. For example, Ben's changes to the inter-module interfaces may have eliminated certain features or made them awkward to use. But at first glance, the benefits of the "pros" seem worth the cost of the "cons", so it would at least be worth looking more deeply at the details.

2. Answers to the Exercises of Chapter 2

Ex. 2.1a There are two name spaces involved here: (1) a name space of telephone numbers and (2) a name space of telephone lines, the wires leading to particular telephones. The telephone company binds telephone numbers to telephone lines, and forwarding modifies those bindings. *Follow me* rebinds Ann's telephone number to Bob's telephone line. If Bob hasn't forwarded his phone yet, Bob's telephone number and Ann's telephone number are now *synonyms*. When Bob later forwards his telephone number to Mary, he rebinds his telephone number to Mary's telephone line, but Ann's telephone number remains bound to Bob's telephone line, and continues to ring there.

Delegation rebinds Ann's telephone number to Bob's telephone number (rather than Bob's telephone line), thus creating an *indirect name*. When Bob invokes forwarding, he rebinds his telephone number to Mary's telephone number. Someone calling Ann's telephone number now goes through two layers of indirection and rings Mary's telephone.

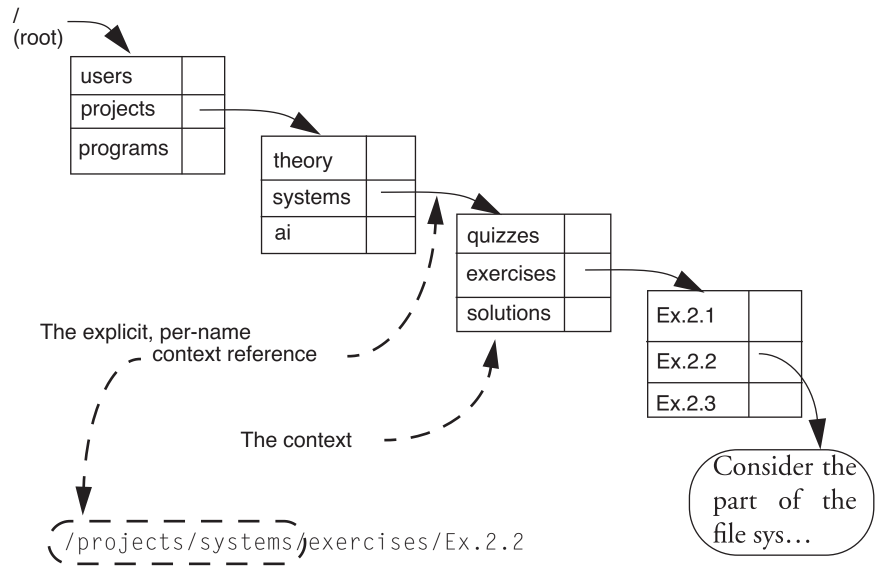
Ex. 2.1b When Ann requests forwarding, the only name she has available to describe where she wants the forwarding to go is Bob's telephone number. If Bob forwards his phone to Mary *before* Ann forwards her phone to him, under *follow me* Ann's telephone number will end up bound to Mary's telephone line, and all calls to Ann will now ring Mary. Ann will be surprised not to get any calls while baby sitting at Bob's house. Under *delegation* there are no surprises. Calls to Ann go to Mary's telephone line while Bob is visiting Mary, and go to Bob's telephone line when he turns off forwarding.

Ex. 2.1c Telephone lines connect the telephones in a neighborhood to a nearby telephone switch. The scope of a telephone number is an entire area code, but the scope of a telephone line is a single switch. *Delegation* binds one telephone number to another telephone number, and a recursive name resolver can easily perform the indirection. *Follow me* rebinds a telephone number to another telephone line. But if the target telephone line is attached to a different switch, this switch doesn't have a name for that line, so it has no way to express the new binding. Implementing *follow me* would require that the telephone system implement a universal name space for telephone lines.

Ex. 2.2a The context reference identifies the context that one should use to resolve the name `exercises`. There are two things that identify that context to the name interpreter, depending on the level within the interpreter:

- the path name `/projects/systems/`
- the inode number (shown in the figure as an arrow) that is associated with the name `systems` in the superior directory

The context is the file system directory identified by that context reference.


/
(root)
users
projects
programs
theory
systems
ai
quizzes
exercises
solutions
Ex.2.1
Ex.2.2
Ex.2.3
The explicit, per-name context reference
The context
/projects/systems/exercises/Ex.2.2
Consider the part of the file sys…

Ex. 2.2b Because the context reference came as part of the path name, it is both *explicit* and *per-name*.

Ex. 2.3a The trouble with synonyms is that if *name1* and *name2* refer to the same object, {*name1, object*} is in the cache, and someone refers to *name2*, we are likely to not realize that we already have *object* in the cache, and we will look it up again and add it to the cache as the pair {*name2, object*}. We have wasted time, we have used up cache space, and we have set ourselves up for a disaster if the system permits modifying objects by name: a modification to the copy of *object* associated with *name1* should be propagated to the copy of *object* associated with *name2*, but we don't have enough information to realize that.

Ex. 2.3b If each object has a unique ID one could change the cache to hold {name, object, UID} triples. Then, whenever someone modifies an object by name, search the cache for any other entries that have the same value for UID in their third field and either modify or invalidate those copies.

Ex. 2.4 Louis has mistakenly assumed that repeated searches for the same name always resolve to the same object. There are several situations in which this assumption may be false; here are a few examples:

- When one of the items in the search path is the working directory, and the object is found in the working directory. If the user changes the working directory, and tries to use an object that happens to have the same name in the new working directory, the ROT will have priority in resolving the name and the user will get the object from the old working directory, which is probably not what he or she expects.

- When the user changes the search path with the intent of changing the way names get resolved, the ROT will interfere with that intent; all names resolved previously will continue to be resolved the way they were before the change.
- A user refers to an object through a link, causing the path name of the object to be placed in the ROT with the name of the link, and then changes the link to point to something else. The next time the user refers to the link the ROT will lead to the thing the link used to refer to rather than the thing it currently refers to.

In each of those cases, the user will notice a difference. Louis might try to patch around these problems by, for example, resetting the ROT whenever the current thread changes the search path, the user changes the working directory, or anyone on the system modifies a link. But these fixes might eliminate much of the speedup that Louis hoped to obtain, and there are probably other cases he hasn't discovered yet.

Ex. 2.5 A Web browser interprets the contents of a Web page, which is coded in the HTML language. When the browser downloads a new web page, it sequentially examines one HTML element at a time (the instruction reference). Its repertoire is the list of HTML elements that it is prepared to interpret. Its environment reference is the URL of the retrieved page; it obtains both instructions and data from that page and if it encounters a name of another page it interprets that name relative the the URL of the retrieved page. This name interpretation is described in detail in Section 3.2.2.

3. Answers to the Exercises of Chapter 3

Ex. 3.1a The UNIX system stores system metadata about a file, such as its physical location, owner, file permissions, etc., in the inode for that file.

Ex. 3.1b In the UNIX system, although the user may be able to control the values of some of the system metadata such as the owner, the only significant user-provided metadata for a file is the name or names of the file. The file system stores those names in directory entries.

Ex. 3.1c The UNIX system stores metadata about a file system, such as the size of the file system, in the super block.

Ex. 3.2 The UNIX system always allows the name "`..`" any place that a file name is expected, so answer **A** is incorrect. The premise of answer **B**, that Alice's home directory now has two parents, is wrong; a symbolic link does not create a parent. Answer **C** is the correct one. The reason is that

```
/home/alice
```

is a symbolic link pointing to

```
/disk1/alice
```

so the command

```
cd /home/alice
```

actually sets Bob's working directory to `/disk1/alice`. When Bob then types the command

```
cd ../bob
```

The system interprets the "`..`" as meaning the parent of `/disk1/alice`, so it looks for

```
/disk1/bob
```

which does not exist and Bob receives an error message: "No such file or directory". Answer **D** incorrectly confuses symbolic links with hard links. One of the purposes of symbolic links is to allow naming of files and directories on other disks. It is hard links to other disks that the UNIX system disallows.

Ex. 3.3a Bob will end up in the directory `/home/bob`. Using syntactic path names, the system will implement "`..`" by removing `Alice` from `/home/Alice`, leaving `/home/`, and then the `cd` command will change the working directory to `bob` in the directory `/home/`.

Ex. 3.3b

A. No. An undirected graph doesn't restrict the potential naming networks. In particular, the naming networks in the previous examples are possible, and we have seen that semantic and syntactic path names resolve differently in those examples.

B. Yes. The previous examples induce naming networks that are not trees, and thus are disallowed by the question's assumption. If the naming network is a tree, then each node can have only one parent, so there are no opportunities for semantic and syntactic path names to resolve differently.

C. Yes. If there are no synonyms, then the induced naming networks have nodes with only one parent, and thus the same reasoning as in answer **B** applies.

D. Yes. This question is the same as question **C**, and the answer for **C** applies.

Ex. 3.3c

A. Yes. Resolving a syntactic path name requires reading every directory in the path name, which may require several disk accesses. Resolving a semantic path name requires reading only one directory, the one in which ".." must be resolved. Thus, syntactic path names may require more disk accesses. If the file system has a naming cache, however, many of the disk reads may hit in the cache, in which case the number of disk accesses could be the same.

B. Yes. See the answer to question **A**.

C. No. Hard links apply only to files in the UNIX system, and there is no difference between semantic and syntactic path names in resolving a file name.

D. No. Bob's proposal doesn't affect resolution of "**.**", so there is no difference between semantic and syntactic path names that involve "**.**"

Ex. 3.4

A. TRUE. The reference count is used to keep track of the number of (hard) links and determine when it is safe to reclaim the inode and file blocks following an unlink operation.

B. FALSE. The reference count of an inode counts the number of hard links that refer to that inode. There are always at least two hard links that refer to a directory: one from its parent, and one from the entry "**.**" within the directory, so the reference count of a directory is at least two. Beyond that, if the directory contains any subdirectories, those subdirectories have an entry "**. .**" that is a hard link to its parent, increasing its reference count further.

C. FALSE. The directory itself is a file, and its blocks contain the names of the files in the directory together with their associated inode numbers.

D. FALSE. The inode of a symbolic link contains the path name of the link's target rather than its inode number.

E. FALSE. See answer **C**, above.

F. FALSE. The inode number is an offset in the inode list. That list is stored on the disk, but the inode number is not directly usable as a disk address.

G. \FALSE. Inodes are located in an inode table, in a part of the disk distinct from file contents.

Ex. 3.5 There are several possibilities. Here are three: (1) Choose as the name of the "no picture available" image an identifier that has very little chance of being a student name, e.g. "1uj8902!lkp.jpg". (2) For the described system to work at all, the name space of student images must assign every student image a unique name, so there must be an unmentioned scheme to assure that uniqueness. Adjust that scheme to know that the name of the "no picture available" image has been allocated. (3) Store the "no picture available" image in some name space other than the name space of student images and change the system that creates the visual class list to understand that when resolving the name of a student's image it should handle a "not found" response by retrieving the "no picture available" image from the other name space.

4. Answers to the Exercises of Chapter 4

Ex. 4.1 False. An incorrect answer from a service can affect the client. Enforced modularity restricts interactions between service and client to just their defined interfaces, but it is still possible for a wrong value to be passed across a defined interface.

Ex. 4.2a *Modularity.* Client/service organization separates the client and service address spaces (they may even run on separate machines). If either client or service crashes or otherwise malfunctions, the other will be affected in a limited way, if at all.

Ex. 4.2b *Abstraction.* RPC hides the mechanical details of argument marshalling and network communication behind an interface that resembles a local procedure call with some new error signals.

Ex. 4.3 The essential difference between remote and local procedure call is that with remote procedure call the caller and callee can fail independently. From the caller's point of view, this means there is a new potential outcome that the runtime system can report, so answer **A** can't be right. Particular RPC implementations sometimes exhibit some of the other suggested differences, but only answer **E** is generic to all RPC implementations.

Ex. 4.4

A. True.
B. False. Usually, an X client uses *asynchronous* communication with the X server for better interactivity.
C. False. The X server is the program that manages your display, keyboard, and mouse.

Ex. 4.5

A. True only when M's cache misses *and* the prefix of the name has nothing in common with the domain that M serves.
B. True only if the time-to-live of the DNS record cached by *M* has not expired. If the binding found by M in its cache for www.`cslab.scholarly.edu` has expired, then M must contact at least one other name server to resolve the domain name.
C. True only if M has no valid information in its cache for `www.cslab.scholarly.edu`.
D. Not usually true, because the `scholarly.edu` name service will normally refer *M* to the `cslab.scholarly.edu` name service, rather than answer the query itself. (Though it is possible to configure a name service to directly contain the data for one or more of its subdomains.)
E. Normally true, though it is possible that the name is a synonym that requires `cslab.scholarly.edu` to ask yet another name server for help.

Ex. 4.6

A. False. Although M had a query for that same IP address, the time-to-live on the cached name record may have expired, in which case M may have contact another name server and ask for a new name record.

B. True. The name server for `cslab.scholarly.edu` should be authoritative for the name `www.cslab.scholarly.edu`, so it doesn't need to ask any other name server.

Ex. 4.7

A. True. Because an NFS server is stateless, it must assure that the write to the disk has completed before responding to the client.

B. True. If a client does not receive a response to a request, for example because its request was lost, it retransmits it.

C. False. NFS servers don't have to run any recovery procedures upon being restarted after a crash. The NFS protocol is stateless, so there is no state to coordinate with its client.

Ex. 4.8

A. True. When the server will get the resent message, it will look in its reply cache, discover that it has already processed the message, and return the same response to the client. The question indicates that the server records **all** previously executed operations. In practice, no server can ever afford to guarantee to record all operations in its memory reply cache because that would force the server to keep a potentially unbounded amount of state around forever. One way around this problem is to keep a fixed amount of space for the cache and discard results of old operations when space gets tight.

B. False. If the server processes the resent message with an empty reply cache, it will always try to execute the request in the message. In this case it will always return "file not found" because it removed the file when it processed the first request. The file is therefore no longer present.

C. True. A stateless server does not maintain a reply cache, so it will try to execute whatever request comes in. In particular, it will try to remove the file, find that it is not present, and return "file not found".

D. False. As the previous answers indicate, REMOVE is not an idempotent operation in NFS.

5. Answers to the Exercises of Chapter 5

Ex. 5.1a There are two good answers.

Abstraction. Virtual memory hides the management of disk, page tables, etc. from the user, giving the appearance of a single, seamless address space (which may be larger than the size of physical memory).

Modularity. Virtual memory separates activities from one another by giving them different and protected address spaces, thereby minimizing the chance that one activity can accidentally or intentionally harm another.

Ex. 5.1b *Modularity.* A microkernel partitions an operating system's functions (such as the file system) into separate user-level modules, thereby minimizing and making well-defined the interactions among those functions.

Ex. 5.2 Everything in the universe is an example of an object, so we could associate item D with all five of the mechanisms in the list on the left. Ignoring that trivial interpretation, we have:

1–F. A page map is a context for the page-number-to-page-location entries that it contains.

2–E. A virtual address names a location in virtual address space.

3–D or E, depending on one's point of view. From the point of view of the virtual memory system, a physical address is the object to which a virtual address is bound. From the point of view of the underlying physical memory system a physical address is the name of the place that holds the data.

4–G. Although a TLB entry is located in a context that is used in a kind of search, it is not itself a list of contexts to search, so it does not qualify as a search path. A TLB entry is a single name-to-object mapping, so letter G seems to be the best choice.

5–C. The PMAR identifies which page map, and thus which context, to use, so it is a context references.

It is hard to interpret any of the mechanisms in the list on the left as a naming network, so item B on the right is not used.

Ex. 5.3a Data types **A** and **B** have meanings that are independent of the current addressing context, so they are no problem. Troubles can arise when the items communicated are or include addresses, so we need to investigate data types **C**, **D**, and **E** more carefully. Data type **C** is a reference to something else inside the communication region; since the communication region is in the same absolute position of every virtual address space, a virtual address within it will resolve the same way for every user. Data type **D** represents a problem, because it refers to a location in the private virtual memory of the user who put it in the communication region. If someone else tries to resolve it, they will undoubtedly end up with some different object in their own address space. Whether or not a procedure of data type **E** will run correctly depends on whether all of the addresses it uses are within the communication region. If they are, it should run without incident. But if any address that it uses (for example, a variable stored on the running thread's stack) is outside the communication region, that is an example of data type **D**, with the same problem.

Ex. 5.3b Ben has muddled things up for data types **C** and **E**. Data type **C** no longer works, because the value of the virtual memory address will depend on whose virtual memory it came from. It would be possible to invent a protocol in which the user who places a virtual address in the communication region also places next to it the block number of the communication region in his own address space. Then other users could deduce the proper offset to add to the virtual address in their own address space, a process known as *relocation*. Data type **E** is even more problematic, because all addresses it uses, even to refer to itself, will have to be relocated.

Ex. 5.4 False. With a microkernel, there will be more context switches per unit time than with a monolithic kernel. But context switches cause TLB invalidations and thus would lead to a lower hit rate. In addition, one might expect a microkernel plus the non-kernel components of the operating system to use more memory pages than a monolithic kernel, causing an even greater load on the TLB.

Ex. 5.5

A. True. Using SEND/RECEIVE communication links between threads will eliminate any races caused by shared variables whose access is not properly coordinated. (However, using communication links may slow down Louis's program).

B. True. The one-writer rule is a good strategy for helping to avoid race conditions.

C. True. Ensuring that a shared variable is always protected by a lock can avoid certain race conditions.

D. False. Always acquiring locks in the same order will prevent deadlocks. While deadlocks are arguably a form of race condition, we are told that Louis's program always completes, so the program must not have any deadlocks to prevent.

Ex. 5.6

A. True. Preemptive scheduling can prevent applications that are in an infinite loop from stalling other applications.

B. True. The kernel is responsible for mediating between programs by managing shared memory, inter-program messages, etc.

C. Normally false. Virtual memory is nearly always supported by hardware features that resolve most memory references without involving the operating system. Only when a page fault exception is encountered is the kernel invoked.

D. False. In order to enforce modularity between address spaces, the kernel must set the PMAR register before switching to user mode and beginning execution of the target application.

Ex. 5.7a No, because there is no possibility that two or more threads each hold a lock at the same time that another thread needs. If GLORP acquires either or both locks before some other thread, say thread C, it will run to completion, because, by assumption, thread C can only lock at most one lock at a time (and thus cannot have a lock needed by GLORP). If thread C acquires the lock first, GLORP may be delayed, but C will finish and eventually release the lock (we know that C proceeds because it could only be waiting for one lock, which in this case we assumed it has already acquired). Either way, some thread is able to run to the point of releasing its lock(s). Threads A and B cannot conflict, since only one is in GLORP at once.

Ex. 5.7b Yes. Deadlock can occur if thread A acquires *lock_a* at the beginning of the procedure and thread B acquires *lock_b* in the middle. Each will hold a lock needed by the other thread to proceed, so neither can make forward progress. Since without forward progress neither thread can release the lock needed by the other thread, the threads are deadlocked.

Ex. 5.8a No. All three threads acquire their locks in the same (in this case, alphabetic) order. That ordering protocol prevents deadlock on the first pass through each loop. Each thread releases all of its locks before acquiring any more. That release assures that alphabetic re-acquisition on the next pass will also prevent deadlock.

Ex. 5.8b No. The previous argument did not depend on the order that the threads release the locks, only that they all be released before the next re-acquisition.

6. Answers to the Exercises of Chapter 6

Ex. 6.1a The processor clock rate of 100 megahertz corresponds to a cycle time of 10 nanoseconds. Setting this cycle time equal to the average latency of the two-level memory, we can calculate the needed hit rate *h* as follows:

$$\begin{aligned} 10ns &= h \times 1ns + (1-h) \times 101ns \\ h &= 0.91 \end{aligned}$$

Ex. 6.1b If we installed a large enough cache that $h = 1.0$, we would have an average latency of 1 nanosecond, which would allow use of a 1 gigahertz processor.

Ex. 6.2a Each object occupies 100 pages. Let us assume that if the program makes 1000 randomly distributed accesses to such an object, every one of the 100 pages is touched. If so, every time the program begins to use a new object, all 100 pages of that object will have to be brought into the fast memory if they are not already there. The fast memory has a capacity of 1000 pages, so it can hold 10 objects. Every time the program begins to page in a new object, we can expect the new object's 100 pages to replace the 100 pages of whichever object in the fast memory has been used least recently, so we can expect that, at the point when the program finishes touching an object, the fast memory contains exactly the 10 most recently used objects. Since the next object the program will use is chosen randomly, there are 10 chances in 100 that the new object will already be in the fast memory.

Thus with probability 0.1 the 1000 references to the new object will all be hits, and with probability 0.9 the 1000 references will result in 100 misses (one for each page that must be fetched). Thus, the average number of misses per 1000 references is 90, the miss ratio is 0.09 and the hit ratio is 0.91.

Ex. 6.2b The number of objects that fit in the fast memory is unchanged, but the number of misses required to page in an entire object has been cut from 100 to 10. As a result, the hit ratio will go *up* to 0.991.

Ex. 6.3 Some likely candidates:

- On every kernel call it will be necessary for the kernel to save and restore twice as many registers, which will slow things down. Since the most popular operating system is organized as a micro-kernel, there will be a lot of context switches, so this effect will probably be the most significant.
- The bottleneck in the system may not be memory access rate, but somewhere else such as disk access, the network, or even the processor itself.
- Many of the targets of loads and stores are likely to be in the processor cache, so reducing the number of loads and stores won't save as much time as Metoo expected.

Ex. 6.4 Here are two likely sources for the degradation in performance Mike observes:

1. Many missing-page faults, because the total physical memory requirements of the multiple applications exceed the amount of physical memory available. Missing-page faults involve disk accesses, which take a long time (figure 10 milliseconds).

2. Context switching overhead, since to run all the applications the operating system must context switch among them. Context switching requires loading a new page table (the P97 has only one), saving and restoring registers (if the P97 has any), etc., etc. Context switching overhead is likely to be negligible when there's only one application.

So Mike can improve the P97 system's performance by addressing either of these problems, or both. There are many, many ways to do this, falling into 4 main categories.

1. Improve hardware:

- Buy more RAM (reduces number of missing-page faults)
- Buy a faster disk (reduces missing-page fault handling time)
- Partition the hardware page table and allocate one partition to each application (if applications have a small working set, this will improve performance)
- Add more hardware page tables to avoid reloading the page table on every context switch
- Add more processors
- Use a page map address register to switch between multiple page tables (avoid reloading the page table on each context switch)
- Reduce the number of registers (reduces the context switch time)
- Add a cache to the disk to hold recently used disk blocks

2. Improve the virtual memory system:

- Reduce the page size (helps to defragment memory so more of it can be used)
- Use a multiple-level page table (reduces the amount of physical memory used up by page tables if the working set of applications is small)
- Use a better page replacement policy (perhaps an application-specific one)
- Implement prepaging (may reduce the number of missing page faults)
- Add a TLB to improve address translation time

3. Improve the thread scheduler:

- Use preemptive scheduling (prevents threads from hogging the CPU)
- Modify the scheduler to allocate longer time slices to threads (makes context switches less frequent)

4. Other:

- Reduce the size of the kernel (freeing up physical memory for applications)

Ex. 6.5 The latency introduced by RPC comprises two network transit times, time in the client stub that marshals arguments, time in the network layers that add and remove headers, time in the service stub that unpacks the arguments, time spent creating threads, and time in context switches. So making the network faster will reduce only one, possibly unimportant, part of the total RPC latency. (The called procedure at the far end also doesn't get any faster when the network speeds up, but presumably Ben is smart enough to realize that time in the application program isn't part of the RPC-introduced latency.)

Ex. 6.6 Transferring large blocks to and from the disk amortizes the seek and rotational delays over a larger number of bytes, reducing the time required to read or write the same amount of information.

But if only a small amount of information in each page is actually used by the program, then large block transfers to and from disk will yield needlessly long transfer times, and bring into main memory information that is not needed.

Ex. 6.7a

Seek time:	6.5 milliseconds	
1/2 rotation time:	4.0 milliseconds	7500/60 = 125 rps, or 8 ms per revolution
Read time:	0.4 milliseconds	(8 x 512 bytes)/(10 megabytes per second)
Total time:	10.9 milliseconds	

Ex. 6.7b We have been told only the latency of the RAM; we don't know the number of bytes transferred per cycle. If the number of bytes transferred per cycle is four, then the transfer rate would be 4 bytes/25 nanoseconds = 160 megabytes per second.

Ex. 6.7c Since we are given the hit ratio, the size of the cache does not enter into the calculation. In RAM-to-RAM transfers of a whole sector, the latency of the first byte is a negligible component, so we can ignore it. The time to transfer a sector from RAM to RAM is 512 bytes/160 megabytes per second = 3.2 microseconds. And sometimes we have to transfer a sector from disk; since the sector is randomly chosen, a seek is usually required, but that transfer will take a little less time than when we transferred 8 sectors at once, 10.55 milliseconds. Assuming that the system first reads a missing disk sector into the cache, and then copies it into the user's read area,

$$T_{\text{exp}} = 100 \times [(1-h) \times (10.55) + 0.0032] \text{ milliseconds}$$

If the system somehow reads the disk sector in parallel into both the cache and the user's read area, the second term should be multiplied by h.

Ex. 6.7d In order to increase the size of the disk cache, Ben must have reduced the memory available for other things such as virtual memory, heap space, or application storage. These things are now working harder to utilize their smaller allocation. And since the cache is now much larger than the designer expected, its search algorithm may not have scaled up well and the time spent searching the cache may now be significant.

Ex. 6.7e This is fundamentally a question about incommensurate scaling.

- Don't visit the lab when Louis turns the fast-spinning disk on for the first time, because it will probably explode. The force required to keep the edges from flying away probably will exceed the tensile strength of the disk platter.
- Assuming Louis switches to titanium platters to avoid the first problem, the edge of the disk is moving at Mach 1.6, while the innermost track is moving at about Mach 0.5. The seek arm may have some trouble moving across the region of the platter where it crosses the sound barrier.
- The data will be passing under the read heads an order of magnitude faster than it used to, and the heads probably will require a complete redesign for the higher bandwidth.
- Since the change didn't do anything to improve the seek time, the overall performance may not improve nearly as much as Louis hopes.

Ex. 6.8a Only one block write is required. All of the information about the file is contained in the first block of the file.

Ex. 6.8b Initially the three structures are cleared. Then the disk is scanned once and each block is processed. If the block is free (i.e., *fid* is zero), it is inserted into FREE. Otherwise MAP is probed to see if an entry with the same (*fid, sn*) exists. If no entry exists, the block is added to MAP. If an entry exists, the block with the older timestamp is added to RECYCLE, and the block with the newer timestamp is inserted into MAP. If the block is the first block of a file and the directory *fid* is zero, the MAP is probed for the other blocks of the file and they are removed from MAP and placed in the RECYCLE list.

Ex. 6.9a Assuming that for purposes of style analysis sentences are independent, while one client thread is waiting for a response from the remote server a different client thread can be parsing a different sentence. It may also be possible for a thread to initiate a parallel RPC even though the response from an earlier RPC has not returned, in which case still more client threads can make progress on other sentences. The first mechanism overlaps parsing with RPC latency, whether caused by the network or the server. The second mechanism overlaps network latency with server activity.

Ex. 6.9b The second thread exploits some, but not necessarily all opportunities for overlapping. The third thread exploits some more. But each additional thread finds fewer opportunities for overlapping, and eventually client parsing, network latency, and server time are overlapped to the maximum extent possible. If still more threads are added they can't help, but they can begin interfering with one another, perhaps because of increased scheduling and thread switching overhead, mutual exclusion and coordination overhead, or increasing virtual memory traffic for the extra thread stacks.

Ex. 6.9c There are several possibilities:

- The client could cache the result of a request to the server, in the hope that a writer will tend to use the same style patterns over and over.

- The client could parse several sentences, then send several requests to the server at once by constructing a special client RPC stub that fires off several RPC request messages and then waits till all the responses have returned.
- Stop using RPC; change to a stream model of interaction between client and server that allows the client to work on the next sentence without waiting for the result of analyzing the previous sentence.
- Buy a faster processor for the client.

Ex. 6.10a (1 millisecond x 100 MIPS) / 2000 instructions = 50 threads.

Ex. 6.10b (1 millisecond x 100 MIPS) / (2000 + 500 instructions) = 40 threads.

Ex. 6.10c Within a Web browser, application threads will tend to be interdependent (perhaps waiting for one another because of competition for disk, network, or display), and the application I/O will tend to be bursty. "Average" run times are thus subject to wide variance, and the browsing process will afford many idle moments.

Ex. 6.10d Almost any typical system activities will qualify, e.g.,

- paging
- multiprocessing overhead, sub-optimal scheduling
- I/O bursts in audio visual subsystems

network delays (transit time, load on remote servers)

Ex. 6.11 **A** and **B**. LRU requires some computation at every reference to memory, to maintain things in order of last use, and it also requires space for counters, timestamps, or list pointers. Clock approximates LRU keeping only one extra bit of information per object, and reducing the computation needed at reference time to just forcing that bit ON. Answer **C** would imply that the right thing to approximate is MRU, not LRU.

Ex. 6.12a Because prepage-OPT maintains a total ordering of all pages in the multilevel memory system, it is a stack algorithm.

Ex. 6.12b Because it is a stack algorithm, we can develop a single table for prepage-OPT that shows the stack, and then by examining the stack quickly identify the page movements required for each possible memory size.

Table Sol.1: Stack contents and page movements for the prepage-OPT policy.

time		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	
reference string	**preload**	**0**	**1**	**2**	**3**	**0**	**1**	**4**	**0**	**1**	**2**	**3**	**4**	
stack contents after reference	0 1 2 3 4	1 2 3 0 4	2 3 0 1 4	3 0 1 4 2	0 1 4 2 3	1 4 0 2 3	4 0 1 2 3	0 1 2 3 4	1 2 3 4 -	2 3 4 - -	3 4 - - -	4 - - - -	- - - - -	number of page move- ments
size1 move-in	0	1	2	3	0	1	4	0	1	2	3	4	-	12
size 2 move-in	0,1	2	3	0	1	4	0	1	2	3	4	-	-	12
size 3 move-in	0,1,2	3	0	1	4	-	-	2	3	-	-	-	-	9
size 4 move-in	0,1,2,3	-	-	4	2	-	-	3	-	-	-	-	-	7
size 5 move-in	0,1,2,3,4	-	-	-	-	-	-	-	-	-	-	-	-	5

As in the tables in Section 6.2, the "move-in" rows tell the number of the page that must be moved in to the primary device to correspond with the stack contents after the reference. Column 0 identifies the pages required to preload the primary device.

Ex. 6.12c There are several possible measures of "better". Consider first page movements. Prepage-OPT has caused more page movements than demand-OPT, so Louis needs to come up with a name for this policy that doesn't include the letters "OPT". For example, with a primary memory of size 2, the move-in at time 1 of page 2 correctly anticipates the need for page 2 at time 3, but it unnecessarily forces page 0 out, so page 0 has to be brought in again at time 5. It would have been better to bring page 2 in only when the reference string demands it, because that movement would have kept page 0 in the primary device. The startling result is that prepaging can cause unnecessary page movements even when one has perfect knowledge of the future!

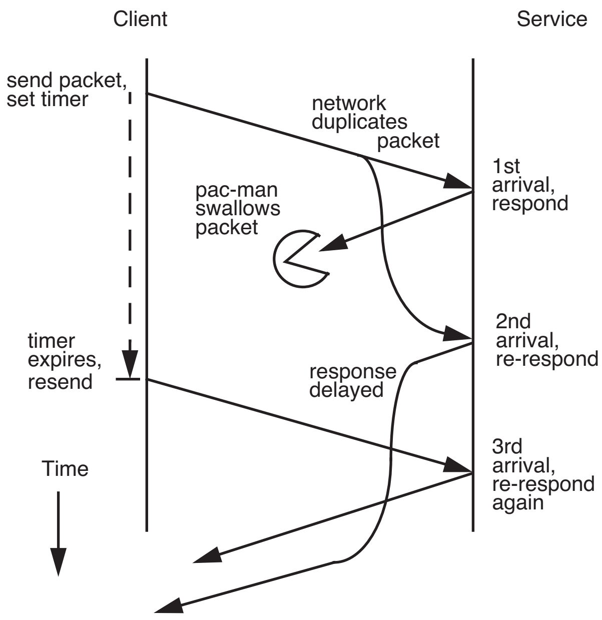
On the other hand, since prepage-OPT moves pages into the primary device before they are needed, the I/O operations involved in that movement could be overlapped with execution of the application program. With demand-OPT, the I/O operation to bring in a page does not begin until the program touches the page, so overlapping is not possible. This consideration might give prepage-OPT some slight advantage. But, since both policies require knowledge of the future, neither is implementable in practice, so the prospective performance advantage of prepage-OPT disappears. Louis should look elsewhere for applications of prepaging, and be cautious in using it.

7. Answers to the Exercises of Chapter 7

Ex. 7.1 *Layering.* Layers transform one set of functions into another. Layers also interact only with adjacent layers. Protocol stacks are organized in exactly this manner: each protocol layer transforms the underlying layer's function into a more convenient form.

Ex. 7.2 It is an argument that helps guide placement of function, rather than a rule about where to place function, so answer **A** is the appropriate one. Anonymity is not involved, and by itself it is not a debate (though it may trigger one).

Ex. 7.3 Answer **E** is over-ambitious; an *end-to-end argument* does not suggest moving *all* functions to the upper layers, just those that the upper layers have to do anyway. The chain manufacturer of answer **C** matches the problem statement quite well: checking individual links doesn't tell whether or not the links will be correctly assembled. Only an end-to-end check of the completed chain can do that. And if the end-to-end check is going to be done anyway, one could probably omit checking of the individual links.



Ex. 7.4 The timing diagram at the left shows two things that could lead to duplicate packets arriving at the service: the network may duplicate the request internally or the network may lose or delay the service's response to the request. In either of the latter two cases, the client will time out and resend the request, causing it to arrive at least twice at the service.

Ex. 7.5a No. A source of failure to deliver frames is collision. Shortening the retransmission wait will only increase the probability of collision.

Ex. 7.5b No. If deliveries are lost or delayed because of congestion, retransmitting lost frames more frequently will only make the congestion worse.

Ex. 7.6 False. Isochronous networks have uniform delay by definition. (The root *iso* means equal, *chronous* means time.)

Ex. 7.7a At least 200 milliseconds, the median round-trip time, plus whatever time is required for B to generate an acknowledgment. If the link is not private, still more time should be added to allow for variations in queuing delays, since the median queuing delay will be exceeded by about half of the frames. But A should not wait much more than that because longer waits will just delay recovery when a frame really is lost.

Ex. 7.7b Each frame from A arrives undamaged at B with probability $(1-\alpha)$. Each acknowledgment from B arrives undamaged at A with probability $(1-\alpha)$. Thus the probability that A will receive an acknowledgment for a given frame that it sends is $(1-\alpha)^2$. If A has *N* frames to send to B, A will have to perform enough send operations to eventually receive *N* acknowledgments from B (one for each of the *N* frames). But since some packets are lost, A will have to send $N/(1-\alpha)^2$ frames to receive *N* acknowledgments. Therefore, A will have to send each frame an average of $1/(1-\alpha)^2$ times.

Ex. 7.8 **A** and **C**. Answer **B** sounds vaguely plausible—the end-to-end layer does tell the network layer what end-to-end protocol it is using. However, the purpose of this information is solely to allow demultiplexing at the delivery point; the network layer really should not be using it for other purposes such as implementing separate queues. (Recall that the only reason this information appears in the network-layer header is to avoid having a separate multiplexing layer.) Answer **D** does not work at all, because the link layer wouldn't know what end-to-end protocol was being used, unless it looked inside the network header, which would violate the layer abstraction.

Ex. 7.9 **D**. Answer **B** is the definition of *overload*, not congestion. Answer **C** might indicate congestion in the mail system, which is store-and-forward, but not in the receive-and-forward network. The flow control cycle of answer **A** would be an end-to-end problem, not a network problem. And answer **E** is just bogus.

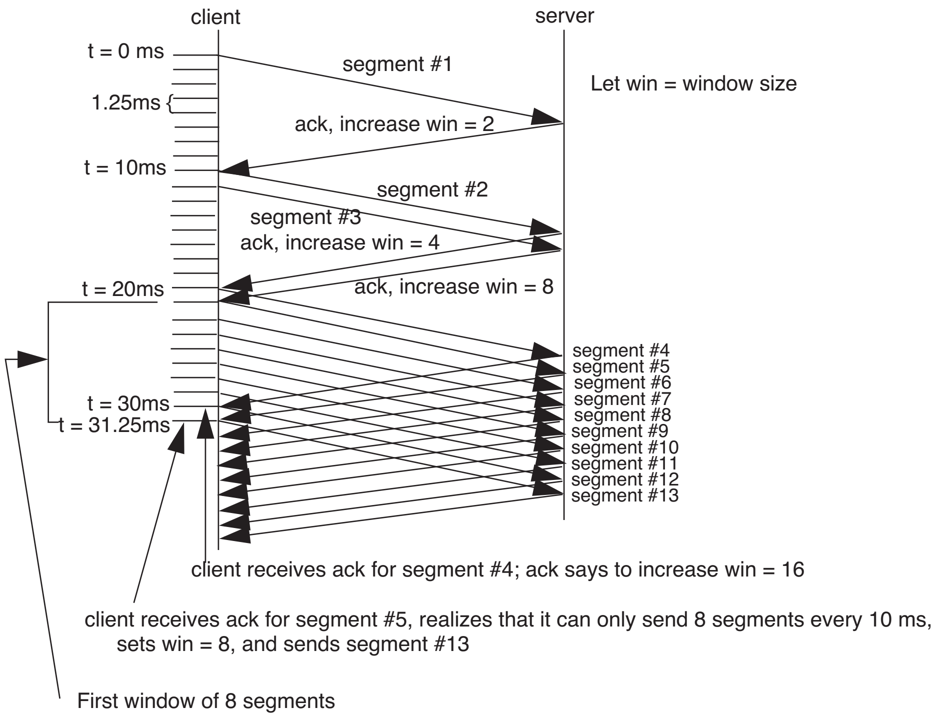
Ex. 7.10 Alice sends segment 1, which Bob receives twice. So Bob sends back two acknowledgments saying "OK to send block 2". Alice receives four of these messages, so she sends the segment number 2 four times. Bob receives 8 blocks numbered 2, Alice receives 16 messages saying "OK to send block 3", Bob receives 32 blocks numbered 3, and Alice has to ignore 64 go-aheads for block 4.

Ex. 7.11 The problem is that the network may be duplicating window opening messages, and the protocol is not idempotent. If a window opening message gets duplicated on its way from B back to A, A will increase the window more than once, and go on to overrun B's buffers. The protocol designer should also worry about lost, or reordered messages. If a window opening message gets lost, then A will never again use that part of the window. Reordered window opening messages have little or no effect, because addition is commutative. The protocol can be made idempotent by having B send window-opening messages that specify the cumulative total window, rather than the amount to add. This change fixes the duplicate problem and some, but not all, instances of the lost message problem.

Ex. 7.12a In order to achieve 800,000 bytes per second throughput, the client must send 800 segments per second, or 8 segments per 10 milliseconds. If the round-trip time is 10 milliseconds, then the client must send 8 segments before it receives an acknowledgment for any of them, making the window size 8 segments.

Here is another way to calculate the answer: on an 800,000 bytes per second link the client takes 1.25 milliseconds to send one 1000 byte frame. Thus, the client can send 8 frames in 10 milliseconds and saturate the link bandwidth.

Ex. 7.12b The client realizes it has reached maximum throughput after 31.25 milliseconds, because that is when it begins receiving acknowledgments while it is still transmitting; at that instant it has eight segments in the pipeline, so it knows that is the largest window needed. The figure below illustrates.



Ex. 7.12c The client will receive the acknowledgment for the first segment at the end of 10 milliseconds, at which instant it has 8 segments outstanding, so it continues to use that number for its window size.

Ex. 7.13a The transit time is 1 millisecond (transmission time) plus 249 milliseconds (propagation time) so if we assume that R can absorb message segments instantly the round-trip time is 500 milliseconds. We can move one segment every 500 milliseconds and there are 1000 segments to move, so it will take at least 500 seconds.

Ex. 7.13b Measuring time in milliseconds, at t = *0,* R sends a go-ahead. At t = *250*, T begins transmitting, and at t = *500*, data starts to land on R. R immediately sends a pause; that pause will arrive at T at t = *750*, by which time 500 segments will already be in flight. So B needs to be able to hold a minimum of 500 segments. The 500th segment will arrive at R at t = *1000.*

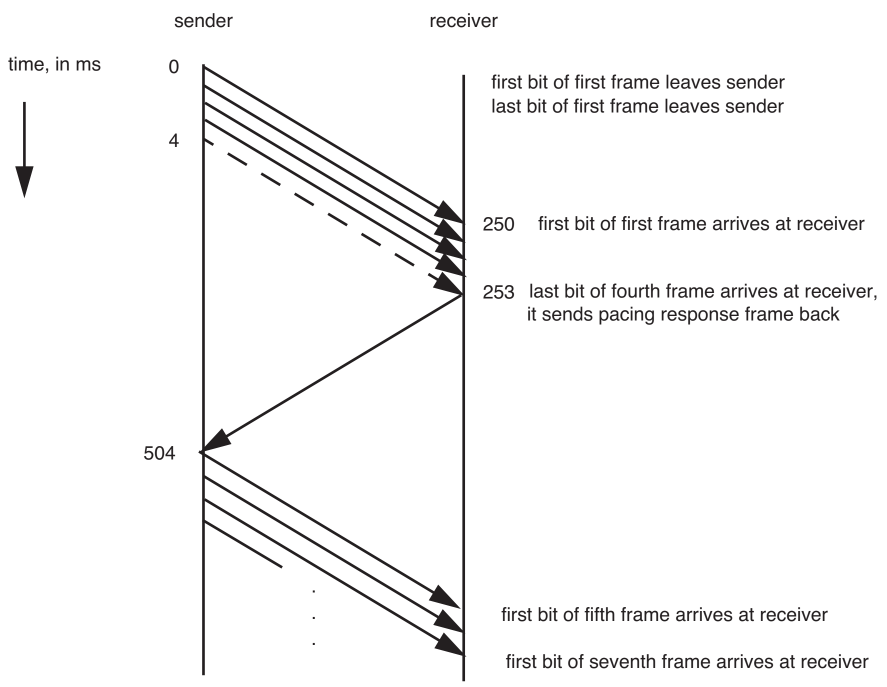
Ex. 7.13c Each round of the bang-bang protocol takes one second, and will result in 500 segments delivered to R. Again assuming that R can absorb segments instantly it will take 2 seconds to deliver the file.

Ex. 7.14 The only reasonable answer is **A**. If the end-to-end protocol needs to assure delivery accuracy, it can do so by including a copy of the destination address in the end-to-end header where the receiving end can verify it. Answer **B** is an over-stretch of the end-to-end argument; the function of the network layer is routing, and it wouldn't make sense to have a network layer if there were a good argument for having the end-to-end layer do it. Answer **C** would require the network layer to peer inside its end-to-end payload, a violation of the layer abstraction. Answer **D** is just handwaving with buzzwords; the sender determines the contents of all the fields of the end-to-end header, it doesn't look at those contents to make decisions.

Ex. 7.15 For maximum space reduction, there must be links among hierarchically related packet forwarders. If such links are available, a single table entry can tell where to dispatch all packets that are not destined for attachment points located in the hierarchy below this packet forwarder. There can in addition be cross links, but for them to be used the packet forwarder must have extra table entries that describe what set of attachment points can be reached more effectively by those cross links.

Another way of thinking about this trade-off is that we are taking advantage of the hierarchical shape of the names to do a lossless compression of the routing tables. If the network were organized hierarchically, the compression would be maximal. If the network is not hierarchical, but we compress as though it were, then we would have a lossy compression scheme, for which the losses would show up as not knowing the best available routes. Curiously, everything would still work "correctly," except that the performance would be suboptimal.

Ex. 7.16a


sender
receiver
time, in ms
0
4
504
first bit of first frame leaves sender
last bit of first frame leaves sender
250 first bit of first frame arrives at receiver
253 last bit of fourth frame arrives at receiver, it sends pacing response frame back
first bit of fifth frame arrives at receiver
first bit of seventh frame arrives at receiver

Ex. 7.16b We are transmitting 4 out of every 504 milliseconds, so the utilization is a tiny bit less than 0.8%

Ex. 7.16c If the pacing window is of size N segments, we will transmit for N milliseconds, wait for 500 milliseconds, then begin transmitting again. Our utilization will be $N/(N + 500)$. For a utilization of 50%, we must use a pacing window of $N = 500$ segments.

Ex. 7.17

A. True: hosts that reside behind a NAT have addresses that hosts outside the NAT cannot utter.

B. True: the NAT has to know how to examine payload data in order for it to find and translate all examples of network addresses, but it is typically installed inside the network rather than at an end point, so it must violate the layering abstraction.

C. False: NATs reuse Internet addresses, so they actually reduce, rather than increase, consumption.

D. True: the method of allocation of Internet address blocks tends to waste addresses and there is now a shortage.

E. True for the same reason as **B**: if a new end-to-end protocol places addresses in the payload, it won't work through any NAT that hasn't been updated to know about the new protocol.

F. False: The Ethernet checksum is in the link-layer header. The link layer doesn't calculate the checksum until the NAT box calls on it to send the packet, so the checksum calculation automatically includes any address translations the NAT box did. At the other end of the link, the link layer may receive a packet with a bad checksum and discard it, but the reason for the bad checksum can't be that the NAT box translated some network-layer address after calculation of the checksum.

G. False: Because public Internet addresses don't need translation, a NAT box translates only Internet addresses belonging to hosts behind it. Hosts outside the NAT cannot know Internet addresses of hosts behind the NAT, so their packet payloads can't contain those addresses.

H. False: The path between a client and server that are both behind the same NAT should not require address translation. If the NAT for some reason is in that path (for example, it is also acting as an ordinary forwarder) it will not look in its translation table for addresses within the private network. And in its role as an ordinary forwarder it should have a route to the server.

Ex. 7.18 The two-port structure minimizes byte-swapping for all possible combinations of big-endian and little-endian clients and services, resulting in better overall performance. Specifically, if both client and service are same-endian, then there will be no byte-swapping. If they are different-endian, then multibyte data such as long integers will be swapped exactly once.

The alternative, in which the service accepts only little-endian data, requires a big-endian client to byte-swap its data into little-endian form before sending it, even if the service happens to also be big-endian and will have to byte-swap it again. This additional byte-swap might result in a substantial performance decrease.

An even better alternative would be to use a single port, and have the service and the client negotiate at the outset which one, if any, will do byte-swapping. (Exercise: assuming the two ends have different byte-swapping costs, design a negotiation protocol that always minimizes the cost of byte-swapping.)

Ex. 7.19 False. Contention control in the Ethernet is actually quite decentralized—each transceiver independently plays its own part. The lack of scalability of the Ethernet comes from its broadcast nature, and the resulting need for every signal to propagate to every point on the net, the speed of light being the fundamental limiting factor.

Ex. 7.20

- **A.** Michelson and Morley established that there is no luminiferous ether.
- **B.** The original Ethernet used Manchester encoding, to simplify carrier detection, but modern higher speed Ethernets that use full-duplex point-to-point links don't.
- **C.** Ethernets use exponential backoff.
- **D.** Ethernets do retransmissions, but it is because of congestion, rather than to avoid it.
- **E.** Yes, carrier sense and collision detection are the responsibility of each station on an Ethernet.
- **F.** An ideal property that no real network can satisfy.
- **G.** Another ideal, unrealizable property.
- **H.** Although linking Ethernets with routers would permit unbounded physical range, the result is not an Ethernet, it is a network consisting of several Ethernets. Each individual Ethernet remains limited in physical range.

Ex. 7.21 E. When used in the Internet, the only uniqueness requirement is that all of the stations on a single Ethernet have different station addresses, and those station addresses are used in, among other things, an address resolution protocol that helps map Ethernet station addresses to Internet addresses.

Ex. 7.22 There may be repeated collisions, or the power may fail, or a plumber may sever the cable. In the real world there is never a guarantee of success, so **A** must be wrong. On the other hand, the random exponential back-off algorithm used in the Ethernet makes it very likely that all the stations will eventually get their packets through, so **B** is a better answer than **C** or **D**.

Ex. 7.23 It made a certain amount of sense to modify RPC to pass pointers in a more general way, because procedure calls frequently involve pointers. But many other protocols such as file transfer or remote login pass pointers only very rarely, or perhaps not at all. Requiring all end-to-end protocols to support this feature will add complication and may incur needless performance penalties. Instead, only those protocols that regularly pass pointers should be modified to add this feature. If an application using one of the unmodified protocols has to pass a pointer, it should explicitly include the necessary context reference. (This is an example of an *end-to-end argument*.)

Ex. 7.24a If a packet is in transit to mobile host A at the time when A moves from location X to location Y, that packet will arrive at X and find that A has left. Since there is no information at X indicating where A went, the packet will be lost. Similarly, any packet that the home node forwards after A has moved but before A's rerouting message gets to home will go to location X and be lost. All packets that home forwards to A from about one network transit time before the move until about one network transit time after the move will probably be lost.

A more disastrous scenario, though less likely and therefore probably not the cause of messages "regularly" getting lost, happens when a mobile host A moves from X to Y, and then on to a new location Z. The rerouting message that A sends from Y may be delayed in transit long enough that the rerouting message it sends from Z gets home first. In that case, when the rerouting message sent from Y finally arrives, home will forward all future packets to the wrong place; A will be completely cut off, at least until its next move. The briefer the visit of A to Y, the greater the chance of this scenario.

Ex. 7.24b *No and Yes* (a typical systems answer!).

No, packets will not get lost: Suppose mobile host A moves away from N' just after N' responds to the search request N sent. N will forward the original packet to N', but when it arrives, N' will follow the standard procedure: If it is in contact with A it will pass the packet along to A; however, since it is not, it broadcasts another search request. Assuming A is always in contact with some node, the original packet will thus follow A around. If A ever rests long enough for the packet to catch up with it, it will then be delivered.

Yes, packets will get lost: Search broadcasts have a transit time, and any given broadcast doesn't get to all nodes at exactly the same time. Suppose node N broadcasts a search request, and while the broadcast is in transit, A moves from node X to node Y. The following time sequence creates a problem:

1. N broadcasts "Where is A?";
2. Y receives broadcast, ignores it because A is at X;
3. A moves from X to Y;
4. X receives broadcast, ignores it because A is now at Y;
5. N concludes from the lack of response that A has retired from active service and trashes the original packet.

And the following briefer example of the above scenario, in which Y is actually N, can lose packets even if broadcasts are delivered everywhere simultaneously. (We assume that mobile hosts don't actively notify locations when they arrive; instead, the locations try to locate them when packets for them are received.)

1. N broadcasts "Where is A?";
2. A moves from X to N;
3. X receives broadcast, ignores it because A is now at N;
4. N concludes from the lack of response that A has retired from active service and trashes the original packet.

Ex. 7.24c There are lots of ways to lose packets with this protocol. Here are three of them (numbered for future reference).

P1. Since the underlying network can reorder packets, the "delete forwarding pointers" message may overtake a packet already in transit along the forwarding chain. Then, when the packet gets to its next forwarding point it will find that the forwarding information has already been deleted.

P2. If mobile host A creates a long forwarding chain, sends an "I'm over here at node X" message home, then immediately moves to a location Y that is earlier in the chain and then on to somewhere else, the "delete forwarding pointers" message will come along and delete the current pointer in Y, leaving A unable to receive anything.

P3. A version of the disastrous scenario of problem *Ex. 7.24a* also applies. A moves from X to Y, sends a message home, then moves to Z and sends another message home. The second message home gets there first, causing the chain out to Y (including the forwarding pointer from Y to Z) to be deleted. Then the first message gets home, telling home to send everything to Y.

Ex. 7.24d Here are a number of different solutions.

S1. Since the latest "I'm here" message from mobile host A to its home short-circuits any forwarding pointers in other nodes, the home node could just omit sending the message that deletes old forwarding pointers. That would solve all three problems of part 2c, which is good enough. However, it would leave dead forwarding information lying around, using up space, with no way to know when it can be reclaimed (call that problem P4).

S2. If the home node knew the maximum transit time of the network, it could delay the sending of the "delete forwarding pointers" message for that length of time. This approach would take care of problem P1 and problem P4 and, with a clever home node, might help reduce the likelihood of problem P3. Unfortunately, it will increase the likelihood of problem P2.

S3. The home node could include sequence numbers in all forwarded packets. Then, when it gets a rerouting message from a mobile host, it can immediately begin forwarding new packets directly to the new location, and it can also send a message to the mobile host telling it the sequence number of the last packet sent via the old route, and asking for an explicit acknowledgment when all packets up to the sequence number have arrived. (Note that a timestamp isn't good enough here; we need a way of being sure that **all** intervening packets have been delivered.) When that acknowledgment returns to home, send the "delete forwarding pointers" message down the old chain. Problems P1 and P4 are solved. Problems P2 and P3 remain.

S4. Instead of sending a "delete forwarding pointers" message, the home node could send a "change forwarding pointers" message, alerting everyone along the old forwarding chain to adjust their forwarding pointer to A's new location. That would eliminate problems P1 and P2, but problems P3 and P4 remain.

S5. The home node could include its own address in every forwarded packet, and all other forwarders could follow a new rule: if a packet is undeliverable as originally addressed, send it back to the home node. The home node can then rescue and reroute any packets that would have been lost under the other methods. In addition, when it gets a rerouting message from a mobile node, it can immediately send a "delete forwarding information" request down the old route, confident that any dallying packets will be returned to it for rerouting. This approach takes care of problems P1 and P4, and some instances of P2. It also means that when problem P3 (or the bad instances of P2) arise, any packets that the home tries to forward to the mobile will loop indefinitely between home and the last place home thought the mobile was connected; if the mobile ever checks in again, these packets will eventually be delivered. Whether this is a satisfactory state of affairs is debatable.

S6. The mobile host could count the number of rerouting messages it sends home, sending a sequence number with each message. These sequence numbers impose an order on rerouting messages, so problem P3 is solved. The last sequence number the mobile sent is attached to each forwarding pointer it creates. When the home node receives a rerouting message with sequence number S, where S is greater than the last sequence number home saw from that mobile host, it sends out a message "delete forwarding information with sequence number smaller than S." This solves problem P2. Combine this method with solution S5 (undeliverable packets are sent back to the home node) and we have solved all four problems, P1 through P4!

Note that none of the suggested solutions involve adding an acknowledge + timer-expiration + retransmit scheme to the forwarders. That approach would have the side effect of producing duplicates in a network that is certified never to produce duplicates, so it would require modifying the end-to-end protocol implementations to suppress these duplicates.

Ex. 7.25a It takes 1,000 microseconds to push a 1,000 byte segment into the network. Since the propagation delay is 500 microseconds, the last byte of the segment will arrive at the receiver after 1,500 microseconds. The acknowledgment takes 500 microseconds, so the time to send one segment and receive an acknowledgment is 2,000 microseconds. The maximum throughput therefore is 500,000 bytes per second.

Ex. 7.25b The sender can push one segment into the network every 1,000 microseconds. The send window will never close, since before the sender has sent 8 segments, acknowledgments will have come in for earlier segments, keeping the window open (assuming the network is full duplex). Therefore, the throughput is 1,000,000 bytes per second.

Ex. 7.25c For a large message the maximum throughput is equal to the rate at which the sender can push segments into the network, since the one acknowledgment at the end is negligible compared to sending many 1,000-byte data segments. Since the sender can push one 1,000 byte segment into the network every 1,000 microseconds, the maximum throughput is again 1,000,000 bytes per second.

Ex. 7.25d **C**. Because messages can be very large and the receiver might take a long time to process a segment, Blast (answer **B**) might overrun the 20,000-byte buffer at the receiver, causing segments to be dropped. Flow-control (answer **C**) will not overrun the 20,000-byte buffer, since its send window is only 8,000 bytes (8 segments). Since Blast might be unreliable and Send-and-wait (answer **A**) is slower than Flow-control, Flow-control is the right product to use.

Ex. 7.26a (480 bytes)/(0.1 seconds) = 4,800 bytes per second

Ex. 7.26b To send 100 blocks requires that 200 packets traverse the network, of which an average of 2 will be lost, so 2 blocks will require retransmission. The average time to send 100 blocks (48,000 bytes) is thus

```
(98 blocks) x (0.1 seconds)   =     9.8 seconds
+ (2 blocks) x (1.1 seconds) =     2.2 seconds
                 total        =    12.0 seconds
average time is 120 milliseconds per block
480/.12 = 4,000 bytes per second
```

(This approximation ignores the possibility that 2% of the retransmitted blocks will also be lost. A more precise answer (about 3990 bytes per second) isn't worth the effort because the 1% packet loss figure probably isn't that accurate anyway.)

Note that we must average the transmission times, not the transmission rates. The reason is that although 2% of the blocks are retransmitted, much more than 2% of the time is spent on retransmitted blocks.

Ex. 7.26c If we send 100 blocks on a Tuesday, an average of 50 of them will require 100 milliseconds, and the other 50 will require 200 milliseconds. Total time for 100 blocks is

```
50 x 100 milliseconds        =    5 seconds
50 x 200 milliseconds        =   10 seconds
              total              15 seconds, or 150 milliseconds per block
(480 bytes)/(.150 second)  =   3200 bytes per second
```

Again, we must average times, not data rates.

Ex. 7.26d If the sender is willing to accept the first packet that is responsive to a request, then the data rate does not change; there are a number of duplicate packets flying around but they always arrive at times when the client or the server are waiting for something else and can discard or re-respond without interfering with useful activity. Rate = 3200 bytes per second.

If the sender insists on ignoring responses that don't match with specific requests, the calculation is harder. The time to send one block will be 100 milliseconds, or 250 milliseconds, or 400 milliseconds, or 550 milliseconds, etc., depending on how many tries are needed to get an undelayed round trip.

At this point, one can either approximate by taking the first few terms of the series or one can compute the exact series sum. Here is an elegant way to get the exact sum:

Let *t* be the expected delay.
There's a 50% chance the round trip takes 100 milliseconds.
There's a 50% chance a retry will be started after 150 milliseconds.
The retry has the same expected delay, starting from when it was sent.
Therefore, working in milliseconds:

$$t = 0.5 \times 100 + 0.5 \times (150 + t)$$

Solving for *t*:

$$2t = 100 + 150 + t$$
$$t = 250 \text{ milliseconds}$$

The data rate is (480 bytes)/(0.25 seconds) = 1920 bytes per second

Ex. 7.26e The data rate of the slowest link is one of the things that contributes to the round-trip time, so it is already included in the given parameters.

Ex. 7.27a The technical definition of "Transmission delay" is just the time required to send the data. That time is

$$\frac{\text{RPC request size in bits}}{\text{Channel bandwidth in bits per second}} = \frac{50 \text{ bytes} \times 8 \text{ bits per byte}}{1.5 \times 10^{6} \text{ bits per second}} = 267 \times 10^{-6} \text{second}$$

Ex. 7.27b In the steady state, everything running as fast as possible, the following times add up to the round trip taken by one RPC:

client computation + req. ransmission delay + req. communication latency +
server computation + reply transmission delay + reply comm. latency =
$100 \times 10^{-6}\text{sec} + 267 \times 10^{-6}\text{sec} + 12.5 \times 10^{-3}\text{sec} +$
$50 \times 10^{-6}\text{sec} + 267 \times 10^{-6}\text{sec} + 12.5 \times 10^{-3}\text{sec} = 25.684 \times 10^{-3}\text{sec/RPC}$

Inverting gives a sustained maximum rate of 38.9 remote procedure calls per second.

Ex. 7.27c The assertion in the problem that "pipe calls return immediately to the caller" is the key. With that assumption, the number of calls is limited only by the client computation time between calls:

$$\frac{1}{100 \times 10^{-6}\text{sec/call}} = 10{,}000 \text{ calls/sec}$$

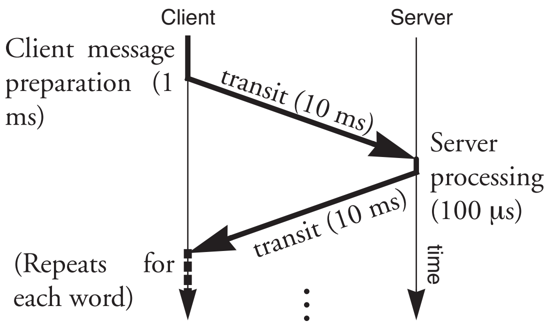
Since the problem mentioned that the application runs in a single thread, one could instead assume that the thread making the pipe calls must spend time sending the 1000 byte pipe request message after each 96 pipe calls. In that case, the total time for 96 calls would be

$$96 \times \left(100 \times 10^{-6}\text{seconds}\right) + \frac{1000\text{ bytes} \times 8\text{ bits per byte}}{1.5 \times 10^{6}\text{ bits per second}} = 14.93 \times 10^{-3}\text{seconds}$$

giving a sustained maximum rate of $\frac{96}{14.93 \times 10^{-3}\text{sec}} = 6428.6$ calls per second

Ex. 7.27d C.

Ex. 7.28a For RPC, each call blocks at the client until the server reports that the procedure has been completed. All events happen sequentially, so the times for message preparation, client-to-server transit time, server processing, and server-to-client transit time can be simply added as shown in the figure:

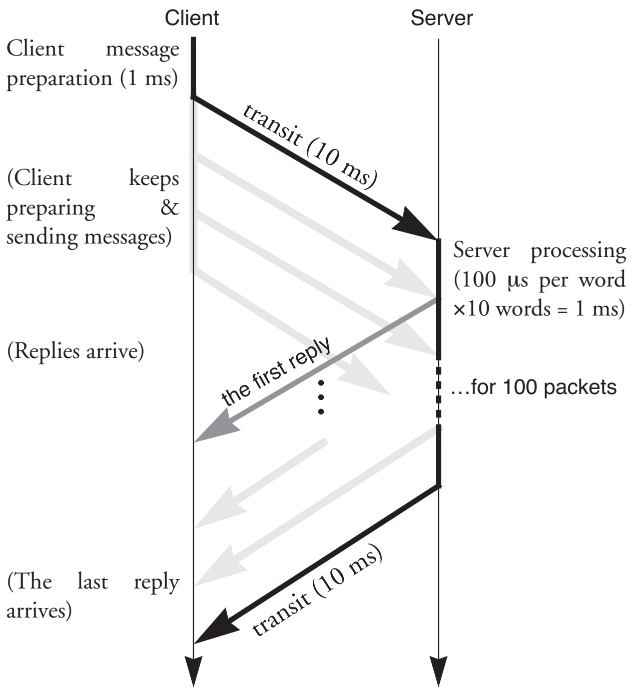


Cost per message = 1 ms + 10 ms + 0.1 ms + 10 ms = 21.1 milliseconds.
21.1 milliseconds per message × 1 message per word × 1000 words = 21.1 seconds.

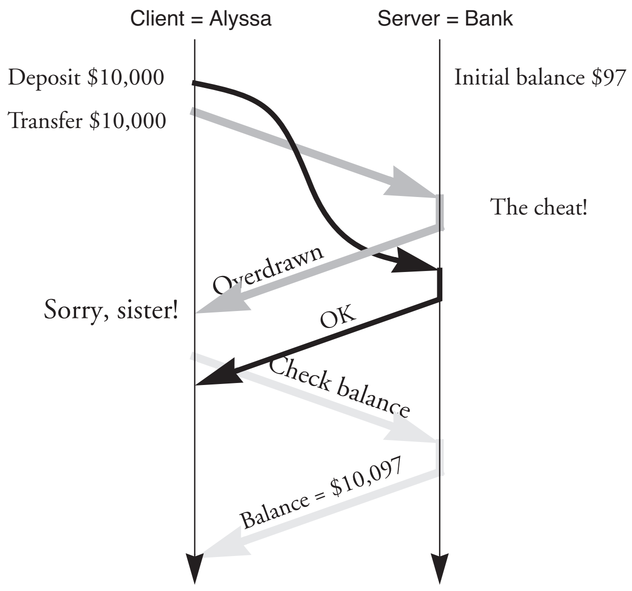
Ex. 7.28b Grouping 10 words per message cuts down on the number of messages, and hence, the effect of network transit time. Note that client preparation time is 1 millisecond per message, not per word, whereas server processing time is 100 microseconds per word, not per message. (Assuming the server has a single processor, so it can't check the 10 grouped words in parallel.)

Cost per message = 1 ms + 10 ms + 10 words per message × 0.1 ms per word + 10 ms = 22 ms.
22 milliseconds per message × 1/10 messages per word × 1000 words = 2.2 seconds.

Ex. 7.28c Since we are using an asynchronous protocol, the client can continue immediately after it sends a message. This implies that requests and replies will overlap each other on the network, and that client and server processing can take place simultaneously. Specifically, we can't just add client and server times together, and we can't add transit time for each message. For a grouping of 10 words per message, both the client preparation time and server processing time are 1 millisecond: both are the bottleneck (so having a multiprocessor server won't help very much). The first and last messages are special cases that add to this time. The figure at the right illustrates.


Client
Server
Client message preparation (1 ms)
transit (10 ms)
(Client keeps preparing & sending messages)
Server processing (100 μs per word ×10 words = 1 ms)
(Replies arrive)
the first reply
…for 100 packets
(The last reply arrives)
transit (10 ms)

Time for client to prepare first request + transit time from first request + server processing time for 100 messages + transit time for last reply
= 1 ms + 10 ms + 100 messages × 1 ms per message + 10 ms = 0.121 seconds!



Ex. 7.28d Referring to the figure at the left, the problem arose when the network reordered the deposit and transfer requests. Alternatively, the deposit request may have been lost and retransmitted later, resulting in the same effect. The check-balance request then occurs after the deposit has succeeded.

8. Answers to the Exercises of Chapter 8

Ex. 8.1 **C**. A *failure* is when a module fails to meet its specification. Latent faults don't cause failures, so that excludes **A**. If a module succeeds in completely containing an error, it meets it specification, so that excludes **B**. When a fault causes an error that error may or may not cause a failure, depending on whether or not it is successfully masked, so **D** is not correct.

Ex. 8.2

A. Not tolerated. If the software has a deterministic error, all versions will compute the same incorrect result, and the voter will return a unanimously wrong answer.

B. Tolerated. Even though the twelve computers with corrupted memory may compute incorrect results, the 18 uncorrupted computers are enough to constitute a majority and the voting procedure will ensure that Ben gets a correct result.

C. Tolerated. Because of the hourly backup to disk, no matter how many times the power fails and the computers have to restart, all of them can continue calculating correctly. The worst thing that happens is that each power failure sets back the computation as much as an hour, so if there are a lot of power failures it might take longer than Ben hopes to get the result.

Ex. 8.3 **D**. Ben's design will not call the fire department unless four or more of the smoke detectors report the fire. Smoke detectors can fail either by giving a false alarm (perhaps because of accumulated dust) or by not giving an alarm when they should (because their batteries have died). While Ben's design probably eliminates the false alarms, a dead battery will probably delay notifying the fire department, and four dead batteries will ensure that the fire department never gets called. Ben should have used smoke detectors that report that their batteries are alive (that is, they are fail-fast for missed alarms) and a voter that doesn't count dead smoke detectors (that is, it considers only correct inputs, in this case those that say their batteries are alive) when it is assessing a majority. Answer **C** is misguided; if a voter ignores inactive inputs, then any one active input will be a majority, which isn't what Ben wanted.

Ex. 8.4 Because the failure process is memoryless, the conditional failure probability for flight A is 1/6000 per hour and for flight B is 1/5000 per hour. The probability of the plane for flight A failing during its flight is thus 6/6000, and of the plane for flight B is 5/5000. In both cases, the probability of failure during the flight is 1/1000. Either flight is an equally good choice.

Ex. 8.5 An NMR system consisting of three fail-fast replicas with repair can do the job. Since with fail-fast replicas (and presuming that the voter ignores replicas that say they are down) the system continues to work properly unless all three replicas are out of service at the same time, $MTTF_{system}$ is approximately

$$\frac{MTTF_{replica}}{3 \cdot MTTR_{replica}} \times \frac{MTTF_{replica}}{2 \cdot MTTR_{replica}} \times MTTF_{replica} = \frac{1}{6} \cdot \frac{(MTTF_{replica})^3}{(MTTR_{replica})^2}$$

Plugging in the numbers given for $MTTF_{replica}$ and $MTTR_{replica}$ we get an $MTTF_{system}$ of about 10^8 hours, substantially more than required. Duplex with repair is not quite good enough, since then the $\text{MTTF}_{\text{system}}$ is approximately:

$$\frac{1}{2} \cdot \frac{(MTTF_{replica})^2}{MTTR_{replica}} = \frac{1}{2} \cdot 10^6 \text{ hours}$$

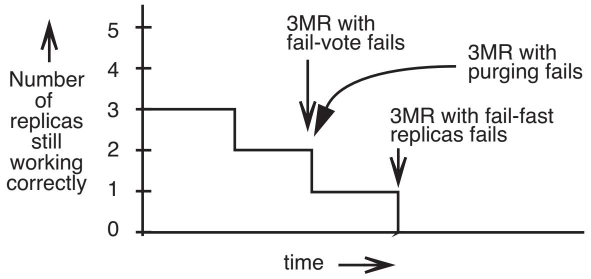
Ex. 8.6a The daisy chain is up if both links are up. Each link will be up with probability (1-p), so the probability that both are up is $(1\text{-}p)^2 = 1 - 2p + p^2$. If p is small, the probability of being down is approximately $2p$.

Ex. 8.6b The fully connected network is up if two or more links are up. The probability that all three links are up is $(1\text{-}p)^3$. The probability that links 1 and 2 are both up and 3 is down is $p(1\text{-}p)^2$; similarly for 1 and 3 up and 2 down; and again for 2 and 3 up and 1 down. These possibilities are mutually exclusive, so we can add their probabilities:

$$\text{Prob(network up)} = (1-p)^3 + 3p(1-p)^2 = 1 - 3p^2 + 2p^3$$

and if p is small, the probability of being down is approximately $3p^2$.

Ex. 8.6c The daisy chain will be down with probability $2 \cdot 10^{-6}$ and the fully connected network will be down with probability $3 \cdot 10^{-8}$. The fully connected network is better, despite its lower-reliability links.



Ex. 8.7 See the figure at the left. With only three replicas, purging does not help, because it requires at least two working replicas. So 3MR with purging fails at the same point as 3MR with fail-vote.

Ex. 8.8 First we derive a formula for the expected time S_k, measured in program steps, for a program of k steps to complete if there is a probability f of failure on each step. Then we reexpress the formula in multiples of the MTTF and, finally, apply the formula to the two cases.

Assume a program has managed to complete $(k-1)$ steps and consider for a moment how many tries, *T*, it will take to complete just the *k*th step. If the probability of being interrupted is *f*, the chance of success on the first try is $(1-f)$, on the second try is $f(1-f)$, on the third try is $f^2(1-f)$, etc. The expected number of tries is thus

$$T = (1-f) + 2f(1-f) + 3f^2(1-f) + \ldots = \frac{1}{(1-f)}$$

Of course, on each such try the program will first have to complete all $(k-1)$ previous steps. So how many steps of execution will it take to get to the point that *k* program steps are complete? Let S_k be the expected number of execution steps required to complete *k* program steps. If each program step fails independently of the next (that is, *T* is independent of *k*) we can simply add the expected values (this claim seems reasonable, but its proof, known as Wald's theorem, takes some effort), which allows us to make a recursive statement about S_k:

$$S_k = T + T \times S_{k-1}$$

That is, the total number of execution steps will be the expected number of tries required to complete that last program step plus the expected number of execution steps to complete the $(k-1)$ previous program steps, which we will, unfortunately, have to repeat *T* times. The recursion terminates by noting that a program of zero steps requires zero execution steps, so $S_0 = 0$. This recurrence can be rewritten

$$S_k = T + T^2 + T^3 + \ldots + T^k$$

which series sums, for finite, positive *k* and any value of *T*, to

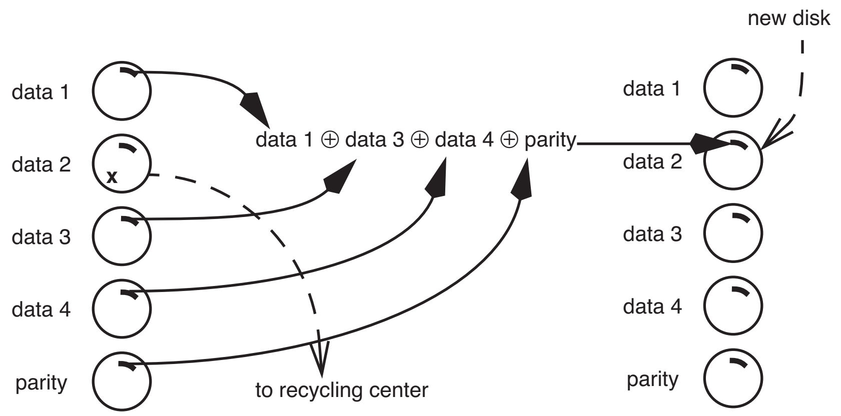
$$S_k = T \times \frac{T^k - 1}{T - 1} = \frac{1}{f}\left(\left(\frac{1}{(1-f)}\right)^k - 1\right)$$

Exact evaluation of this expression is tedious, but for small *f*, the first two terms of the Taylor series $e^{-f} = 1 - f + \ldots$ give us the very good approximation $S_k = \frac{1}{f}(e^{fk} - 1)$. Noting that $1/f$ is just the MTTF, and expressing *k* as a multiple *n* of the MTTF, we can reexpress the time required to complete the job as $S_n = MTTF(e^n - 1)$. We note with alarm that the time required to finish (or to get to the next checkpoint) increases exponentially with the MTTF multiple. Returning to our problem, $n = 10$ for Louis's plan and $n = 1$ for each segment of Ben's plan, but Ben's plan has ten segments, so we have

Louis:	$22,026 \cdot 10^{12}$	machine operations
Ben:	$17.2 \cdot 10^{12}$	machine operations

we conclude that Louis's plan will take, on average, 1,280 times as long as Ben's. Even so, Alyssa points out that Ben has missed an opportunity: if saving state really takes zero time, it would be wiser to save state following every successful program step. Then we would expect to see about ten failures, each losing only the time to execute one operation, with a total execution time of just $10^{13} + 10$ machine operations.

Ex. 8.9



Recovery of a sector on failed disk 2 of a five-disk RAID 4 system. To reconstruct the sector on data 2, we note that the parity sector was originally computed by:

$$\text{parity} \leftarrow \text{data 1} \oplus \text{data 2} \oplus \text{data 3} \oplus \text{data 4}$$

Since XOR is commutative and $x \oplus y \oplus x = y$, we can obtain data 2 by XORing parity with the corresponding data sectors on disks 1, 3, and 4.

Ex. 8.10a **B** and **C**. Every write must be made to both a data sector and a parity sector, so by distributing the parity sectors on different disks, RAID 5 reduces competition for the parity sectors. There is also a small improvement in read performance. Since RAID 5 places useful data on every disk, there is more opportunity for concurrent reads. Since statement **A** is true for both RAID 4 and RAID 5, it isn't a difference. **D** doesn't represent a difference either—in both configurations, a file map will contain an entry that says, e.g., that a file block is on sector 4173 of drive 2. **E** identifies a slight disadvantage of RAID 5. The disk sector allocation map has to avoid using the parity blocks used by the RAID 5 controller. With a RAID 4 controller, the disk sector allocation map just omits disk N completely. Answer **F** is incorrect; RAID 4 and RAID 5 require exactly the same amount of disk space. As for answer **G**, since **B** and **C** represent advantages, **G** must be false.

Ex. 8.10b Yes, but it requires a really contrived workload. Suppose, for example, one application intensely reads disk sectors, all of which are located on disk 1, and a second application writes only sectors located on disk 2. Under RAID 4, the reads cause no interference with the writes, since the writes involve writing a sector on disk 2 and reading and updating the corresponding sector on the parity disk. On the other hand, under RAID 5, at least some of the writes will require reading and updating a parity sector on disk 1. Those writes will encounter queuing delays because of the high disk 1 traffic generated by the other application.

Ex. 8.10c A two-replica RAID 1 configuration and a two-disk RAID 5 configuration can both store the same amount of data—the amount that would fill a single disk. But for $N > 2$, an N-replica RAID 1 configuration still can hold only the amount of data that would fill a single disk, while an N-disk RAID 5 configuration can hold an amount of data that would fill $N - 1$ disks. So in terms of storage space a RAID 5 configuration is always at least as efficient as RAID 1, and in most cases RAID 5 would be more efficient.

Ex. 8.10d With RAID 1 (the simple replica configuration), no matter how many replicas there may be, all disk operations needed to do either a single-sector GET or a single-sector PUT can always be done in parallel. Moreover, GET operations can be directed to whichever replica will have the smallest latency (queuing time + seek time + rotation time). With RAID 5, PUT operations require preceding GET operations in order to know how to update the parity sector and two PUTs, one to write the data sector and one to write the parity sector. A GET must be directed to the one disk containing the desired sector, so there is no opportunity to choose a disk with lower latency. Thus RAID 1 should have higher performance than RAID 5 under any workload, and significantly higher performance if the workload is heavy on PUTs.

Ex. 8.11 The rate of vibration failures is 1/30 failures per day, the rate of surge failures is 1/60 failures per day. The total failure rate is 1/30 + 1/60 = 1/20 failures per day. So the MTTF is 20 days.

9. Answers to the Exercises of Chapter 9

Ex. 9.1a Simple locking protocol:

```
procedure REGISTER_TWO (hx, hy)
    ACQUIRE (hx.slock)
    ACQUIRE (hy.slock)
    status ← REGISTER (hx)
    if status = 0 then
        status ← REGISTER (hy)
        if status = –1 then DROP (hx)
    RELEASE (hx.slock)
    RELEASE (hy.slock)
    return status
```

Ex. 9.1b Two-phase locking protocol:

```
procedure REGISTER_TWO (hx, hy)
    ACQUIRE (hx.slock)
    ACQUIRE (hy.slock)
    status ← REGISTER (hx)
    if status = 0 then
        ACQUIRE (hy.slock)
        status ← REGISTER (hy);
        RELEASE (hy.slock)
        if status = –1 then DROP (hx)
    RELEASE (hx.slock)
    return status
```

Ex. 9.1c Moving ACQUIRE and RELEASE to subroutines REGISTER and DROP makes it difficult to use those subroutines as components of larger transactions such as REGISTER_TWO. The first clue that there may be a problem is when you notice that the locking sequences that result do not conform to either the simple locking protocol or the two-phase locking protocol. As an example of what can go wrong, suppose that classes 21W731 and 21W732 each have one space available, and Jim requests

```
REGISTER_TWO ("21W731", "21W732")
```

at about the same time that Mary requests

```
REGISTER_TWO ("21W732", "21W733")
```

Jim takes the last slot in 21W731, then Mary takes the last slot in 21W732, then Jim notices that 21W732 is full, then Mary notices that 21W731 is full. Each then goes on to drop the class they previously registered for, and both go away to look for other classes with space. Although the registrar's data has not been corrupted, this is not an outcome that could have occurred under any serial ordering of these two requests—one of them should have been registered for both classes.

Ex. 9.2 Both Ben and Alyssa can be right, depending on what is meant by "won't work".

Ben's point is that if a transaction fails to call MARK_POINT_ANNOUNCE as early as possible, it will unnecessarily delay later transactions, with the result that the mark point discipline won't show a significant performance improvement compared with simple serialization. Since the only reason for using the mark point discipline is to improve performance, Ben can argue that abusing it in this way causes it not to work.

Alyssa is making a different point: If a transaction fails to call MARK_POINT_ANNOUNCE, the effect is to delay the next transaction until the first one commits or aborts. If all transactions behave this way, the result will be exactly the same as with simple serialization, so all of the transactions will have the before-or-after property.

Ex. 9.3 Since the lock already assures that the entire procedure has the before-or-after property, NEW_OUTCOME_RECORD can simply add one to its previous *id* value. For this plan to work correctly, the variable *id* must be a static variable so that its value is available on the next call.

```
procedure NEW_OUTCOME_RECORD (starting_state)
    declare id static initially 0
    ACQUIRE (outcome_record_lock)   // Entire procedure is before-or-after atomic.
    id ← id + 1
    allocate id.outcome_record
    id.outcome_record.state ← starting_state
    id.outcome_record.mark_state ← NULL
    RELEASE (outcome_record_lock)
    return id
```

Ex. 9.4 C. This problem is interesting because the second desideratum (not surprising the programmer, who wrote the program for a single-issue machine) requires using a stricter definition of correctness than the serializability that is acceptable for many database applications. Answer **A** would avoid surprising the programmer, but it would not have the best possible performance, because some sequences of instructions with overlapping register sets can safely be performed concurrently. Answer **B** is the database concept of correctness, and it would surprise a programmer by performing instructions correctly but out of order. **D** again would avoid surprising the programmer but it would not have any concurrency. **E** would really surprise the programmer—”some set of instructions” allows every possible result. Answer **F** can’t be right, because whenever two instructions issued at the same time modify the same register what is loaded in the register will depend on which result happens to get there first.

Ex. 9.5a After the commit, *x* is 4 and *y* is 3. If the transaction on the left had been performed completely before the one on the right, the values would have both been 3; if the transaction on the right had been done before the one on the left, the values would have both been 4. Since we are getting a result that is not the same as any possible serial ordering of the two transactions, this coordination protocol is not providing serializability.

Ex. 9.5b In practice, access to variables shared between transactions that might run at the same time tends to be quite simple and stylized. The rules as given do assure before-or-after atomicity between two transactions that read and then modify the same variable, which is probably the most common case of shared variables. Other kinds of sharing, such as reading from one shared variable and then modifying a different one are actually not very common, so the chance that two different transactions might be doing that kind of sharing simultaneously is actually quite remote. Finally, if a database management system produces wrong answers only very rarely, it will probably never be noticed, or blamed on something else.

Ex. 9.6a If the system is using rollback recovery, it should scan back to the start record of transaction 52, so that it can be undone.

Ex. 9.6b If the system is using roll-forward recovery, it should scan back to the start record of transaction 51 (which committed after the checkpoint but whose installs may not have been completed), to make sure that everything it has changed has been redone.

Ex. 9.6c Losers: transactions 52, 55, and 57. Winners: transactions 17, 51, 53, 54, and 56. This answer does not depend on which kind of recovery protocol is being used.

Ex. 9.6d $x = 2$, because transaction 56 committed that value. $y = 4$, because transaction 54 committed that value. Transaction 55's attempt to change y has been rolled back.

Ex. 9.7 The only shared variables are x and y, so they must be the clue. Reconstructing their version history up to the point where transaction 55 updates y, we run across the following remarkable state, where $a \rightarrow b$ means that the transaction identified in the label at the top of the column updates this variable from old value a to new value b:

transaction	17	51	52	53	54	55	56
x				5→9			
y				5→6	6→4	4→3	
commit state	A	P	P	C	P	P	P

Transaction 55 seems to be using as its old value for y the new value that transaction 54 has proposed, but transaction 54 has not yet committed. If transaction 54 aborts, transaction 55 would be violating the all-or-nothing requirement that there should be no evidence that transaction 54 ever existed. Perhaps transaction 55's real purpose is to force y to 3 no matter what its previous value. But if transaction 54 now aborts, it will restore the value 6 to y, which is not what transaction 55 intended. The only way this could work is if the system is using some kind of optimistic coordination procedure that provides cascaded aborts, so that aborting transaction 54 forces aborting transaction 55, and the resulting undos are carefully ordered.

Ex. 9.8 **B**. A roll-forward recovery manager performs only redos, so answer **A** can't be right. Answers **C** and **D** are diversions; they have nothing to do with this problem.

Ex. 9.9 **B**. The purpose of the two-phase locking discipline is to ensure that all resources needed to abort a transaction remain acquired until the commit point, so as to assure atomicity. One can still create a deadlock, so answer **A** does not apply. When compared with simple locking, two-phase locking can reduce the time that resources are locked, but there is no assurance that it minimizes that time, so **C** is not correct. Answers **D** and **E** are concatenations of buzzwords that sound like they might have something to do with the question.

Ex. 9.10a Protocol III is the first one that can be called a two-phase commit protocol.

Ex. 9.10b Step d of protocol III places some individual participants in a PREPARED state, in effect tentatively committing them, but because other participants may send apologies rather than PREPARED responses, the transaction as a whole cannot commit until Pat knows that everyone is PREPARED. The sending of Pat's ON/OFF message in step e is the commitment step for the entire transaction. Steps g and h are a refinement that allow Pat to discard her notes after the last acknowledgment comes back, but they contribute nothing to the 2-phase commit protocol itself.

Ex. 9.11a Overwrite semantics violate the *golden rule of atomicity*:

1. If the disk crashes during a write, a partial write results; the data will be corrupt and there is no way to reconstruct it.

2. If a commit requires multiple writes and a crash occurs between these writes, there is not enough information to abort the transaction on recovery by undoing its changes. (Similarly, there is not enough information on the disk to handle recovery after a nested transaction commits, when the top-level transaction has not committed: we should abort the nested transaction, but how will we do this?)

Ex. 9.11b As changes are made over time, the log will grow larger and larger. The disadvantages are:

1. Slow recovery, since the entire log must be replayed.

2. Slow reads, since the whole log may need to be examined to find the current value for a variable.

3. A lot of disk space is consumed.

Ex. 9.11c

	Undone	*Redone*	*Neither*
T1	√		
T2		√	
T3			√
T4			√
T5		√	

Ex. 9.11d

	Undone	*Redone*	*Neither*
T1	√		
T2	√		
T3	√		
T4			√
T5		√	

Ex. 9.12 **A**, **B**, and **C**. It is OK for Bob to abort any time before he sends message 7 saying that he is PREPARED. If Alice has not received Bob's PREPARED message, she will not tell Charles to COMMIT. If Alice has received Bob's PREPARED, she may tell Charles to COMMIT. Since the goal is to ensure that either both or neither of Bob and Charles COMMIT, Bob must be willing to COMMIT once he sends PREPARED. Although at that point Alice can still decide to abort, Bob cannot abort unilaterally. Thus **D** and **E** cannot be correct.

10. Answers to the Exercises of Chapter 10

Ex. 10.1a Since all three copies were being written at the same time, the power failure has caused all three copies to contain a muddle of old data partially overwritten by an unknown amount of new data.

Ex. 10.1b Write the copies sequentially, rather than concurrently, so that a power failure can affect at most one copy. We still have to consider the possibility that the power fails while writing the sector on D3, and then the next time someone writes that same sector the power fails again but while writing D2.

Write procedure:

- Do the check-and-recover-if-necessary procedure on the sector you plan to write.
- write the sector on D_1, then on D_2, then on D_3.

Check-and-recover-if-necessary procedure:

- Read the sector from D_1 and D_2 and D_3 and compare the copies.
- If all three copies are different, copy the sector from D_1 to D_2 and D_3 (completion; D_1 was new, D_2 garbled, and D_3 old.)
- If the first two copies are the same, copy that sector to D_3 (completion; D_3 was garbled or old.)
- If the last two copies are the same, copy that value to D_1 (back out; D_1 was garbled or new; we can't tell which, so we have to discard it.)

Read procedure:

- Do the check-and-recover-if-necessary procedure on the sector you plan to read.
- Return the now-unanimous value.

The purpose of running check-and-recover-if-necessary before writing is to establish the precondition that there is unanimous agreement on the old value. This precondition is necessary to assure all-or-nothing atomicity if there should happen to be a failure in the write that is about to happen.

This procedure may do the wrong thing if, in addition to a power failure that corrupts one sector while writing, decay strikes the same sector on a different disk.

Ex. 10.1c

```
for each sector N do
    read sector N from D1 and D2 and D3
    if all three copies are different then report "untolerated error"
    if any two are the same then copy the agreed-on value over the odd one
```

The detected but untolerated error occurs if we are unlucky enough that the same block *N* is corrupted on two or more different drives. Another (undetectable) untolerated error occurs if we are unlucky enough that the same block *N* is corrupted in exactly the same way on two or three different drives. It is interesting that the response to the "all three copies different" case is not the same as in the answer to exercise *Ex. 10.1b*; the reconstruction procedure here is designed to tolerate a different set of expected events.

Ex. 10.2 A is correct. If conflicts were common, reconciliation would be labor-intensive, and users would quickly tire of using it. RECONCILE detects conflicts and alerts the user to resolve them (so answer **B** is also correct), but it does not try to automatically resolve them, so answer **C** is wrong. Answer **D** is just wrong.

Ex. 10.3 **D**. While Mary is absent, the only operations that the mail system performs to her inbox and outbox are to add and delete files, and similarly, while she is traveling the only operations she performs on mail are to add new messages to the outbox and delete messages from the inbox. The mailer must somehow keep track of which messages she has read and replied to, but the file that records that information when she is traveling doesn't need to be modified by the desktop mail system, so it will never create a conflict. Thus RECONCILE will happily and automatically reconcile both the inbox and outbox, but because it reconciles deletions by restoring the deleted file, Mary may not be any happier with the result. She needs a more powerful reconciliation program, one that maintains a list of the files in each directory at the time of the previous reconciliation.

Ex. 10.4

A. No, for two reasons. First, RECONCILE reports a conflict only when it detects separate modifications of the two file copies. Second, it detects modifications by looking at file timestamps, not by looking at file contents.

B. Yes. The file named X that you created and then deleted in file set 1 will not appear in any of RECONCILE's lists, so the file named X that you created in file set 2 will appear to be a new file that RECONCILE should copy back to file set 1.

C. Yes. Every file that appears in both file sets will appear to have been modified on both sides since the last time that RECONCILE ran, so every file will be flagged as a conflict. In addition, every file that appears in only one of the file sets will be copied to the other file set, thus undoing all file deletions.

Ex. 10.5

A. RECONCILE uses the language feature "**copy** x **from** a **to** b" for all file operations, so the answer depends on what guarantees that language feature provides. If **copy** has the all-or-nothing atomicity property, then RECONCILE will maintain the invariant.

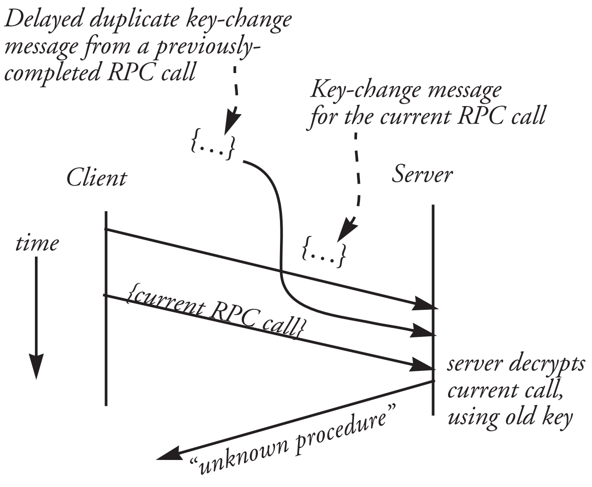
B. If the **copy** language feature does not have the all-or-nothing atomicity property, then RECONCILE does not maintain the invariant.

C. No. It uses file modification time, not file creation time.

D. No. It could send just a list of file names and modification times.

11. Answers to the Exercises of Chapter 11

Ex. 11.1 The problem, illustrated in the figure on the right, is that the service is trying to decrypt a procedure call using the wrong key—the client and the service have gotten out of sync because a duplicate key-change message has been delayed. If the service doesn't know what to expect for a nonce, and the arguments depend on what procedure is being called, the only clue that the decryption did not go well is that the procedure name doesn't appear in the list of known procedures.


Delayed duplicate key-change message from a previously-completed RPC call
Key-change message for the current RPC call
{…}
Client
Server
time
{…}
{current RPC call}
server decrypts current call, using old key
"unknown procedure"

There are at least two ways to fix this protocol. Both involve tying the key to the corresponding procedure call more tightly.

1. Include an authentication key for this call in the key-change message, and have the service keep a table of recently-used encryption and authentication keys. Then, when a procedure call arrives, decrypt it with the newest key in that table and verify it with the corresponding authentication key to confirm that the decryption worked. If it didn't, go through the table trying earlier keys to try to find a set that works. If none work, discard the packet.

2. Combine and send the two messages from client to service in a single packet:

```
client ⇒ service: {ENCRYPT ({K_tn}, K_cs),
                          ENCRYPT ({nonce, procedure, arguments}, K_tn)}
```

This second method is a little flaky: an active attacker can play havoc with this scheme (How?). But Louis says that he is worried only about eavesdroppers, and this method does meet that requirement.

Ex. 11.2a Since there are 26 lower-case letters, there are 26^8 passwords. It will take

$$T_1 = \frac{26^8}{2 \cdot 600 \cdot 10^6} = \frac{(2.6)^8}{12} = 174 \quad \text{seconds, about three minutes.}$$

Ex. 11.2b This time it will take $T2 = \frac{78^8}{2 \cdot 600 \cdot 10^6} = \frac{(7.8)^8}{12} = 1.14 \cdot 10^6$ seconds, about 13 days.

Ex. 11.2c The ratio of the numbers of passwords is $78^8/26^8 = 3^8$. Thus eight triplings of performance are required, and that will take 16 years. (A legalistic reading of the question yields the answer, “Never. No matter how fast the processor gets, it will always take 3^8 times as long to crack the new password as it takes to crack the old one.”)

Ex. 11.3 The main problem with Tracy’s idea is that changing a user’s password requires either changing the user’s name, or changing the password key, which means that everyone else’s password also changes. A less-significant problem is expecting the user to remember and type in an imposed, meaningless password (though this problem might be addressed by recording the name and password on a magnetic stripe card.) Another nagging problem is that the entire security of the system depends on keeping one key secret. Finally, a group of users might get together and, with the help of the pool of names and corresponding passwords, break the cryptographic transformation and deduce the password key.

Ex. 11.4a No. Assuming the cryptographic algorithm itself is reasonably good, all that a passive intruder will see is packets encrypted with unknown, never-before-used keys.

Ex. 11.4b Yes. Lucifer need merely trap the first message, ENCRYPT_C (M, K_a), and send it back to Alice. According to the protocol she will decrypt it and send the result to Bob. Lucifer can also intercept that message, which is the plaintext version of key M. If Lucifer wants to ensure that Alice doesn’t become suspicious, he can encrypt the first packet in a key he knows before sending it back to Alice. He can similarly mimic the protocol with Bob to ensure that Bob ends up with a copy of M and is none the wiser about Lucifer’s intervention. The problem with this protocol is that it provides no authentication whatever.

Ex. 11.5a

- 100-digit number ==> 300 workstations
- Work doubles with every three additional digits.
- 155-digit number leads to 55/3 doublings so we need $300 * 2^{55/3}$ workstations

This is about 99 million workstations.

Ex. 11.5b

- Performance doubles each year.
- 2001 – 1993 = 8 years, which implies 8 doublings
- Dividing the answer from part (a) by 2^8, we now need
- $300 \times 2^{40/3}$ workstations

This is about 400,000 workstations, each of which is running about 750 MIPS.

Ex. 11.5c

Time per unit data = time to decrypt + time to send
= *1/250 seconds per kilobyte + 1/800 seconds per kilobyte*

= 21/4000 seconds per kilobyte

The new throughput is the inverse of this number, i.e., data per unit time:

4000/21 kilobytes per second = 190.4 kilobytes per second

We were asked for the *reduction* in throughput, which is

Old throughput - new throughput =
800 kilobytes per second – 190.4 kilobytes per second = 609.6 kilobytes per second

The new system has only (800-609.6)/800 = 24% of the throughput of the old system.

Ex. 11.5d It is better to compress a file first, then encrypt it. There are several reasons for this. Among them are the following, ordered strongest to weakest.

- Effective compression depends on eliminating redundancy (e.g., patterns of character frequency) in the file being compressed. However, effective encryption eliminates patterns in order to disguise content. Therefore, if the file were encrypted first, little or no compression would be achieved.
- Compression reduces the size of the file, whereas encryption usually does not. Assuming both operations take time proportional to the length of the source being operated on, it will be faster to compress the file to create a smaller source for the encryption operation.
- If the file is compressed before it is encrypted, it may be harder for an attacker who is guessing keys to verify that he has correctly decrypted the file, since the result of decryption is probably not immediately intelligible. Verifying requires understanding and possibly even running the decompression algorithm, which would increase the work factor for cracking the cryptography.

Ex. 11.6a

A. Yes. The one-time pad encrypts each bit of the entire message independently, so there is no correlation between the encryption of the data bits and the header bits.

B. Yes. Assuming that the key really is random, and all she has available is the ciphertext *c*, there is no way for Eve to determine the value of any of the plaintext bits of message *m*.

C. No. The scheme provides no authentication at all. An active attacker could change bits in ciphertext *c* to c_{modified} and there would be no systematic alert to Bob that the decrypted message m_{modified} was not the original message *m* that Alice sent. Bob might notice that m_{modified} says something unexpected, but that would be the only clue.

Ex. 11.6b

A. Yes, since $c3 = r2 \oplus m1$ and Bob knows *r2*.

B. Yes. Lucifer can use the same technique as Eve (see next answer).

C. Yes. Eve can discover *r2* by intercepting *c1* and *c2* and calculating $c2 \oplus c1$. Knowing *r2,* upon intercepting *c3* she can then discover *m1* by calculating $c3 \oplus r2$.

Ex. 11.7 Choice A allows the bank to establish the following, two-step line of reasoning:

1. Jim **speaks for** Ben [given]

   Louis **speaks for** Jim [choice A]

Reasoning rule #2 (chaining of delegation) says that

(A **speaks for** B) and (B **speaks for** C) implies (A **speaks for** C)

Applying this rule tells the bank that

(Louis **speaks for** Ben)

2. Louis **says** (Ben **says** (Transfer $1,000,000 to Alyssa)) [given]

   Louis **speaks for** Ben [from step 1]

Reasoning rule #1 (delegation of authority) says that

(A **says** (B **says** X)) and (A **speaks for** B) implies (B **says** X)

Thus if step 1 is sound, the bank can be sure that

Ben **says** (Transfer $1,000,000 to Alyssa)

which is exactly the authorization the bank needs to perform the transfer. Thus the bank should attempt to establish the truth of choice A.

Ex. 11.8 *Performance*: Every use of a timed capability will require comparing its expiration field with the current clock. In some situations this cost might be negligible, but in others (such as a processor that has capabilities as part of its addressing architecture) this looks like a performance burden because it involves moving a lot of bits around on every reference.

Propagation: Assuming that copying the capability includes copying its expiration field, the expiration feature helps reduce the effect of propagation, since inappropriately propagated capabilities will eventually expire rather than come back to haunt the owner.

Revocation: Timed capabilities reduce the need for revocation, since an unrevoked capability will expire.

Auditing: Asking the question "who can access this object" becomes much easier; after one delays E seconds, the answer is limited to holders of untimed capabilities and those who have received new capabilities since the question was asked.

Ease of use: This could be problematic, since the expiration period is arbitrary, and an capability could expire in the middle of some useful operation; at that instant there may not be any convenient way to get a new, valid copy.

Ex. 11.9a The basic problem is that the protocol sends several related pieces of information in separate messages with nothing to prove that the messages have any connection with one another. As a result, an active intruder can record one instance of the protocol, and then when the protocol runs again, capture one of the messages and insert a replacement for it from the earlier instance of the protocol.

Scenario A: To increase or decrease the amount of a transfer, simply replace message three of one protocol instance with a message three from a recording of another transaction. (To cause a specific amount to be transferred, the attacker could perform a transaction for that amount on his or her own account, and make a recording of that transaction.) Scenario B: To cause a transfer to occur more than once, simply send a set of three recorded messages and capture the "OK" response message. Scenario C: To make a transaction undetectably disappear, capture and discard all three messages coming from the bank and send back a copy of a previously recorded "OK" message.

Ex. 11.9b Change the protocol so that a single message from bank 1 to bank 2 contains all of the information needed to complete a transaction, and includes a nonce to prevent replays. The response message should *be explicit*: confirm all the information in the transaction, including the original nonce. Both messages should be signed and encrypted; in addition, the two banks should use different keys for messages going in different directions, so if an active attacker tries to reflect a message back to the originating bank the intrusion can be detected.

Bank 1 ⇒ Bank 2: $\{\{$"Do Transfer Z from Y to X", $nonce\}_{K2}\}^{K1}$
Bank 2 ⇒ Bank 1: $\{\{$"Did Transfer Z from Y to X", $nonce\}_{K3}\}^{K4}$

Ex. 11.10a This is a broadcast system, so all clients must receive the same shared-secret key in their start messages. But once one client knows the shared-secret key, that client can create bogus messages that appear to be authentic and send them to another client.

Ex. 11.10b The idea is interesting, and it almost works, but there is no assurance that the client will actually receive D_i before the service sends K_i. The scheme depends on the end-to-end network transit time always being less than the time between sending of message i and message $i + 2$.

To fix this problem, the service could include a timestamp in each message, being careful to ensure that the timestamp is protected by that message's authentication tag. If (and this is a big if) the client's clock is known to be synchronized with the service's clock to within d seconds, then the client can accept as authentic any message whose authentication key is timestamped by the sender at least d seconds after the message was received as measured by the clock at the receiver. That is, a message is accepted as authentic if and only if

- $T_{received,i} + d < T_{i+2}$ and
- VERIFY (P_i, K_i) = ACCEPT

Ex. 11.11 **A** and **B**. **A** and **B** are fundamental properties of capabilities and access control lists, respectively. **C** is backwards; revocation of particular permissions is easier in an access control list system, because there is exactly one place to look for the information--in the access control list. In a capability system it may be necessary to search all data accessible to the principal to locate the capability. **D** is wrong; the permission bits associated with a UNIX file are a form of access control list.

Ex. 11.12 **C**. The system checked for permission when Bob issued his query, but did not check again later when it actually performing the notification he requested. In between these two times, the access control list changed, resulting in a classic case of the time-of-check to time-of-use bug. Answers **A**, **D**, and **E** all describe weakness that would require an attack to exploit, but the question indicates that Bob received the call-back in the ordinary course of business, so we can exclude them. Similarly, we can exclude answer **B** on the basis that the question specifically asks about a blunder made by the system designer. Answer **B** would be a blunder made by Alice.

Ex. 11.13a **C**. From the graph, when the worm has saturated the vulnerable hosts those hosts are sending nearly 10,000 probes per second to Ben's part of the network. Ben's network receives only 1/256 of all Internet traffic, so the Internet as a whole is receiving around 2,560,000 probes per second. A single infected host sends 100 probes per second, so the number of infected hosts must be around 25,600.

Ex. 11.13b **C**. The graph starts with a jump to 100 probes per second, which indicates that at the beginning the worm used a hit list. For Ben's monitor to see 100 packets per second while observing 1/256 of the address space, the worm must have started with a hit list of about 256 vulnerable machines.

Ex. 11.14a **B**. There are 2^{32} possible IP addresses, and 2^{15} vulnerable hosts, hence roughly one in 2^{17} probes will be directed at a vulnerable host. Since one worm instance can generate 2^9 probes per second it will take 2^8 seconds to generate 2^{17} probes. Thus, near the beginning of an attack (before saturation effects start to take hold) each worm will find a vulnerable target and double the number of active worms in about $2^8 = 256$ seconds.

Ex. 11.14b **C**. The goal is that the threshold be crossed at least 15 minutes before the outbreak reaches half of of the 2^{15} vulnerable hosts. From the answer to part *11.14a*, 15 minutes is roughly 4 doublings. Thus the threshold should be set for the worm traffic level generated by the infection of 1/16th of half the hosts, or about 1000 hosts. That many hosts can generate 512,000 probes per second, of which Ben's traffic monitor would see 1/256th, or 2000 probes per second. So the threshold must be set to a value smaller than 2000.

Ex. 11.15 All of the methods work, because the authenticity of a valid certificate is independent of the source from which it was obtained. No spammer (or other unsavory party) could forge the trusted certificate authority's signature.

Ex. 11.16a Louis is almost correct. Neither Message 1 nor Message 2 contains any secrets, so passive eavesdropping won't reveal anything. Eve may, however learn something from traffic analysis: that Alice has asked for Bob's public key. If Alice had hoped to keep that fact secret, this was not a very good protocol to use.

Ex. 11.16b Lucifer can actively attack the protocol in several ways:

1) He can intercept and replace Message 1 with, "What is Lucifer's public key?" which will cause Message 3 to be encrypted with Lucifer's public key. He will then be able to decrypt Message 3.

2) He can intercept and replace Message 2 with Lucifer's public key instead of Bob's public key, which will cause Message 3 to be encrypted with Lucifer's public key. He will then be able to decrypt Message 3.

Ex. 11.16c The protocol lacks two things:

- Authentication: Alice needs evidence that the message she is receiving is from PKPI and not from Lucifer. The PKPI should sign Message 2, for example with the private key of PKPI, K_{Spkpi}. Anyone can verify this signature but only PKPI could have constructed it.
- Connection with Bob's key: Alice needs evidence that the key she is receiving in Message 2 is Bob's, not Lucifer's. The PKPI should include the name "Bob" inside the now-signed Message 2: $\{\text{Bob}, K_{\text{Pbob}}\}_{K_{\text{Spkpi}}}$

Note that if public keys rarely change, replays aren't of much concern—even if Lucifer intercepts and replaces Message 2 with a month-old copy of Message 2, as long as that stale message was signed by PKPI and contains Bob's name it is probably OK.

Ex. 11.17 So many problems. Where to start?

- If Norris's service goes down, the entire universe (well, at least the universe of nonce users) could come to a screeching halt. His service represents a single point of failure. Replicating it would be a good idea, but it will be a challenge to ensure that the replicas never give out a replicated nonce.
- Even without replication, when the service comes back up following a crash, it is essential that it remember the last nonce it gave out. It is hard to do this and also maintain enough performance to handle the huge stream of requests that the service is expecting.

- Many uses of nonces are local in scope, so they only need to be unique within that scope. Most people don't need Norris's service. (So maybe performance isn't such an issue after all.)
- There are many preexisting uses of nonces, most of which didn't expect them to be 200 bits long, so those users won't be able to benefit.
- Norris didn't say anything about authenticating the nonce service. Without authentication, anyone who depends on nonces being unique can be easily tricked. The nonces need to be signed.

Solutions to problem sets

1. Bigger Files

Q 1.1 Method **A** doesn't work. The maximum file size is bounded to approximately one gigabyte by the number of block numbers in an inode and its indirect blocks. The size field already allowed for larger (about four gigabyte) files, so increasing the size of the size field alone won't help.

Method **B** does work. If the block size is *b*, the one triple-indirect block can by itself refer to up to $b*(b/4)^3$ bytes, which for 2048-byte blocks would be 256 gigabytes. However, before reaching that number another ceiling interferes: the inode contains the file size and it is stored in a 32-bit field. So the maximum file size would be 2^{32} bytes, or about four gigabytes.

Method **C** doesn't help. Increasing the number of inodes increases the number of files the file system can contain, but does not increase the maximum size of a single file.

Method **D** helps. We give up one block for the triple indirect block, but the triple indirect block allows referring to 2^{21} blocks, which with 512-byte blocks provides addressability of 1,073, 741,824 bytes. Added to the original addressability, that will almost double the maximum file size.

Q 1.2 Method **A** is counterproductive—it will reduce the maximum file size by about half, because indirect blocks will be reduced in capacity from 128 block numbers to 102 block numbers. The triple indirect block, which provided most of the addressability, can now refer to only 102^3 blocks. With 512-byte blocks that provides addressability to just 543,338,496 bytes, about half a gigabyte.

Method **B** works because it increases the number of block numbers that each indirect block can contain from 128 to 170. The one triple indirect block can now refer to 170^3 = 4,913,000 blocks. The double indirect block adds 170^2 = 28,900; the single indirect block adds 170 blocks. The total number of block numbers is now 4,942,080, which allows addressability of 2,530,344,960 bytes. The 3-byte (24-bit) block numbers do not impose a limit: 24 bits can identify 16,777,216 different blocks.

Method **C** goes too far. Although it increases the number of block numbers that an indirect block can contain to 256, the total number of blocks is now limited by the size of the 2-byte field that contains the block number. The biggest block number is now

65,535. That many blocks allow addressability to only 33,553,920 bytes, substantially fewer than the original limit of about one gigabyte.

2. Ben's Stickr

Q 2.1 **A**. False. *At-least-once* will keep trying until it succeeds, forever, if necessary. When it does succeed, it will always return triples if they exist in the system.

B. True, but with a qualification. If the RPC returns OK, it is certain that the given triple is now present in the system. However, if the triple was already there, INSERT could return OK without inserting the triple.

C. True. Suppose the first DELETE succeeds, tries to return TRUE but the network loses the response message. When the *at-least-once* RPC retries the DELETE the triple will already be deleted and the DELETE will now return FALSE.

D. False. Deleted triples will not be in the system and therefore cannot be reported.

Q 2.2 **A**. False. At most once tries exactly once. If the RPC does not time out, it must have succeeded on the first try. Thus, if matching triples were present, they must be reported.

B. Using the same reasoning as in part *A*, the RPC must have succeeded and it is now certain that the triple is present. However, as before, if the triple was already there, INSERT could return OK without inserting the triple.

C. False. If it does not time out, the RPC must have succeeded on the first attempt. The response from the DELETE could not have been lost, and will return FALSE if and only if the triple was not present.

D. False. Again, the RPC must have succeeded, and the FIND cannot return a triple that had been deleted.

3. Jill's File System for Dummies

Q 3.1 Answer **A** cannot be correct because UNUSED has its own special value. The correct answer is **B**. The empty string indicates an open file on the client host machine. The library saves "" in the *handle_to_host* table if the path name does not begin with "/fsd/" which means it is not a remote FSD file. The empty string has nothing to do with either an end-of-file condition or an error condition, so **C** and **D** are both incorrect.

Q 3.2 **A** will work: The FSD client library will use the local file descriptor as the FSD file handle.

B will work: The FSD client library will use the remote file descriptor as the FSD file handle.

C will work: The FSD client library will use the local file descriptors as the FSD file handles.

D will work: The FSD client library will use the remote file descriptors as the FSD file handles.

E will fail: The FSD client library will use the remote file descriptors as the FSD file handles. Problems can occur because the different FSD servers may return identical file descriptors. If two servers return the same file descriptor for different files, the client library will incorrectly assume the file handle maps to the remote host corresponding to the second OPEN.

F will fail: Local file descriptors may conflict with the remote file descriptors and cause buggy file handle aliasing in the FSD client library.

Q 3.3

inode numbers? NFS: Yes, inode numbers help NFS server remain stateless. FSD: No, FSD handles include file descriptors, not inode numbers.

idempotence? NFS: Yes, the RPCs message content alone is sufficient for the server to service the call. FSD: No, the FSD server uses file descriptors as state. If the server crashes, it loses state like file descriptor offsets which are not included in the FSD RPCs.

read after delete? NFS: No, an "open" NFS file does not prevent the server's file system from deleting the file. FSD: Yes, since the FSD server actually opens the file on the server, the kernel increases the file's reference count, so the file system will keep the file around while the remote client has the file open.

mount remote file system? NFS: Yes, the NFS client needs the remote file system's root directory inode number so that it can bootstrap LOOKUP RPCs. FSD: No, FSD does not need to mount remote FSD file systems. The client library just intercepts OPENs and redirects them to the FSD server; FSD does not need to do any bootstrapping.

Q 3.4 The underlying problem is that with *at-least-once* semantics, a lost RPC response on a READ causes the RPC stub to time out and resend the READ request. The server doesn't realize that the request is a duplicate, so it moves the read cursor ahead and responds with the next batch of data, rather than resending the lost batch of data. So Jill needs to do something different when a duplicate request arrives at the server.

Client caching will not do anything about duplicate requests arriving at the server. Adding a cache to detect duplicate requests on the server will eliminate duplicates but changes the semantics. Consider an application that reads a file by repeatedly issuing READ (*host*, "read", *handle*, 4096): the cache will respond with the same answer every time, fooling the client into thinking that the file is just the same 4096 bytes repeated over and over. Adding a sequence number to each request changes nothing by itself, however it opens the possibility for a cache to distinguish between duplicate and similar requests. Thus, **E** is the correct answer: sequence numbers help the server distinguish repeat requests and the response cache allows the server to send a repeat response without the cursor-moving effect of repeating the READ system call.

4. EZ-Park

Q 4.1 **A**. Yes. If two FIND_SPOT () threads run concurrently they may both end their search of the *available* array at the same index *i* before either one has had a chance to go on to change *available*[*i*] to FALSE. Since both threads would then end up with the same value for *i,* two cars would be allocated the same parking spot, which violates correctness specification *A*.

B. No. Although the two threads may concurrently read *available*[*i*] using the same value of *i*, RELINQUISH_SPOT always forces *available*[*i*] to TRUE. No matter when that change occurs, it cannot cause FIND_SPOT to allocate the same space twice, so specification *A* cannot be violated.

C. Yes. Two threads running either of the programs concurrently may read the same value for *numcars*, modify that value, and write the new value back, in which case the second one to write back will overwrite the first one. The first update will be lost, leaving NUMCARS with a wrong value, which violates correctness specification *B*.

D. No. Even though the *average* time between requests may be larger than the *average* processing delay, particular requests may still arrive closely enough in time to encounter one of the race conditions described above.

Q 4.2 Yes. Because every access to the *available*[] array and the *numcars* variable is protected by the calls to ACQUIRE (*avail_lock*) and RELEASE (*avail_lock*) there are no race conditions.

Q 4.3 No. Ben's change introduces a serious bug because certain ACQUIRE () calls don't have a matching RELEASE (). Suppose a thread acquires the lock near the beginning of FIND_SPOT () but the parking lot is full and *available*[*i*] is FALSE for every *i*. Before the next search of the parking lot the program tries to again acquire the lock it already holds, and it will deadlock on itself. The result is that the car won't be able to get a spot even though one may have become available, violating specification C.

Q 4.4 Yes! Ben has now arranged so that every ACQUIRE has a matching RELEASE.

Q 4.5 No. Ben has arranged that *available*[*i*] and *numcars* are completely protected by *avail_lock*, so there are no race conditions. In addition, every ACQUIRE has a matching RELEASE, so there is no danger of self-deadlock. But because *avail_lock* is held from the time a car arrives until the time it leaves, there can be at most one car in the parking lot at any one time! This limitation violates correctness specification C.

Q 4.6 *A*. Yes. If there are no spots, the server searches over and over, not returning a response until it finds a spot.

B. Yes. If other cars keep entering the system, they may systematically get the relinquished spots ahead of this client, who may never get a spot. This phenomenon is known as starvation, and it will come up again in Chapter 6.

Q 4.7 **A**. Yes. The server creates a new thread every time a client makes a request. If a client times out and retries, the old server thread continues to run so there are now two server threads running on behalf of the same client car.

B and **D**. Yes. Because a car can have multiple threads running at the server, it may be allocated multiple parking spots. In addition, the server will increment *numcars* for each allocation even though the car can be parked only in one of those spots!

C. Yes. Since Ben built the timers into the RPC system, a call to RELINQUISH_SPOT may also time out and retry, in which case a single car leaving may cause *numcars* to be decremented more than once.

Q 4.8 **A**. No. Since Ben designed the system so that all the threads run in the same address space there is no need to save the PMAR. But each thread has its own stack, PC, and current register contents, so they do have to be saved.

B. Yes, using the same reasoning.

C. No. Even though there is no return value, each thread still has its own stack, so SP must be saved. And RELINQUISH_SPOT uses registers, so they must be saved, too.

D. No. If saving a thread's stack were done by copying it, then this answer might be true, but that isn't how it is done: all that is needed to save the stack is to save the SP, which is always the same size.

5. Goomble

Q 5.1 No. Since there is only one thread, there can be no race conditions.

Q 5.2 Yes. She should notice that the variable *account.Balance* is shared by multiple threads, which both read and write it, and it is not protected by a lock. Worse, LUCKY reads the variable, and then, depending on its value, reads it again to change its value. This way of using an unprotected shared variable should raise a red flag—there is clearly potential for a race condition.

Q 5.3 Since *Balance* is a 32-bit unsigned integer, if the system does not trigger exceptions on arithmetic underflows the following 4 steps could lead to this result:

1. T1 evaluates "**if** *account.Balance* > 0", finds statement is true
2. T2 evaluates "**if** *account.Balance* > 0", finds statement is true
3. T2 decrements *account.Balance* by 1, and the balance is now 0
4. T1 decrements *account.Balance* by 1. The arithmetic of unsigned integers in a fixed-width field wraps around and the value is now the largest possible positive number, $2^{32} - 1$ = 4,294,967,295. Willie's account has been replenished beyond his wildest dreams.

Q 5.4 Proposal 2 still has a race. Proposals 1 and 3 eliminate the race by using a lock in LUCKY, the only program that touches *account.Balance,* to protect that variable from before it is tested until after it is decremented. If there are several threads concurrently running LUCKY, the region of code surrounded by ACQUIRE and RELEASE exhibits before-or-after atomicity.

While proposal 2 surrounds both the read of *account.Balance* and its subsequent decrement by an ACQUIRE/RELEASE pair, it releases the lock between the read and the subsequent write. A concurrent thread working on this same account may decrement *account.Balance* during the time the lock is not being held, thus invalidating the test that *account.Balance* be greater than zero when decremented.

Q 5.5 *Proposal 3*. We can exclude proposal 2 on correctness grounds, so proposals 1 and 3 are the only plausible choices. The use of a single global lock in proposal 1 serializes all executions of LUCKY, lowering performance to something approaching that of the single-threaded server. Proposal 3 avoids this serialization by using a separate lock for each account, so it would have better performance.

6. Course Swap

Q 6.1 Only **B** is possible. The server is single threaded and there is no possibility of races, hence the server will process the two requests one at a time, in the order in which they arrive. The result will be two swaps and the state of the server is back where it started.

Q 6.2 **A** is possible. If each server starts processing its request at about the same time, so that their SET-AND-GET messages cross, then neither will be able to respond to the other's set-and-get. Both of them will be waiting for a response in CROSSEXCHANGE (), not waiting for a request in SERVER (). The result will be deadlock, and thus no changes to the lecturer assignments.

B is possible. If the servers finish one request before they start the other, then both swaps will happen as requested.

C is not possible. If one swap completes, there is nothing to stop the other swap from also completing, so it should take place, too.

Q 6.3 **A**. No. This is the starting state, which implies either that no exchanges happened or that the requested exchanges cancelled each other out. Since there is nothing that would act to prevent exchanges, and the two requested exchanges are for disjoint lecturers that can't cancel each other, this state should not be found.

B. Yes. Multiple threads eliminate the deadlock of question Q 6.2 state A. One thread can wait for the other server's reply to SET-AND-GET while a second thread is processing the other server's SET-AND-GET request, so both exchanges will finish.

C. No. Version 3 of the code is susceptible to race conditions, owing to its use of threads and concurrent modification of shared variables. But since the particular requests in this question involve disjoint variables, no race condition can occur.

Q 6.4 Each EXCHANGE or CROSSEXCHANGE operation acquires two locks, and the order in which they acquire the locks depends on the order of the client's arguments. If two operations occur at about the same time, each of the operations may start by acquiring a lock that the other operation needs. When the operations then try to acquire their second locks they will each wait for the other, producing a deadlock.

Q 6.5 **A**. No. Both clients have supplied their arguments in the same order, so the server threads will attempt to acquire locks in the same order.

B. Yes. Each EXCHANGE could acquire one of the two needed locks, and then both will wait forever waiting for the other lock.

C. No. Because the four lecturer name arguments are disjoint, and there is a separate lock for each lecturer name, none of the ACQUIRES will encounter a wait, so there can't be a deadlock.

D. Yes. Each crossexchange could acquire its local lock at about the same time, thus blocking the other's SET-AND-GET from proceeding. Neither can continue (and release its lock) until the other finishes.

E. No. While one of these operations may need to wait for the other to release a lock, there is no cyclic dependency, and thus no deadlock.

7. Banking on Local Remote Procedure Call

Q 7.1 Even though the call to YIELD may start out in user (that is, client or server) address space, by the time it gets to the code that schedules the processor it must be in the kernel address space with the user-mode bit OFF. That is how that the kernel implementation enforces thread modularity.

Q 7.2 For Ben's LRPC to work correctly, both the server and the client must be able to share the physical address *transfer*. Since the server and the client each have their own independent page maps, the virtual addresses *shared_client* and *shared_server* can be different in each address space, yet by proper setting of the two page maps both virtual addresses can map to the same physical address. There is no constraint that might force a virtual memory with page maps to map different virtual addresses to different physical addresses.

Q 7.3 Two threads. Let's count them: the server's initial thread is blocked in a call to YIELD in the procedure SERVER. The client thread is running in the procedure at the address *receive* in the server address space (it got there via the TRANSFER_TO_GATE call). Since the system does not create any new threads after the initial two, we've accounted for all of them.

Q 7.4 Just one supervisor call. The only supervisor call that LRPC makes is a call to TRANSFER_TO_GATE. To copy the data the client doesn't have to make a supervisor call, because the block at *shared_client* is mapped into its address space.

Q 7.5 **A**. True. First, the client can only end up at the address it registered with the kernel. Second, the client doesn't transfer control on its own: only the kernel can transfer the thread to another address space. **B**. False. The client program is running in the server address space when executing the RPC and can overwrite any memory in that address space. **C**. False. For example, the thread in the server address space might issue an illegal instruction or go into an infinite loop. **D**. True. An advantage of remote procedure call is that it offers the opportunity to enforce modularity by checking responses for validity. If the client doesn't check the validity of responses, a variety of problems can arise including a client crash or even a malicious server taking control of the client program.

Q 7.6 **A**. The kernel does no checking of *addr*. A buggy user program might provide an invalid address. Alternatively, the user program might have passed in a bogus stack pointer trying to cause the kernel to store to an invalid address. The kernel should be more defensive. **B**. If a bogus address was passed to REGISTER_GATE (which then pushed the address on the stack), it will be popped off by the **return** and that **return** is executed in user mode in the service's address space. **C**. The description of the gate system indicates that all PMAR entries have the user-bit switched ON. **D**. This is precisely the goal of gates: to allow limited access to an address space. Ben managed to get this right.

Q 7.7 **A** is false, but **B** and **C** are true. Consider the following scenario for **A**, which is the challenging one: client X is running the procedure at the address *receive,* which invokes a procedure to execute the requested operation. This procedure may call YIELD on behalf of client X. In the meantime, client Y calls LRPC, transfers into the server address space, and calls the procedure at the address *receive*. As long as receive copies its arguments from *shared* to a private copy (as is done in reverse for the response), this scenario works fine. But **B** and **C** are true. The scenarios described in B and C add more concurrency to the system and allow for race conditions: for example, the scenarios allow for concurrent writing of the shared buffer and cause the shared block to become internally inconsistent. **D** is false. Threads are virtual processors, and one processor can support many threads.

8. The Bitdiddler

Q 8.1 **A**. No. If the modified inode was not yet written to the disk, the old inode on the disk would not contain a newly-allocated block. The result would be a leak of a block from the free list but not garbage data in a file.

B. No. This scenario can't happen because a reallocated inode is marked with 0 data blocks before that inode is written to disk.

C. Yes. WRITE writes the inode to the disk before it writes the contents of the data block. Thus a crash between writing the inode and writing the data block will result in the file containing a block with data that has not been updated. That block probably contains residue from a previously deleted file.

D. No. The ordering is explicit: allocate from the free list, update inode, then write data. Since disk writes are synchronous, this scenario cannot occur.

Q 8.2 **A**. Yes. By following pointers from allocated inodes, all in-use blocks can be discovered and checked against the free list.

B. Yes. After discovering in-use blocks as above, all remaining blocks are unused and thus should be on the free list.

C. No. There is no way to determine if the data in a file is correct.

D. Yes. If a block is reachable from multiple inodes, it is used in multiple files.

9. Ben's Kernel

Q 9.1 The value the supervisor needs is in one of the fields of the SVC instruction in the user program. Although the supervisor is running in kernel mode, the user-mode registers are still accessible. The value in the user PC, *upc*, is the address of the next instruction to be executed in user space, so (*upc* – 1) should be the location of the SVC instruction. Thus the code in the supervisor should retrieve the contents of that address and extract the SVC number. However, it should first check to be sure that the instruction stored there really is an SVC; it is possible that the reason the supervisor was called is that the user program has performed an illegal instruction. There also needs to be a way of distinguishing interrupts from SVC's, but the architecture description doesn't mention how that works; somehow the number –1 magically becomes available to be stored in *n*.

Q 9.2 **A**. Yes. When the kernel is in control, the current address space is determined by the *kpmar* register, so writing *kpmar* changes the current address space. **B**. No. When the kernel is in control, the current address space is determined by the *kpmar* register and *upmar* is inactive, so writing it has no effect on the current address space. **C**. Yes. This bit tells the processor which of the *upmar* or *kpmar* registers it should use, so changing it switches the address space from one to the other. **D**. Yes, but not because the register values changed. Attempting to write a value in either of these registers in user mode is illegal, so it causes an exception and a consequent switch from user to kernel mode, thus switching the current address space. **E**. No. DOYIELD, which is in the kernel, saves and restores *upmar*, which does not switch the current (i.e. kernel) address space.

Q 9.3 **A**: No. KERNEL is in the kernel and, further, does not call itself. **B**: No. MAIN calls YIELD and it is YIELD that executes SVC 0. The return from that kernel call will be to the last instructions in YIELD not to MAIN. (Assuming that Ben doesn't have a compiler that inlined YIELD.) **C**: Yes. The SVC 0 returns to YIELD, namely to the instruction after SVC 0 which is probably a **return** instruction. **D**: No. DOYIELD is in the kernel and does not execute SVC 0.

Q 9.4 **A**: Yes. Separate address spaces prevent applications from damaging one another. **B**: Yes. Without a user-mode bit, programs could modify their address spaces and damage other programs. **C**: No. The scheduler, as defined so far, is not preemptive. **D**: No. Small size has something to do with simplicity, but is not related to modularity.

Q 9.5 **A**: No. **B**: No. **C**: Yes. **D**: Yes. **E**: Yes. Except for alternative **C**, the same reasoning as in question *Q 9.3* still applies. The only change is that the clock can now cause an interrupt during the execution of MAIN.

Q 9.6 Again, except for answer **C**, the same reasoning as in question *Q 9.4* applies. **A**: Yes. **B**: Yes. **C**: Yes. Preemption provides modularity by preventing one thread from dominating the processor and starving other threads. **D**: No.

Q 9.7 **A**. Yes. If *t* = 1 and thread 1 gets interrupted after 100, and thread 2 doubles the value of *t*. Then thread 1 will resume at instruction 104 and end up adding t = 1 + 2 =

3. **B**. Yes. If *t* = 2 and thread 1 gets interrupted after instruction 100, and thread 2 doubles the value of *t*, then thread one will set *t* to 2 + 4 = 6. **C**. Yes. This is the behavior if thread 1 is not interrupted. **D**. No. Neither thread will print the value of *t* without first doubling it. Recall that *t* was statically initialized to the value 1.

Q 9.8 C is the only possibility. The code from 100–112 now executes atomically, so a thread always doubles the value of *t*. Since *t* starts out at 1 the program can generate and print only powers of 2.

Q 9.9 **A**. Yes. If the first thread is within the 100–112 region and it gets interrupted, then its PC is set to 100 (which is within the inclusive range 100–112). The second thread can then run and enter the 100–112 region. **B**. No. DOINTERRUPT is executed in kernel mode and in kernel mode interrupts are ignored. DOINTERRUPT is therefore executed atomically. **C**. No. DOYIELD is also executed atomically by the kernel. **D**. No. A thread calls YIELD only outside the region 100–112.

Q 9.10 **A** is possible: If the thread runs without being interrupted the values of *a* and *b* are swapped. **B** is not possible: No matter how many times the code region is restarted eventually *a* is assigned the value of *b*. Therefore *a* is always set to 2 in the end. **C** is possible: If the code is interrupted between 104 and 108, then *a* has been set to 2. Now when the code restarts from instruction 100, *x* gets the new value of *a* (i.e. 2) and in instruction 108, *b* gets the value 2. **D**. Not possible, for the same reason as **B**.

10. A Picokernel-Based Stock Ticker System

Q 10.1 **A**. We are told that the numbers are the addresses issued by the processor, so they must be virtual addresses.

Q 10.2 **A**. Answer **B** couldn't be correct, because we are told that in the simplePC, SP points to the value on the top on the stack and a **return** instruction includes popping SP. **C** and **D** can't be correct, because we are told that the stack contains only return addresses.

Q 10.3 **A**. This is the only possibility because the return point in the stack is the address of the instruction after a call to PRINT_MSG.

Q 10.4 **A**. The procedure PRINT_MESSAGE is doing I/O by writing into a variable in memory. This is the definition of memory-mapped I/O.

Q 10.5 **A**. Based on the value at the top of thread 0's stack, thread 0 is currently in PRINT_MSG, but it will soon call YIELD from location 33. Based on the value in thread 1's stack, it is in a call to YIELD. So when YIELD returns, it will be running thread 1 and it will return to location 34.

Q 10.6 **B**. We were told that the only things that go on the stack are return addresses.

Q 10.7 **A**. **C** is always TRUE, so it would cause an endless loop, and **D** is always FALSE, so the threads would never change. **B** would cause a bug, because YIELD just set the current thread to WAIT, the test for RUNNABLE would fail, and the scheduler would incorrectly redispatch the current thread. **A** creates a spin loop that keeps the processor busy until one of the threads has something to do.

Q 10.8 Both **A** and **B**. **A** is necessary in order for the processor to be redispatched, and **B** is necessary in order for thread 1 to run. **C** doesn't help thread 1.

Q 10.9 Since the interrupt handler uses the stack to keep track of where the program was interrupted, any address in the program may now appear there. However, since the interrupt handler cannot be interrupted, the only addresses inside the handler that can appear on the stack are for return points of the procedures it calls. Inspecting the handler, we see that it calls NOTIFY, so the address of the next instruction (the handler's **return** instruction) can be on the stack. Considering that the handler is a procedure, answer **C** is therefore correct.

Q 10.10 If device 0 interrupts thread 0 after it checks *input_available* but before it gets to the statement in YIELD that changes its state to WAIT, the interrupt handler will run and call NOTIFY, then YIELD will set the thread's state to WAIT, where it will remain forever, because it missed the NOTIFY. So scenario **A** can deadlock. Similarly, if the interrupt occurs between statements 35 and 36 of READ_INPUT, then the interrupt handler will set *input_available* just before READ_INPUT resets it, and READ_INPUT will loop forever waiting for *input_available* to become true. So scenario **C** can also deadlock. As for scenario **B**, once YIELD changes the thread's state to WAIT, interrupts are again safe, so that scenario cannot deadlock. If the interrupt occurs during the running of SCHEDULE_AND_DISPATCH, it will change thread 0 to RUNNABLE; that event will occur either before or after SCHEDULE_AND_DISPATCH inspects that variable, but either way, thread 0 will eventually run. So scenario **D** does not cause a problem.

11. Ben’s Web Service

Q 11.1 The largest useful value for the size of the segment table is equal to the maximum number of different segment numbers that the hardware can handle. That would be 2^{18}.

Q 11.2 The index identifies a byte within a segment, counting from the beginning of the segment, so a segment cannot usefully be larger than the largest index that the hardware allows, which is 2^{16}.

Q 11.3 The number of bits in a physical address determines the maximum number of bytes of physical memory that can be attached, but that is unrelated to the size of virtual addresses. The design specifies only the size of the virtual addresses, so it places no constraint on the size of physical addresses.

Q 11.4 **A**. The program has issued a virtual address containing an index greater than the segment length, so we should signal an illegal address fault. None of the other answers make any sense.

Q 11.5 **B**, **C**, and **D**. The compiler is not involved in computing physical addresses, but it does have to insert the correct virtual addresses in programs it compiles, and **B**, **C**, and **D** are three examples.

Q 11.6 **A** is correct. A STORE instruction could overwrite an address stored in a variable (or in executable program code). The resulting address could point anywhere, including into the library segment. If the program stores data using the corrupted address, it will then overwrite the library segment. **B** is also correct. A LONGJMP could jump into the middle of the library executable code. If it happens to land on a STORE instruction that uses register addressing, and the register happens to contain the segment number of the library segment, the library segment could write into itself. **C** is not correct. Lack of paging can't cause the library segment to be overwritten. **D** is not correct. An endless loop would not itself cause a write into the library segment.

Q 11.7 Extension **A** does not solve any of Ben's problems, since it does not protect the library segment from writes, and it does not prevent incorrect an LONGJMP into the library segment. It might be useful in enforcing some of the other solutions, but by itself it doesn't help. Extension **B** is a reasonable solution for problems A or B, since a write-protect bit can be used to prevent all changes to the library segment. Extension **C** is a reasonable solution for problem B, since it could be used to prevent the execution of inappropriate instructions in the library segment. Extension **D** is irrelevant, and extension **E** doesn't address any of the observed problems.

Q 11.8 The implementation described for events allows a race following a test of step A that comes out TRUE. If, just after step A completes, a preemption allows steps D and E to be executed, the increment by step D will happen too late to be noticed by step A, and the call to NOTIFY of step E (which wakes up only those threads that are already in the queue for the event) will happen before step B has a chance to queue the Web server thread. So the sequence that causes the problem is

… A, D, E, B, …

The Web server is now waiting even though a packet has arrived and needs to be processed. When the next packet arrives, the network manager will execute steps D and E again, and the Web server will wake up to find two queued packets. (One might wonder if the network manager would overwrite the first packet with the second one. Since the problem says that the first packet is sometimes handled late, rather than saying it is lost, overwriting must not be a concern.)

12. A Bounded Buffer with Semaphores

Q 12.1 **A**, **B**, **C**, *and* **D** *are all* true. **A**: Two threads that invoke RECEIVE_MESSAGE concurrently on different ports will access different elements of *port_infos*. They therefore access disjoint regions of the memory, and there are no data races.

B: DOWN holds *threadtable_lock* between lines 12 and 19. The first statement of UP tries to acquire *threadtable_lock*. If one thread invokes UP when another thread is between lines 12 and 19 in DOWN, the thread that invoked UP will wait until after the thread in DOWN releases *threadtable_lock* at line 19.

C: If the instructions for changing *sem* in UP and DOWN were interleaved at all, UP and DOWN would not be atomic.

D: Consider the following execution sequence:

1. INTERRUPT is called enough times to fill up *d.buffer* completely.

2. RECEIVE_MESSAGE executes until just before the statement $d.out \leftarrow d.out + 1$; The thread running RECEIVE_MESSAGE is now preempted and does not run until later.

3. INTERRUPT is called. At this point INTERRUPT will throw the message away, even though it could safely stick the message in the buffer. Also note that d.int and d.out are both long integers. Changes to long integers are not atomic—they may require multiple instructions to appropriately update the upper and lower halves of the long value stored in the integer.

Consider the following sequence:

1. interrupt and receive_message are called (perfectly interleaved, with interrupt called first in each pair) $2^{32} - 1$ times. At this point there are no messages in the buffer and *d.in* = *d.out* = $00000000ffffffff_{hex}$

2. interrupt is called and inserts a message into the buffer. *d.in* = 0000000100000000_{hex}

3. RECEIVE_MESSAGE is called. It takes the message out of the buffer and proceeds on to the statement $d.out \leftarrow d.out+1$. It executes the first half of the update and *d.out* = 0000000000000000_{hex}.

The thread running receive_message is now preempted and does not run until later. Note that the second half of the update will eventually complete the increment of *d.out* and set it to 0000000100000000_{hex}, but this will not occur until the thread runs later.

4. INTERRUPT is called. It computes that $d.in - d.out \geq$ NMSG, and so discards the message even though it could safely stick the message in the buffer.

Q 12.2 **A** doesn't work: If *mutex* is initialized to 0, the call to DOWN (*d.mutex*) will always block and the program will deadlock.

B is the correct answer—the proper initial value is 1. Only 1 thread at a time will proceed past the call DOWN (*d.mutex*), and atomicity is preserved.

C doesn't work: Two threads may proceed concurrently past the call DOWN (*d.mutex*). The statements of these two threads could interleave in such a way as to cause a race condition; for example, both could return the same message.

D doesn't work: The call to DOWN (*d.mutex*) will always block and the program will deadlock.

13. The Single-Chip NC

Q 13.1 An endless wait can occur if thread number one is in WAIT_FOR_MESSAGE, and immediately after it executes line 11, thread number two executes all of MESSAGE_ARRIVED. The trouble is that **after** thread number one has convinced itself that *message_here* is FALSE (in line 11), *message_here* becomes TRUE (line 6) and the condition gets signalled (line 7). Since thread one is still running, it misses the signal; it then waits forever (line 12) for the signal that it missed.

The minimal sequence that leads to an endless wait is **11–6–7–12**.

Q 13.2 To fix the problem we need to make sure that the shared variable *message_here* can't be changed between the time that WAIT_FOR_MESSAGE tests it and WAIT_FOR_MESSAGE reaches its WAIT_CONDITION statement. WAIT_FOR_MESSAGE has done its part—it has acquired lock *m*, and it has released lock *m* atomically as part of going into the wait state. But MESSAGE_ARRIVED isn't respecting the lock protocol.

The minimal repair, then is to replace line 6 with the following three lines

```
6.1    ACQUIRE (m)
6.2    message_here ← TRUE
6.3    RELEASE (m)
```

In terms of the solution to Q 13.1, this change guarantees that line 6 must be executed either before line 11 or after WAIT_CONDITION releases the lock in line 12.

Note that although it wouldn't hurt anything, it is not necessary to continue to hold the lock while calling CONDITION_NOTIFY. The simple explanation is that surrounding line 6 with the same mutual exclusion lock that protects lines 10 through 12 is sufficient to eliminate the bad execution ordering.

An interesting exercise to explore is whether or not placing the lock only around line 7 would also be sufficient.

14. Toastac-25

Q 14.1 Suppose thread 1 gets to the COOK_TOAST step. It is holding the accelerator lock. Thread 2 then enters the server, grabs the message lock, and waits for thread 1 to release the accelerator lock. Thread 1 then goes for the message lock, only to find it locked by thread 2. We have the deadlock described in scenario **A**. Scenario **B** describes a delay, not a deadlock. The atomicity of COOK_TOAST has nothing to do with deadlocking, so answer **C** does not apply, and answer **D** is just handwaving.

Q 14.2 **A**. The other answers are not even remotely plausible.

Q 14.3 All workloads will be sped up some by the asynchronous protocol, but grouping will cause some workloads—the bursty workloads—to be sped up much more.

A workload that is sped up: any workload with average distance between requests < 2 milliseconds will be sped up more on average: every once in a while, 2 requests will be sent in one message, saving one latency (i.e., the second piece of toast will be made 1 latency earlier than it would have otherwise). A specific example: 2 requests every millisecond.

A workload that isn't sped up as much: any workload with distance between requests > 2 milliseconds. There's no grouping, so we still have to pay one latency cost per request. A specific example: 1 request every 3 milliseconds.

Q 14.4 Here are some reasons to agree with Louis's decision.

- On his high-latency network, the performance of toasting clients will be greatly improved, since they don't have to wait for replies. If the request pattern is bursty, the grouping feature will improve performance even more.
- If each slice of bread is essentially identical, who cares if we burn the 4th slice of 97, and 5 more get cooked or burned before the "Malfunction 54" reply reaches us? We'll try again, and eventually, we'll get our 97 slices of toast.

And here are some reasons to disagree.

- Asynchronous systems are generally more complex than synchronous systems (and RPC is synchronous). This increases the probability for bugs in server or client, and makes the system harder to debug.
- We want to know right away if the Toastac has failed, to minimize later toast lossage, and perhaps to prevent it from catching fire! (This assumes that Toastac failures are not transient.) Even if failures are transient, we may want to report each failure to the user before allowing her to make more requests.
- Depending on the protocol used, we may get a failure report and match it up with the wrong toasting request (the brown request instead of the black request, for instance). (Most asynchronous protocols don't let this happen.)

15. BOOZE: Ben's Object-Oriented Zoned Environment

Q 15.1 Procedure OBJECT_TO_ADDRESS sets the value of *addr* when it brings an object in from disk, which excludes answer **D** but makes answer **C** correct. The disk address is in *blocknr*, so **B** is excluded. The code doesn't use the address of the object map, so that excludes **A**.

Q 15.2 PAGE loads up memory with unused objects, while BOOZE brings in only things that are used, so answer **A** applies. The relation of speed to address size isn't clear, but UID's are probably bigger than virtual memory addresses, so **B** can't be right. Since all disk transfers are 4 kilobytes BOOZE transfers full disk blocks, just like PAGE; that excludes answer **C**. **D** is handwaving; the applications will have the same locality of reference in both systems, but BOOZE is taking better advantage of the locality of reference.

Q 15.3 Implementation 1 has a serious bug. It writes the object into the disk block starting at the first byte of the block, even though the place the object belongs may be in the middle of the block. (It also writes unrelated objects that happen to be in memory into the rest of the disk block.) So answer **A** is the only acceptable one.

Q 15.4 Answer **A** won't reduce the number of writes; it may increase the number. Answers **B** and **D** may reduce the number of writes (because an object may be modified twice but only has to be written to the disk once) but there won't be any place in memory to put other needed objects when they are fetched, so it just won't work. **C** works very nicely.

Q 15.5 If all the objects have been touched in the last 30 seconds, then they will all have their *dirty* flags on and FINDFREESPACE, which now accepts only clean objects, will fail. Solution **A** fixes the problem. **B** might reduce the frequency of failure, but it wouldn't eliminate it. Solution **C** would work if FINDFREESPACE repeats it when the evicted object is smaller than the one to be loaded.

16. OutOfMoney.com

Q 16.1 The time required is

```
(seek + average rotation +   transmission)         x   (number of blocks)
(5    + 6/2              +   8,000/10,000,000)     x   131072
      = 1153 seconds, about 20 minutes.
```

Q 16.2 Since each video is used only once, the cache does not help, so the time is unchanged. Each video still takes 20 minutes.

Q 16.3 **B**. The problem is that the cache is too small, so its hit ratio is low. The performance is disk-limited, so a faster CPU (answer **D**) or a smaller cache (answer **A**) wouldn't help.

Q 16.4 Since the cache is large enough to hold an entire video, and no one else is using it, the hit ratio in the cache is 100%.

Q 16.5 Ben is right. Because ADD_TO_CACHE does not call YIELD, the thread package is not preemptive, and there is only one processor, there is no way for two threads to enter ADD_TO_CACHE at the same time. If any of those conditions changed, then there might be a need for a lock.

Q 16.6 Since the hit ratio is 0.9, the miss ratio is 0.1. When the video is in the cache, the time is zero; when it isn't it must be retrieved from disk, which takes the time we calculated in question *16.1*. So the average time is 0.1 of that time, or 115 seconds.

Q 16.7 **E**. The cache is large enough to hold the entire video, so the replacement policy doesn't matter. The stuff about all users starting at the same time and the threads all reading the same blocks is just a distraction.

Q 16.8 **A**, **B**, and **C**. If an interrupt occurs while driver has acquired *bufferlock*, then the procedure named INTERRUPT will make another call to DRIVER which will again attempt to acquire *bufferlock* and loop forever, because *bufferlock* is already set. The same thing can happen if an interrupt occurs while RECEIVE_MESSAGE has acquired *bufferlock*. And since REMOVE_FROM_BUFFER is always called with *bufferlock* set, an interrupt while in that program will again cause DRIVER to hang forever waiting for the lock to be released.

This answer holds whether the implementation of ACQUIRE and RELEASE is by spinning or by waiting. Since the interrupt handler runs as the current thread, an AWAIT in the interrupt handler means that the current thread cannot make any progress until the interrupt handler returns. But in all three cases the current thread already holds *bufferlock*, so the condition the interrupt handler must wait for will never happen; it is in a deadlock.

Answer **D** is bogus; An interrupt between a test that comes out true and the ensuing call to YIELD simply provides an opportunity for the test to come out false next time around the loop, it does not cause the thread to miss anything.

Q 16.9 **B**. If Mark deletes all the code dealing with locks, the deadlock problem will disappear, but it will be replaced by a worse problem: the program will start doing bad things, such as incorrectly updating the (shared) state variable that shows how much space is left (the exact list of bad things depends on how ADD_TO_BUFFER and REMOVE_FROM_BUFFER coordinate their activities). So even though **A** eliminates the symptom, it doesn't seem a good candidate for the best fix. **B** eliminates the deadlock without adding any new problems, so it is by far the best suggestion in the list. **C** sounds tempting, but INTERRUPT and DRIVER don't need to be coordinated. (It is possible that one could come up with a way of using an eventcount to coordinate RECEIVE_MESSAGE and DRIVER, but even that is questionable, because waiting on anything while in an interrupt is likely to be a source of trouble.) Answer **D** wouldn't work. If the buffer is full, driver can't hold onto the packet till there is space in the buffer, because it has possession of the processor in interrupt mode, so RECEIVE_MESSAGE won't ever get a chance to provide the necessary space.

Q 16.10 **A**. The system is a victim of receive livelock. Because the interrupt handler runs immediately, the processor is spending all its time reading and discarding messages and doesn't have any time left to process the ones already in the buffer. Answers **B**, **C**, and **D** are all activities that, if the processor is doing them, contribute to throughput rather than causing it to decline.

17. Quarria

Q 17.1 Connectionless. Each request is completely self-contained and doesn't depend on a connection for its context.

Q 17.2 Blocking would cause the client to hang forever even on transient errors (lost messages); a timer expiration error code would allow the client to provide a recovery procedure.

Q 17.3 Retransmitting would cause the client to hang forever on a permanent failure (e.g., server crash); a timer expiration error code would allow the client to provide a recovery procedure.

Q 17.4 Setting Q above the time bound for a request/response guarantees that no retransmission will occur unless the original request, or its response, was actually lost. (This doesn't eliminate the need for at-most-once server semantics, since the server may still see duplicate requests.)

Q 17.5 In general, retransmission makes message duplication possible. In the absence of a time bound, duplicate responses may be seen by the client.

Q 17.6 **C** and **D**. Since the server is already flooded with client requests, the bottleneck is the server and/or the network.

Q 17.7 The degradation is probably due to thread overhead (e.g., thread creation costs).

Q 17.8 Yes; it allows Trade requests, which may involve long (blocking) steps in their execution, to be interleaved. Server time otherwise wasted on I/O waits can be profitably used on other requests, improving throughput.

Q 17.9 Answer **B** seems to provide the best trade-off because it allows interleaving of long requests and at the same time avoids thread creation overhead on short ones.

Q 17.10 Assuming a single-threaded client, no server changes are necessary since at most one client request can be processed at any time and server variables are not shared among requests from different clients.

Q 17.11 To keep several requests from the same client from interfering with each other, the body of TRADE should be made before-or-after atomic, e.g. via an ACQUIRE/RELEASE pair surrounding the critical 3 lines. Preferably, each client would have a separate lock to minimize contention.

Q 17.12 TRADE requires at-most-once, since it causes a change in server state. BALANCE, being idempotent, does not.

Q 17.13 Add a sequence number, supplied by the client, to the request. The client increments the sequence number on new requests but not on retransmissions, and the server uses cached responses for requests with previously-seen sequence numbers.

18. PigeonExpress!.com I

Q 18.1 To fly 50,000 meters at 50 meters per second takes 500 seconds, and it takes another 500 seconds for the acknowledgment to get back. In 1000 seconds, up to 640 megabytes of data (one CD) can be transmitted and acknowledged. But the protocol inserts a 2000 second delay between pigeons, so the peak data rate is 640 megabytes per 2000 seconds, or 320 kilobytes per second.

Q 18.2 Both data packets and acknowledgments can be lost, but there is a loss-recovery procedure, which contradicts answers **B** and **C**. According to the program, the sender repeatedly sends the same data until an acknowledgment appears and changes the sequence number of the CD it should send next, so answer **A** is correct, rather than answer **D**.

Q 18.3 **D** can't be right, because there are certainly conditions under which the given code sends the same data several times. And **C** isn't sufficient, because an acknowledgment pigeon may also get lost, in which case the algorithm will still send a duplicate. But if acknowledgments always get back before the 2000-second retransmission, then each CD will be delivered exactly once, so **B** applies. Answer **A** seems to have more to do with at-least-once than at-most-once.

Q 18.4 **B**. The protocol doggedly refuses to send the next CD until it has an acknowledgment of the previous one, so it assures that the first copy of each CD arrives in order.

Q 18.5 **A**. We shouldn't send more CD's than there are to send, and we shouldn't send more than one window's worth, so FOO should compute the *minimum* of its arguments.

Q 18.6 **A**. Although traditionally Ben comes up with broken code and Alyssa has all the right answers, you can't depend on this cultural phenomenon.

Q 18.7 Answer **A** is in the right direction, but insufficient, because the receiver also must be able to handle the higher data rate. Answer **C** is in the wrong direction. If *both* sender and receiver can handle the higher data rate, then the pipelining effect of the window begins to pay off, so the answer is **B**.

Q 18.8 **B**. This is clearly a network layer function.

Q 18.9 We need a congestion management system. Since there may be several customers sending pigeons through the same hub, having each customer limit their window size to 100 pigeons won't be sufficient, so we can exclude answer **C**. Answer **A** is promising, but since it forever increases the window size, it will eventually get into trouble. Answer **B** reduces the total number of pigeons flying around, but it doesn't directly do anything about congestion. Answer **D** provides complete congestion management.

19. Monitoring Ants

Q 19.1 About 48 hours. Transmitting continuously, the mote draws 12 milliamperes. It has a battery with 575 milliampere-hours, so 575 mAh/12mA = 47.9 hours.

Q 19.2 1.9 microjoules. At 19.2 kilobits per second, it takes 1/19,200 seconds to transmit one bit. When transmitting, the mote consumes 3 volts x 12 milliamperes = 36 milliwatts. Finally, 36 milliwatts x 1/19,200 seconds = 1.875 microwatt-seconds = 1.875 microjoules.

Q 19.3 **A**. Yes. If all the packets are lost, no routes are set up and E's path is NULL.

B. No. All paths must lead to A, i.e., have A as the last entry.

C. Yes. Scenario: A advertises [A] to B and D, advertisement to B gets lost. D advertises [D,A] to E, E accepts advertised path.

D. Yes. Scenario is as in C, except D and B are swapped.

E. Yes. Scenario: A advertises [A] to B and D, advertisement to B gets lost. D advertises [D,A] to C and E, advertisement to E gets lost. C advertises [C,D,A] to B and D, D rejects path. B advertises [B,C,D,A] to E. E accepts advertised path.

F. No. E cannot receive advertisements from C, so it will never have a path that starts with C.

G. No. D and A occur twice in this path. For this path to occur at E, A must accept the path [B,C,D,A], then advertise the path [A,B,D,A] to D, which accepts the path and advertises the path [D,A,B,C,D,A] to E. But neither A nor D will accept the path, since a condition in RECEIVE rejects an advertised path if the node receiving the message is already in the path.

Q 19.4 **A**. After the routing path reaches a stable state, every mote will have a path to A. But when the system starts, all paths (except A's) are NULL. So it depends when you look. **B**. Yes. Receive always accepts a new path if it is shorter than its current path. Eventually this means that all motes will have a shortest path to A. **C**. No. A forwarding cycle occurs if a packet can visit the same mote twice. This can't occur in the stable state because all routes follow the shortest path to A and the routes don't change. **D**. Yes. The longest shortest paths in the graph are all two hops long, so the longest routing path is two hops long.

Q 19.5 **A**. No. If all packets are lost, no mote has a routing path to A except A.

B. Yes. Remember that a routing cycle occurs when a packet visits the same mote twice. Here is how a routing cycle could occur:

- The system establishes the following path from C to A: [D,E,B,A], and sends a packet from C to D, then from D to E.

- The paths at B,C, and D time out. They are then reestablished so that D has path [A], C has path [D,A], and B has path [C,D,A].
- E still has path [B,A], so it routes the packet to B.
- B routes to C, then D, and the packet has visited D twice.

C. Yes. Routes may always time out and be established differently.

D. Yes. Consider the following situation: C has path [D,A], then sends a packet to A. But the path at D has timed out and not been reestablished, so there is no route to A from D and the packet can't get through to A. There is no guarantee that B will ever reestablish a path to A.

Q 19.6 In a stable state the routing protocol will construct a tree rooted at A. The intermediate nodes in the tree are B and D. C and E will form the second level of this tree. A report from B results in one packet from B to A, and a report from D results in one packet from D to A. A report from C results in two packets (one to B or D, and then one to A). Similarly, a report from E also results in two packets. Thus, the total number of packets is 6 and the energy cost is 6(s + 21j).

Q 19.7 **A**. No. A nonce will make the packets bigger and use up more energy. **B**. Yes. By piggybacking multiple reports on a single report packet, the cost of starting the radio can be amortized over multiple reports. **C**. Yes. There are only 5 nodes in the system, thus one can use 4-bit node IDs, which makes the packets shorter. **D**. No. This change may make Alice's system more reliable, but it won't save energy.

Q 19.8 **A**. No. Reports are not encrypted (they are only signed), so every mote can read a report. **B**. Yes. A mote can duplicate the signed report. This duplicate will verify correctly, because the signature covers only the counter and the source. **C**. No. A signature is verified using a public key. **D**. No. Alice won't be able to determine if the report is fresh, since the report doesn't contain a nonce, and intermediate nodes can duplicate packets.

20. Gnutella: Peer-to-Peer Networking

Q 20.1 **A**. If one does not know the maximum propagation time of the Gnutella network, there is no way to establish an upper bound on how long it might take for all of the QUERYHIT messages to return. If the node ever deletes a QUERY entry from its message table, it will incorrectly discard any later QUERYHIT for that QUERY that come along and the client will never learn that there are additional copies of the file available. Also, the original QUERY might return a second time to this node via a roundabout path. If a QUERY returns after the node has deleted that QUERY's entry in the message table, the node will broadcast it again to all its neighbors, repeating the cycle, and that QUERY message will probably circulate forever. By mutual exclusion, **B**, **C**, and **D** must be incorrect.

Q 20.2 **D**. The counterexample in answer **D** is correct, so **A** and **C** must be wrong. The question asks about path length, not speed, so the reasoning of answer **B** is not applicable.

Q 20.3 **B** and **C**. **B** expresses the normal decrement of TTL on each hop. **C** expresses the invariant relation between TTL and Hops: At each hop, TTL is decremented and Hops is incremented. Answers **A** and **D** have no connection with reality.

Q 20.4 **A**. A null QUERY will induce a QUERYHIT in every node it passes through, it will also be forwarded on to all other attached Gnutella nodes, and the payload of a null QUERYHIT is identical to the payload of a PONG, so the PING/PONG pair is redundant. By mutual exclusion, answers **B**, **C**, and **D** must all be wrong; in addition the reasons given in answers **C** and **D** contradict the specifications.

Q 20.5 **A**, **C**, and **D** all represent ways of partitioning the network with the client C in one partition and the server S in the other partition. Because there can be many paths between C and S, loss of a single node will not usually be sufficient to partition the network, so **B** cannot be correct.

Q 20.6 **B** and **C**. Because there can be many paths between a client and a server, no single other node can deny service by discarding messages; even a node containing the file cannot always deny service, since other nodes may also have a copy. Thus **A** is excluded. **D** is excluded by the same reasoning used in the answer to *Q 20.5*. **E** is excluded because the flood-control strategy of other nodes will discard messages they have already seen, no matter what the value of the TTL.

Q 20.7 None of the answers are correct! Ben is right, but he is having trouble expressing the reason. To protect against eavesdroppers requires a key-based reversible transformation, and CSHA would be neither key-based nor reversible. A shared-secret encryption algorithm would be both key-based and reversible, so it can in principle do the job, assuming Ben can come up with a secure way of delivering the shared-secret key to the recipient of the file. (The reason that **A** is not correct is that you can't compute anything useful from the output of a cryptographic hash function, even if you have the original input available. All you can do is recalculate the hash and verify that it matches.)

Q 20.8 **E**. Unfortunately, signing every payload and download does not prevent any of these attacks. There are three problems: (1) Ben has not provided a secure way to communicate the public key to the recipient of a download, so a malicious node can replace a self-signed public key in a PONG message with a different self-signed public key. (2) Ben signed only the payload of the Gnutella messages, so Gnutella headers are still vulnerable to malicious changes. (3) Even if Ben had solved problems (1) and (2), the resulting system isn't capable of withstanding any of the four specific attacks. A malicious node could pursue attack **A** by sending its own public key in a PONG message, responding with a false claim to a QUERY message, and then sending a Bach fugue, signed with the corresponding private key, in response to the download request. It could also pursue attack **B**, because the recipient of a QUERY does not know even an insecurely communicated public key needed to verify the integrity of that QUERY. Attack **C** would still be possible because lost messages are undetectable, whether or not they are signed. Attack **D** would still be possible because the protocol still allows any node to claim it has a copy of any song.

21. The OttoNet

Q 21.1 **A**, **C**, and **D**. Conclusion **A** follows from the definition of a "best-effort" network. **B** is not correct. Even when there's no congestion, packets could be lost. For example, noise could corrupt packets, routes could fail, a car's computer could fail, etc. **C** is correct, because a best-effort network makes no guarantees on packet delay. **D** is also correct. An OttoNet client makes a query to a location, and the query could be delivered to the server application of more than one car at that location. Each of those servers will send a response to the client.

Q 21.2 **X** should be 0. When the client does not care or know which server to use, setting 0 in the *dst_id* field causes the message to be delivered to all the cars at the location specified in *dst_loc*, in best-effort fashion. **Y** should be *query_packet.src_id* to assure that the response goes only to the client application that initiated the query.

Q 21.3 The GPS location is an address, not a pure name. It is overloaded with location information that helps the OttoNet find a suitable car to process a query. In a packet containing a response message, the GPS location helps locate a car with a specific ID in the OttoNet. A car's ID is not overloaded, and by itself offers no clue where the car is located; it is a pure name. Both the GPS location and the ID are unique identifiers. The GPS location uniquely identifies a region, defined by the precision of the GPS system, on the surface of the Earth. The car ID is a unique identifier of a particular vehicle.

Q 21.4 **A** and **C**. Setting a timer and retrying a QUERY if the timer expires is an end-to-end loss recovery technique, and an end-to-end checksum is a great example of an end-to-end error detection technique. Method **B** might improve performance and reduce the number of losses observed by a client or server, but it is not an end-to-end technique. Similarly, the suggestion of **D** of running a reliable transport protocol like TCP between cars is like running TCP connections between Internet routers. There is no end-to-end mechanism present in such an approach!

Q 21.5 Any timer expiration setting that is larger than the largest possible round-trip time between client and server is "safe". The round-trip time in the OttoNet is the sum of the per-hop transmission and processing times, the per-hop queuing delays, and the speed-of-light propagation time of the packets containing request and response messages.

Both transmission and processing time are 10^{-7} seconds per bit, and they are not overlapped. For a packet containing a request, the transmission + processing time at each hop is

$$10^3 \text{ bits} \times (10^{-7} + 10^{-7}) \text{ seconds per bit} = 0.2 \text{ milliseconds}$$

with a total of 20 milliseconds to complete 100 hops.

For a packet containing a response, the transmission + processing time is

$$10^4 \text{ bits} \times (10^{-7} + 10^{-7}) \text{ seconds per bit} = 2 \text{ milliseconds}$$

with a total of 200 milliseconds to complete 100 hops.

For a packet to be allowed to enter a queue whose maximum size is four, there can be no more than three packets already in the queue. The worst-case queuing delay is when every queue has three unprocessed packets in it at the instant that the packet whose delay is being estimated arrives, and they all contain response messages. Thus the worst-case queuing delay at each hop is 3 x the processing + transmission delay of a packet containing a response, or 6 milliseconds. The total round-trip queuing delay is therefore equal to

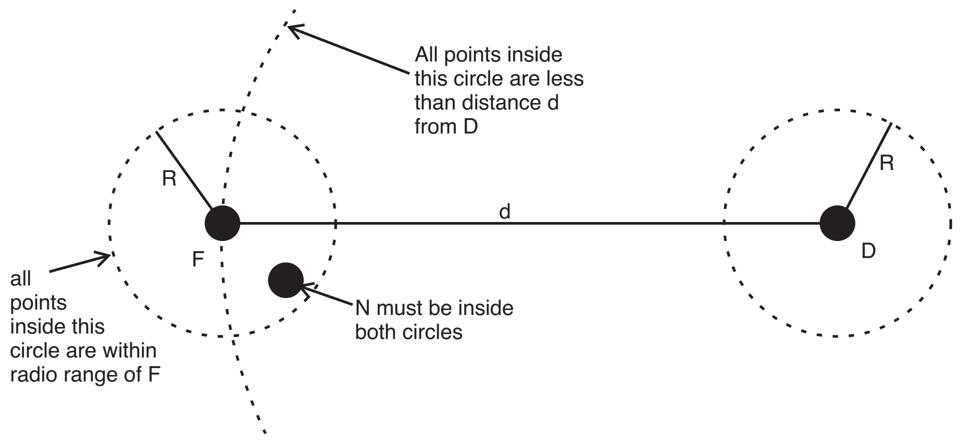
$$2 \times 100 \text{ hops} \times 6 \text{ milliseconds per hop} = 1200 \text{ milliseconds.}$$

Hence, the maximum possible round-trip time is 20 + 200 + 1200 ms = 1420 milliseconds = 1.42 seconds.

Compared to these numbers, the propagation time is negligible, because the maximum distance is 2 x 100 x 250 = 50000 meters, and radio (light) travels such distances in less than 1 millisecond.

Q 21.6 To be able to forward a packet using rule F3, two conditions must be true: first, the next hop, N, must be closer to the destination, D, than the source, F; second, N must be within F's communication radius. To identify points that are closer to D than F, let's label the distance between D and F as d. Any point that is less than distance d from D will be within a circle of radius d centered at D.

If we take the intersection of this circle with F's communication radius, we get the possible locations where N can be.


All points inside this circle are less than distance d from D
R
R
d
F
D
all points inside this circle are within radio range of F
N must be inside both circles

Q 21.7 A is wrong. The response to a query does not include the *dst_loc* that the sender used, so the *dst_loc* in the packet containing the query cannot be used to differentiate replies. The *src_loc* in the packet containing the response cannot be used either, since the car which responds to a query is not guaranteed to be at the *dst_loc* of the query packet. The client can't simply match a *reply.src_loc* to the nearest *query.dst_loc* because the same server may reply to two different queries addressed to two different locations (if, for example, that server is the nearest car to both destination locations). **B** is correct. A nonce is a unique identifier that the client can use to differentiate responses. **C** is wrong. With choice B the client can already match responses to queries, so there is no need for additional mechanism such as having the server add a different nonce to the reply. Since **B** was correct, **D** must be wrong.

Q 21.8 **A** won't work. There are two possibilities for the timer expiration value. First, suppose that Ben used the answer to *Q 21.5* to set the timer. Given a fixed timer value, lengthening queues would increase, not decrease, the chance of congestion collapse. The longer queues may cause clients to time out and resend their requests, even though a response may already be on its way back. Second, suppose that Ben adjusted the timer for the longer queues. Doubling queue lengths certainly doesn't prevent congestion collapse, because congestion collapse can occur with queues of any length. There is no a priori reason to believe that it is less likely with 8-packet queues than with 4-packet queues. Increasing the size of the queue to 8 packets might have a positive effect: some packets that would otherwise have been dropped might eventually reach their destination. However, it might also have a negative effect: packets that would otherwise have been dropped remain in the system and may cause congestion elsewhere. **B** works. Exponential backoff reduces the injection rate of packets to a level that the network can tolerate. **C** will also work. If this question had said "current" rather than "maximum" rate, it would have exactly been exponential backoff. Reducing the maximum rate eventually produces the same end result. **D** is bogus. Flow control windows apply to streams of data. OttoNet requests are not streams, they are independent messages, each one of which may be delivered to a different server, so a flow control window is not applicable. Moreover, flow control is an end-to-end mechanism to ensure that a slow receiver's buffers don't get overwritten by a fast sender. But the problem states that the server and client processing are both infinitely fast, so adding flow control would not accomplish anything.

Q 21.9 **A** is true. Since the message is not encrypted, anyone who handles the packet can read the payload. Since the message is signed, and the corresponding public key (the ID of the client car) is inside the signed message, anyone who handles the packet can verify that the source is as claimed and that the payload has not been modified since it left the source. **B** is false. The destination fields are not signed. A malicious node that is forwarding the packet could modify the destination field, or could modify the payload (say, by substituting one that is encrypted for someone else). **C** is true. A malicious node in the network can fabricate and encrypt an arbitrary payload with the client car's public key (the destination), and use the result to replace the original payload. **D** is false, based on the answer to part C. In addition, malicious nodes can also perform replay attacks or change the unprotected destination location and ID fields.

22. The Wireless EnergyNet

Q 22.1 The double vertical bars demarcate fields belonging to different layers:

Start of frame	*recvid*	*sndid*	*dstaddr*	*location*	*time*	*payload*	*e2e.chsum*	*ll.cksum*

Q 22.2

A. No. For instance, a fault in the network layer could cause an error that corrupts the payload, but since the link-layer checksum is recalculated at each hop the corruption of the payload would not be detected.

B. Yes. Assuming that a corrupted field *and* a corrupted checksum don't accidentally match, the *e2e.cksum* should be sufficient to ensure that the *location*, *time*, and *payload* fields have not been corrupted in transmission. Such a match should be a low-probability event unless one of the intermediate nodes is malicious. (We will learn how to protect against that possibility in Chapter 11.)

C. No. The end-to-end checksum is set and verified only by the end-to-end layer.

Q 22.3

A. True. For instance, if the timer expiration period is too short, an ACK may not arrive at the sender until after it has already retransmitted a frame.

B. True. For instance, an ACK may be lost, causing the retransmission of a frame that was already received.

C. False. Link-layer retransmissions don't introduce any reordering, and we were told that each node's network layer queue is FIFO.

D. False. The end-to-end argument does not preclude duplicating a function in a different layer, and duplication may have a benefit, such as a performance improvement.

Q 22.4 The Manchester scheme of phase encoding requires two level transitions to transmit each bit. A transition frequency of 500 kilohertz means that there can be 500,000 transitions per second, so the maximum data rate would be 250 kilobits per second.

Q 22.5 1:B. 2:D. 3: A and C.

Q 22.6

A. Yes. Not receiving an ACK is probably an indication of congestion at the intended receiver, and exponential backoff reduces the offered load, so it could help reduce congestion.

B. No. Provisioning queue sizes to ensure that no packets are dropped for want of queue space has no effect on congestion, and it increases delay.

C. Yes. Each node reacts to rising congestion by slowing down. This approach is called "hop-by-hop congestion control". (Contrast this approach with the "end-to-end" approach discussed in the text).

Q 22.7 Rounding the answers to the nearest second allows us to ignore the exact time taken for the advertisement to be received and processed.

Earliest: t = 10 seconds. Latest: *t* = 16 seconds. For the earliest time, suppose that the previous five advertisements from the parent were lost. In that case, destruction of the parent would mean that the advertisement at *t* = 10 seconds doesn't happen, so N would remove the entry for the parent the next time it runs REMOVE_OLD_ENTRIES, at *t* = 10 seconds.

For the largest possible time at which N would remove the entry for its parent, the parent could have successfully sent an advertisement just before being destroyed, with that advertisement reaching N at slightly after *t* = 10 seconds. N's once-per-second check, as part of BROADCAST_ADVERTISEMENT, runs at *t* = 10, 11, 12, … seconds. The *t* = 10 advertisement will be five seconds old at *t* = 15 seconds, and REMOVE_OLD_ENTRIES will deREMOVE_OLD_ENTRIESlete it at *t* = 16 seconds.

Q 22.8 Here is a scenario that leads to a loop. Soon after B fails, the first advertisement heard by A is from R, saying that R can reach the sink with a certain quality. When A removes B as its parent, it will pick R, because the quality will be higher than any other choice. We now have a routing loop, because A and R each have the other as its parent.

Q 22.9 The simplest way to eliminate this routing loop is for each advertisement to include the path to the sink (that is, use a path-vector approach). A node ignores an advertisement whose path already includes it.

An apparently attractive way to solve the problem would be for each node to include its current parent in its advertisement, and for a node to ignore any advertisement on which it is the named parent. (There are colorful terms given to variants of this approach in the networking literature, including "poison reverse" and "split horizon".) Unfortunately, this method only prevents two-hop routing loops and does not prevent loops involving three or more nodes.

One might also think that adding a "hop limit" or "time to live" field in the packet header, decrementing that field on each hop, and discarding a packet when this value reaches zero, might eliminate routing loops. That is not correct; this approach only ensures that no packet circulates in the network forever, but it does nothing to prevent the routing loop.

Q 22.10 There are three paths from R to S each with different end-to-end packet delivery probabilities: R–A–S ($p = .99 \times .33$), R–A–B–S ($p = .99 \times .99 \times .99$), and R-C-S ($p = .75 \times .8$). Sara's protocol picks the path with the largest delivery probability, R–A–B–S.

Q 22.11 The path that minimizes the expected number of radio transmissions is R-C-S. Consider again the three paths:

R–A–B–S: The expected number for this path is $3/0.99 \approx 3$.

R–A–S: The expected number for this path is $1/0.99 + 1/0.33 \approx 4$.

R–C–S: The expected number for this path is $1/0.75 + 1/0.8 \approx 2.58$.

Although the chance of success is greater for the path R–A–B–S, that path always takes at least three transmissions, whereas path R–C–S succeeds in two transmissions often enough that on average it wins.

Q 22.12 A C B. Suppose x is the number of sensor nodes that fail before they are replaced. Since m > k > 1, the availability of the room's data is a decreasing function of x; replacing a faulty sensor after a smaller number of failures improves availability.

Q 22.13 Let T_f be the MTTF of a sensor node. The expected time for the first of m nodes to fail is

$$\frac{T_f}{m}$$

When that happens, the expected time for the next sensor node to fail is

$$\frac{T_f}{m-1}$$

because the failure process is memoryless. The room becomes unavailable when only $(k-1)$ sensor nodes are up, which means that the room's MTTF is

$$T_f\left(\frac{1}{m} + \frac{1}{m-1} + \ldots + \frac{1}{k}\right)$$

Notice that the argument underlying this calculation is essentially the same as the one made for airplane engines in Chapter 8.

23. SureThing

Q 23.1

A. Yes. If an intermediate router discards a packet, the packet will not arrive at its destination.
B. Yes. If a packet is corrupted in transit, the IP checksum may fail, causing an intermediate router to discard the packet.
C. No. The RPC payload is not examined by the routers. Routers act on the IP addresses found in the network layer header, but they do not examine RPC payloads. As a result, any IP addresses found in that payload, whether correct or not, are not seen by the router and cannot affect Internet forwarding.
D. Yes. In the Internet, forwarding loops are always a possibility, and the network layer includes a time-to-live field that forces a packet caught in a forwarding loop to eventually be discarded.

Q 23.2 Answer **B**. In the best case, the first computer that GET_LOCATION queries has in its *successors* table the immediate successor of c, so only one remote lookup is required.

Q 23.3 Answer **E**. In the worst case, the computer executing GET_LOCATION is the next computer after the immediate successor of c. In that case, GET_LOCATION must query every fourth computer in the system. Thus, the number of lookups needed is ~$n/4$, which is O(n).

Q 23.4

A. Yes.
B. Yes.
C. No. It may be that the immediate successor of *c* is the node just before the local node. In that case, the local node will return its own copy rather than doing a remote GET of the other copy.
D. Yes.

Q 23.5

A. Yes. If one of the computers has crashed, it might not answer a call.
B. Yes. Both of the successors may have crashed.
C. No. The two nodes that stored the content may both have crashed.
D. Yes. As long as either of the two computers is up, GET should be able to retrieve the file.

Q 23.6 The correct answer is **A**, zero. In the best case, the *node_cache* at the computer where GET_LOCATION is called will contain the immediate successor of the requested content ID *c*. In this case, GET_LOCATION will return the IP address of the immediate successor of *c* without having to make any RPCs.

24. Sliding Window

Q 24.1 The maximum data rate occurs when there is a continuous flow of data, so the window size should be at least equal to the round-trip time multiplied by the bottleneck data rate, as explained in Section 7.5.6.3. The round-trip time comprises the two-way propagation delay (2 seconds) plus the transmission time for one segment and its ACK. Since we can send 10 segments per second, it apparently takes 0.1 second to transmit a segment. The problem statement doesn't tell us the time required to transmit the ACK but presumably it is smaller in size than a data segment—call its transmission time ε. The round-trip time is thus $(2.1 + \varepsilon)$ seconds. The bottleneck data rate is 10 segments per second, so the smallest sliding window that achieves the full capacity is

$$(2.1 + \varepsilon) \text{ seconds} \times 10 \text{ segments per second} = 21 + 10\varepsilon \text{ segments.}$$

Since we assume that $\varepsilon < 0.1$ second, 10ε must be less than 1 second, so the window should be between 21 and 22 segments in size. Since a window size is expressed as an integer number of data segments, 22 is the smallest window size that is at least as large as needed.

Q 24.2 **A** is the correct answer. The receiver sends the first ACK immediately after receiving the last bit of the first segment, which is also just before it receives the first bit of the second data segment. It sends the second ACK just after it receives the last bit of the second data segment. So the interval I between the sendings of the two ACKs, and thus also the interval between arrivals of the two ACKs at the other end, is exactly the time to transmit the second data segment. That time is the data segment size K divided by the link data rate B, so $I = K/B$. The sender knows K and can measure I, so it can calculate B as $B = K/I$.

Q 24.3 **A** is the correct answer. Source 1's window is always 22 data segments, and the router's queue can hold about 220 data segments. Source 2 will open up its window until it begins missing ACKs, which will be when the router begins discarding data segments, at which point Source 2 will cut its window size in half and then gradually open it up again. If we look at the router queue at the instant that it first overflows, we would expect to find ~22 segments from Source 1 and ~198 segments from Source 2. The Source 2 window will thus vary between about 100 and 200 data segments. Since between 80% and 90% of the packets in the router's queue will be from Source 2, that source will capture between 80% and 90% of the link capacity.

25. Geographic Routing

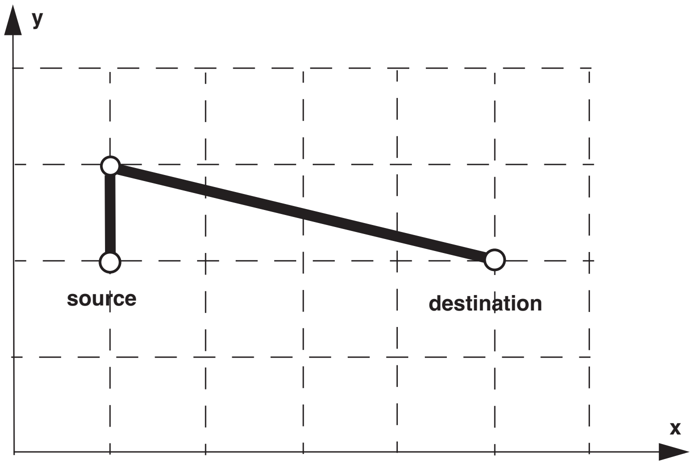
Q 25.1

A. True. Since each hop brings the packet either strictly closer to the destination, or drops the packet, there is no opportunity for a loop.

B. True for some failures, false for others. A node that fails in such a way that it continues running but sends packets in the wrong direction could create a loop. However, a node that detects that is has failed and stops running (in Chapter 8 this behavior is called *fail-stop*) can at worst lose some additional packets; it cannot cause a loop.

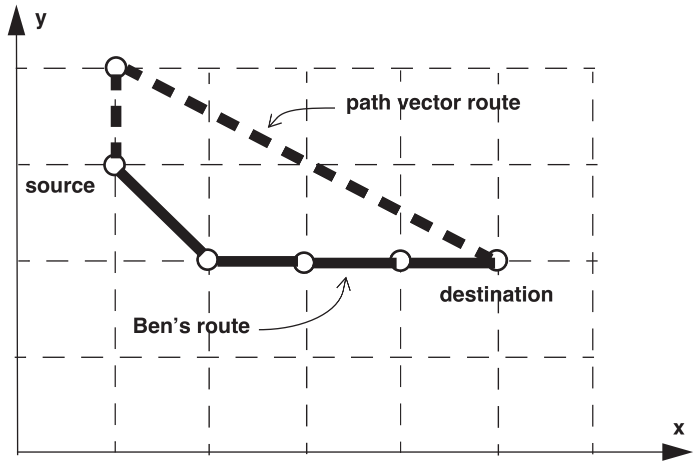
C. False. A newly added node will either drop the packet or forward it strictly closer to its destination. There is no opportunity for creating a loop.

Q 25.2 No. Here is a counter example.



Ben's algorithm will never send a packet along the first hop in the path, since this hop would take the packet further away from the destination.

Q 25.3



The path vector protocol will select the upper path because it has the smallest number of hops. Ben's algorithm will choose the lower path because every step takes the packet closer to its destination.

26. Carl's Satellite

Q 26.1 **A**. Any time a bit is corrupted by interference, the corresponding character will print as something different from that intended. As described, there are no mechanisms in the communication system or in the programs that re-transmit, and there are no delays, so there is no way for a story to be repeated or received out of order. In addition, we are told that a bit is received for every bit sent, so the receiver will receive a (possibly incorrect) complete copy of every story.

Q 26.2 For a story to be received correctly, every one of the 1024×8 bits must be received correctly. The chance of receiving a single bit correctly is $1 - 10^{-5}$. Since each bit is independent of all the others, the probability that all of the bits are received correctly is $(1 - 10^{-5})^{1024 \times 8}$. Using the Taylor series approximation $(1 - \varepsilon)^k \cong 1 - k\varepsilon$, this number is about 0.92, so about 92% of the stories will be printed correctly.

Q 26.3 **A** and **D**. Even though there is a checksum, if there are two transmission errors in the same story their effect on the checksum may cancel, so stories with errors will still slip through and be printed. In addition, the new system discards incoming stories when the checksum doesn't verify, so some stories can be completely missing. Carl hasn't added any retransmission or delay mechanism, so we still can't have duplicate or out-of-order stories.

Q 26.4 Scheme 1 would, on average, print more correct stories. This question illustrates the same principle as the three-engine airplane example of Chapter 8[on-line]: when you add redundancy to a system, the additions can fail and thereby lead to new modes of system failure. Scheme 1 prints every story, whether or not it is correct, so it is assured that it prints all of the correct stories. Scheme 2 doesn't print stories that fail to pass its error detection filter. As a result, it will occasionally throw away a perfectly correct story because of a failure in its error detection system—a reception error in a bit of just the checksum that the sender appended to the story. The chance that it will happen is small—about 1 story in 10,000 will have just a bad checksum, but when there are 10,000 stories, the effect may be noticeable.

Q 26.5 **A**, **B**, and **C**. As before, stories with errors can still slip through. In addition, Carl has added a retransmission mechanism, and since this is a broadcast system, all the receivers hear the retransmissions. So if site A misses story 9 and requests a retransmission, site B will see story 9 arrive twice. Further, the only way that a receiver knows to request retransmission of a missed story is that it receives a story with a higher sequence number. Therefore, the retransmission is certain to arrive out of order. As for answer **D**, we are told that the first and last story has been received by everyone, and that everyone has stopped asking for retransmissions, which means that all stories have been received. So no stories are missing.

Q 26.6 Three different stories have been requested, so Carl will resend exactly those three.

Q 26.7 When only one story has a checksum error, the receiving site can reconstruct the damaged story by XOR-ing the other three stories and the parity story. Since each site is missing at most one story, all the damaged stories can be reconstructed at the receiving ends and no retransmissions are needed.

Q 26.8 With just the parity story, the receiver could discover that one of the five stories contained an error, but would have no idea which one, so it would not have enough information to correct the error. Carl's system has lost a useful feature, so Carrie must be wrong. In his original system, each story carries its own checksum, so a bad checksum indicates something is wrong with that particular story, and the receiver knows which one needs to be reconstructed.

27. RaidCo

Q 27.1 Blank spaces in the table have been filled in and underlined. Explanations follow.

	R0	R1	R2	R3	R4	R5
Block size (kilobytes) exposed to GET/PUT	1 kB	1 kB	8 kB	11 kB	1 kB	1 kB
Capacity, in blocks	1,200,000	600,000	100,000	100,000	1,100,000	1,100,000
Max time for a single 100 megabyte GET (seconds)	1/12 s	1/2 s	1/8 s	1/11 s	1 s	1 s
Time for a 1-block PUT (milliseconds)	10 ms	10 ms	10 ms			20 ms
Typical number of microdrives involved in a 1-block GET	1	1	12	11	1	1
Typical number of microdrives involved in 2-block GET	2	2	12	11	1	1
Typical number of microdrives involved in 2-block PUT	2	2	12	12	2	3

Block size times # of blocks is the capacity of the array. Max time for a single 100 megabyte GET is a measure of the data striping/redundancy. Time for a 1-block PUT is based on whether the array must do a read–modify–write. Typical number of microdrives involved in a 1-block GET is based on block size. The number of microdrives involved in multi-block GETs and PUTs differs from single-block due to striping (data and parity).

R0 has no parity disks or redundancy; hence all 100,000 sectors of each of the 12 microdrives constitute blocks, totaling 1,200,000 blocks The 12-way parallelism allows

100 megabytes to be read in 1/12 the time to read from a single microdrive, or 1/12 second. A single-block GET involves a single sector of a single microdrive, requiring one 10 millisecond seek time. A 2-block GET or PUT is striped across 2 disks.

R1 has 2X redundancy, for 600,000 total blocks. This organization allows twice the read data rate of a single microdrive, so a 100 megabyte GET takes 1/2 second. A 1-block GET accesses one of the 2 copies from a single sector of a single microdrive; a 2-sector GET can be optimized slightly by reading a single block from each of two disks, involving two disks. The mirroring requires writing two microdrives on any PUT.

R2 uses bit-level error detection and correction, for groups of bits spread over the 12 microdrives. The 12 bits of a group must be partitioned into D data and P parity bits, such that $2^P \geq D + P + 1$ to allow single-bit

error detection and correction within each group, since there need to be sufficient distinct combinations of parity bits to encode no error as well as an error in any of the D + P bits. This requires 4 parity disks, leaving 8 data disks. The disks are thus organized as 100,000 8 kilobyte blocks, each comprising 8 data sectors and 4 parity sectors. The 8x parallelism afforded by the striping allows a 100 megabyte GET in 1/8 second. Access to even a single 8K block involves all 12 disks, as the parity disks must be read for error detection (recall that the microdrive-level error detection is disabled).

R3 stripes data across 11 disks, with a single parity disk; the data is organized into 100,000 blocks of 11 kilobytes, each comprising a single sector of each microdrive. The 11-way striping gives 11x read parallelism, or 1/11 second for a 100 megabyte GET. A single 11 kilobyte block can be written via a single PUT to each of the 12 disks, requiring a single seek time of 10 milliseconds. A GET requires reading each of the 11 data disks, while a PUT involves all 12 disks since the parity must be rewritten as well as the data.

R4 uses 11 data disks and a single dedicated parity disk, for a total of 1,100,000 blocks of 1 kilobyte, each corresponding to a single sector of a microdrive. No striping means 1 second for a 100 megabyte GET. A 1-block PUT requires updating the parity disk, which can be done by a GET followed by a PUT and takes 20 ms. A 2-block PUT involves writing the data disk (only 1; no striping) and the parity disk, thus two disks.

R5's distribution of parity causes the common case of a 2-block PUT to involve a single data disk plus two different parity disks, for a total of 3 disks. On a few 2-block writes, one of the parity blocks will reside on the same disk as the data, thus involving only two disks.

28. ColdFusion

Q 28.1 The system maintains complete replicas, so answer **B** is appropriate. The description of the system does not mention anything about returning acceptable values when correct values are not available, so fail-safe design (answer **A**) can not apply. Pair-and-compare is specifically for two replicas, and this system may have more than two, so **C** is eliminated. There is no mention of replicas being separated in time, so answer **D** is out.

Q 28.2 Choices **A** and **B** do not write to enough servers to produce a majority. Choice **C** writes to half the servers. Although it reads all of the servers, the half that PICK_MAJORITY chooses to use may not include any of those in the half written to, so it doesn't work when the number of servers is even. Choice **D** always works; it writes to at least 3/4 of the servers and reads at least half, so unwritten servers cannot constitute a majority of those read. Choice **E** writes to all servers, so GET can read successfully from any server.

Q 28.3 **C** is correct. The failure characteristics of PUT are congruent to those of the multiengine airplane described in Chapter 8[on-line]. The PUT MTTF coefficient when there are $2k-1$ servers is $1/(2k-1)+1/(2k-2)+\ldots+1/k$. When there are $2k+1$ servers ($k \geq 1$), the MTTF coefficient is $1/(2k+1)+1/(2k)+\ldots+1/(k+1)$. The difference between the two is equal to $1/k-\{1/(2k)+1/(2k+1)\} = 1/(2k)-1/(2k+1) > 0$. Hence, the PUT MTTF decreases as the number of servers increases, if we have an odd number of servers.

Q 28.4 **A** is true and **B**, which contradicts **A**, is false. Although the successful PUT wrote to a majority of the servers, the later failed PUT may overwrite and thereby destroy part of that majority. **C** is not true, because we might get lucky and the later failed PUT might overwrite only servers that weren't written by the successful PUT.

Q 28.5 With the specified values of *A* and *B*, answer **C** is guaranteed to be true. Each PUT writes to 10 servers. Even if three servers crash, at least seven servers hold copies of the data, and there are 12 left for GET to choose from; it will select 11 of them, of which at least six, a majority, will have copies of the user's data. Since **C** is correct, answer **A** is automatically excluded. The scenario of answer **B** can fail because there is no coordination between writers working on the same data at the same time. Both writers may think they succeeded when in fact each has been overwriting the other's data. **D** can fail if there are several different unsuccessful attempts to PUT the same new data in the file; each attempt may increase the number of copies that have the new data. A later GET may then find a majority and return SUCCESS, even though the last successful PUT was for the previous data.

Q 28.6 With the addition of the log, scenarios **B**, **C**, and **D** now all work correctly.

Q 28.7 Choice **A** is quite broken; getting a response from some other server tells us nothing about whether or not the transaction completed. Choice **B** is almost correct, but not quite. The recovering server also needs to check that the data obtained for the file is the same as the locally stored version in READY state. Choice **C** does this correctly.

Q 28.8 Choice **A** is the classic technique for avoiding deadlocks, and it works in this situation. Choice **B** is a very bad idea; for the same reason that **A** works, **B** will tend to create deadlocks. Choices **C** and **D** also both work; they are equivalent. Both put the actions, rather than the locks, in some universal order. Actions wait only on locks set by other actions that precede it in the ordering, so the first action in the ordering can always make progress. When it completes, some other action becomes first, and can make progress; eventually all actions will complete.

29. AtomicPigeon!.com

Q 29.1 **A** is true. Since the sender and receiver never fail, *next_sequence* and *expected_sequence* are nonces, and the check in the code is correct. **B** is false, since *expected_sequence* is incremented only when a new sequence number arrives. When an old sequence number arrives, it has to be acknowledged, otherwise the sender won't increment *next_sequence*. **C** is false. It doesn't matter how many times the sender retries, the receiver will execute PROCESS only once. **D** is false. Pigeons with the same data are retransmitted, but the protocol defends against that.

Q 29.2 **A** is true. If sender fails before calling BEEP, then TRANSFER will not succeed. **B** is true. If the sender and receiver don't fail, the TRANSFER will complete. **C** is false, since the sender and receiver don't restart, this cannot happen. **D** is true. If the sender and receiver don't fail, and no messages are lost this can happen.

Q 29.3 Line 3. The "obvious" answer, line 5, is wrong. The way to analyze this question is to first replace STARTED and COMMITTED with F1 and F2, so that the mnemonics don't distract from the semantics. There are two things at risk from a failure: *next_sequence* and the transmission of the CD. Writing F1 to the log durably records the current value of *next_sequence* as intending to be used; no matter what happens after that point *next_sequence* will end up set to the current value plus 1 and the data will be sent. The recovery procedure merely checks to see whether or not TRANSFER ever got around to incrementing *next_sequence*, and by the problem statement someone magically continues the BEEP if the acknowledgment containing the current value of *next_sequence* hasn't come back.

What we have here is a redo log, in which F1 is the OUTCOME record and F2 is the END record. The recovery procedure knows the only variable that can change, so it wasn't necessary to record its change in the log.

Q 29.4 **A** is true. If the COMMITTED record is missing, then BEEP hasn't completed, and thus the TRANSFER might not have happened. **B** is false. If an ABORTED record is in place, that means that the transactions completed, but unsuccessfully (which doesn't happen in this problem). **C** is false. If the COMMITTED record is in place, then BEEP succeeded and TRANSFER must have been completed. **D** is false. That cannot happen; a COMMITTED record is always preceded by its STARTED record.

Q 29.5 **A** is true, since this would be an example of an execution without failures. **B** is true, since sender might write STARTED 1, fail, write STARTED 1 again, complete, and write COMMITTED 1. **C** is false, since a second transfer will not start until the preceding transfer has been completed. **D** is false, since a COMMITTED record will never be written twice.

Q 29.6 **A** is correct; if RECOVER_SENDER sees a COMMITTED record, that means that BEEP completed and therefore *next_sequence should* be set to the logged sequence number plus 1. For the same reason, all other answers are false.

Q 29.7 **A** is true. If the machine keeps failing during RECOVER_SENDER, then no progress will be made. **B** is true. If there are no failures, then TRANSFER will complete. **C** is false. The receiver never fails, so *next_sequence* will be a true nonce and PROCESS will be called only once. **D** is true. If TRANSFER succeeds, it will happen exactly once for the same reasons that **C** is false.

Q 29.8 Line 3 should be a STARTED record and line 6 should be a COMMITTED record. The sequence number in the STARTED and COMMITTED records should be the same. Since *expected_sequence* is incremented between lines 3 and 6, the statements should be

```
3       ADD_LOG (STARTED, expected_sequence)
6       ADD_LOG (COMMITTED, expected_sequence – 1)
```

or (since *h.sequence_no* is not incremented between lines 3 and 6),

```
3       ADD_LOG (STARTED, h.sequence_no)
6       ADD_LOG (COMMITTED, h.sequence_no)
```

Q 29.9 **D** is true. Suppose that *expected_sequence* is 100. If the receiver fails after the completion of PROCESS but before writing the COMMIT record, then the recovery code will reset *expected_sequence* to 100. Since the sender hasn't received the acknowledgment for 100, it will resend the transfer with *h.sequence_no* set to 100. When the receiver receives the resent message, it will think it is a new message, because *h.sequence_no* will be equal to *expected_sequence*. Since **D** is true, **A** must be false. As for **B**, although it is true that with this protocol acknowledgments might be retransmitted, that is not a reason why PROCESS could be executed multiple times. **C** is false. First, requests are not always retransmitted; they are retransmitted only if the sender doesn't receive an acknowledgment within 2,000 seconds. Second, even though requests might be retransmitted, that doesn't imply that process might be executed twice, because PROCESS_REQUEST filters duplicates.

Q 29.10 In the answer to the previous question we established that PROCESS_REQUEST can be called more than once, and answers **A** and **B** would both pass multiple calls along to PROCESS, so they are clearly false. **C** is true. If PROCESS is a nested transaction, then its completion will be dependent on the completion of PROCESS_REQUEST, which completes when it writes the COMMIT record at line 6. Thus, if a failure happens after PROCESS completes but before the COMMIT record in PROCESS_REQUEST is written, then the recovery procedure will roll back both PROCESS_REQUEST and PROCESS. This guarantees that PROCESS will complete only if PROCESS_REQUEST completes, and thus PROCESS will appear to execute exactly once. **D** is false, because in this case PROCESS will not be dependent on PROCESS_REQUEST, and the problem described in the answer to the previous question can still happen.

30. Sick Transit

Q 30.1 **D**. Gloria is doing redo logging.

Q 30.2 **D**. The following sequence leads to [$x = 1$, $y = 2$]: *t1* reads x (value 0), *t2* reads x (value 0), *t1* completes (writing 1 to x), and then *t2* completes (writing 2 to y).

Q 30.3 Because there is only a single lock, which is acquired at BEGIN and released at COMMIT, this strategy provides before-or-after atomicity. And with only one lock, deadlock is impossible.

Q 30.4 If a transaction *t1* releases the lock protecting a variable immediately after a write, the result of that write could be observed by another transaction *t2* before transaction *t1* commits or aborts. Thus, this strategy does not provide before-or-after atomicity. Deadlock is impossible though, because this strategy doesn't require any transaction to hold more than one lock at the time.

Q 30.5 This strategy is a simple form of two-phase locking, which provides before-or-after atomicity. This strategy might deadlock, however, because transactions might acquire locks in conflicting orders (e.g., *t1* might acquire locks in the order {A, B}, while *t2* might acquire the same two locks in the order {B, A}, opening the window for deadlock).

Q 30.6 This strategy is again a simple form of two-phase locking, which provides before-or-after atomicity. Because locks are acquired by all transactions in the same order, deadlocks are impossible.

Q 30.7 This strategy doesn't provide before-or-after atomicity, because transactions don't acquire locks for variables that they read. As a result, one transaction might read a variable that has been changed by another transaction and that other transaction might later abort. Since all transactions acquire locks in the same order, deadlock is impossible.

Q 30.8 **C**, if the restore algorithm is clever; **D** otherwise. The problem is that checkpointed values may include the results of a transaction that was in progress at the checkpoint but aborted later. If, while scanning backwards, a note is made of variables restored to old values by an abort, and one is careful not to restore those variables when the checkpoint is reached, then answer **C** is correct. If the algorithm simply restores all values found in the checkpoint, then answer **D** provides for undoing the restoration of values set by transactions aborted later.

Q 30.9 Yes. The value returned is the last value of x that was written by any transaction that committed prior to BEGIN (t). (If there is no such value then 0 is returned.)

Q 30.10 Simplicity's scheme returns the correct value for a READ from the perspective of all-or-nothing atomicity; READ returns the last value committed by any transaction, or the last value written by the current transaction. However, the scheme doesn't have any mechanism to provide before-or-after atomicity. Answer **B** is therefore true.

Q 30.11 B.

31. The Bank of Central Peoria, Limited

Q 31.1 **B**. The omitted lines update the log to show that winning transactions were recovered successfully. This doesn't affect recovery from the first crash, but without those lines, any recovery from future crashes will try to redo those transactions again, writing old data to the disk.

Q 31.2 **D**. RECOVERY performs roll-forward recovery using redo information it finds in the log. The volatile-memory list named *intentions* is not used by RECOVERY; *intentions* is a cache that speeds up committing or aborting the transaction. The first step of COMMIT should thus be to log the transaction as COMMITTED. Then, after installing the updates in the database, it can log an END record to show that this transaction does not require recovery if there is a crash.

Q 31.3 **B** and **C**. Since COMMIT has not yet happened, it doesn't matter whether the intentions list or the log is written first. Choice **A** forgets to update the intentions list, so the data record will never be updated. Choice **D** updates the data record, effectively committing the transaction because this system has no provision to undo updates.

Q 31.4 Answers **A**, **C**, and **D** would all work, because the only thing that needs to be done is to clear the intentions list. Placing an ABORT record in the log might be handy for later auditing, but it isn't needed for correct operation, and the RECOVERY procedure need do nothing but ignore it.

Q 31.5 All four answers are correct!

Q 31.6 **C**. The problem will be with uncommitted transactions that don't get a chance to complete because of a crash. Ben is now writing data records before COMMIT, the LRU buffer cache management algorithm may push an unCOMMITted data record to the disk, and the crash recovery procedure has no UNDO provision.

Q 31.7 **D**. Transaction A adds 1 to the account, transaction B adds 2 to the account. Whichever order they run, they will add 3 to an account that started at zero. The lock assures before-or-after atomicity for the two transactions, so the result is guaranteed to be 3.

Q 31.8 **B**, **C**, or **D**. By releasing the locks before COMMIT, Ben has broken the before-or-after atomicity mechanism. If transaction B is waiting on the lock that A is holding, it can read the old data value (0) before COMMIT gets a chance to write the new value. B will then overwrite A's 1 with a 2, giving result **C**. A similar scenario holds if A is waiting for B, with A overwriting B's 2 with a 1, giving result **B**. If neither is waiting on the lock, then there is a good chance that the two transactions will actually run serially, and produce the correct result, **D**.

32. Whisks

Q 32.1 Given an address A and some new data, we want to choose some sequence of operations such that the Whisk block A/0 will either have the new data, or its old contents—no partially written stuff—even if the system crashes during the ATOMIC_PUT. We can use the atomic CHANGE_TAG operation to do this in just two steps. The commit point is on line 3:

```
1        procedure ATOMIC_PUT (A, data)
2            WRITE (A/1, data)
3            CHANGE_TAG (A/1, 0)
```

Q 32.2

```
4  procedure AA_COMMIT (t)                    procedure AA_ABORT (t)
5      PUT (ComRec/t, COMMITTED)                  PUT (ComRec/t, ABORTED)
6      for i from 1 to num_dirty do               // We don't have to worry
7          CHANGE_TAG (dirty[i]/t, 0)             //   about the garbage!
```

Line 5 is the commit point of AA_COMMIT. It writes the commit record, and installs all of transaction *t*'s updates (in blocks *dirty*[*i*]/*t*) into the durable database state by changing those blocks' tags to 0. Note that the commit record **must** be written **before** the updates are installed. Imagine that you installed the updated blocks first. Then, if a crash happened in the middle of the installs, the recovery procedure would have no way of distinguishing the following two situations, since in both cases, the commit record would just say PENDING.

1. The transaction had not yet committed. It must be aborted; its updates should be thrown away.

2. The transaction was in the middle of committing. Some of its updates have been installed in the non-volatile database, so its remaining updates still need to be installed.

Q 32.3

```
procedure AA_RECOVER ()
    for i from 1 to num_committed do
        t ← committed[i]
        for A from 0 to N do                      // assume ComRec isn't in 0 to N
            if EXISTS (A/t) then CHANGE_TAG (A/t, 0)
```

The idea here is to redo the work of AA_COMMIT in case there was a crash while it was running. Unfortunately, the *num_dirty* array and *dirty* variable that AA_COMMIT used doesn't exist any more—it was in volatile storage and was lost in the crash. So we conservatively run over the entire Whisk, looking for blocks *A/t* where *t* is a COMMITTED transaction. These blocks contain updates a COMMITTED transaction made: we should install them in the database, by changing their tags to 0, just as AA_COMMIT would have.

We don't care about any PENDING or ABORTED transactions. Any updates they made are stored in locations with non-zero tags, so they don't affect the database state even if we don't do anything at all.

33. ANTS: Advanced “Nonce-ensical” Transaction System

Q 33.1 **A**, **B**, and **D**. **C** is wrong because final value of X would increment only by 1, not 2, which is not a result achievable by any one-at-a-time execution.

Q 33.2 **D**. The change suggested by **B** doesn't work; suppose T_1 writes x, then T_2 reads x, then T_1 aborts; now T_2 has read a value that is not equal to that written by the last committed transaction.

Q 33.3 **A**. WRITE must abort if a higher-numbered transaction has written the variable. If it instead wrote the variable, the next transaction to read the variable wouldn't get the value written by the highest-numbered transaction. **C** is wrong because T_h may already have committed. We can't undo T_h because some later transaction may have already read the new value and used it to update some other variable.

Q 33.4 **D**. Suppose x starts with the value 7. Then T_1 writes 8 to x, then T_2 writes 9 to x. Then T_1 aborts, then T_2 aborts. The correct final value of x has to be 7. But the abort of T_2 will write 8 to x, since that was the old value of x that T_2 saw. If T_2 aborts before T_1, T_1 will correctly restore the old value of x, so answer **C** does not demonstrate the incorrect result.

Q 33.5 **A** and **C**. Since line 15 in WRITE ensures that $tid \geq x.WriteID$, answer **C** is equivalent to answer **A**.

Q 33.6 **B** and **D**. Answers **A** and **C** both contradict the order of interleaving proposed by the figure.

Q 33.7 **A** and **D**. The BEGIN operation ensures that **A** is true, and the WRITE operation aborts any transaction that would violate **D**. But COMMITs and ABORTs of independent transactions can occur in any order.

Q 33.8 **A**. Suppose a transaction sets x to 10, then COMMITs, then the system fails before it writes x to cell storage. RECOVER will not undo this transaction, because the log has a COMMITTED record in it. RECOVER doesn't have enough information to redo since WRITE logs only the old value.

34. KeyDB

Q 34.1

A. Neither. The RELEASE can't be just anywhere—it has to be after all ACQUIREs are done. Moreover, to ensure proper behavior on a transaction ABORT, the lock for an updated data item can't be RELEASEd before the COMMIT point.

B. Sufficient. This rule is the simple locking protocol.

C. Sufficient. This rule is the two-phase locking protocol. It is sufficient, but not necessary, because other locking rules that violate two-phase locking may, in particular cases, guarantee before-or-after atomicity.

D. Sufficient. This rule, together with the problem statement that locks are ACQUIREd before first use of a data item is again two-phase locking, with an additional constraint on locking order that might avoid deadlocks but has no effect on before-or-after atomicity.

E. Neither. Suppose the transaction discovers that it must abort after it RELEASEs a lock that is needed to abort. Because another transaction can update a variable and then commit between the time when an uncommitted transaction releases a lock and then subsequently reacquires that lock in order to abort, this rule disrupts the assurance that the all-or-nothing atomicity property holds. Without that assurance, the transaction may not be able to do either all or nothing, in which case there is no way that it can assure correct participation in the implementation of before-or-after atomicity.

Q 34.2

A. No. Two or more transactions could attempt to ACQUIRE locks in different orders and deadlock.

B. No. Same reason as in **A**.

C. Yes. This scheme prevents deadlocks from occurring. The reasoning is explained in Section 9.5.5.

D. Yes. This scheme ensures that every transaction that could make progress if run by itself will eventually make progress in the concurrent environment.

35. Alice's Reliable Block Store

Q 35.1

A. True. In ALL_OR_NOTHING_PUT, the CAREFUL_GET on line 2 and the test on line 3 ensure that if one copy of the data is uncorrupted ALL_OR_NOTHING_PUT will write to the other copy first. This test preserves the invariant that at least one copy of the data is always valid.

B. True. In ALL_OR_NOTHING_GET, the test on line 2 ensures that ALL_OR_NOTHING_GET returns an uncorrupted copy.

C. False. Consider the following scenario: initially, *x.D0* is uncorrupted, and the system fails after completing the first CAREFUL_PUT to *x.D1* on line 4. Then *x.D0* will contain the old data and *x.D1* the new data. However, ALL_OR_NOTHING_GET will return the old data from *x.D0*.

Q 35.2 Line 5. After the new data is written to *x.D0*, ALL_OR_NOTHING_GET will return the new data. Line 4 is not the commit point because of the scenario described in the answer to question *Q 35.1*, part C.

Q 35.3 Line 7. As soon as the new data is written to *x.D0*, ALL_OR_NOTHING_GET will return it, even though *x.D1* still contains the old data.

Q 35.4

A. True. This state will occur if the system fails after successfully completing both ALL_OR_NOTHING_PUT operations.

B. False. In state 1, *x.D0* is valid, so ALL_OR_NOTHING_PUT will overwrite *x.D1* before touching *x.D0*, so if *x.D0* has changed *x.D1* must be new, not old.

C. True. This state will occur if ALL_OR_NOTHING_PUT fails in the middle of the CAREFUL_PUT of *x.D0* on line 5.

D. False. Since *x.D0* initially contains a bad value, ALL_OR_NOTHING_PUT would not replace it with a new value, not an old one.

E. True. This state can occur if ALL_OR_NOTHING_PUT fails after completing the CAREFUL_PUT of *x.D1* on line 4, but before beginning the CAREFUL_PUT of *x.D0* on line 5.

Q 35.5 No, the system will not work properly. Suppose that the system crashes during SIMPLE_PUT's second CAREFUL_PUT (to *x.D1*), leaving *x.D0* as the only good copy. If, on the next call to SIMPLE_PUT, the system fails during the first CAREFUL_PUT (this time to *x.D0*), both copies will be corrupted.

Q 35.6

A. True. The head crash can damage at most one disk, leaving all of the data on the other disk available to ALL_OR_NOTHING_GET.

B. False. If the two copies are on the same disk, a single head crash can wipe out both copies of the data, leaving no good copies available to ALL_OR_NOTHING_GET.

C. False. The same damaged version of the data will be written to both *x.D*1 and *x.D*0, leaving no good copy available to ALL_OR_NOTHING_GET. Worse, both copies of the damaged data will be written to the disk correctly, so ALL_OR_NOTHING_GET will decide they both contain good data.

Q 35.7

A. True. SALVAGE will run frequently enough that at most one of the two duplicate sectors is likely to be corrupted in the time between checks.

B. False. There may be long intervals between calls to ALL_OR_NOTHING_PUT, leaving plenty of time for both copies of the sector to decay.

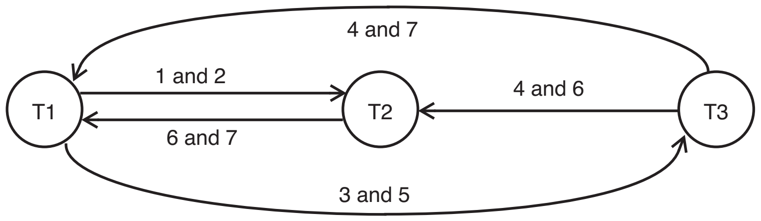
C. False. The intervals might be shorter, but the same reasoning as in part **B** still applies.

D. False. Fail-stop failures can also occur at arbitrarily long intervals, allowing both copies of the sector to decay in the meantime.

36. Establishing Serializability

Q 36.1

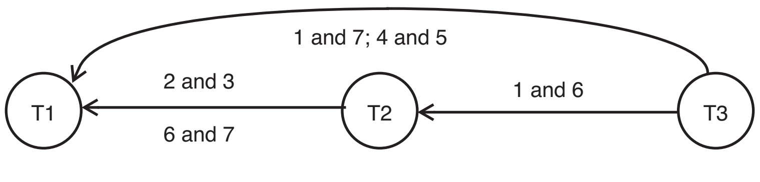
Arrow	Exists?	Steps
T1 → T2	Yes	1 and 2
T1 → T3	Yes	3 and 5
T2 → T1	Yes	6 and 7
T2 → T3	No	
T3 → T1	Yes	4 and 7
T3 → T2	Yes	4 and 6



Q 36.2 This schedule is not serializable because there are cycles in the action graph.

Q 36.3

Arrow	Exists?	Steps
T1 → T2	No	
T1 → T3	No	
T2 → T1	Yes	2 and 3 (also 6 and 7)
T2 → T3	No	
T3 → T1	Yes	1 and 7 (also 4 and 5)
T3 → T2	Yes	1 and 6



Q 36.4 There are no cycles, so this schedule is serializable. The serialization is T3, T2, T1.

Q 36.5 No. Schedule 2 could not have been produced by two-phase locking because two-phase locking does not allow a transaction to acquire a lock after it has released any locks. In schedule 2 this rule is violated. For example, T2 must acquire the lock on *x* in step 2, and release it before step 3 (since T1 needs to acquire this lock in step 3), but T2 needs to acquire the lock on *z* in step 6. This example demonstrates that two-phase locking can be too conservative.

Q 36.6 Following recovery, $x = 1$, $y = 2$, $z = 1$. Since record 7, but nothing after it, made it to disk, the only transaction with a logged COMMIT record is T3. So after recovery, only T3 will have affected the system state. T3 changes *y* from 1 to 2, but *x* and *z* keep their old values because they were modified only by not-yet-committed transactions T1 and T2.

Q 36.7 Following recovery, $x = 1$, $y = 2$, $z = 2$. This time, record 9 makes it to disk, but not record 10, so T2 and T3 committed. Thus *y* and *z* have updated values, but since T1 never committed, its changes to *x* and *z* will not appear in the system state after recovery.

Q 36.8 No. Ben's plan does not work. The reason is that the increment operation is not idempotent. To see what goes wrong, suppose transaction T1 increments object *x* and commits. Since *x* might or might not have been flushed to disk from the cache, the recovery manager may or may not need to redo the update to *x*. However, it has no way to tell whether it needs to do so, because the update record does not contain the previous state of *x*. Similarly, if a transaction does not commit, the transaction manager does not know whether or not it needs to undo the increment operation.

37. Improved Bitdiddler

Q 37.1

A. This protocol works. Replaying the log in forward order will result in every file having its most recent contents. Since the log is keyed on filename, a CREATE that returns an inode different from the one when it was originally executed will not cause a problem (file descriptors are volatile and thus do not persist through a crash). With this protocol, the disk could even be wiped clean before recovery and the recovery operation would work correctly.

B. This protocol fails miserably. Replaying the log in reverse order will result in all files having no contents. The reason is that the earliest log record relating to each file is a create record, which when replayed sets the number of data blocks in the file to 0!

C. This protocol not only works, it is also very efficient. Since file system operations are synchronous and only one operation is in progress at a time, there can be only one operation interrupted by a crash. Thus, reapplying the most recent log record is sufficient to completely recover the disk contents.

D. This protocol recovers only newly created files. It misses all changes to already existing files.

E. False. As in answer **D**, this protocol doesn't recover old files with recent writes. In addition, new files end up empty, as with the protocol of answer **B**.

Q 37.2

A. This protocol still works for the same reason as it did in the answer to *Q 37.1* part **A**. The full replay eliminates any asynchrony issues.

B. Replaying the log in reverse order results in the same disaster described in *Q 37.1* part **B**.

C. This scheme no longer works. With asynchronous disk block WRITEs, an unknown set of previous updates may not yet have been written to disk at the time of the system crash. For this reason, a full log replay (answer **A**) is necessary.

D. This scheme didn't work without a cache, and adding the cache doesn't help.

E. Same answer as for part **D**.

Q 37.3

A. This protocol works. When there are no operations in progress and the file system disk cache contains no modified blocks, the state of the disk matches the state of the log. It is safe to write a checkpoint log record at this point. There is no need to include open files in the checkpoint, since open files do not persist across a crash, but it doesn't hurt.

B. This protocol works, using the same reasoning as in the answer to part **A**.

C. This protocol may fail. Since there are operations in progress, the state of the disk and the state of the log may be out of sync. Noting the list of open files will indicate which files might have issues, but there may be log records before the checkpoint log record that must be replayed to ensure all-or-nothing semantics.

D. This protocol may fail. With modified blocks in the cache, the state of the disk and the state of the log are out of sync, potentially by an arbitrary amount. Any log record before the checkpoint might need to be replayed to ensure all-or-nothing semantics.

Q 37.4

A. No. Files may be closed before commit is called. Imagine T_1 READs file A, CLOSEs and re-OPENs it, then WRITEs to it. In between the CLOSE and the re-OPEN, a different transaction T_2 could WRITE to file A. T_2's write would then be neither before nor after transaction T_1.

B. No. Suppose transaction T_1 got to point B, released its locks, then stalled for a while. In the meantime, another transaction T_2 reads values that T_1 produced and then T_2 successfully commits. Then the system crashes. T_1 should be aborted because it did not get a chance to finish its remaining updates and call COMMIT. However, T_2's results, which were based on the results of aborted transaction T_1, did get written. T_1's changes are neither all nor nothing.

C. Yes. Locks can be safely released only after the commit point.

Q 37.5 Trace 1. There is no deadlock. Since T3 opens file 'baz' before T2, T2 will have to wait for T3 to commit before it can open 'baz' and continue. The apparent serial order will be as if T1 ran before T3, which ran before T2.

Q 37.6 Trace 2. T2 opens 'bar', then T4 opens 'foo', then T2 tries to open 'foo' but has to wait for T4 to commit. Then T4 tries to open 'bar' but has to wait for T2 to commit. Deadlock.

Q 37.7 Trace 3. No deadlock. T4 opens 'foo' after T1 releases its lock and before either T2 or T3 express an interest in that file, so T2 and T3 will have to wait for T4 to commit. The apparent serial order is thus T1 before T4, before T2 and T3. Which of T2 or T3 runs nextdepends on which one first captures the lock on 'foo'.

Q 37.8

A. Action C1. Since there is no COMMIT record, it is safe for the coordinator to abort the transaction. If it sent PREPARE messages, the workers will check back and receive the abort notification.

B. Action C3. A COMMIT record on the coordinator means the transaction did commit. By resending the commit message, the workers will ACKNOWLEDGE the commit and the coordinator will be able to write an END record.

C. Action W3. The worker does not know if the transaction committed or aborted, and thus must ask the coordinator.

D. Action W1. If the worker does not find a PREPARE or COMMIT record in its log, it doesn't know if it actually finished executing the transaction, so it must abort it.

38. Speedy Taxi Company

Q 38.1

A. If both crashes occur right after Arnie presses the button, before DISPATCH_ONE_TAXI gets a chance to even execute BEGIN_TRANSACTION, the log will be empty.

B. If both crashes occur just after COMMIT_TRANSACTION but before DISPATCH_ONE_TAXI reaches the **display** action, this log record would remain.

C. This log record is not possible. There is no C101, meaning that the first transaction did not commit, so the list cannot be empty after the second transaction.

D. This log record is not possible The C101 means that the first transaction did commit, so it must have left the list with just one element. Even though the second transaction did not complete, it would have modified the list to be empty.

E. If both crashes occur just before DISPATCH_ONE_TAXI reaches COMMIT_TRANSACTION this log record would remain.

Q 38.2

A. This message could appear if the crash occurred before the first transaction attempt reached COMMIT_TRANSACTION. None of the actions of the first attempt had any effect, so the second attempt dispatches the first taxi to the first address.

B. This message cannot appear. If the second transaction attempt finds that the list contains a_2, that implies that the first attempt must have committed (but not displayed), in which case the first attempt also allocated a_1 to taxi 0.

C. This message cannot appear. It implies that the first transaction attempt allocated an address to taxi 0, but if it did it would have allocated address a_1.

D. This message could appear if the crash occurred after the first transaction attempt reached COMMIT_TRANSACTION but before it reached the **display** action. The transaction took the first address off the list, assigned it to taxi 0, and committed, so the second attempt finds both database variables have changed.

Q 38.3

A. This message could appear, using the same reasoning as in *Q 38.2* answer **A**.

B. This message could appear if the crash occurs after the first transaction of DISPATCH_ONE_TAXI commits and before the second transaction of DISPATCH_ONE_TAXI gets to its BEGIN_TRANSACTION. In that case, the first attempt to run DISPATCH_ONE_TAXI would have taken a_1 off the list, but not gotten far enough to assign it to taxi 0.

C. This messsage is not possible. It implies that the first run of DISPATCH_ONE_TAXI allocated an address to taxi 0, which means that it must have committed its second transaction. In that case it must have committed its first transaction also, which would have taken a_1 off the list.

D. This message could appear, using reasoning like that of *Q 38.2* answer **D**.

39. Locking for Transactions

Q 39.1 **A.** The locks cause the two transactions to execute one at a time. The two transactions happen to commute, so the result is the same regardless of order.

Q 39.2

- **A.** Yes. If a crash occurs after one transaction commits, but before the other one starts it would leave this result.
- **B.** Yes. Suppose one transaction gets as far as releasing both locks, but not as far as committing. Then the second transaction runs and commits, writing only a new X=2 to the log. Then the system crashes, loses its state, and recovery re-does the second transaction's update to X—but doesn't redo the update to Y, since the first transaction did not commit.
- **C.** Yes, if the crash occurs after both transactions commit.
- **D.** No. Although the misplaced RELEASE statements foul up the all-or-nothing property of the transaction, they still preserve the before-or-after property, which assures that Y can be incremented only once.

Q 39.3

- **A.** T_2 and T_2: The result must be correct. Since neither copy of T_2 changes M, we can ignore the first half of T_2. The second halves are forced to run one at a time by the lock on N, which means they are serialized and the result by definition is one that could have been obtained by a serial execution.
- **B.** T_2 and T_3: The result must be correct. The only potential problem case is if T_3 runs after T_2's RELEASE of the lock on M, but in that case the result is the same as if T_3 ran entirely after T_2.
- **C.** T_2 and T_4: The result may be wrong. Suppose T_4 runs after T_2's RELEASE of M and before its ACQUIRE of N. In that case, the result is M=1, N=3, which is not a result that could be obtained either by running T_2 before T_4, nor by running T_4 before T_2. The before-or-after property has been lost.

40. “Log”-ical Calendaring

Q 40.1 All-or-nothing atomicity. The ADD code is all-or-nothing; if the system crashes or ADD returns, the effect is as if ADD either finished adding the appointment fully, or not at all.

Q 40.2 P2: Durability. P3: Before-or-after atomicity. P4: Constraint (P4 is an example of an invariant).

Q 40.3 C. Here’s why: Since the code for the SHOW procedure is not available for examination, some reasoning is needed to establish how it must work, based on its two-line text description. In particular, we must deduce what “end of the log” means when SHOW looks in the log. To start, we have already established that ADD provides all-or-nothing atomicity, which means that its additions are either fully installed or else not visible to SHOW at all. The description of the master sector is that it always contains the sector number where the next log record will be written, which in turn means that the last log record that was successfully written will be in the previous sector. For ADD to be all-or-nothing, SHOW must begin scanning with the last log record that was successfully written. As a result, it will not see a record that is in the process of being written until ADD successfully completes the ALL_OR_NOTHING_PUT in line 7. But as soon as ALL_OR_NOTHING_PUT in line 7 completes, a subsequent SHOW will be able to see this change.

(Conversely, if SHOW were to mistakenly start by examining the all-or-nothing sector pointed to by the master sector, it would at some times be looking at an all-or-nothing sector that has never been written, at other times at an all-or-nothing sector that is in the process of being written, and at still other times a sector that was written by an ADD that decided to abort.)

Q 40.4 **A** and **D** are true, **B** and **C** are false. **A**: Writing the second physical sector (S_2) of the all-or-nothing master sector controls what SHOW reads; only ADD modifies that physical sector, and ADD makes that modification by overwriting. Because one cannot read a disk sector at the same time it is being written, reads and writes to a single sector are naturally serialized, so P3 is satisfied even in the absence of locks. **B**: If two ADD operations run concurrently, they may both read the same value for *end_of_log*, and then both write their appointment in the same log sector, thus violating P3. **C**: If P3 does not hold, then P4 cannot hold, because two concurrently running ADD procedures for the same timeslot without proper before-or-after atomicity may end up committing both of them. That violates P4. **D**: P3 ensures that a set of concurrent ADD and SHOW actions is equivalent to some serial order. In that serial order, each ADD that tries to add an appointment to a timeslot that already has one will not commit. Therefore, P4 will hold if P3 holds.

Q 40.5 Disk B. The disks are said to be burned-in, so they are past the six-month knee of their conditional failure rate curves. The disks are said to be in their first year of service, and the durability requirement is three years, so we care about the probability of failure between 6 months and 4.5 years. Disk B has a lower conditional failure rate over that entire time span.

Q 40.6 **A** and **D** are true, **B** and **C** are false. **A**: This choice is the write ahead log protocol. If a crash or abort occurs, the log has the information required to undo any changes. Hence, SHOW will display only the appointment corresponding to the last committed ADD. **B** and **C**: The reason these choices do not work is that OVERLAPPING could abort. If, at the time this abort occurs, the log does not have information to undo the changes made by the ADD, then a subsequent SHOW may observe the changes made by this uncommitted ADD. That would violate P1. **D**: Write ahead logging is correct whether the cell storage is in volatile or non-volatile memory.

Q 40.7 Scheme **A** follows the simple locking protocol, and it works. Scheme **B** violates the two-phase locking protocol, which provides a hint that something may be amiss. If RELEASE (λ_t) happens just after line 5, then another transaction may be able to view the changes made to the table by this call to NEW_ADD. Now, if NEW_ADD were to ABORT inside COMMIT, then property P3 is violated.

Q 40.8 **A** and **B** are true. **C** and **D** are false. **A**: NEW_ADD () will not add entries to the primary that overlap with one already present in the primary for that timeslot. **B**: If an appointment added on a disconnected client overlaps with one present on the primary when reconciliation begins, after reconciliation that appointment will not appear in the log or table on either replica. **C**: When C2 is the primary, any overlapping appointments will be reconciled in favor of C2, but when C1 is the primary, those same overlapping appointments will be reconciled in favor of C1. As a result, the server can end up with a different set of appointments in the two cases. **D**: All appointments will be present on the primary after all clients reconcile, but clients that reconciled early will not know about appointments that were found on clients that reconciled later.

Q 40.9 **A** and **C** are true, **B** and **D** are false. **A**: NEW_ADD will not add any entries to the primary which overlap, so appointments already present on the primary remain after reconciliation. **B**: This procedure will miss records added since the last reconciliation. **C**: The only changes that need to be made to the table are new entries contained only in the primary log, or appointments in the client that are removed as a result of reconciliation. Since any conflicting appointment also has a new entry on the primary, no client table entries need to be deleted. All such appointments needing consideration follow the last RECONCILE record. Note that this choice makes the assumption that the "previous RECONCILE record" mentioned in the choice has the same client identifier as the one from the current reconciliation.

41. Ben's Calendar

Q 41.1 ClientRead is affected only by S1, which is fast and reliable, so choices **B** and **D** are incorrect. S3 stops working for a few minutes at a time, since ClientWrite doesn't give up until it succeeds; this merely slows down ClientWrite. Therefore, answer **A** is the only one that applies.

Q 41.2 Since the code executes strictly sequentially and ClientWrite doesn't return until it succeeds, the second call to ClientWrite must have completed before the call to ClientRead. The string returned should be "Breakfast".

Q 41.3 The key observation is that ClientWrite recovers from RPC failures by retrying forever, if necessary, and it writes S1, S2, and S3 sequentially. Therefore, if a power failure occurs in the middle of a ClientWrite, the only possible outcomes are that {S1}, {S1 and S2}, or {S1, S2, and S3} have been updated (partially written records are not of concern, because the writes are actually done by the servers, which are unaffected by the power failure.).

Ben can get outcome **A** if the power failure occurs between the last ClientWrite updates of S2 and S3. If the power failure happens between the fourth and fifth calls to ClientWrite, he would get outcome **B**. Outcome **C** is not possible, since it implies that the test got as far as the last ClientWrite, which updated at least S1, which in turn means that the fourth ClientWrite must have already updated the 11 a.m. record on all of the servers and "Free at 11" could not have been returned by ClientRead. A similar line of reasoning rules out outcome **D**.

Q 41.4 If the test got at least partway through the last ClientWrite then outcome **A** is possible. And if the power failed after the third but before the fifth ClientWrite, outcome **B** could occur. **D** is also a possible outcome: Ben's test could have failed during the 3rd ClientWrite updating say S1 with "Talk to Frans at 10", leaving the other servers with "Free at 10". The first read might contact S1 and get "Talk to Frans at 10"; the second read might encounter an RPC failure on the attempt to contact S1, and move on to S2, which still says "Free at 10". Choice **C** is not a possible outcome; in order for the fifth ClientWrite to cause "Breakfast at 10" to be returned, the third ClientWrite must have previously updated all the servers with "Talk to Frans at 10" and "Free at 10" must have been overwritten.

Q 41.5 With this change, a ClientWrite might return after having updated 0, 1, 2, or all three of servers. Thus, choices **A**, **C**, and **D** are possible outcomes. Choice **B** is not possible, because it implies returning "Z" before writing "Z".

Q 41.6 This time, only choice **A** is possible, because a successful write in Ben's original system would always update all copies.

42. Alice's Replicas

Q 42.1 **A** ,**B**, **C**, **E**, and **G**. **A** & **B**: A file whose content has decayed (some bits changed from, for example, gamma radiation) or has been modified intentionally will pass the test "CSHA (CONTENT (*path*) ≠ *info.hash*", and thus be added to *changeset*. **C**: A file that has been created since the last reconciliation will pass the "*info* = NULL" test and be added to *changeset*. **E**: A file that has been deleted will be detected by the COMPUTEDELETESET procedure, and added to *deleteset*. **G**: Files that have been renamed will pass the "*info* = NULL" test and processed as newly created files, and will be added to *changeset*.

D and **F** are not correct. **D**: A file whose inode has been modified will not pass the "CSHA (CONTENT (*path*) ≠ *info.hash*" test, because CSHA is computed over the content of the file, not the attributes. This file won't be added to *changeset*. And since the file exists, it won't be added to *deleteset*. **D**.: A file that has been deleted but recreated (with the same path name) with identical content will not be added to *changeset* because the "CSHA (CONTENT (*path*) ≠ *info.hash*" test won't succeed. The COMPUTEDELETESET procedure will not add the file to *deleteset*, because its test won't succeed.

Q 42.2 **A** and **D**. **A**: Files created on the laptop will be added to *changeLeft*, and won't appear in *changeRight* or *deleteRight*, because the files don't exist on the server. Thus, these files will be added to *additionsRight*. **D**: Similarly, these files will be in *changeLeft* and not in *changeRight* or *deleteRight*, and thus added to *additionsRight*. **B**, **C**, and **E** are not correct. **B** & **C**: Files removed might be added to a remove set, but not to an additions set. **E**: These files will end up in the conflict list since these files show up in both *changeLeft* and *changeRight*.

Q 42.3

A. Yes. See the answer to *42.2*, part **E**.
B. Yes. These files are in the *changeLeft* set, but not in *changeRight* or *deleteRight*, thus they won't be added to *conflicts*.
C. No. These files are in the *deleteLeft* and *deleteRight*, and the second **for each** loop will not add them to *conflicts*.
D. No. These files are only in the *changeRight* set, and thus won't be added to *conflicts*.
E. Yes. These files will be in the *changeRight* and *changeLeft* sets, and thus in the *conflicts* set.
F. Yes. These files, if they have the same path names, will be in the *changeRight* and *changeLeft* sets, and thus in the *conflicts* set.

Q 42.4 **A** or **D**. RECEIVE commits by executing RENAME; it aborts by executing DELETE. Since file system operations are said to be atomic, the commit point is at the end of whichever of those two operations the program decides to perform.

Q 42.5 **A**. Alice wants RECEIVE to be atomic, which means that if it fails to reach its commit point it should leave no evidence that it was ever there. If a crash occurs after RECEIVE calls CREATE_FILE and before RECEIVE reaches its commit point, there will be a uniquely named temporary file that may remain indefinitely unless the recovery procedure removes it. (In some file systems, if you place the temporary file in a special directory intended for temporary files, a system recovery procedure will automatically remove it.) Answer **C** is something that could be done, but is not required to achieve atomicity. Answer **D** is incorrect because RENAME is the mechanism to commit the transfer. The crash may have happened in the middle of the file transfer, in which case the transfer should not be committed.

Q 42.6 None of these are advantages.

A. Neither version of RECONCILE detects decayed files, but this one, unfortunately, replaces the good file with the bad one, which is not the usual meaning of "repair".

B. Both versions of RECONCILE can run on any standard file system without change.

C. Neither version requires a log.

D. True, but the other version does the same thing, so it isn't an advantage.

E. False. This RECONCILE calculates a cryptographic hash of every file, which requires reading everything in the file system, while the other version simply looks at the time stamp of last modification of a file, so it would be expected to do reconciliations much faster, especially on big files that haven't changed.

Q 42.7 **A**, **B**, and **C** are correct but **D** is not. To avoid confusion between scenario identifiers and participants, call the participants Alyssa, Ben, and Charlie. The original file is *f*, and its hash is found in both $fsinfo_{A\text{-}B}$ and $fsinfo_{B\text{-}C}$. Ben creates state *f '* at noon, Charlie creates state *f ''* at 2 p.m, and Alyssa creates state *f '''* at 4 p.m. At the 1 p.m. Alyssa-Ben reconciliation, Alyssa receives and $fsinfo_{A\text{-}B}$ records state *f '*. The 3 p.m. Ben-Charlie reconciliation discovers that states *f '* and *f ''* are both different from state *f* as found in $fsinfo_{B\text{-}C}$, so there is a conflict whose outcome depends on the scenario:

Scenario A: Charlie receives state *f '*.

Scenario B: Ben receives state *f ''*.

Scenario C: Ben and Charlie receive state *f ''''*.

Scenario D: Ben and Charlie do not have an *f*.

In each scenario, $fsinfo_{B\text{-}C}$ records the new state (or absence) of *f*, but that is irrelevant to the question, because the 5 p.m. reconciliation uses $fsinfo_{A\text{-}B}$, which still shows file *f* at state *f*.

We have the following results:

	Alyssa	$fsinfo_{A\text{-}B}$	Ben	report	answer
Scenario A	*f '''*	*f '*	*f '*	no conflict	yes
Scenario B	*f '''*	*f '*	*f ''*	conflict	yes
Scenario C	*f '''*	*f '*	*f ''''*	conflict	yes
Scenario D	*f '''*	*f '*	absent	conflict	no

43. JailNet

Q 43.1 **B** and **C**. The initial string serves to alert everyone that a new packet is coming, so it is framing a packet. And by counting, one can tell where the first bit of the first byte will be, so it also provides byte framing.

Q 43.2 Since this is a public key system, and in all existing public key systems one can in principle compute $KEY from KEY, so answer **A** is out. Answer **C** would mean that it is too easy to compute $KEY. Public key systems don't depend on lack of knowledge of anything but the private keys, so **D** is excluded. Answer **B** is exactly the goal of a public-key system.

Q 43.3 **D**. Annette intends to read the encrypted text, so it needs to be decrypted, so answers **A**, **B**, **C**, and **E** are immediately excluded. It was encrypted with her public key and she decrypts it with her private key, so the answer is **D**.

Q 43.4 None of the offered explanations is nearly as likely as this one: Ty's protocol includes no authentication, so Annette can't tell who really sent that message. Anyone, including a guard, can send a message that appears to come from Ty.

Q 43.5 Ty is signing the message, so he should provide his private key, choice **C**.

Q 43.6 Since VERIFY returned ACCEPT, the message must have originally been signed by Ty, and it couldn't have been modified since then, so **A** and **F** are both true. But we don't know who sent it or how it got here—someone else may have relayed it. And it was not encrypted, so anyone else could have read it.

Q 43.7 The protocol has no protection against replay attacks. A guard recorded the message yesterday and replayed it today, without having any idea what it means. If Ty had said "let's escape tonight, February 14" then Annette would have realized that the message was a replay when she received in on February 15. The inmates should extend their protocol to always include freshness guarantees.

Q 43.8 Since no one outside knows what key Pete might be using, he is somewhat incommunicado. In particular, he can't sign anything, so no one can be sure that things he sends came from him (answer **B**). If Pete can recall Ty's public key, then he can verify that Ty's nightly signed broadcast of everyone's public key is really from Ty (answer **C**). Ty could even remind Pete of his old public key, but unless Pete can reconstruct his private key he still can't sign anything, and if someone else sends him a message encrypted with his public key he won't be able to decrypt it.

44. PigeonExpress!.com II

Q 44.1 **B**. All an attacker needs to do is trap two pigeons that are carrying CD's that were secured with the same KCD. By XORing those two CD's together the KCD will drop out and the attacker is left with the XOR of the two plaintexts, which is likely to be easy to cryptanalyze using frequency analysis. Worse, if one of those plaintexts is known, the other can be recovered with just one more XOR. Ben needs to study the part of Chapter 11 that discusses one-time pads.

Q 44.2 **A**. The message containing *k* is signed but not encrypted, so anyone who intercepts a pigeon can know *k*; even though *k* is used only as a seed to generate the real key, the pseudorandom number generator is well-known, so anyone else will also be able to reconstruct the key. The protocol is insecure but answer **C** is *not* the reason why. A message that is only signed is not intended or expected to be confidential, so it is not appropriate to describe the widespread availability of the key needed to verify the signature as a cause of insecurity.

45. WebTrust.com (OutOfMoney.com, Part II)

Q 45.1 **A**. The string `http` is a protocol name, and `index.html` is a file path name. there are no internet host addresses in the URL.

Q 45.2 **A**. This source of context references is a standard defined by the URL specification.

Q 45.3 **A** and (probably as an incidental side effect rather than as a primary goal) **B**.

Q 45.4 Unfortunately, all of these attacks work. **A** counts even though it is a passive attack; active attackers are allowed to use the tools of passive attackers, too. **C** qualifies because an attacker can use runaway account creation to cause the account database to overflow or run out of space. Finally, because DNS is not an authenticated service, an active attacker can send to a victim DNS response packets that contain the network address of a site controlled by the attacker.

Q 45.5 **A**, **C**, and **D**. **A** is possible because the POST does not contain anything that assures freshness, so the browser has no way of detecting and rejecting a later copy replayed by an attacker. **B** does not work: The POST contains a user's name and password, so simply replaying it will log in that user; it can't log in any other user. **C** is a difficult attack, but it can be accomplished. The thing that makes it hard is that the registered user knows how the real `webtrust.com` responds, so the attacker must behave in exactly the same way, even to the extent of rejecting wrong passwords. But it can do just that by standing in the middle, passing some or all of the user's requests to the real `webtrust.com`, and passing that site's responses back. **D** is possible, too; all the attacker needs to do is reject the login the same way that `webtrust.com` does.

Q 45.6 **A** and **D**. **B** and **C** refer to private communication, but in this protocol nothing is encrypted, so nothing could be communicated privately.

Q 45.7 **B** and **C**. **A** is false: an attacker can replay any cookie to gain illicit access, until the cookie expires. **B** and **C** are two ways of saying the same thing, since the cookie contains only the expiration time.

Q 45.8 **A**, **B**, and **C**. Because a cookie is a really a message from the server to itself, only the server knows the MAC key; thus the server will detect an attacker's attempts to create or modify cookies. The server creates a cookie saying "Alice" only if someone successfully logs in with Alice's user name and password; thus **B** is true. The timestamp allows the server to reject a cookie if it wasn't created recently. **D** is not true: An attacker may have intercepted the cookie on the way from the server to Alice's browser and is now sending it back to the server.

Q 45.9 **D**. The server can't know whether Alice's browser, or some other software running on her computer, has given away the cookie. But the server can know that Alice's browser sent the cookie, since it arrived over an authenticated connection, and the question states that only the browser and server know the authentication key.

Q 45.10 **A** and **B**. A multiuser system may have mechanisms in place to keep users separated, but those risks merely reduce the amount of increased risk; they don't eliminate it. **C** and **D** do not qualify because the trusted computing base already included the local operating system and the hard disk: the virtual memory of the browser probably includes the hard disk, and the binary of the operating system and the browser were almost certainly stored on the hard disk.

Q 45.11 You should immediately recognize that this question is unanswerable; it is entirely a matter of judgment of the designer and users, not a strictly technical question.

Q 45.12 Lucifer can create an account, log into the server, change his cookie's user name to "Alice" (or any other user name he knows), and send the modified cookie to the server. The server will believe that he is Alice, since the MAC doesn't cover the user name.

46. More ByteStream Products

Q 46.1 The premise of **C** is wrong, and even though **D** is a true statement it is irrelevant to this question. Answer **B** would sound plausible if the key were shorter, but even if Eve could try all possible keys, she cannot verify whether a guessed key is correct or not, since every message is equally likely. Answer **A** is the correct one.

Q 46.2 Suppose C is the output of the PRNG when seeded with key K, and M is Alice's message. Lucy knows the first 500 bits of the message M and the first 500 bits of $M \oplus C$, so she can XOR these two things together and recover the first 500 bits of C. She can now start guessing keys K', generate a stream C' (since the PRNG code is publicly available), and verify whether her guess is correct by comparing the first 500 bits of C' with those of C. If they are identical, then K' is almost certainly the same as K, and she can decrypt the complete message M. But since it is unlikely that Lucy will quickly find the correct 200-bit key, the appropriate answer is **B**.

Q 46.3 Suppose C is the output of the PRNG when seeded with key K, M is Alice's message, and R is Bob's response. The encrypted request is then ($M \oplus C$). The encrypted reply is ($R \oplus C$), since Bob seeds the PRNG with the same key K. ($M \oplus C$) $\oplus$ ($R \oplus C$) is ($M \oplus R$), so answer **C** is certainly correct. Whether or not answer **D** is also correct depends on whether or not Eve has enough additional information to cryptanalyze ($M \oplus R$). If either M or R has no redundancy whatever, and Eve can't persuade Alice or Bob to include some known plaintext in one of M or R, then Eve is out of luck. But if either of those conditions is false, Eve may succeed in cryptanalysis.

47. Stamp Out Spam

Q 47.1 Alyssa intends that this check will prevent malicious receivers from mounting a space exhaustion attack on the quota enforcer. Without the check a receiver R could call TEST_AND_CANCEL (*garbage*, R), causing different garbage bits to be installed in *num_uses* and thus exhaust its storage space. Unfortunately, since CHECK_STAMP_VALIDITY involves cryptographic operations, it probably consumes a lot of machine cycles, so keeping that check opens the enforcer to a different resource exhaustion attack: running it out of processor capacity.

Q 47.2

A. False. Nothing ties a stamp to a particular message. In fact, S could have sent a message with a stamp to T, and T could have used S's stamp without cancelling it on a message to R.

B. True. The certificate is signed with SA_{priv}, and we assume that no private key is compromised, that the cryptographic algorithms are not broken, and that digital signatures can't be forged. With those assumptions, any tampering of CS will be detected.

C. False. Again, if private keys are not compromised and the cryptographic algorithms are not broken, and a digital signature cannot be forged, R cannot manufacture a proper stamp that belongs to S unless R has already seen that stamp (for example, by receiving the stamp on a message from S).

D. True. And, that possibility might prevent users of the compromised machine from sending their own messages. One redeeming feature of this situation is that users will notice, and thus have an incentive to keep their machines up-to-date with security patches!

E. False. While S can send a message to R, wait until it thinks that R received it, and call TEST_AND_CANCEL, S cannot draw any clear conclusion from the result. If the quota enforcer returns CANCELLED, then S might think that R received the message, but it can't be sure because some other entity en route could have cancelled the stamp. If the quota enforcer does not return CANCELLED then S might think that *R* has not received the message, but again it can't be sure, because R may not have bothered to cancel the stamp. Moreover, by doing a TEST_AND_CANCEL of a stamp that it used, S runs the risk of R discarding the message as spam when it later does check the stamp.

F. True. The described procedure is the standard one that provides confidentiality with a public-key system.

Q 47.3

A. G1 No. Superficially, it appears that the only thing that the stamp authority learns is a unique, random identifier and the identity of the receiver, R, and to associate the unique identifier with the sender S it would be necessary to cryptanalyze the first argument, an operation we assume is not feasible without knowing S_{priv}. But if the stamp authority makes a guess as to the identity of S, and can get its hands on a copy of the original stamp (perhaps by eavesdropping, since stamps aren't confidential), it can verify that guess by itself computing DETERMINISTIC_ENCRYPT (u, S_{pub}) and comparing the result with the value of e_1 that R supplied. Unfortunately, deterministic encryption, which so neatly achieves goal G2, is not secure against known-plaintext verification. Louis has stumbled into yet another bad idea.

G2 Yes. Although TEST_AND_CANCEL cannot decrypt e_1, it doesn't need to; it simply uses e_1 to uniquely identify the stamp. If a different receiver checks another copy of the same stamp it will calculate the same unique identifier.

B. G1 Yes.

G2 No. Because each different receiver R has a different public key R_{pub}, each would compute a different value for e_2 thereby making it impossible for TEST_AND_CANCEL to detect the reuse of a stamp.

C. G1 Yes. The only thing that the stamp authority learns is a unique identifier and the identity of the receiver, R. To associate the unique identifier with the sender S, it would be necessary to invert the cryptographic hash function, an operation we assume is not feasible.

G2 Yes. Every receiver R will calculate the same *h* for the same stamp. Thus *h* uniquely identifies that stamp.

Q 47.4 Let's first consider the number of RPCs per second. The average request rate is 1 million per second, so to handle the peak rate requires $\frac{10 \times 10^6}{50,000} = 200$ servers.

Now let's consider RAM storage. We need to store cancelled stamps for two days, so we need to store $200 \times 10^9 \times 20$ bytes. Each computer has 10^9 bytes of stamp storage, so the number of servers needed is $200 \times 20 = 4000$.

The bottleneck is thus the RAM capacity. Alyssa will need 4,000 servers.

Q 47.5

A. True. The thread may have updated an entry in *num_uses*[*s*], and that update would need to be backed out.

B. False. An ABORT may involve an UNDO, but never a REDO.

C. False. Since the in-RAM database is wiped out when a server crashes, nothing needs to be undone during recovery.

D. True. Since the in-RAM database is wiped out when a server crashes, to maintain the two-day durability requirement of cancelled stamps from committed transactions, the recovery process needs to REDO committed winners from the log.

E. True. By construction, the operations in the log correspond to a serial execution order, because the log was written by transactions holding the appropriate locks. (This claim assumes, of course, that the locking protocol and implementation are bug-free.) Because new threads are not allowed to run until recovery completes, there's no need to ACQUIRE any locks during recovery.

Q 47.6 Suppose the system did not provide before-or-after atomicity and S sends multiple messages to different receivers with the same stamp. Each receiver would call TEST_AND_CANCEL on the same stamp, which could cause multiple threads to run on the same server. Each thread could execute lines 4 through 7 of TEST_AND_CANCEL before any thread executes line 8. In that case, S's reuse of the stamp would go unnoticed.

Q 47.7 Alyssa wants to reduce global spam by a factor of 10, to 6.6×10^9 messages per day. Because spammers are willing to spend $11 million a day to spend to send spam, that's the largest amount they can spend on stamps. So, if we price each stamp at

$$\$\frac{11 \times 10^6}{6.6 \times 10^9} = 0.167 \text{ cents}$$

and if the quota enforcer is able to do its job, we might be able to reduce worldwide spam by a factor of 10. That estimate assumes, of course, that the starting numbers are in the right ballpark.

48. Confidential Bitdiddler

Q 48.1

A. Yes. 32 bits is rather small these days. You can easily have more than 2^{32} files on a disk, in which case collisions are not just accidental, they are inevitable.

B. Yes. By knowing the value of the prime number *P*, it is easy for an adversary to construct another block that hashes to the same value.

C. No. Blocks are indexed by their block number, not their hash value, so a coincidental equality of a hash value to a block number has no effect on file system correctness.

Q 48.2

A. A. 160 bits is enough space that is very unlikely that accidental collisions will occur. However, a non-cryptographic hash function probably still allows an adversary to easily construct another block with the same hash.

B. A and B. A 160 bit cryptographic hash is large enough to avoid accidental collisions. Furthermore, a cryptographic hash (like SHA-1) is collision-resistant, meaning that it computationally difficult for an adversary to find a second block with the same hash. (It would be better to use SHA-256, to make it computationally more difficult.)

C. A and B. Checking the blocks for equality prevents both problems, at the cost of a minor performance loss.

Q 48.3

A. True. She can try encrypting her guess with the same hash function used for convergent encryption. If the result matches the block stored on disk, she knows her guess was correct.

B. True. If Alyssa notices that one of her files uses the same block(s) as one of Ben's files, Alyssa knows exactly the contents of Ben's block(s).

C. True. If an adversary changes an encrypted block on the disk, when the file system decrypts that block using the stored key, the decrypted output will be different from the original data. To check, the file system can hash the decrypted output and compare the result with the key for that block. If they don't match, the block is not authentic; if they do match, the block is OK (assuming of course that the hash function is cryptographically strong, so that it is not vulnerable to collision attacks.)

Q 48.4

A. False. She must try all possible keys to decrypt the block. That is not easy.

B. False. Knowing the block number gives no information about the content of the blocks in Ben's files, since those blocks are encrypted with Ben's key.

C. False. The block will decrypt into some value, but the file system has no way of verifying that it is different from the value it originally encrypted.

49. Beyond Stack Smashing

Q 49.1

A. Yes, since the simple buffer overrun requires the attacker's instructions in the buffer to be at an address that the attacker can predict.

B. No. A trampoline attack works regardless of stack address, since it takes advantage of a register that contains an address in the stack at a known offset.

C. No. Arc injection doesn't involve executing instructions in the buffer overwritten by the attacker, so unpredictability of the buffer's address doesn't help.

Q 49.2

A. Yes, since the attack tries to execute instructions in a buffer on the stack.

B. Yes, since the attack tries to execute instructions in a buffer on the stack.

C. No, since the attack executes only instructions that are already present in the victim's address space, and that therefore have the necessary execute permission.

Q 49.3

A. Yes for all three, since they all change the procedure return address by overrunning the array.