5c. (*6 points*) An upgrade to a supercomputer fixes that problem, but it's too late—Louis is obsessed with performance. He switches from RPC to an asynchronous protocol, which groups several requests into a single packet if they are made within 2 milliseconds of one another. On his network, which has a very high latency, he notices that this speeds up some workloads far more than others. Describe a workload that is sped up and a workload that is not sped up. (An example of a possible workload would be one request every 10 milliseconds.)

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 ms will be sped up more on average: every once in a while, 2 requests will be sent in one packet, 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 ms.*

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

5d. (*6 points*) As a design engineering consultant, you are called in to critique Louis's decision to move from RPC to asynchronous client/server. How do you feel about his decision? Remember that the Toastac software sometimes fails with a "Malfunction 54" instead of toasting properly.

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

1. 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.

2. 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.

1. 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.

2. 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.

3. 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.)

Your answer—agree or disagree—depended on how you weighted these reasons and others you may have come up with.

5. (*24 points*) Louis P. Hacker-Reasoner (no relation) bought a used Therac-25 for $14.99 at a yard sale. After some slight modifications, he's hooked it up to the Ethernet as a computer-controllable turbo-toaster, which can toast one slice in under 2 milliseconds. (Louis likes his toast black.) Louis decides to use RPC to control the Toastac-25. Each toasting request starts a new thread on the server, which cooks the toast and returns an acknowledgement (or perhaps a helpful error code, such as "Malfunction 54"). Here's what the server thread looks like:

```
server thread:
        lock(message_buffer_mutex);
A -->   decode message;
        lock(accelerator_mutex);
        unlock(message_buffer_mutex);
B -->   cook toast;
        lock(message_buffer_mutex);
        put acknowledgement in message buffer;
        send message;
        unlock(accelerator_mutex);
        unlock(message_buffer_mutex);
        exit
```

5a. (*6 points*) To his surprise, the toast server stops cooking toast the first time it is heavily used! What has gone wrong? Refer specifically to the code above.

If two threads, A and B, execute the code above and reach the points indicated, they will ***deadlock****: A has message_buffer_mutex and wants accelerator_mutex, B has accelerator_mutex and wants message_buffer_mutex.*

5b. (*6 points*) Once Louis fixes the multithreaded server, the Toastac gets more use than ever. However, when the Toastac has many simultaneous requests (i.e., there are many threads), he notices system performance degrading badly—much more than he expected. Performance analysis shows that competition for locks is not the problem. What could be going wrong? Explain briefly.

Here are two good answers and some OK answers.

1. ***Context switching.*** *Since there are many threads, the system might be spending most of its time switching between them (swapping page tables, etc.). This was particularly expensive on the PDP-11 (the Therac's control unit).*

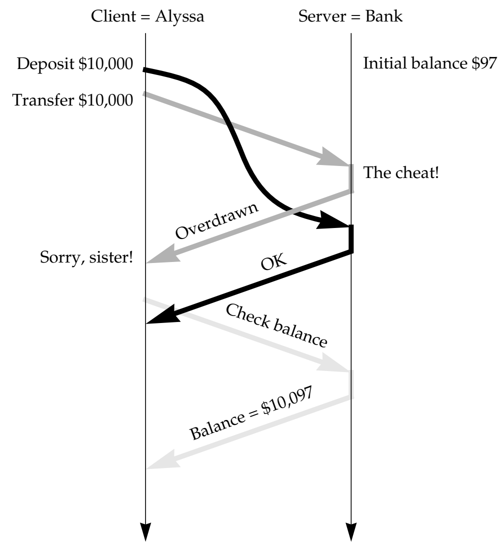
2. ***Network collisions.*** *Assume the Toastac is on an Ethernet. Ethernet performance degrades badly when many packets (i.e., toast requests) are being sent simultaneously.*

The OK answers included network latency, not enough server computing power, etc. Presumably Louis would have "expected" these.

Several incorrect answers implicated competition for locks, which we specifically said wasn't the problem. For instance, the problem was not that the Toastac can only cook one slice at a time, since that boils down to competing for locks.

4d. (*6 points*) Alyssa is so pleased with the performance of this last design that she decides to use it (without grouping) for a banking system. The server maintains a set of accounts and processes requests to debit and credit accounts (i.e., modify account balances). One day Alyssa deposits $10,000 and transfers it to Ben's account immediately afterwards. The transfer fails with a reply saying she is overdrawn. But when she checks her balance afterwards, the $10,000 is there! Draw a time diagram explaining these events.

Referring to the figure, 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.

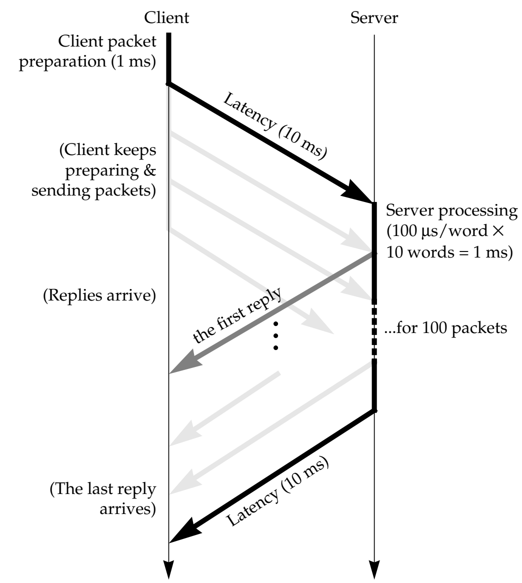


4c. (*6 points*) Alyssa decides that grouping words still isn't fast enough, so she wants to know how long it would take if she used asynchronous messaging (with grouping words) instead of RPC. How long will it take to spell check the same file? (She is careful to resend packets that are not acknowledged but, for your calculations, you may assume that packets are not lost or reordered.)

With asynchronous messaging, the client can continue immediately after it sends a packet. 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 latency for each packet. For a grouping of 10 words/packet, both the client preparation time and server processing time are 1 ms: both are the bottleneck (so having a multiprocessing server won't help very much). The first and last packets are special cases which add to this time.

Time for client to prepare first request + latency from first request + server processing time for 100 packets + latency from last reply

= 1 ms + 10 ms + 100 packets × 1 ms/packet + 10 ms = 0.121 seconds!



4. (*22 points*) Alyssa P. Hacker is implementing a client/server spell-checker. The client scans an ASCII file, sending each word to the server in a separate message. The server checks each word against its database of correctly spelled words and returns a one-bit answer. The client displays the list of incorrectly spelled words.

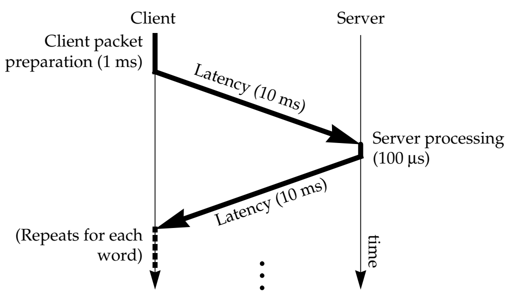
4a. (*5 points*) The client's cost for preparing a message to be sent is 1 millisecond, regardless of length. The network latency is 10 milliseconds, and network bandwidth is infinite. The server can look up a word and determine whether or not it is mispelled in 100 microseconds. Since the server runs on a supercomputer, its cost for preparing a message to be sent is zero milliseconds. Both the client and server can receive packets with no overhead. How long will Alyssa's design take to spell check a 1,000 word file if she uses RPC for communication (ignore acknowldgements to requests and replies, and assume that packets are not lost or reordered)?

First off, the latency. "Network latency" means one-way latency. However, you didn't lose points if you explicitly assumed 10 ms was the round-trip latency.

For RPC, each call blocks at the client until it's completed. All events happen sequentially, so the times for packet preparation, client-to-server latency, server processing, and server-to-client latency can be added directly:

Cost per packet = 1 ms + 10 ms + 0.1 ms + 10 ms = 21.1 ms.

21.1 ms/packet × 1 packet/word × 1000 words = 21.1 seconds.



4b. (*5 points*) Alyssa does the same computations that you did and decides that the design is too slow. She decides to group several words into each request. If she packs 10 words in each request, how long will it take to spell check the same file?

Grouping 10 words per packet cuts down on the number of packets, and hence, the effect of network latency. Note that client preparation time is 1 ms per packet, not word, whereas server processing time is 100 µs per word, not packet. (You could explicitly assume the server runs on a multiprocessor, which would mean checking 10 grouped words takes only 100 µs.)

Cost per packet = 1 ms + 10 ms + 10 words/packet × 0.1 ms/word + 10 ms = 22 ms.

22 ms/packet × 1/10 packets/word × 1000 words = 2.2 seconds.

3. (*21 points*) Mike R. Kernel is designing the Intel P97 computer system, which currently has one page table in hardware. The first tests with this design show excellent performance with one application, but with multiple applications, performance is horrible. Suggest 3 design changes to Mike's system that would improve performance with multiple applications, and explain your choices briefly. You cannot change processor speed, but any other aspect of the system, from disk to MMU design, is fair game.

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

1. Many page faults, because the total physical memory requirements of the multiple applications exceed the amount of physical memory available. Page faults involve disk accesses, which take a long time (about 10 ms).

2. Context switching overhead, since to run all the applications the operating system must context switch between them. This 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 3 main categories.

Improve hardware:

- *Buy more RAM (reduces number of page faults)*
- *Buy a faster disk (reduces 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 table pointer (page map address register) to switch between multiple hardware or physical memory 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 save recently used disk blocks*

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 algorithm (perhaps application-specific)*
- *Implement the working set model (prefetch pages, etc.)*
- *Add a TLB to improve address translation time*

Improve the thread scheduler:

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

Other:

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

2d. (*5 points*) How many interconnections are there now? Include module-to-module and submodule-to-submodule interconnections.

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 $4 \times 3 = 12$.

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/main module × 3 calls/ 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.*

2e. (*6 points*) Was using hierarchy a good decision? Why or why not?

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 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 new instructions absorb cycles, so performance usually goes 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 will significantly 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, but at first glance, the benefits of the "pros" seem worth the cost of the "cons", so this was probably a good decision.

(Obviously, most accepted answers were much shorter.)

2. (*18 points*) Ben Bitdiddle has written a program with 16 major modules of code. (Each module contains several procedures.) In the first implementation of his program, he finds that each module contains at least one call to every other module. Each module contains 100 lines of code.

2a. (*1 point*) How long is Ben's program in lines of code?

16 modules × 100 lines/module = 1600 lines.

2b. (*4 points*) How many module interconnections are there in his implementation? (Each call from one module to another is an interconnection.)

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 $16 \times 15 = 240$.

Ben decides to change the implementation. Now there are 4 main modules, each containing 4 submodules in a one-level hierarchy. The 4 main modules each have calls to all the other main modules, and within each main module, the 4 submodules each have calls to one another. There are still 100 lines of code per submodule, but each main module needs 100 lines of management code.

2c. (*2 points*) How long is Ben's program now?

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

1. (*15 points*) We've discussed 4 principles for dealing with complexity: modularity, abstraction, hierarchy, and level definition. For each system organization below, choose *the most important* principle it uses or implements that makes it effective, and explain *briefly* why the system organization is an example of that principle. *An answer without explanation will receive no credit.* If you think two or more principles are equally important, choose one, and explain it well.

1a. *(3 points)* Protocol stack

Level definition. *Levels (or layers) transform one set of functionalities into another. Levels also interact only with adjacent levels. Protocol stacks are organized in exactly this manner: each layer transforms the underlying layer's functionality into something new. Note that levels are more than simple modularity.*

1b. *(3 points)* Client/server

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

1c. *(3 points)* Virtual memory

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 isolates processes from one another by giving them different and protected address spaces, thereby minimizing the chance that processes can harm one another.*

1d. *(3 points)* RPC

Abstraction. *RPC makes client/server computing look just like local computing by hiding it beneath a procedure call, hiding the communication mechanisms from the user.*

1e. *(3 points)* Microkernel

Modularity. *A microkernel partitions an operating system's functionality into isolated user-level modules (like the file service, etc.), thereby minimizing and making well-defined the interactions between the partitions.*

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.033—Computer System Engineering Handout 13—March 12, 1996

Quiz 1 Solutions

This quiz contains five problems whose relative weights are given below. Please write your answers in the space provided, and hand in this booklet at the end of the examination.

Put your name on this cover sheet AND at the bottom of each page of this booklet. Be sure that you don't put part of the answer to a problem on the back of a sheet for another problem.

Some parts of some problems may be much harder than others. Read them all through first and attack them in the order that allows you to make the most progress. Some problems are intentionally ambiguous; be sure to write down any assumptions you make. Show your work, or you risk losing partial credit. Be neat. If we can't figure out your answer we can't give you credit. On answers that involve numbers, be sure to clearly specify the units of your answer.

Circle your recitation:

10:00	1. Shrira/Litt			
11:00	7. Corbató/Fu	5. Ganger/Yerushalmi	12. Shrira/Kohler	2. Wetherall/Legedza
12:00	6. Wetherall/Yerushalmi			
1:00	3. Agarwal/Legedza	11. Corbató/Litt	9. Ganger/Fu	8. Saltzer/Engler
2:00	4. Agarwal/Kohler	10. Saltzer/Yerushalmi		

For Official Use Only

Problem	Your Score	Maximum Score
1		15
2		18
3		21
4		22
5		24
Total		100

Name: *Nancy Drew, 6.033 Cluemaster*