DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.033—Computer System Engineering Handout #9—February 24, 1999

Old Quiz 1 Solutions — 1994 to 1998

This document contains all of the solutions to the old Quiz 1's from 1994 to 1998. The solutions are not in order, to motivate you to solve the problems before looking over the solutions. Some of the provided answers are more verbose than were required, to cover all possible right answers. These are usually noted.

The wide range of question styles is not completely accurate. True/False and multiple choice questions are very likely, and will be complemented by a set of short answer or essay questions.

Good luck on the quiz Friday. If you have been missing everything, here are the details given on the quiz announcement:

1. Quiz 1 will be from 2-3pm on Friday, March 5, 1999.

2. The quiz will be held on the third floor of Walker for students with last names from A-R, and in 34-101 for those with names S-Z.

3. The quiz will cover all the material up to and including the March 2nd recitation.

4. There will be a quiz review in 34-101 from 7-9pm, Tuesday, March 2, run by the TAs, during which they will go over an outline of the covered subjects and explain a few questions from the practice quizzes.

5. Yes, the quiz is open-book, open-notes.

6. One question from the practice quizzes will be on the quiz.

 Name ____________________

*. A reconfiguration might occur after the sender launches a packet, but before the Autonet delivers it. Similarly, a hardware fault may occur along the path just as the packet is coming by.

*. The target host, which is required to accept the packet whether it is prepared to do so or not, may discard it for lack of buffers.

*. 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/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.

*. The Autonet hardware or software may experience a fault that results in the packet being destroyed or damaged so badly that it isn't clear where to deliver it any more.

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

*. The alternative, in which the server 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 server 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.

*. FALSE
When a microkernel is used, there will be more context switches per unit time than with a monolithic kernel. If anything, this would lead one to expect more pages in use, more frequent TLB invalidations, and overall more, rather than less, load on the TLB.

*. FALSE.
No, mutual friends would make it more of a "web" or a "complete graph".

*. FALSE.
They DID reuse software from the previous machines, but removed hardware interlocks.

*. FALSE.
An incorrect answer from the server would still affect the client. Firewalls attempt to keep such effects to a minimum.

*. TRUE.
They keep bugs within a module/abstraction layer, and thus control propogation.

*. FALSE.
Isochronous networks have equal delay by definition. (ISO=equal, chronous=time)

*. 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.

*. FALSE
Hierarchy has no direct bearing on module size. Instead, it provides structure for the interconnections between modules. Its primary impact on complexity comes from constructing stable subassemblies and thus reducing the number of modules that interact at each assembly point.

*. FALSE
Modularity has no effect on incommensurate scaling. Its impact on complexity comes from reducing the number of interconnections.

*. FALSE.
The paper was on building complex systems, anticipating that failures will occur, and how to cope with problems.

*. TRUE
The speed of the components also increases with the inverse of feature size. Assuming that computation capacity is roughly measured by the product of number of components and their speed, the computation capacity then increases with the inverse cube of feature size.

One can plausibly argue that it is hard to actually realize this increased capacity, since more components will require more interconnection and coordination overhead.

*. 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.

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

*. 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.

*.

1. The client could cache the result of a request to the server, on the chance that a writer will tend to use the same style patterns over and over.

2. 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 packets and then waits till all the responses have returned.

3. If available, use a non-blocking RPC, and work on the next sentence while awaiting results on the previous sentence.

4. Use multiple processes (as contrasted with threads) in conjunction with shared memory to coordinate which process does which sentences.

5. Buy a faster processor for the client.

*. (480 bytes)/(0.1 seconds) = 4,800 bytes/second

*. The formula is 20 + n * 10-4 msec
Other formulas which define M and K as 220 and 210 respectively also received full credit.

*. 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 ms.
                     480/.12 = 4,000 bytes/second
```

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

Note that you 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.

*. 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 ¥ 15 = 240.

*. The time for a 1-Kbyte page = 20.1 msec
The time for a 10-Kbyte page = 21.0 msec

*. If we send 100 blocks, an average of 50 of them will require 100 ms., and the other 50 will require 200 ms. Total time for 100 blocks is

```
50 x 100 ms = 5 sec
50 x 200 ms = 10 sec
              15 sec, or 150 ms/block
(480 bytes)/(.150 sec) = 3200 bytes/second
```

Again, we must average times, not data rates.

*. 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 rerespond without interfering with useful activity. Rate = 3200 bytes/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 ms, or 250 ms, or 400 ms, or 550 ms., 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 ms.
There's a 50% chance a retry will be started after 150 ms.
The retry has the same expected delay, starting from when it was sent.
Therefore, working in milliseconds:

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

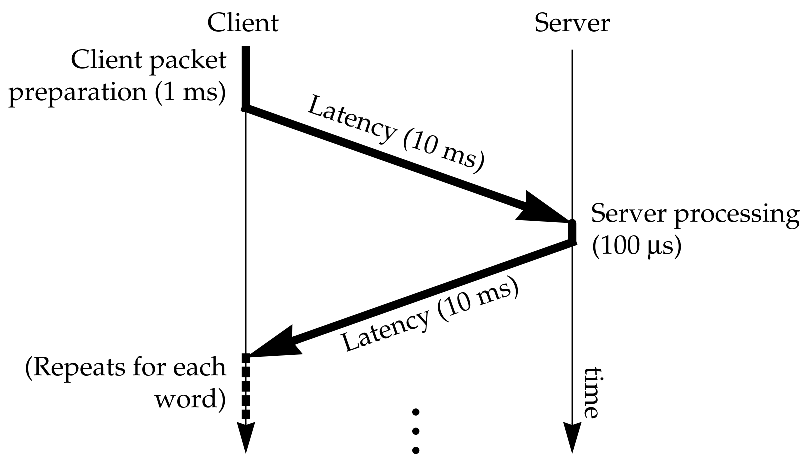
Solving for t:

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

The data rate is (480 bytes)/(0.25 sec) = 1920 bytes/sec

*. The data rate of the slowest link is one of the things that contributes to the round trip latency, so it is already included in the given parameters.

*. 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.



*. 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.)

*.Ben might better off using smaller page size, because with smaller pages more pages fit in memory. If multiple applications are using only a small parts of a page, then fewer page faults will be incurred and less memory will be wasted than with using big pages.

The cost for doing the LRPC from the kernel to a trusted server is neglible compared to the time for dealing with a page fault. Therefore, the VM system might as well be implemented in a trusted server to improve modularity of the system.

*. 16 modules ¥ 100 lines/module = 1600 lines.
*. Backing up the saved program counter on a supervisor call.
*. Exhibit a property called "near decomposability".
*. Fast typing by the operator.
*. The demotion of the window manager to ordinary-program status.
*. The buffering demanded of a network node exceeds its capacity.
*. Do not use faster client machines.
*. Do not use multiple client threads (each making the Balance request) on each client.
*. Use a faster server machine.
*. Use faster network technology.

*. To fix the problem we need to make sure that the shared variable packet_here can't be changed between the time that Wait_For_Packet tests it and Wait_For_Packet reaches its wait statement. Wait_For_Packet has done its part--it has seized lock m, and it has told Thread.Wait to release lock m atomically as part of going into the wait state. But Packet_Arrived isn't respecting the lock protocol. (Eraser would have a fit.)

The minimal repair, then is to replace line 7 with the following three lines

```
LOCK m DO                          7a
    packet_here := TRUE;           7b
END                                7c
```

In terms of the answer to part a, this change guarantees that line 7 must be executed either before line 14 or after Thread.Wait releases the lock in line 15.

Note that although it wouldn't hurt anything, it is not necessary to continue to hold the lock while calling Thread.Broadcast. The simple explanation is that surrounding line 7 with the same mutual exclusion lock that protects lines 13 through 15 is sufficient to eliminate the bad execution ordering.

An interesting exercise to explore is whether or not placing the lock only around line 8 would also be sufficient.

*. Wait-forever can occur if Thread number one is in Wait_For_Packet, and immediately after it executes line 14, Thread number two executes all of Packet_Arrived. The trouble is that after Thread number one has convinced itself that packet_here is FALSE (in line 14), packet_here becomes TRUE (line 8) and the condition gets signalled (line 9). Since thread one is still running, it misses the signal; it then Waits forever (line 15) for the signal that it missed.

The minimal sequence that leads to an endless wait is 14-7-8-15 .

*. Call an RPC passing as an argument the serial number that was sent on the last pipe call, and design the remote procedure called to not return

until a pipe call with a given serial number had been processed.

*. Are an essential part of the 6.033 educational experience, and participation in recitation discussions contributes to your final grade.

*. Were dependent on timing.

*. Let newly created processes inherit pipes to facilitate inter-process communication.

*. May miss synchronization errors.

*. A remote procedure call will always be executed at most once, which includes the chance that the call may not be executed at all or may be partially executed.

*. We accepted two answers for this question. The answer we expected was

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

Many students interpreted “transmission” in the dictionary sense of moving data from sender to receiver, rather than in the correct technical sense of sending only, giving

$$\begin{aligned}&\text{transmission delay + communication latency}\\ &= 267 ¥ 10^{-6}\text{sec (as above)} + 12.5 ¥ 10^{-3}\text{sec} = 12.767 ¥ 10^{-3}\text{sec}\end{aligned}$$

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

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

Inverting this gives a sustained maximum rate of 38.9 RPC/sec.

*. We accepted two answers for this question. The answer we expected was based on the assertion in the problem that “pipe calls return immediately to the caller.” If this is so, then the number of calls is limited only by the client computation time between calls:

It was also reasonable to assume, since the problem mentioned the application runs in a single thread, that the thread making the pipe calls must spend time sending the 1000 byte pipe request message after

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

each 96 pipe calls. The total time for 96 calls is thus

$$96 \times (100 \times 10^{-6}\text{sec}) + \frac{1000 \text{ bytes} \times 8 \text{ bits/byte}}{1.5 \times 10^{6} \text{ bits/sec}} = 14.93 \times 10^{-3}\text{sec}$$

giving a sustained maximum rate of

$$\frac{96}{14.93 \times 10^{-3}\text{sec}} = 6428.6 \text{ calls/sec}$$

*. Because the client is single-threaded, 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. (One ought to check ConfirmTrade for potential locks, but we did not ask for it.)

*. Under these circumstances, the body of Trade should be made atomic, e.g. via lock/unlock pair surrounding the critical 3 lines. Preferably, each client would use a separate lock to minimize contention.

*. Trade requires at-most-once, since it causes a change in server state. Since Balance doesn't change state, at-most-once is not needed for it.

*. Add a sequence number, supplied by the client, to the request. The client increments sequence number on new requests but not on retransmission, and the server keeps a table of past responses which it uses for responses to subsequent requests with identical sequence numbers.

*. Yes, it is appropriate for a connectionless network model; each request is completely self-contained and doesn't depend on a "connection" for its context.

*. The latter approach would cause the client to hang forever even on transient errors (lost packets); the timeout code allows the client to dictate a recovery procedure.

*. The degradation is due to thread overhead (eg, costs associated with thread creation, deactivation, and context switching).

*. 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.

*. Don't use threads for Balance requests, but use them for other requests.

*. The latter approach would cause the client to hang forever on a permanent failure (eg, server crash); the timeout code allows the client to dictate a recovery procedure.

*. 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.
[Note that this doesn't eliminate the need for at-most-onceserver semantics, since the server may still see duplicate requests...]

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

*. 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.

*. 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 ms per word, not packet. (You could explicitly assume the server runs on a multiprocessor, which would mean checking 10 grouped words takes only 100 ms.)
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.

*. 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.

*. 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.

*. 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.

*. 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.

*. No, because there is no possibility that two or more threads each holds a lock at the same time time that another threads needs. Call the code fragments in the problem "our_func()". If our_func() gets one 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 "our_func()"). If thread C gets the lock first, our_func() 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 gotten). 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 our_func() at once.

*. Yes. Deadlock can occur if thread A acquires mutex_a at the beginning of the function and thread B acquires mutex_b in the middle. Both will hold a lock needed by the other thread, so neither will make forward progress. Neither lock will ever be released, so the threads are deadlocked.

*. 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.

*. 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)

*. 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.

*. 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.

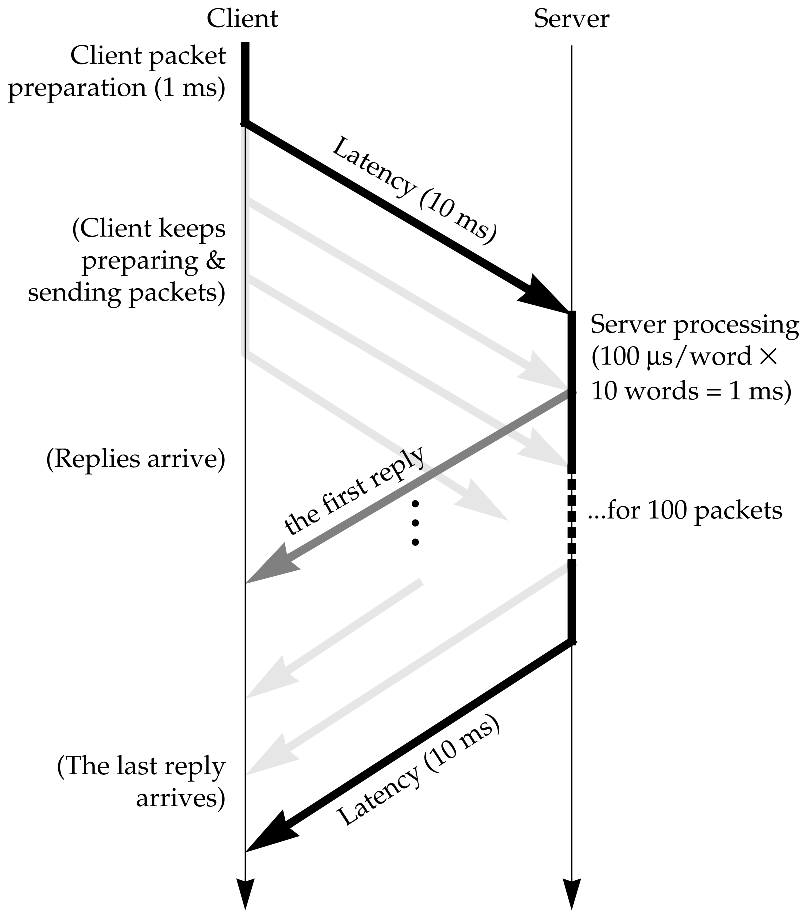
*. All workloads will be sped up some by the asynchronous protocol, but grouping will cause some workloads—the bursty workloads—to be sped up much more.

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.

*. 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!


Client
Server
Client packet preparation (1 ms)
Latency (10 ms)
(Client keeps preparing & sending packets)
Server processing (100 µs/word × 10 words = 1 ms)
(Replies arrive)
the first reply
...for 100 packets
(The last reply arrives)
Latency (10 ms)

*. Some likely candidates:

1. On every kernel trap it will be necessary for the kernel to save and restore all the extra 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 be significant.

2. 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.

3. 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.

*. 50 threads [= (1msec * 100 MIPS) / 2000 instructions]
Adding one more (51) was acceptable -- most who did this added a bonus thread for a scheduling role, and mentioned that.

*. 40 threads [= (1msec * 100 MIPS) / (2000 + 500 instructions) Again, a bonus thread (41) for overhead was acceptable if qualified. Assuming that a context switch requires 2C=1000 total instructions (ie., to get in, and get out) gives you 33.33333 threads. A few people did this, and it's a reasonable alternate answer.

*. Within a WWW browser, application threads will tend to be interdependent (perhaps blocking on disk, network, or display), and the application i/o will tend to be bursty. Any "average" latencies are thus subject to wide variance, and the browsing process will afford many idle moments.

*. A nice bullet list of general system vicissitudes was fine; e.g.,
- paging
- multiprocessing overhead, sub-optimal scheduling
- i/o bursts in audio visual subsystems
- network delays (percolation through internet, load on remote servers)

*. To "execute the RPC on the current running thread" must mean that the system pre-empts the currently running thread from whatever it was doing and puts it to work on the newly arrived RPC request. If the pre-empted thread happened to be holding a lock at the instant of pre-emption, and the new RPC request requires it to try to seize that same lock, the thread will deadlock on itself. More complicated scenarios involving other threads are also possible, but they all involve the new RPC request somehow waiting for release of the lock held by the pre-empted thread.

*. The solution to this problem depends on the assumption whether a thread can overlap communication with computation. If it can, the solution as handed out is correct.
If computation and communication are not overlapped, the right answer is 51

instead of 50. If communication and computation are not overlapped one needs three threads instead of two in the following simple case (1 millisecond computes, followed by a 2 millisecond latency before the reply from the WWW is received):

Consider a time line:

```
Thread 1      *****----------*****---------- ...
Thread 2            *****----------****----- ...
Thread 3                  *****----------***** ...

              Time -->
```

Each character is 0.1msec, asterisk represents a busy 0.1msec, each dash represents a thread waiting for a reply, each line is a thread. From the diagram, we can see that three threads are needed. If, however, communication and computation are overlapped then we need only two threads.

*. The latency introduced by RPC comprises network transit times, time in the client stub that marshalls arguments, time in the server stub that unpacks the arguments, time in the network layers that add and remove headers, and time in context switches in both the client and the server. 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-induced latency.)

*. The timing diagram below shows several things that could produce this effect: the network may duplicate the request or the network may swallow or delay the server's response to the request. In either of the latter two cases, the client will timeout and resend the request, causing it to arrive

`twice at the server.`