CS414 Fall 2004 Homework 7. Solutions

1. Suppose that you have written a program that needs to transfer small files from a nearby server. You do a test and discover that for a 1GB transfer, the network connection runs at 100MB per second. But when you transfer your files (using a separate TCP connection for each file), the file transfer speed is only 4MB per second. Explain where the factor of 25 was lost.

   TCP runs in a "slow start" mode for a short period after it has been idle, including right after a new connection is established. In this situation, the entire file was probably sent before TCP was able to crank the transmission rate up to anything close to the limit.

2. In class, we learned that the TCP protocol uses a sliding window algorithm for two reasons. First, this allows it to overcome packet loss. But the algorithm also permits the sender to match its rate to what the network can accommodate, and also to the rate of the receiver. Specifically, TCP operates by increasing its sending rate until packet loss is detected, then reducing the rate. It does this by making the sliding window size larger, or smaller, respectively.

   a) Suppose that you were to use TCP over a wireless network, in which packet loss occurs frequently. How will TCP behave?

      Packet loss can fool TCP into throttling back by reducing its window size. It will run in "slow motion" at a rate much lower than the speed of the link.

   b) Is the behavior you described in (a) the right reaction to packet loss in a wireless network?

      Nope. A smart move would be to retransmit any lost packets very aggressively, and basically blast the lossy link with data. TCP is doing the opposite of what it should, because it was optimized for the Internet (where the main cause of packet loss is overload, not a wireless link).

   c) Now suppose that you are designing a large corporate network with mostly wired links, but some wireless links to mobile machines "at the edge." What might you consider doing to avoid the problem you described in part (a)?

      If it were me, I would put a special purpose "tunnel" protocol in between the wireless device and the wired network, and have that protocol intercept the IP traffic and retransmit lost data very aggressively. This would effectively hide the lossy link.

   d) [Extra credit*] Read about the "End to End Argument". (J. Saltzer, D. Reed and D. Clark: "End-to-end arguments in system design", ACM Trans. Comp.

Sys., 2(4):277-88, Nov. 1984). Does your answer to part (c) violate the principle advocated in this paper? Explain.

The paper talks about a general strategy and argues that the only place where link-level recovery matters is on a lossy link. But here we have a lossy link.

3. A computer system consists of five machines, named M0 … M4. Each machine has an attached input device, and each reads a single bit from its device (hence, 0 or 1).

   a) Assuming that no failures occur and that messages are delivered promptly and reliably, design a protocol whereby the machines can "vote" for its value (0 or 1). Your protocol should pick a value that at least one machine actually voted for (you can't just say "always pick 0"), Each machine must know the outcome of the election at the end (you can't just say "process 0 now picks the first value it received" – all need to know, and all need the same value). Design your protocol to run in "rounds", such that in each round, each machine sends four messages (one to each of its counterparts), and receives four (one from each of its counterparts). Assume that no failures occur.

   In the initial round, call it "round 1", the protocol process on machine i sends its value to all the other machines, and receives an input from each other machine. In round 2, every process has identical data: all the votes. Pick the majority. The outcome is the value we picked (note that every machine will pick the same value).

   b) Modify your protocol from (a) to work correctly even if a single machine fails while the protocol is running (you may assume that the faulty machine crashes, halting at some step in the protocol and that this type of failure is "reported" to each operational machine, in the form of a special message from the network that says "process 2 has failed")

   Round 1 is the same: every machine sends its value to every other machine. In round 2, however, we may start with some machines having five values and some having four, since a machine might have crashed, and there is also a possibility that everyone has four values (if the crash happened very early in round 1). So: in round 2, every machine sends its full set of votes to every other machine.

   Now think about the state at the beginning of round . If every machine was healthy in round 1, everyone has 5 votes, received in round 2. If some machine crashed in round 1, but anyone had all 5 votes, it will have relayed them in round 2, so now everyone has all 5 votes (since there is only one crash, we don't have to worry about a crash in both rounds 1 and 2). So: if you have 5 votes, pick the majority. If, in round 3, you only have four votes, then <u>everyone</u> only has four votes. In this case, pick the majority and if there is a tie, pick 0.

   c) Modify your protocol from (a) to work correctly even if some machine might behave maliciously, for example by ignoring some messages, telling one machine that its input was a 0 and another that its input was a 1, etc. It cannot, however, impersonate some other machine – and more generally, you may assume that the faulty machine cannot prevent the correct machines from running rounds, or tamper with messages sent from one correct machine to another correct machine.

With "evil" processes, things get more interesting! By the way, this is called a "Byzantine Agreement" problem. The general version of this was a major area of research for almost a decade, starting in the late 1970's. But when we narrow it down to 5 processes, the whole problem is simplified.

Our solution will run in three phases.

Phase 1: Each process sends to the other processes "My name is Mi and my vote is v", where i is the id of the process (the machine), and v is the value it read from its input device. By the end of phase 1, each process has either 4 or 5 of these messages, depending on whether or not there is a faulty process and, if there is, how it behaved.

Phase 2: Now each process sends to the other processes: "My name is Mi and during phase 1, I received votes {v0, …, v4}"

Phase 3 now starts. We'll say that a vote v from process Mi is "supported" at process j if process j received three witness messages for that vote. A process can't witness its own vote – a witness needs to be some other process. If a vote isn't supported, we'll say it is "unsupported".

Let's think for a moment about the correct processes, and completely ignore the impact of the faulty process (if any). In fact, let's go further – without loss of generality, we'll assume that process M4 was faulty. Let's erase every message sent by M4 and ignore all the witness stuff about M4.

Now, the other four processes are definitely correct. Notice that any vote from a correct process will be supported at all the other correct processes. This is because each correct process sent its vote (correctly) in phase 1, and hence it was received by the three other correct processes. In phase 2, they all will have sent witness messages about that vote. So at the start of phase 3, any correct process is in possession of four supported votes from correct processes.

There are five possible "cases" to consider {0,0,0,0}, {0,0,0,1}, {0,0,1,1}, {0,1,1,1}, {1,1,1,1}. Let's say that in the first three cases we pick 0 as the outcome, and in the last two we'll pick 1.

Now let's ask what nasty tricks M4, our traitor, is able to play. First, remember that a process isn't allowed to witness its own vote. If M4 wants its vote to be supported, it needs to send that vote to three or four correct processes.

So M4 is in a bit of a corner. It can either fool around, and have its votes completely ignored, or it can send some vote to three or four correct processes. Doing so will get its vote added to the list, but all the correct processes will end up with the same list! We now get the following cases: {0,0,0,0,0}, {0,0,0,0,1}, {0,0,0,1,1}, {0,0,1,1,1}, {0,1,1,1,1},{1,1,1,1,1}. So, let's say that in the first 3 cases we'll pick 0, and in the last three we'll pick 1.

Since we never pick a value unless it was voted for by at least two processes, we've satisfied our non-triviality requirement. And because M4 can't witness its own vote, there isn't any way that it can cause some processes to end up with 4 votes in their list of supported votes, and some to end up with 5. The first four elements of that list are going to be there no matter what: they came from correct processes. As for the fifth element, well, M4 can cause its own vote to be included in every list, or excluded from every list, but that's the limit of its powers.

d) Can the problem from part (c) be solved with just three machines instead of five? Either explain how to do it, or explain briefly why this isn't possible.

The problem can't be solved with just three processes, one of which might be faulty. This can be proved formally, but the brief answer centers on the "way" that part (c) solved the problem: by overwhelming the faulty process using three correct processes. Our protocol works by exploiting the indirect observations of a large number of witness processes. We can't do that with just three processes – if M0 is correct, it knows that one of M1 and M2 is correct, but has no way to figure out which one to trust. The formal proof works by pretending that there is a way to solve the problem, and then showing that this leads to a contradiction – specifically, to a case in which both correct processes vote 0, but M2 tricks one of them into picking 1, while the other picks 0.

If you would like to see how this is done, the reference is M. Fischer, N. Lynch, and M. Merrit. Easy impossibility proofs for the distributed consensus problem. Distributed Computing, 1(1):26--39, 1986. It works like this. They imagine a system with process A/0, C/0 and B/1 in which B is faulty (my notation means that A is running with input 0, etc). Under these conditions, A and C need to agree on 0 (obviously) no matter what messages B/1 sends. You can draw a picture of a triangle if you like. Imagine that you made a note of what messages are traveling over the connections – what A sends to B, what C send to A, etc.

Now imagine a system C/1 B/1 A/0 in which A is faulty. C and B need to reach agreement (on 1).

Finally, set up a system A/0 B/1 C. On its connection to A, C "behaves" like it did in the first run, when it had input 0. This looks identical to scenario 1 in A's eyes, so A picks 0. On its connection to B, C "behaves" like it did in the second scenario, when it was a correct process with input 1. B picks 1. So: C was faulty, and it tricked A *and B into picking different values! This proves, quite formally, that with three processes and one Byzantine fault, it is impossible to achieve agreement. You can generalize it to show that with f faulty processes you need at least 3f+1 processes in total to overcome such faults.*

4. Suppose that a program has a virtual address space of size 20MB and is running on a computer with 2MB of memory (so: if you run *k* instances of the program, the VM load on the computer would be 20*k* MB, but the computer would still have 2MB of physical memory; for this problem *k* is equal to 1). You would like to speed up the program, so you begin to add memory. At first, each time you add 1MB of memory you find that the program indeed speeds up. But after the physical memory size reaches 10MB, adding memory stops having any effect.

   a) Explain why extra memory stops helping, even though the virtual memory of the program is twice as large as the physical memory of the computer.

   The performance bottleneck apparently has shifted from paging to something else. For example, perhaps we were spending a noticeable amount of time paging, but now paging is no longer a noticeable overhead and the CPU speed dominates. The "working set" apparently fits within 10MB.

   b) Your close friend Doug "The Bug" Crump drops by and shows you that inside your computer, the "motherboard" has a speed setting. By changing it, he is able to double the speed of the CPU, without affecting the speed of anything else. To your surprise, you now find that adding physical memory now "helps" again, and you can obtain a further speedup. Explain why this happened. (Hint: This is actually not at all easy. If nobody gets it, Ken won't be surprised. Second hint: if you try this, don't be surprised if your computer overheats and melts down!).

   Having eliminated the CPU bottleneck for a few days (since that CPU will probably fry, but perhaps not right away), the VM subsystem again is the limiting factor. But the extra memory will soon have us back to an in-memory working set and once we get there, the CPU will again be the bottleneck.

5. When modifying the Linux kernel late one night, Doug accidentally modifies the context switching software so that sometimes, when context switching from process P0 to process P1, the system forgets to flush the TLB and main-memory (L2) cache.

   a) What sorts of problems might result from this mistake?

   In such a situation, data from process P0 might be "visible" to process P1. For example, the page that process P0 had at location 1000 might show up in the address space of P1, or actual data from a memory location in P0 would turn up in the address space of P1. The applications will crash.

b) Doug realizes what he has done, but in fixing it, modifies Linux to flush the TLB and cache not just on every context switch operation, but also on every interrupt (including page faults, system calls, device interrupts, etc). What impact would you expect this to have on the system?

This is likely to make the system slow, since it will run with a cold cache.