15-440 Distributed Systems
Midterm

Name:
Andrew: ID

December 8, 2009

- Please write your name and Andrew ID above before starting this exam.
- This exam has 13 pages, including this title page. Please confirm that all pages are present.
- This exam has a total of 100 points.

Question	Points	Score
1	3	
2	3	
3	3	
4	4	
5	3	
6	3	
7	2	
8	3	
9	24	
10	28	
11	24	
Total:	100	

A Short Answers

1. (3 points) RPC tries to make remote procedure calls look the same as local procedure calls. But the illusion isn't perfect. Circle all of the options that correctly describe differences between a local function call and an RPC:

   A. RPC calls require an extra parameter to identify the server.

   B. RPC calls may have higher or variable latency.

   C. RPC calls are limited to call-by-value.

   D. RPC calls have different (more) failure modes.

   E. Local function calls require a `malloc` of a heap object to represent the return pointer.

   Solution: A,B,D

2. (3 points) Why does the condition variable call to wait take a mutex as an argument?

   Solution: The call to wait accepts a mutex so that it can atomically lock the mutex and put the thread to sleep, because otherwise we could have a lost wakeup.

3. (3 points) Recall that NFS uses a timeout-based mechanism to provide weak consistency, where AFS uses a callback mechanism to provide close-to-open consistency. Assume a file server has 100 clients. These clients have all opened the same file read-only. They have no other open files. In this scenario, would an NFS server or an AFS server experience lower request load? Why (briefly)?

   Solution: In AFS, clients that write to the file cause the server to initiate a callback to all clients. If most clients are opening the file as read-only, *only* writes to the file cause the server to have to do work. NFS in comparison would require all clients, even those that are only reading a file, to periodically check the server for updated status.

4. (4 points) Using the 5 year-old single-disk computer in his office, Prof. Evil von Ahn repeatedly measured the read throughput of a 1 GB file on two systems: an unknown remote file server in the SCS machine room, and the local disk in Prof. Evil's computer itself. The throughput from the remote server was 12 times higher than Prof. Evil's local disk.

   Of the possible differences between the unknown remote server and Prof. Evil's computer and the trends in computing hardware, what factors *alone* could likely account for this performance difference? (Circle all correct answers.)

   A. The remote server disk(s) spin 12 times faster than the disk in Prof. Evil's computer.

   B. The remote server uses RAID 0 and has many disks.

   C. The network bandwidth is 12 times higher than the disk bus in Prof. Evil's computer.

   D. The remote server uses RAID 5 and has many disks.

   E. The remote server serves the file from a memory cache while Prof. Evil's computer repeatedly serves the file from disk.

   F. Prof. Evil runs Windows.

   G. The remote server is an AFS server.

   H. The remote server is a Sun NFS server.

Solution: B, D, and E are correct. A and C are technologically infeasible: no commercial server disks spin 12 times faster than even the slowest commercial disks five years ago, and modern deployed networking systems do not approach 12 times the speed of modern computer data buses. (Also, the network traffic in a typical computer also must use the same data bus as disk traffic.) Windows, AFS, and NFS alone do not significantly affect disk performance without other major architectural differences (such as RAID).

5. (3 points) Name 3 reasons why you would use threads instead of processes for a problem requiring concurrency (doing multiple tasks in parallel).

Solution: 1. Switching between processes is much more expensive than switching between threads. 2. Threads share memory, so they can coordinate more effectively regarding their tasks. 3. Threads are easy to create and destroy.

6. (3 points) List and describe the three characteristics that a solution to a synchronization (mutual exclusion) problem must have.

Solution: 1. mutual exclusion - only one process in critical section at a time 2. progress - don't wait for an available resource 3. bounded waiting - can't wait forever

7. (2 points) Describe two disadvantages of a ring-based algorithm (one that passes a token around a ring) for implementing a distributed mutex?

Solution: There could be a very long delay for a large ring. Also quite unreliable because a process could die and it would take a very long time to generate another token. Not necessarily fair.

8. (3 points) For which of the following applications would UDP be preferable over TCP, ignoring annoying concerns such as firewalls that might block some protocols?

A. Streaming a live video over the internet	TCP / UDP
B. Instant messaging/email	TCP / UDP
C. Logging in to your bank website	TCP / UDP
D. Voice over IP	TCP / UDP
E. Large file transfers	TCP / UDP
F. Looking up a very small value from a directory service	TCP / UDP

Solution: UDP, TCP, TCP, UDP, TCP, UDP

B Going RAIDing

9. The growth of disk capacity has been outpacing the growth in speed, both for seek latency and transfer rate. A few years ago, you built a RAID array to reliably store your collection of course notes (you take very detailed notes – many hundreds of gigabytes of them). At the time, you built the system as a RAID5 (rotating parity) array using six drives. You picked a bunch of cheap 120GB drives off of newegg.com. The parameters of the drives are:

Capacity	120GB
Seek latency	7ms
Rotational delay	3ms
Transfer speed	60MB/sec
MTTF	1,000,000 hours

Assume where it matters that the disks are full—your data occupies *all* of the available capacity on the array—and that the parameters are given in SI units.

(a) (4 points) Using RAID5, and ignoring filesystem overhead, etc., what is the usable capacity of your RAID array?

> **Solution:** There are six disks; one is for parity. Therefore, the array can hold: $120GB * 5 = 600GB$

(b) (3 points) How long will it take to write one byte of data to this RAID, assuming you write it to a randomly chosen location in the filesystem?

> **Solution:**
> The transfer time for one byte is roughly negligible (it's 1 byte / $6 * 10^7$ bytes/sec, which is 0.016 μs. The dominating factor, then, is the write latency.
> With RAID5, to update one byte, we either have to read the byte and the parity block and then write both, OR we have to write all blocks. It probably makes more sense to do the read and the write. Therefore, the answer is approx 20ms.
> (We don't want this question to be tricky, so we'll accept any variant of "2 * 10ms", or "one seek plus two rotational delays" == 13ms, or even an expected half rotational delay the first time, etc. All of these are acceptable assumptions depending on how smart the RAID is.)

(c) (4 points) Pittsburgh goes to the superbowl again. You live near Atwood, and you worry that in the ensuing riots, your computer may be destroyed, so you decide to copy all of the data off of your computer to a remote server on the Internet. How quickly can you read all of the data off of the raid in huge sequential blocks to make a copy of it?

> **Solution:**
> RAID5 lets you read all of the data disks in parallel. Therefore, using this RAID, you could get 60MB/sec * 5 = 300MB/sec of big sequential read throughput. $\frac{600GB}{300MB/sec}$ = 2000seconds = 33.3minutes

(d) (4 points) Disaster strikes! While you were copying your data, a power surge blew up one of the disks in the array. Your array is now "degraded." You have a spare disk, and you put it in the array in place of the dead one. How long will the rebuild process take? State any assumptions you make.

Solution: To restore, we can read from all five working disks in parallel and recompute the disk blocks for the 6th disk. Therefore, we can write as fast as the one disk will allow – in other words, the answer to this question is the same as to the previous question: $\frac{120GB}{60MB/s} = 2000\text{seconds} = 33.3\text{minutes}$.

(e) (3 points) It's now 2009, and you have a lot more course notes. You decide to build a new RAID array using modern disks:

Capacity	1000GB
Seek latency	5ms
Rotational delay	2ms
Transfer speed	80MB/sec
MTTF:	1,000,000 hours

You've once again filled up your array of six disks. How long would it take to do a RAID rebuild using this new array?

Solution: $\frac{1000GB}{80MB/sec} = 12,500\text{sec} = 208.3\text{min} = 3.47hours$

(f) (4 points) Assume that disk failure probabilities are completely independent. What is the probability of your RAID array experiencing a second disk failure *during* the rebuild? (Hint: Use the MTTF, and state any assumptions you make.)

Solution: Assume that the disks are operating in the "good" part of the bathtub curve (e.g., they aren't all starting to die together). The MTTF is 1 million hours, so the probability of a failure in 3.47 hours *per disk* is approximately 3.47 / 1,000,000. But the array will die if any of the five disks dies. Because the numbers are small, we can approximate the failure probability as: $\frac{5*3.47}{1{,}000{,}000} = .00001735$ or about a 1 in ten-thousand chance of having the entire array die and lose its data permanently during the rebuild. That's not an astoundingly good failure probability.

(g) (2 points) Do you expect that probability to be higher or lower in reality? Explain your answer briefly (1–2 sentences).

Solution: In reality, the failure probability will be higher. The failure of the first disk *may* indicate some source of correlated failures – perhaps power or environmental, or perhaps the disks are starting to age, or came from a bad batch, etc. It doesn't have to indicate this, but in practice, the likelihood of a second disk failure given a first is often much higher.

C The Rowing Cartographers

10. Adam, Becky, and Cartman decide to try out rowing crew. Unfortunately, there's only one boat left that can seat three people (it leaks a bit), and there are only four oars. A rower must have two oars in order to row, or else the boat will go in circles.

    Adam proposes that to arbitrate access to the oars, they should be placed in the center of the boat, and every rower will follow a simple protocol:

```
while (!at destination) {
  recover_strength();
  grab_one_oar();  /* may block */
  grab_one_oar();  /* may block */
  row();
  row();
  row_your_boat();
  drop_one_oar();  /* will not block */
  drop_one_oar();  /* will not block */
}
```

    (a) (5 points) Explain clearly why this plan will *not* lead to deadlock. Note that a one sentence answer probably isn't enough, but six sentences starts to look like too long an answer. Refer to the characteristics of deadlock and mutual exclusion.

    Solution: Deadlock can occur if all rowers have grabbed one oar and are *all* blocked trying to grab another oar. With 3 rowers (each with two arms) and 4 oars, one rower will always be able to grab the 4th oar, make progress, and release oars for the others to make progress.

(b) (5 points) After rowing down the river, an octopus jumps into the boat in a spare seat. The octopus wants to row too – but it has eight arms. Let's generalize the solution a bit to handle an arbitrary number of rowers, R, where each rower needs a particular number of oars in order to row.

Let's characterize the scenarios by:

- R - the number of rowers
- A - the total number of arms summed across all rowers

For example, the boat with Adam, Becky, Cartman, and Ozzy the Octopus, would have $A = 14$ arms and $R = 4$ rowers. The generalized rowers use this protocol:

```
int n_oars = my_number_of_arms();

while (!at destination) {
  recover_strength();
  for (int i = 0; i < n_oars; i++)
    grab_one_oar();  /* may block */

  row();  row();  row_your_boat();

  for (int i = 0; i < n_oars; i++)
    drop_one_oar();  /* will not block */
}
```

Phrased in terms of A and R, what is the smallest number of oars you need to put in the boat to ensure that deadlock cannot occur? Explain. Your answer must be organized and convincing.

Solution: Deadlock possible when all rowers have 1 oar left to grab and there are no more oars.
A = total arms, R = number of rowers. $A - R$ is the max that would lead to deadlock (each waiting for the last oar), so $A - R + 1$ is the number needed to prevent deadlock.

(c) (10 points) Let's build this code. Provide code for three functions:

```
typedef struct boat { ... } *boat_p;
void boat_init(boat_p b, int n_rowers, int n_oars);
void grab_one_oar(boat_p b);
void drop_one_oar(boat_p b);
```

You may use only mutexes and/or condition variables, NOT semaphores or lower level atomic primitives. Assume that the environment is "error-free" (allocations always succeed, etc.). The mutex and cond var operations are:

```
mutex_init(mutex_t *mp);
mutex_lock(mutex_t *mp);
mutex_unlock(mutex_t *mp);

cond_init(cond_t *cp);
cond_wait(cond_t *cp);
cond_signal(cond_t *cp);
cond_broadcast(cond_t *cp);
```

Declare the struct boat here, and write the init function (on the next page). Assume that n_oars has been set appropriately for the needs of the rowers:

```
typedef struct boat {









} *boat_p;
```

```
void boat_init(boat_p b, int n_rowers, int n_oars) {









}
```

(d) (8 points) Write `grab_one_oar(boat_p)` and `drop_one_oar(boat_p)`:

Solution:

```
typedef struct boat {
  mutex_t lock;
  cond_t cond;
  int num_oars;
} *boat_p;

void boat_init(boat_p b, int n_rowers, int n_oars) {
  mutex_init(&(b->lock))
  cond_init(&(b->cond));
  num_oars = n_oars;
}

void grab_one_oar(boat_p b) {
  mutex_lock(&(b->lock));
  while (b->num_oars == 0) {
    cond_wait(&(b->cond));
  }
  b->num_oars--;
  mutex_unlock(&(b->lock));
}

void drop_one_oar(boat_p b) {
  mutex_lock(&(b->lock));
  b->num_oars++;
  cond_signal(&(b->cond));
  mutex_unlock(&(b->lock));
}
```

D Shamport Clocks

11. Suppose each computer in a distributed system keeps an approximate real-time clock R_i in addition to a Lamport-like clock L_i. A computer's real-time clock R_i is an always-increasing integer (i.e., for any two reads r_1, r_2 of R_i such that $r_1 \rightarrow_i r_2$, we have $r_2 > r_1$), but the real-time clocks at different computers may drift relative to each other (i.e., $R_j - R_i$ is non-constant for $j \neq i$).

    Consider the following modification, *Shamport*, to Lamport's partial-ordering algorithm:

    Rule 1: Before each event at computer i, set $L_i = \min(L_i + 1, R_i)$.

    Rule 2: When sending a message m, apply Rule 1 and include the time L_i as part of the message (i.e. send (m, L_i) instead of just m).

    Rule 3: When receiving a message (m, t) at computer j, set $L_j = \max(L_j, t)$ and then apply Rule 1 before timestamping the message-arrival event.

    Let the Shamport global time of an event e at computer i be $S(e) = L_i(e)$.

    (a) (2 points) In the algorithm above, underline the difference between Shamport and Lamport's algorithm. (Underline as little as possible).

    > **Solution:** Shamport's algorithm uses $L_i = \min(L_i + 1, R_i)$ in Rule 1 instead of $L_i = L_i + 1$. Otherwise the two algorithms are the same.

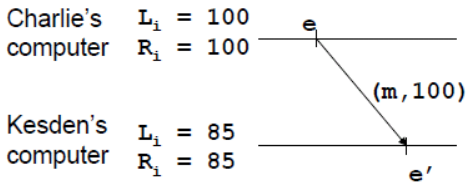
    (b) (8 points) Let e, e' be two events at computer i such that $e \rightarrow_i e'$. Prove that $S(e') > S(e)$.

    > **Solution:** The key observation is that *any* new event advances the logical clock. There are a couple common pitfalls to avoid: (1) e and e' are not necessarily consecutive events; other intervening events can affect the Shamport clock. (2) Events at computer i include the local side of `send` and `receive` events, so you need to include possible applications of Rule 2 and Rule 3 in your proof. And (3), reading the problem correctly to understand that the logical clocks L_i here are *not* Lamport clocks, and thus you cannot blindly apply the rules for Lamport clocks from class. One proof is:
    >
    > Consider the sequence of all events $e \ldots e'$ at computer i. If $S(e') \leq S(e)$ then there must exist at least one pair of *consecutive* events f, f' (WLOG $f \rightarrow_i f'$) such that $S(f') \leq S(f)$; otherwise each new event would advance the logical clock L_i, which would yield $L_i(e') > L_i(e')$ and thus $S(e') > S(e)$. To show $S(e') > S(e)$, therefore, it suffices to show that $L_i(f') > L_i(f)$ for any two consecutive events $f \rightarrow_i f'$.
    >
    > Let (t, r) be the logical and real-time clock times for event f and (t', r') be the times for f'. We must show that $t' > t$ for any events f and f'. We break this into three cases based on whether f' is a normal local event, a `send` event, or a `receive` event. (In each of these cases we allow f to be any type of event.) Notice that in every case we have $r' > r$ because R_i is always-increasing and $f \rightarrow_i f'$.
    >
    > Case 1: f' is a normal local event. This causes a single application of Rule 1, yielding $t' = \min(t + 1, r')$. Briefly consider the rule that set the logical clock for f. Because all three rules last-affect the logical clock with an application of Rule 1, we have that $t = \min(;r)$ and thus that $t \leq r$, yielding $t < r'$. Because $t + 1 > t$ and $r' > t$ we therefore have that $t' > t$ regardless of which term of the min-operation is lesser, yielding that $S(f') > S(f)$ as desired if f' is a local-only event.
    >
    > Case 2: f' is a `send` event. This causes an application of Rule 2, which affects the logical clock only with an application of Rule 1. Thus, Case 2 reduces directly to Case 1.

Case 3: f' is a `receive` event. WLOG suppose the message was (m, t_m). This causes an application of Rule 3 which itself calls Rule 1; the total effect is then $t' = \min(\max(t, t_m)+1, r')$. Because $\max(t, t_m) \geq t$ we therefore have $\max(t, t_m) + 1 > t$. As in Case 1, we still have $r' > t$. Again, regardless of which term of the min-operation is smaller, this will yield $t' > t$.

No matter what type of event f' is, we have $S(f') > S(f)$. By the main argument above, we then have $S(e') > S(e)$ as desired.

(c) (6 points) Draw a time-series diagram in which $S(\text{send}(m)) > S(\text{receive}(m))$ for some message m. Be sure to compute the Shamport-time and label the values of the real-time clocks at all computers for any events in your diagram. Explain in 1-2 sentences why this unwanted behavior occurs.



$L_i(e') = \min(\max(85,100)+1,\ 85) = 85$

Solution:

Because Kesden's clock is always 15 minutes slow, his computer first advances its logical clock to match the time from the incoming message, but then resets its logical clock *back* to match the real-time clock when it applies Rule 1. The fundamental problem is that the logical clock will never advance past the real-time clock at each computer, so skew in the real-time clocks can cause the `receive` to be timestamped earlier than the `send`.

(d) (8 points) Suppose we know that the one-way latency of any message between any two computers i, j is at least x clock-ticks by any local clock.

Also suppose we can synchronize R_i and R_j using an external time-source such that $|R_i - R_j| \leq y$, always, for some integer y. Is there a value of y small enough to guarantee that $S(\text{send}(m)) < S(\text{receive}(m))$ for any message m between i and j? If yes, compute the largest possible value of y to guarantee this property, and explain why your answer is correct. If no, explain why.

Solution: Yes, if $y < x$. Consider a diagram similar to the part above, but with send(m) at (t, r) and receive(m) at (t', r'). Because the propagation delay of m is at least x clock-ticks and the clock-skew is at most y, we have $r' \geq (r - y) + x$ and therefore that $r' > r \geq t$ if $x - y > 0$. Rule 3 is applied to timestamp the `receive` event, yielding $t' = \min(\max(x, t) + 1, r')$, where x was the value of the logical clock before receiving m. Because $\max(x, t) + 1 > t$ and $r' > t$, we have $t' > t$ and therefore that $S(\text{receive}(m)) > S(\text{send}(m))$.

Solution:

(An aside: Changing Rule 1 to $L_i = \max(L_i + 1, R_i)$ creates an algorithm that always respects the global happens-before ordering (regardless of real-time clock synchronization), and also lets each computer's Lamport clock approximate its real-clock, if all the clocks in the system are approximately-synchronized and the real-clocks are sufficiently granular. Neat!)