15-440 Distributed Systems
Fall 2010 Final

Name:	
Andrew: ID	

November 30, 2012

- Please write your name and Andrew ID above before starting this exam.
- This exam has 11 pages, including this title page. Please confirm that all pages are present.
- This exam has a total of 123 points.

Question	Points	Score
1	22	
2	5	
3	12	
4	6	
5	6	
6	6	
7	10	
8	4	
9	4	
10	6	
11	6	
12	10	
13	0	
14	6	
15	6	
16	4	
17	10	
Total:	123	

A True/False and select the right answers

1. (22 points) True/False: Circle the one that best applies. Grading is 2 points for every correct answer. That means that at the end of the exam, if you don't know the right answer, *come back here and guess randomly!*

Threads and Processes

True, False It is less expensive to create a process than a thread.

True, False Switching between threads is faster than switching between processes.

True, False In order to switch between user threads, you need to enter the kernel.

True, False Threads share a heap and a stack.

Networking

True, False The TCP initial handshake must be completed between the sender and every router on the path to the destination before communication can take place.

True, False The Internet service model allows a small (a percent or two) amount of packet loss.
(In other words, end-hosts must be able to deal with a small amount of loss)

True, False Domain Name Service (DNS) servers remember which clients have cached DNS replies so that the servers can send invalidation messages when name bindings change.

Distributed Mutual Exclusion

True, False Ring-based distributed mutual exclusion is a great solution for a latency-sensitive application running on 10,000 nodes.

Logging

True, False Write-ahead logging is a useful technique for providing persistence for random updates without needing to write to a random location on disk.

True, False For correctness, all write-ahead log records must specify how to both undo and redo the operation they describe.

Tor

True, False The onion routing scheme used by Tor is resilient to an attacker that can simultaneously compromise every node in the Tor network.

2. (5 points) Which of the following design assumptions apply to HDFS/GFS? (Circle all that apply)
   A. Only amazingly powerful servers can handle the size and complexity of the workload.
   B. Any use of a centralized server would cause too much of a bottleneck.
   C. Component failures are expected.
   D. Files are huge by traditional standards.
   E. File contents are often updated and overwritten.

Solution: C, D

B The Dining Octopi

3. (12 points) Recall the Dining Philosophers problem. Now consider what happens when each philosopher is an octopus with eight arms. In the center of the table is a pile of forks and knives.

   - To eat, each octopus must have 4 forks and 4 knives.
   - Each octopus can only grab one utensil as a single operation.
   - Each octopus grabs utensils in random order until they have enough. (But once they have 4 of one type, they will only grab the other type.)
   - Once they are done, the octopus returns all of its utensils at once.

   Diners are implemented as threads that ask for utensils and return them when finished.

   (a) Consider the following pseudo-code:

```
global int num_forks_available = NUM_DINERS*3;
global int num_knives_available = NUM_DINERS*3;
global mutex pile_mutex = init_mutex();
global cond_var pile_cond = init_condvar();

void grab_utensil(boolean want_fork){
  mutex_lock(&pile_mutex);
  if(want_fork){
    while(num_forks_available == 0){
      cond_wait(&pile_cond, &pile_mutex);
    }
    num_forks_available --;
  } else {
    while(num_knives_available == 0){
      cond_wait(&pile_cond, &pile_mutex);
    }
    num_knives_available--;
  }
  mutex_unlock(&pile_mutex);
}

void done_eating(){
  mutex_lock(&pile_mutex);
  num_forks_available+=4;
  num_knives_available+=4;
  mutex_unlock(&pile_mutex);
  cond_signal(&pile_cond);
}
```

   Is it possible for this code to deadlock? If so, provide a series of calls to the above functions that will result in deadlock. If not, explain why not.

(b) Consider the following pseudo-code:

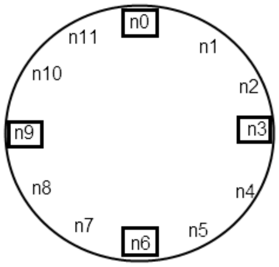
```
global semaphore forks = sem_init(NUM_DINERS*3 + 2);
global semaphore knives = sem_init(NUM_DINERS*3 + 2);

void grab_utensil(boolean want_fork){
  if(want_fork){
    P(forks);
  } else {
    P(knives);
  }
}

void done_eating(){
  for(int i=0; i<4; i++){
    V(forks);
    V(knives);
  }
}
```

Is it possible for this code to deadlock? If so, provide a series of calls to the above functions that will result in deadlock. If not, explain why not.

C P2P



Successor Tables:

n0

i	id+2^i	succ
0	1	3
1	2	3
2	4	6

n6

i	id+2^i	succ
0	7	9
1	8	9
2	10	0

n3

i	id+2^i	succ
0	4	6
1	5	6
2	7	9

n9

i	id+2^i	succ
0	10	0
1	11	0
2	13	0

4. (6 points) Suppose the above Chord setup:

- At what node do you expect to find key 10?
- At what node do you expect to find key 9?
- On average, how many hops are there when searching for a key?
- What route will a search for key 11 take, starting at node 3?

Solution:

5. (6 points) Suppose you want to build a large distributed system to store cookie recipes. Each cookie is stored using the recipe's name ("Grandma's Arsenic Cookie Surprise"). Unlike many P2P systems, in this one, the recipe data is small, so you wanted to store the actual data in the P2P system instead of leaving it on the original nodes. A friend says "Hey, I hear that Chord is very efficient – it doesn't take many lookups to find the node containing the data." Explain to your friend why directly throwing this data into a DHT such as chord, with *key* = recipe name, *value* = recipe data, is a really bad design decision. Brief - two sentences or so.

Solution:

6. (6 points) Tell us what style of P2P system you *would* use, and what assumptions you think are reasonable for your cookie-storing-system:

- Number of nodes
- Number of recipes
- Number of queries/second
- What do queries look like?

Style of system AND a brief rationale for this decision:

Solution:

7. (10 points) At a later point, a hacker breaks in to computers in your P2P Cookie Network and modifies recipes to contain mind-altering substances. To prevent this, you propose a reputation-based security scheme as follows:

   - Every recipe publisher generates a public, private keypair K_p.
   - The publisher signs the recipes they provide using K_p.
   - Publishers who know each other or like each others' recipes sign "yum cards" for the other, of the form: "I think $p2$ is a good cook", signed K_{p1}.
   - When examining a recipe, a reader r looks for a chain of "yum cards" linking them to the publisher p. In other words, r thinks that $c1$ is a good cook, $c1$ thinks $c2$ is a good cook, ..., c_n thinks p is a good cook.

   You want to enable finding this yum-card chain using a DHT over the nodes in your cookie network. Describe an algorithm (pseudocode is fine, so is english) for searching for a path from receiver r to publisher p, describing also what you store as key-value pairs in the DHT. A bad answer would always have to download *all* yum cards in the system every time you want to find a chain (and will not receive a very good score). But you don't have to try to optimize this into oblivion - find a reasonable solution. Awesome solutions will receive a point or two of extra credit depending on their subjective awesomeness to the graders.

Solution:

D Eliminating Redundant Redundancy

The idea of network-level “redundancy elimination” is a form of long-range compression: If object “A” was sent over the network earlier, and we re-send it later, identify that we’re sending a duplicate and tell the receiver to fetch the object from their local cache.

```
Sender ------------------------- Receiver
 {Record of prior               {Cache}
  transmissions}
```

A popular way to accomplish this is to divide the stream of data flowing from sender to receiver into “chunks” (e.g., 1KB each), and to calculate the hash of each chunk. The sender then remembers which chunk hashes it’s sent before to the receiver, and the receiver keeps a cache indexed by chunk hash:

```
Sender ------------------------- Receiver
 {List of transmitted            {Cache:  Chunk ID -> Chunk data}
  Chunk IDs}
```

8. (4 points) Ignoring packet headers, if the system uses 1KB (assume 1KB is 1000 bytes) chunks and uses the 160-bit SHA-1 hash to compute chunk IDs, what is the minimum percentage of the size of the original data stream that this system could send with completely redundant traffic?

Solution: $\frac{160bits}{8bits/byte} = 20bytes$

$\frac{20bytes}{1024bytes} = 1.95\% = \ 2\%$

(Or over 1000 bytes)

9. (4 points) On a 10 Mbit/second (1 megabit = 1 million bits per second) link that had a one-way latency of 50ms, how long would it take to transmit a 10 KB (assume here this means 10,000 bytes) file, with no protocol overhead...

- *Without* using redundancy elimination?

Solution:

- *Using* redundancy elimination, if the whole file had been transmitted previously?

Solution:

10. (6 points) You installed such a system at your house, and you decided to *totally* cheap out on the hard drive you use as a cache for previously received data. You bought it on EBay, and the seller warned that the drive frequently corrupted data that he was trying to store. Justify briefly why you think your decision is acceptable in this case.

Solution:

E Lamport's Alarm Clock doesn't have a snooze button

Consider a system with 4 machines that uses Lamport Timestamps for logical ordering. Let $L_i(m_j, send)$ be the Lamport Logical Time for machine i sending message j. Let $L_i(m_j, recv)$ be the Lamport Logical Time for machine i receiving message j. Each local clock is initialized to 0. Answer the questions below based on the following timestamps:

$L_1(m_1, send) = 1$
$L_1(m_2, send) = 2$
$L_2(m_1, recv) = 2$
$L_3(m_2, recv) = 3$
$L_2(m_3, send) = 3$
$L_3(m_4, send) = 4$
$L_3(m_3, recv) = 5$
$L_4(m_4, recv) = 5$
$L_4(m_5, send) = 6$
$L_1(m_5, recv) = x$ $\longleftarrow$ X is here
$L_4(m_6, send) = 7$
$L_1(m_7, send) = 8$
$L_3(m_7, recv) = 9$
$L_3(m_6, recv) = 10$

11. (6 points) What is the value of x? If it cannot be determined, write "unknown".

Solution:

12. (10 points) For the following, circle which event happened first. If it cannot be determined, circle "unknown."

- $L_1(m_1, send)$ $L_4(m_5, send)$ $Unknown$
- $L_2(m_3, send)$ $L_3(m_4, send)$ $Unknown$
- $L_3(m_4, send)$ $L_3(m_3, recv)$ $Unknown$
- $L_2(m_3, send)$ $L_4(m_6, send)$ $Unknown$
- $L_4(m_6, send)$ $L_1(m_7, send)$ $Unknown$

Solution:

F Fault Tolerant Byzantine Fault Tolerance

13. In this question, we'll explore the links between replication for fail-stop failure resilience and replication for byzantine failure resilience. Recall that Paxos—an algorithm for fault tolerant replication under non-byzantine failures—requires $2f + 1$ replicas to handle f failures. BFT, on the other hand, requires $3f + 1$.

14. (6 points) Why is it sufficient in Paxos to use a majority vote among $2f + 1$ nodes to ensure consistency? (In other words, what property of a majority are we relying on?).

Solution: The key property is that any two majorities intersect in at least one node. Thus, any decision reached by one majority *must* be observed by any other majority, ensuring that the value chosen will propagate even if the other nodes didn't see it.

15. (6 points) Prove by providing a contradicting example that normal Paxos using 3 replicas cannot handle a byzantine fault in an asynchronous network. Use three replicas A, B, and C. For a proof, we want you to sketch a series of communication in which two clients observe an inconsistent result (which violates the requirement of a consistent answer from the system) when there's a single byzantine ("evil") node among the replica set.

Solution: This answer comes from lecture. The key observation for this answer is that the byzantine node can make different statements to different observers. Let B be the evil node:

```
C1 --value 1-->  {A, B},  pkt lost to C
                 {A, B} decide to accept value 1.

C2 --value 2-->  {B, C},  pkt lost to A
                 B pretends to have never seen a value before
                 thus, {B, C} decide to accept value 2

Any client asking {A, B} for the value will hear 1, but any client asking {B, C] for the value wil
```

16. (4 points) Why can BFT can succeed with using $3f + 1$ replicas and requiring a consistent answer from $2f + 1$ of the nodes? (Short answer! One sentence.)

Solution: In 3f+1 nodes, every majority of $2f + 1$ nodes must contain a majority of "good" nodes (f+1, since there can only be f bad nodes). Thus, the good guys overrule the bad guys.

17. (10 points) A friend who took 15-440 last year and who landed a rockstar job at some company that builds a lot of distributed systems tells you that she doesn't think the BFT model is realistic in her scenario. She says that compromise failures are very rare, but machine failures (fail-stop) are much more common. If an attacker compromises one node in a 4-node (f=1) BFT system, and one other node fails, can the attacker...

- Stop forward progress in the system? (Why or why not? One or two sentences.)

  Solution: Yes. For progress, BFT requires 2f+1 nodes to agree at several points in the process. In a 4 node system with one dead node and one evil node, only 2 nodes are guaranteed to give an answer, but 2f+1 requires 3 nodes to answer.

- Cause an inconsistency, or change a previously "consistent" answer? (Why or why not? Show with a tiny example.)

  Solution: This depends a little bit on the assumptions you make about the system, but it's possible to create a system where this is impossible. By storing the signatures from the good nodes that proved that the prior answer had, at some point, become consistent, it can be impossible for the attacker to revoke a previously committed answer. This question is tricky because it depends on details of the view-change protocol that we didn't go in to during lecture. We're grading this flexibly.