15-440 Distributed Systems
Midterm SOLUTION

Name:
Andrew: ID

October 18, 2011

- Please write your name and Andrew ID above before starting this exam.
- This exam has 18 pages, including this title page. Please confirm that all pages are present.
- This exam has a total of 100 points.

Question	Points	Score
1	24	
2	9	
3	8	
4	15	
5	13	
6	14	
7	16	
8	1	
Total:	100	

True/False

1. (24 points) Grading is +2 points for a correct answer and 0 points for either blank or incorrect. In other words, at the end of the exam, if you don't know the answer to any of these, *guess*, because you'll get more points in expectation. If you want to be nice to the course staff, mark the ones you guessed on with a "G" so that we have a better idea of what material to go over. We won't penalize you.

   (a) NFS version 1 (the network file system) tries to keep as much state as possible on the server for things like file read positions, in case the client crashes and restarts. ◯ True √ **False**

   (b) The checksum portion of an IP packet allows the receiver to both detect and correct a single-bit error in the packet ◯ True √ **False**

   > **Solution:** *Checksums can be used to detect errors, but not to correct them.*

   (c) TCP's reliable in-order byte stream abstraction can add unpredictable latency to messages under high packet loss. √ **True** ◯ False

   (d) When creating a UDP connection to a server, the client and server will engage in a handshake protocol. ◯ True √ **False**

   > **Solution:** *There is no setup phase with UDP*

   (e) In a send/acknowledge protocol, such as LSP in Project 1, the bandwidth of a connection when sending a series of packets, each having B bytes of data, is limited to at most $B/2L$, where L is the one-directional message latency. √ **True** ◯ False

   (f) In the synchronous network model, messages may be delayed, lost, and reordered arbitrarily. ◯ True √ **False**

   > **Solution:** *Synchronous networks must have uniform delay.*

   (g) Using a condition variable can eliminate the need for a mutex when protecting shared resources. ◯ True √ **False**

   > **Solution:** *A condition variable must always have an associated mutex.*

   (h) When making a synchronous RPC call, the caller will not return until after the requested operation has been completed, assuming no error is encountered. √ **True** ◯ False

   (i) If the Lamport clock of event e_1 is less than the Lamport clock of event e_2, then there must be a chain of causal events by which e_1 precedes e_2 ◯ True √ **False**

Solution: *Lamport clocks do not capture the case when two events are unordered.*

(j) Global Position System (GPS) satellites use the Network Time Protocol (NTP) to keep their clocks synchronized. ◯ True √ **False**

Solution: *GPS satellites have their own atomic clocks. NTP would not provide sufficient accuracy.*

(k) Lamport's distributed mutual exclusion algorithm will fail if one of its participants fails √ **True** ◯ False

(l) In a transaction processing system, if a process releases one of its locks before completing all of its state updates, then the isolation property of ACID might be violated. √ **True** ◯ False

Short Answers

2. (9 points) In the following, keep your answers brief and to the point.

   (a) One way in which AFS improves upon NFS, the Network File System, is by granting clients leases to files. Briefly (1 sentence) answer: What complication does this addition cause in the event that the server crashes?

   Solution: The server must contact all clients to find out which ones have leases for which files, or it must wait until the maximum lease duration has expired before issuing further leases.

   (b) Cosmic rays and other phenomenon that can add energy to a system may cause memory modules to periodically experience "bit-flips" or soft-memory errors. What is one type of mechanism that is widely commercially available that can seamlessly detect and correct single-bit errors?

   Solution: Error Correction Codes (ECC)

   (c) What is the name of the protocol that provides atomicity for distributed transactions, and ensures that all participants in a transaction agree on whether it should be commited or not?

   Solution: Two-phase commit protocol

Concurrency and Synchronization

3. (8 points) The "dining philosophers" is a classic synchronization problem: Five philosophers sit at a table around a bowl of rice. Between each pair of adjacent philosophers there is a single chopstick. The philosophers think, and from time to time they pause and eat. In order to eat, a philosopher needs two chopsticks: the one to its right, and the one to its left. Each philosopher can pick up an adjacent chopstick, when available, and put it down, when holding it. These are separate actions: chopsticks must be picked up and put down one by one. Since there are only five chopsticks in total, it is clear that at most two philosophers can eat at the same time—they contend for chopsticks, hence the need for synchronization.

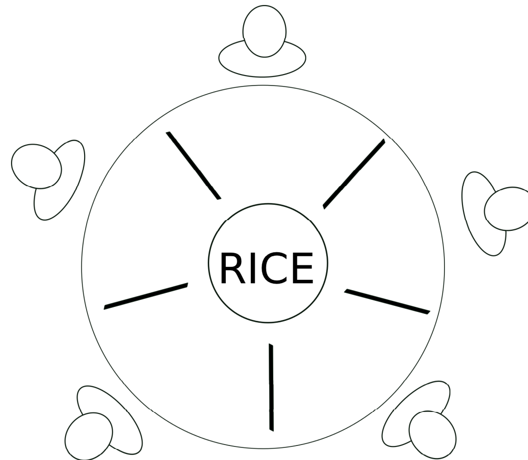

Figure 1: The dining philosophers

Below, we present an implementation in Go, using channels. Each philosopher is represented by a goroutine, and, if we imagine that each chopstick is placed on a chopstick-holder while on the table, each chopstick-holder is also represented by a goroutine. Chopsticks correspond to "chopstick" tokens that are passed through channels. Every chopstick-holder has two channels: one through which it provides the chopstick to one of the philosophers, and one through which a philosopher returns the chopstick.

```
type chopstick string

func ChopstickHolder(taker, giver chan chopstick) {
    for {
        taker <- "chopstick" // wait for philosopher to grab chopstick
        <-giver              // wait for chopstick to come back
    }
}
```

```
func Philosopher(grabLeft, putLeft, grabRight, putRight chan chopstick) {
    for {
        //hungry
        <-grabLeft
        <-grabRight
        //eating
        time.Sleep(1e8)
        //done eating
        putLeft <- "chopstick"
        putRight <- "chopstick"
        //thinking
        time.Sleep(1e8)
    }
}

func main(){
    grabInitial := make(chan chopstick)
    putInitial := make(chan chopstick)
    go ChopstickHolder(grabInitial, putInitial)
    grabLeft := grabInitial
    putLeft := putInitial
    for i := 1; i < 5; i++ {
        grabRight := make(chan chopstick)
        putRight := make(chan chopstick)
        go Philosopher(grabLeft, putLeft, grabRight, putRight)
        go ChopstickHolder(grabRight, putRight)
        grabLeft = grabRight
        putLeft = putRight
    }
    go Philosopher(grabLeft, putLeft, grabInitial, putInitial)

    time.Sleep(1e10) //wait a while
}
```

(a) Explain the problem with this synchronization algorithm.

> **Solution:** It can reach deadlock if all philosophers grab the chopsticks to their left at the same time.

(b) Suggest a simple fix (you don't have to write code). Hint: You don't need other synchronization mechanisms. Instead, you could change something in the behavior of one or all of the philosophers.

Solution: One of the philosophers has to first reach for the chopstick to the right whenever he wants to eat.

Optimizing an LSP-based Password Cracker

4. (15 points) Let's examine the performance of a password cracker such as you implemented for Project 1, with crack clients, worker clients, and a server, all communicating via the live sequence protocol (LSP). Consider the following three parameters that you could control in your system:

   δ: Time between epochs (seconds)

   K: Number of epochs that can elapse without receiving any message from the other end of a connection before having the connection time out.

   w: Typical size of a password cracking job assigned to a worker (seconds).

   For each of the following performance goals, and each of the three parameters, fill in the following table indicating how that parameter should be set to achieve that goal, as follows:

   H: The parameter should be set to a high value.

   L: The parameter should be set to a low value.

   X: The value of the parameter would have little effect on that aspect of the performance.

Goal	δ	K	w
Minimize message traffic in an environment where the clients and the network are highly reliable *High δ minimizes the overhead of epoch events; connections are unlikely to time out; want to minimize traffic to start and finish jobs*	H	X	H
Minimize time to detect crashed worker *Want frequent epoch events and a low timeout threshold*	L	L	X
Minimize wasted effort caused by crashed cracker client *Want frequent epoch events, a low timeout threshold, and minimum work done on behalf of crashed clients*	L	L	L
Minimize chance of prematurely disconnecting one of the clients from the server *Want infrequent epoch events and a high timeout threshold*	H	H	X
Minimize impact of dropped packets without increasing chances of disconnecting clients from the server *Want frequent epoch events, but must then raise the timeout threshold. Also want to minimize traffic between server and workers*	L	H	H

Transactions

5. (13 points) Explain how each of the following three implementations of a transaction breaks the ACID properties in a distributed transactional system: explain what is the problem, and identify which ACID property/properties are violated.

   You must assume that:

   - The given code is run by multiple participants in a distributed transactional system that employs the Two Phase Commit protocol for distributed agreement.
   - All the participants run the exact same code, but on different data. The purpose is for each to subtract the value of its copy of variable *var_B* from its copy of variable *var_A*, but only if *var_A* > *var_B* is true for everyone.
   - There are potentially other transactions running on the same machines that read and write the same copies of variables *var_A* and *var_B* as do some of the transactions participating in this transaction.
   - The convention for lock ordering on each machine is to follow the lexicographical ordering of the lock names.

   (a) (warmup) What do the four letters in ACID stand for?

   Solution: Atomicity, Consistency, Isolation, Durability

   (b)

```
start:
  if (var_A > var_B) {
    vote commit
  } else {
    vote abort
  }

commit:
  lock(var_A)
  lock(var_B)
  var_A = var_A - var_B
  unlock(var_B)
  unlock(var_A)

abort:
  <empty>
```

   Solution: C, I because we are reading *var_A* and *var_B* outside of the critical section, so other transactions may modify them before we lock.

(c)

```
start:
  lock(var_A)
  lock(var_B)
  if (var_A > var_B) {
    var_A = var_A - var_B
    vote commit
  } else {
    vote abort
  }

commit:
  unlock(var_B)
  unlock(var_A)

abort:
  unlock(var_B)
  unlock(var_A)
```

Solution: A, because we may modify *var_A* on some machines and not on the others.

(d)

```
start:
  lock(var_A)
  lock(var_B)
  local_gt = var_A > var_B
  local_tmp = var_A - var_B
  unlock(var_B)
  unlock(var_A)
  if (local_gt == true) {
    vote commit
  } else {
    vote abort
  }

commit:
  lock(var_A)
  var_A = local_tmp
  unlock(var_A)

abort:
  <empty>
```

Solution: D, because *var_A* may have been modified by another transaction in the meantime.

Logical Clocks

6. (14 points) Three computers at CMU, A, B, and C communicate using a protocol that implements the idea of lamport clocks (they include their clock time stamp in messages).

   For reference, if you need a reminder, recall that the three rules of Lamport's algorithm are:

   1. At process i, increment L_i before each event
   2. To send message m at process i, apply rule 1 and then include the current local time in the message, i.e., send(m, L_i).
   3. To receive a message (m, t) at process j, set $L_j = \max(L_j, t)$ and then apply rule 1 before time-stamping the receive event.

   At the beginning of time, all three computers begin with their logical clock set to zero (0). Later, the following sequence of events occurs:

   - A sends message M1 to B: "hi".
   - After receiving M1, B sends message M2 to C: "A told me hi"
   - After receiving M2, C sends message M3 to A: "B is boring"

   (a) Indicate the time included with the messages as they are sent at each step.

   ```
   Send(M1, _ )
   Send(M2, _ )
   Send(M3, _ )
   ```

   Solution:

   ```
   Send(M1, 1)
   Send(M2, 3)
   Send(M3, 5)
   ```

   (b) Maintaining all clock states from the previous question, three ADDITIONAL messages are sent:

   - After receiving M3, A sends message M4 to B: "C is kind of random!"
   - After receiving M4, B sends message M5 to A: "C is boring"
   - A receives message M5

   After all of these messages have been sent and received, what time does each computer think it is?

A	
B	
C	

Solution: A=10, B=9, C=5.
(Remember, receiving and sending are different events!)

(c) Is this a relatively or totally ordered system?

Solution: This is a relatively ordered system.

Reset all clocks to zero. This time:

- A sends message M1 to B.
- The user of machine A is talking on the phone with the user of machine B. She tells him "I just sent you a message online. Please send a message to C right now."
- B sends message M2 to C.

(d) Once all of the messages have been sent and received, what times might C's clock be set to under which circumstance?

Solution: If B received the message before sending the message to C, then C's clock would read 4.

Otherwise, C's clock would read 2.

(e) Explain briefly why the scenarios above could happen even though the events must have happened sequentially in real-time.

Solution: Because the communication over the cell phones was not timestamped using the lamport clock protocol. Therefore, the protocol did not know that the message from A to B had to have happened before the message from B to C.

(f) Which of your two answers for C's clock is more likely to arise? Briefly (1 sentence) justify why.

Solution: It's more likely to be 5. Most of the time, the communication latency between two computers at CMU is likely to be in the few milliseconds range, which is much faster than the two humans can communicate commands in words. Therefore, message M1 is likely to reach machine B well before the user instructs it to send a message to C.

RAID

7. (16 points) You're building a storage array for videos of cats riding Roombas. You've decided to use really cheap disks with a Mean-Time-To-Failure (MTTF) of 10,000 hours ($\simeq$ 1.1 years). The drives have the following performance characteristics:

MTTF	10,000 hours
Sequential read/write speed	50 MB/sec
Capacity	500 GB
Seek time	10 ms

(a) Assume you have built a RAID 1 ("mirrored" – each disk has a copy of all data) system using two disks. Without any humans around to replace dead drives, what is the expected mean time to data loss of this simple RAID 1 system? (Keep your answer simple, using the most simple model of MTTF we discussed).

Solution:

$$\begin{aligned} \frac{1}{2}\text{MTTF} + 1\ \text{MTTF} &= \\ 1.5\text{MTTF} &= \\ 1.5 * 10,000\text{hours} &= 15,000\text{hours} \end{aligned}$$

(b) If one disk fails, how long (in seconds) will it take to rebuild the array by copying all data onto a new disk?

Solution: $\frac{500\ GB}{50\ MB/second} = \frac{500000\ GB}{50\ MB/second} = 10,000\ seconds$

(c) Again, using the simplest model, what are the chances of the second disk dying during this rebuild?

Solution: 10,000 seconds / 10,000 hours MTTF is 1/3600

(d) In practice, the chances are probably higher than the simple calculation above would suggest. Give two reasons why this is the case:

Solution:

1. The load on the disks is higher during repair, which can increase the chances of a failure.
2. Failures are often correlated because they share the same cause—vibration, power fluctuations, etc.

3. Disks do not have a flat lifetime curve. If the disks are approaching the end of their lifetime, the "bathtub curve" will start ramping up sharply, so both disks are much more likely to die than a simple mean would indicate.

(e) You decide instead to build your array using Flash-memory based solid state drives. These drives are fast and silent, but Flash memory behaves differently from disks: It suffers "wear-out." Each time you write to flash memory, it loses a small amount of its lifetime. As soon as you exceed its rated number of writes, the flash is roughly guaranteed to die. However, Flash is often *more* reliable than disk is before the end of its lifetime.

What problem does this cause for your RAID 1 array?

Solution: When one disk dies, the other is extremely likely to die soon.

(f) You decide to fix this problem by giving new instructions to the system administrator for he or she should replace or repair disks in the RAID array. What do you tell them to do in order to drastically reduce the chances of having the whole system fail?

Solution: 1: You could tell them to pre-emptively replace the SSDs, one at a time, before their lifetime ends.

2: You could tell them to create the RAID array using staggered generations of disks, so that neither disk is ever totally at the end of its lifetime.

3: Something else clever here.

Anonymous Feedback

8. (1 point) Tear this sheet off to receive one bonus point. We'd love it if you handed it in either at the end of the exam or, if time is lacking, to the course secretary.

   (a) Please list one thing you'd like to see improved in this class in the current or a future version.

   Solution:

   (b) Please list one good thing you'd like to make sure continues in the current or future versions of the class.