15-440 Distributed Systems
Midterm SOLUTION

Name:
Andrew: ID

October 16, 2012

- 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 115 points.

Question	Points	Score
1	20	
2	12	
3	25	
4	10	
5	13	
6	14	
7	20	
8	1	
Total:	115	

True/False

1. (20 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) The Byzantine failure model allows arbitrary behavior by a faulty system component.
   √ **True** ◯ False

   Solution: That's the definition of the model

   (b) AFS partitions the filesystem by hashing each file's full path and selecting the server the file resides on based on this hash.
   ◯ True √ **False**

   Solution: *AFS partitions files based on (human) administrative decisions at arbitrary directories - often usernames -providing good locality.*

   (c) Callback based caching enables a distributed filesystem to be completely stateless.
   ◯ True √ **False**

   Solution: *A client must register with the server to request that the server sends it a callback when the file is modified.*

   (d) UDP is better than TCP for streaming voice-over-IP data.
   √ **True** ◯ False

   Solution: *Losing some audio bits isn't as bad as dealing with the delays from TCP's retransmission.*

   (e) UDP is better than TCP for RPCs that return medium-sized (100KB) results.
   ◯ True √ **False**

   Solution: *100KB is too big to fit in a single packet. The RPC system would just end up re-implementing much of what TCP does to split this data into packets and ensure that it is received reliably and reassembled. This would therefore be a poor idea.*

   (f) RPCs provide different semantics from local calls because RPCs introduce a different class of errors.
   √ **True** ◯ False

Solution: *RPCs can suffer from network/server failures, so the users need to deal with this.*

(g) Bob finds that his code is blocking on the line `<- signalCh` and causing deadlocks. A clever solution to his problem is to just use a go routine
(e.g. `go func() { <- signalCh }`)
◯ True √ **False**

Solution: *Absolutely not. This rarely (if ever) solves your problem. You might avoid a deadlock, but end up with a race condition or some other synchronization issue. Go routines are not supposed to be used this way and having this in your code is a very likely indication of some bug.*

(h) Bob finds that his code is slow and causing timeouts. Bob remembers reading a magazine about parallelism being the way of the future. Thus, using more go routines will always make his code run faster.
◯ True √ **False**

Solution: *Not necessarily. There are only so many CPUs in a machine and adding more go routines won't make it any faster if you're using all of your CPUs already. Furthermore, there are startup costs and context switching costs that might make it slower. Lastly, there may be some fundamental pieces of sequential code that cannot be run in parallel. If Bob is running into timeouts, then the two most likely causes is 1) the timeout is too low and 2) there is a bug in his code.*

(i) Bob has designed a device that allows a computer to obtain the "true time" with *infinite accuracy*. Ricart & Agrawala's distributed mutual exclusion algorithm still works even if we replace the logical Lamport clock timestamps with the true timestamps.
√ **True** ◯ False

Solution: *Yes, with true timestamps each component sees the same total order.*

(j) Bob bought a new 10Gbps NIC and found its maximum bandwidth is already higher than his hard disk. Thus, in the future it is not necessary for AFS or NFS to keep a local cache any more.
◯ True √ **False**

Solution: *False. First, network performance may still be decreased due to congestion from other sources. Second, even with high bandwidth, the* latency *to the server may slow things down.*

Short Answers

2. (12 points) In the following, keep your answers brief and to the point.

   (a) A round in a distributed mutual exclusion implementation refers to the acquisition and subsequent release of a lock. For n nodes, how many messages are sent in a round using Ricart & Agrawala's algorithm?

   Solution: $2(n-1)$

   (b) Briefly (1-2 sentences) describe a simple example where someone would want to use a condition variable.

   Solution: Many examples exist. One simple example is to support a blocking remove call on a list without using Go channels or spinning. The condition variable would be associated with the list mutex and represent the condition that the list is not empty.

   (c) Google recently announced its globally-distributed database called "Spanner" which replicates data in data-centers across continents. Briefly state (a) what you believe the most important advantage is of having a globally distributed database; and (b) what you believe is the biggest challenge in doing so.
   advantage:

   Solution: it provides global reliability and geographic locality;

   disadvantage:

   Solution: it is very hard to support distributed database transactions at global scale due to long network delay.

   (d) Facebook initially partitioned its user data to different machines according to the college of each user. Briefly state (a) what you believe the biggest advantage is to this partitioning; and (b) what you believe is the biggest drawback to this partitioning approach.
   advantage:

   Solution: more locality in serving one user's queries

   disadvantage:

   Solution: Imbalanced query and storage load; intensive data migration in the beginning of each semester

Debugging Concurrency and Synchronization Bugs

3. (25 points) In the following parts, you will debug a small piece of code and identify the bug (if one exists). Assume calls will succeed so that error checking is omitted. You will mark which category the bug falls (Spin loop, Race condition, Deadlock, Correct) AND explain briefly (1-2 sentences) where the bug is. No explanation is necessary if you mark Correct.

   (a)
```
func ReadHandler(conn *net.UDPConn) {
    bytes := make([]byte, 2000)
    for {
        n, _, _ := conn.ReadFromUDP(bytes)
        go ProcessPacket(bytes[0:n])
    }
}
```

   ◯ Spin loop √ **Race condition** ◯ Deadlock ◯ Correct

   Explain briefly (1-2 sentences) where the bug is. No explanation is necessary if you mark Correct.

   Solution: bytes corresponds to some memory that is reused by ReadFromUDP. Thus, the slice to ProcessPacket could be overwritten while it is being processed.

   (b)
```
// Transfer val from client i to client j
func (am *AccountManager) Transfer(i int, j int, val int) bool {
    am.locks[i].Lock()
    defer am.locks[i].Unlock() // Call unlock on function return

    am.locks[j].Lock()
    defer am.locks[j].Unlock() // Call unlock on function return
    if am.Withdraw(i, val) {
        am.Deposit(j, val)
        return true
    }
    return false
}
```

   ◯ Spin loop ◯ Race condition √ **Deadlock** ◯ Correct

   Explain briefly (1-2 sentences) where the bug is. No explanation is necessary if you mark Correct.

   Solution: This could deadlock if we have concurrent transfers from i to j and j to i (or any circular transfer pattern).

   (c)
```
// Updates inventory quantity and returns whether item is bought
```

```
// - One global mutex causes contention
// - One mutex per item uses too much memory to store the mutexes
// - This uses 100 mutexes as a balance, but does it work?
func (inv *Inventory) BuyItem(id int) bool {
    inv.mutexes[id % 100].Lock()
    defer inv.mutexes[id % 100].Unlock() // Call unlock on function return
    if inv.quantity[id] > 0 {
        inv.quantity[id]--
        return true
    }
    return false
}
```

◯ Spin loop ◯ Race condition ◯ Deadlock √ **Correct**

Explain briefly (1-2 sentences) where the bug is. No explanation is necessary if you mark Correct.

Solution: N/A

(d)
```
func (buf *Buffer) Handler() {
    for {
        select {
        case val := <- buf.insertCh:
            buf.list.PushBack(val) // Insert in list
        default:
            if buf.list.Len() > 0 {
                select {
                case val := <- buf.insertCh:
                    buf.list.PushBack(val) // Insert in list
                case buf.removeCh <- buf.list.Front().Value:
                    buf.list.Remove(buf.list.Front()) // Remove from list
                }
            }
        }
    }
}
```

√ **Spin loop** ◯ Race condition ◯ Deadlock ◯ Correct

Explain briefly (1-2 sentences) where the bug is. No explanation is necessary if you mark Correct.

Solution: The inner select doesn't have a default case so it blocks, but the outer select will spin on the default case when the buffer is empty.

(e)
```
func main() {
```

```
    // Start RandGenerator
    requestCh := make(chan int)
    replyCh := make(chan int)
    go RandGenerator(requestCh, replyCh)
    // Request n random numbers
    n := 5
    requestCh <- n
    // Receive n random numbers
    for i := 0; i < n; i++ {
        aRand := <- replyCh
        ... // do something with aRand
    }
}
// Random number generator
func RandGenerator(requestCh chan int, replyCh chan int) {
    for {
        n := <- requestCh
        replyCh <- rand.Int()
        if n > 1 {
            requestCh <- (n - 1)
        }
    }
}
```

◯ Spin loop ◯ Race condition √ **Deadlock** ◯ Correct

Explain briefly (1-2 sentences) where the bug is. No explanation is necessary if you mark Correct.

Solution: This deadlocks on the line `requestCh <- (n - 1)` as there is no one to receive the request.

Characteristics of a LSP-based Password Cracker

4. (10 points) The following problems relate to a password cracker such as the one you implemented for Project 1, with request clients, worker clients, and a server, all communicating via the live sequence protocol (LSP). Consider the following system parameters:

   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).

   R**:** Minimum roundtrip time for network connections between server and clients (seconds).

   (a) If $w > K \cdot \delta$, then the server may decide the worker client has become disconnected and terminate the connection.
   ◯ True √ **False**

   > **Solution:** *The underlying protocol keeps the connection going via resends, even when there is no application-level communication.*

   (b) Increasing the value of w can increase the amount of wasted work by the worker clients.
   √ **True** ◯ False

   > **Solution:** *Work will be wasted when 1) one worker finds the password and others are working on different ranges, and 2) the request client has become disconnected.*

   (c) If we increase the value of w, then the server will need to make more scheduling decisions.
   ◯ True √ **False**

   > **Solution:** *Just the opposite.*

   (d) If we set $\delta < R$, then no communication will take place.
   ◯ True √ **False**

   > **Solution:** *As long as $K \cdot \delta > R$, the first acknowledgement will arrive before the sender gives up.*

   (e) If a data message gets dropped, then it will take at least time δ for the message to be resent and acknowledged.
   ◯ True √ **False**

Solution: *Not if the drop occurs just before the epoch event.*

(f) The maximum one-way data throughput is $1/R$ data messages per second.
√ **True** ◯ False

Solution: *One complete roundtrip is required for each message sent.*

(g) Decreasing the value of K will reduce the time required to detect a failed worker client.
√ **True** ◯ False

Solution: *The application server will be notified sooner that the client has become disconnected.*

(h) LSP could be extended to be robust, with high probability, in the presence of corrupted UDP packets by appending a checksum to each message and dropping any that are received with an incorrect checksum.
√ **True** ◯ False

Solution: *The resending of messages will ensure that (eventually) a valid packet will be sent.*

(i) If a UDP packet gets delayed by several seconds, then the message it contains could be mistaken for one that occurs much later in the message sequence.
√ **True** ◯ False

Solution: *Due the the wrapping around of sequence numbers.*

(j) Reducing the value of δ will help mitigate the reduction in throughput caused by dropped packets.
√ **True** ◯ False

Solution: *Resending would occur more frequently.*

Transactions

5. (13 points) Josie, a bank employee, has been (pseudo)coding a transaction to simultaneously deduct fees from a subset of global bank accounts, but only if all accounts in the subset have the funds available. Unfortunately, her code is not quite right.

   Using the following assumptions, for each of Josie's implementations, identify one ACID property that is not upheld and briefly explain why that is the case. Even if you find multiple violations, you only need to pick and justify one per section.

   You must assume that:

   - The given code is for this one type of transaction, for a single account, over a distributed transactional system that employs the Two Phase Commit protocol for distributed agreement. This code is invoked concurrently on every account in the set.
   - Each account has its own *balance*. Each account has an internal lock *iLock* controlled by functions Lock() and Unlock().
   - The *amount* passed to OnCommit and OnAbort is correct.
   - This transaction will only begin again sometime after its previous iteration has completely finished.
   - There might be other transactions in progress that also read and update some of the account balances within the set.

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

   Solution: Atomicity, Consistency, Isolation, Durability

   (b)
```
type BankAccount struct {
  Balance int64
  iLock mutex.Lock
}

func global_deduct(acct_id int64, amount int64) {
  account := global_all_accounts[acct_id]

  account.iLock.Lock()
  willCommit := account.Balance > amount
  account.iLock.Unlock()

  if willCommit {
    VoteToCommit(account)
  } else {
    VoteToAbort(account)
  }
```

```
}

func OnCommit(account *BankAccount, amount int64) {
  account.Balance = account.Balance - amount
}

func OnAbort(account *BankAccount, amount int64) {
// <Empty>
}
```

Solution: D. OnCommit creates a Balance race condition with other Transactions.
I. Because OnCommit cannot promise to be done before the next call of this Transaction, so the Balance may remain out-of-date when the next Transaction begins.

(c)
```
type BankAccount struct {
  Balance int64
  iLock mutex.Lock
}

func global_deduct(acct_id int64, amount int64) {
  account := global_all_accounts[acct_id]

  account.iLock.Lock()
  account.Balance = account.Balance - amount
  willCommit := account.Balance > 0
  account.iLock.Unlock()

  if willCommit {
    VoteToCommit(account)
  } else {
    VoteToAbort(account)
  }
}

func OnCommit(account *BankAccount, amount int64) {
// <Empty>
}

func OnAbort(account *BankAccount, amount int64) {
  account.iLock.Lock()
  account.Balance = account.Balance + amount
```

```
  account.iLock.Unlock()
}
```

Solution: I. The transaction is completely detectable by others, especially when Balance is below zero.

(d)
```
type BankAccount struct {
  Balance int64
  iLock mutex.Lock
}

func global_deduct(acct_id int64, amount int64) {
  account := global_all_accounts[acct_id]

  account.iLock.Lock()
  account.Balance = account.Balance - amount

  if account.Balance > 0 {
    VoteToCommit(account)
  } else {
    account.Balance = account.Balance + amount
    VoteToAbort(account)
  }
}

func OnCommit(account *BankAccount, amount int64) {
  account.iLock.Unlock()
}

func OnAbort(account *BankAccount, amount int64) {
  account.iLock.Unlock()
}
```

Solution: A. Only the accounts that didn't have the funds get restored on Abort.
C. Some accounts lose money per aborted Transaction.

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 sending M1, A sends message M2 to C: "hi"
   - After receiving M1, B sends message M3 to C: "A told me hi"
   - After receiving M3 first and then M2, C sends message M4 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, _)
   Send(M4, _)

   Solution:

   Send(M1, 1)
   Send(M2, 2)
   Send(M3, 3)
   Send(M4, 6)

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

   - After receiving M4, A sends message M5 to B: "C is kind of random!"
   - After receiving M5, B sends message M6 to A: "C is boring"
   - A receives message M6

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

A	
B	
C	

Solution: (Remember, receiving and sending are different events!)
initially: A=0, B=0, C=0
send(M1): A=1
send(M2): A=2
send(M3): A=2, B=3
send(M4): A=2, B=3, C=6
send(M5): A=8, B=3, C=6
send(M6): A=8, B=10, C=6
recv(M6): A=11, B=10, C=6.

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

Solution: This is a relatively ordered system.

(d) Write out the vector time representation with the following messages as they are sent at each step.

Send(M1, [ ])
Send(M2, [ ])
Send(M3, [ ])
Send(M4, [ ])

Solution:

Send(M1, [1,0,0])
Send(M2, [2,0,0])
Send(M3, [1,2,0])
Send(M4, [2,2,3])

RAID

7. (20 points) You're building a storage system to hold 2 petabytes (2×10^{15} bytes) of data. The drives you select have the following performance characteristics:

MTTF	12.68 years	(= 111111 hours = 400 million seconds)
Sequential read/write speed	50 MB/sec	(50×10^6 bytes/sec)
Capacity	2 TB	(2×10^{12} bytes)
Seek time	5 ms	(0.005 seconds)

To store 2 PB of data, you will need 1000 "data" disks (not counting extras for mirroring or parity).

You recall from class that creating a RAID-0+1 mirrored/striped scheme *without repair* doesn't help much, so you decide on the following scheme:

- Each disk will be mirrored;
- Each group of 10 data disks (20 total disks!) will have two "spares" that will be automatically brought in to handle repair.

IMPORTANT: In this question, where appropriate, you are encouraged to use the notation **PartA**, **PartB**, etc., as a variable to use in later parts of the question. If you set up the formula properly using the variable, we will give you full credit, even if the numeric value that you then substitute in was incorrect. You may do this even if you can't answer one of the parts.

After showing the formula with the variable, please do substitute in the actual number and carry the calculation as far as seems reasonable with mental math. You don't have to go down to decimals, but if you can cancel out, e.g., $\frac{1000}{100}$ to 10, that would be good.

(a) For a single pair of mirrored disks (2 disks total, both store the same data), what is the expected time to data loss without repair? ("Without repair" means that we don't replace a disk when it dies. The data is lost once both disks die.)

You may express your answer using the variable $MTTF$ to not have to do a lot of ugly calculations. We want to see your formula, not your arithmetic.

Solution: $\frac{1}{2}\text{MTTF} + \text{MTTF} = \frac{3}{2}\text{MTTF}$

(b) For a *group* of 10+10 disks, where each disk is mirrored as in the previous part, but with *no* spares, what is the expected time to data loss?

Solution: $\frac{\text{PartA}}{10} = \frac{3}{20}\text{MTTF}$

(c) Now, let's add in the spares: For a group of 10+10+2 disks (10 mirrored pairs plus two spares), what is the expected time until data loss due to "using up" both spares

and then losing two disks in a mirrored group? Hint: Remember that, until more than 2 disks have failed, there will always be 20 disks in use at any time.

Solution: In order for the system to die, we must now first lose (any) two disks, followed by having any single mirrored group die, as above. In normal operation, there are 20 active disks, so we just wait for two to die first:
$\frac{MTTF}{20} + \frac{MTTF}{20} + \text{PartB} = \frac{5}{20}\text{MTTF}$

(d) For the entire array of 1000 "data" disks (100 groups of 10+10+2), what is the expected time until data loss due to any one group using up both of its spares?

Solution: The answer from C divided by 100.
$\approx \frac{5}{2000}\text{MTTF} = \frac{1}{400}\text{MTTF} = 1\text{million seconds} \approx 11.6$ days

(e) If instead of having two "hot spares" allocated to each single group, you instead could use any of the 200 hot spares to replace *any* failed disk in the cluster, what would be the expected time until data loss due to running out of spares?

Solution: $200 \times \frac{MTTF}{2000} + \frac{1}{100}\text{PartB} \approx 110\text{MTTF} \approx 1.3$ years

(f) During a repair, how long will it take to copy all of the data from the "remaining" disk to one of the hot spares? You can express your answer in whatever time units are most convenient. There are 86400 seconds in a day, 3600 seconds in an hour, and 8760 hours in a year.

Solution: Approximately, using the fact that we have small probabilities:
$\frac{2000\ GB}{50\ MB/second} = \frac{2000000\ MB}{50\ \text{MB/second}} = 40,000\ seconds$

(g) For one mirrored pair of disks, if the first one fails, what is the probability of having a data loss due to the other disk dying during repair?

Solution: $\frac{40000\ \text{seconds}}{400\ \text{million seconds}} = \frac{1}{1000}$

(h) What, therefore, is the probability of the cluster experiencing data loss due to losing a second disk during repair *before* it runs out of spare disks? (Using the scheme from part 'e')

Solution: The cluster will perform 200 repairs before it runs out of disks. What we're really asking, then, is what is the probability of dying during any one of these 200 repairs. The correct answer for this is asking the probability of a

succesful trial in any of 200 trials that each succeed with 1 in 1000 probability. Because the numbers are small, we can handwave and approximate this as: $\frac{200}{1000} = \frac{1}{5}$

(i) A colleague says he's really worried about data loss due to corruption on the disk after writing. He says that your RAID-5 parity scheme can't handle disk corruption. Explain to him, in quick pseudocode, how you could use a function such as CRC-32 or SHA-1 to enable the RAID-5 to recover from corruption:

Solution:

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.