15-440/640 Distributed Systems
Midterm SOLUTION

Name:
Andrew ID:

October 12, 2021

- Please write your name and Andrew ID (IN CAPS) on EVERY page.
- This exam has 15 pages, including this title page. Please confirm that all pages are present.
- This exam has a total of 100 points.

Question	Points	Score
1	21	
2	13	
3	10	
4	10	
5	9	
6	12	
7	14	
8	10	
9	1	
Total:	100	

True/False

1. State whether each statement below is *true* or *false*. ALSO give a *one to two sentence* reason for your answer. Each correct answer is worth 3 points (1 point for the T/F, and 2 points for the reason).

   (a) (3 points) Centralized mutual exclusion algorithms are always safe but may not be fair.

   Solution: True. Centralized algorithms are clearly safe, but their fairness depends on the scheduling policy.

   (b) (3 points) A text messaging application would best be implemented using UDP.

   Solution: False. Text messaging necessitates reliable, in-order service, so TCP should be used.

   (c) (3 points) In Write-Ahead Logging (WAL), the dirty page is written to disk only after the corresponding logs are written to disk.

   Solution: True. WAL writes logs to disk before flushing the dirty page.

   (d) (3 points) A Remote Procedure Call (RPC) enables a computer program to execute a subroutine in its own memory address space.

   Solution: False. RPCs enable subroutine execution in a *remote* address space.

(e) (3 points) In a program representing a card game, a transaction would be considered 'Durable' (D in ACID) if that transaction maintains some global invariants (*e.g.* it does not create any new cards).

> **Solution:** False. The semantics described above correspond to 'Consistency', or the C in ACID. 'Durability' implies that once a transaction is completed, it cannot be undone.

(f) (3 points) DHCP is used to translate hostnames into IP addresses.

> **Solution:** False. Domain Name Service (DNS) is used for name-address translation, while DHCP (Dynamic Host Configuration Protocol) is used for dynamically assigning IP addresses to networked end-hosts.

(g) (3 points) An attacker discovers a vulnerability in LSP (from Project 1) that allows them to change any 'Ack' packet into a 'CAck' packet (only the Message Type is affected). This bug compromises LSP's liveness guarantee.

> **Solution:** True. If one or more lost packets get CAck'd, those packets will never be retransmitted and the receiver will never make progress.
>
> Alternative reasoning: If the attacker tampers with heartbeat messages (ACKs with SN 0), connections over a lossy network will end up getting terminated, preventing the protocol from making any progress.

Short Answers

2. In the following, keep your answers brief and to the point (*i.e.*, 2-3 sentences).

   (a) (3 points) Riccardo believes that the ARIES protocol could be optimized by including only the new value for each record in a transaction, rather than both the old and new values. Do you agree with their assessment? Explain why or why not.

   Solution: No, this modification does not work. We cannot apply UNDO operations if we do not store the previous state of the record.

   (b) (3 points) Which layer(s) of the network stack do UDP and TCP operate at? If you were implementing an RPC service, what would be the benefit of building it atop UDP and TCP, repectively? Answer in one or two sentences for each protocol.

   Solution: UDP and TCP are Layer 4 (Transport) protocols. Building an RPC service atop UDP will impose little communication overhead, and is suitable when the underlying network is already fairly reliable. Building it over TCP allows the system to benefit from the reliability, congestion control, and flow control provided by TCP.

   (c) (3 points) Process p attempts to set its clock using Cristian's Time Sync. At p's local time 10:00:13.14 AM, it sends a request to the time server. At local time 10:00:14.21 AM, p receives a response from the server indicating that the 'real' time is 10:00:13.18 AM. p estimates the *minimum* one-way delay to be 0.40s. Assuming no clock drift for p, what are the earliest and latest possible 'real' times at which p receives the response?

   Solution: Earliest: 10:00:13.58 AM, Latest: 10:00:13.85 AM

   (d) (3 points) How many messages are required for a node to gain access to the critical section (CS) in a system with n nodes that uses the Ricart & Agrawala algorithm for distributed mutual exclusion? Explain how you arrived at your answer.

   Solution: $2(n-1)$ messages. The requesting node must send a request message to all other nodes and receive a grant message from all other nodes in order to enter the CS.

   (e) (1 point) What is the secret nonce you saw during the Paxos lecture? :)

   Solution: AZ107

Andrew ID:

Always Commit Your Work

3. Consider a distributed transaction, T, operating under the two-phase commit (2PC) protocol. Let N_0 denote the coordinator node, and $\{N_1, N_2, N_3\}$ denote the set of participant nodes. The following five messages have been exchanged between the nodes:

Time	Message
1	N_0 to N_1: 'PHASE 1: PREPARE'
2	N_0 to N_2: 'PHASE 1: PREPARE'
3	N_1 to N_0: 'OK'
4	N_0 to N_3: 'PHASE 1: PREPARE'
5	N_2 to N_0: 'OK'

(a) (3 points) Assuming no messages have been dropped in the exchange so far, who should send a message at time 6? Who is the intended recipient of this message?

Solution: N_3 should send a response to N_0.

(b) (3 points) Suppose that N_0 never receives the 'OK' message from N_2 due to a network failure (*i.e.* the message is dropped). Instead, N_0 'times out' after waiting for an extended period of time. What should happen under the 2PC protocol in this scenario? Briefly explain why.

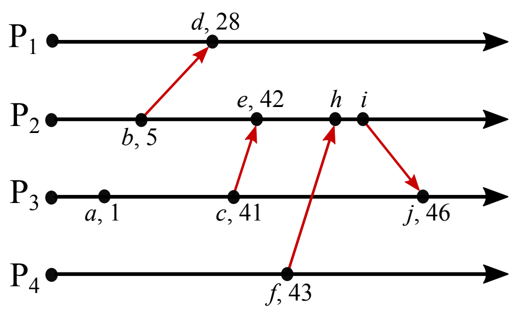
Solution: After the timeout, N_0 will assume that N_2 has failed and it will mark the transaction as aborted. 2PC requires all participants to respond with 'OK'.

(c) (4 points) Suppose that, at some point, N_0 enters Phase 2 and sends out a 'PHASE 2: COMMIT' message to all of the participants. However, N_1 crashes before it receives this message. What is the status of the transaction T when N_1 comes back on-line? Briefly explains why.

Solution: Once the coordinator receives the 'OK' message from all participants, the transaction is deemed to be committed even if a node crashes during the second phase. In this example, N_1 would restore T when it comes back on-line.

Order, Order!

4. The following figure depicts the timeline of multiple events happening across several different processes. The number at each event corresponds to its *Lamport clock* value.



(a) (3 points) Based on their Lamport clock values, what is the relationship between events labelled b and c? Which event happened earlier?

> **Solution:** No relationship, events b and c are parallel ($L(b) < L(c)$ does not imply that b precedes c).

(b) (3 points) Ignore the Lamport clock values for this subpart. Han proposes a much simpler algorithm to determine event ordering: each time two processes communicate, the sender attaches a log of all events it has seen so far, along with their corresponding local timestamps. For example, when P_3 sends event c to P_2, it also sends event a's timestamp. When P_2 receives the message (event e), it can reconstruct a global view of event history across both P_2 and P_3. Based on these logs, can P_2 determine the ordering between events a and b? If so, what is the correct ordering from P_2's perspective up to event e? If not, why not?

> **Solution:** No, because it is futile to compare timestamps across different processes. Since the machine clocks are not precisely synchronized, it is not meaningful to compare the resulting timestamps either. In this example, event a is recorded in terms of P_3's clock while b is recorded in terms of P_2's.
>
> Alternative explanations based on Lamport time will also be accepted.

(c) (4 points) As you may have noticed, event h is missing a Lamport clock value. What are the possible values for the Lamport clock of event h? What are their corresponding *total-ordered Lamport clock* values? Assume that the maximum number of processes is 10, and that process P_2 has ID 2.

> **Solution:** 44 is the only possible Lamport clock value for event h. Therefore, its totally-ordered value is $M * Li(e) + i = 10 * 44 + 2 = 442$.

360 Noscope Gradescope

5. Eunice has decided that Gradescope is too slow, and wants to implement her own autograding system, Dotolab, where students submit their code via RPCs. Since she wants

students to be sure that they have submitted, she wants to implement RPCs using *at-least-once* semantics. Note, however, that students still have a limited number of submissions for their projects!

(a) (3 points) Eunice wrote the following code to implement the 'Submit' RPC (for submitting code to Dotolab). Why does it violate at-least-once semantics, and how can she fix it? (Please keep your answer brief, and do not write code).

```
func (client*) Submit(data []byte) {
  msg, _ := json.Marshal(data)
  client.send(msg)                    // Send message to Dotolab
  ackMsg := client.ReadFromServer()  // Wait for Ack from Dotolab
  ack := json.Unmarshal(ackMsg)
}
```

Solution: Since send is called just once when the student submits, the message may get dropped along the way, and the server may never receive the submission. In order to fix it, Eunice should send the message in a for loop until some sort of acknowledgement is received from the server.

(b) (3 points) Eunice has fixed the code such that student submissions now follow *at-least-once* semantics. However, students are now complaining that they are running out of submissions because of query duplication. What additional information can Eunice include in her messages to make sure that Dotolab discards duplicate submissions?

Solution: Eunice should include a submission ID (sequence number) with each message. If Dotolab receives a message with a submission ID matching a previous value, it should simply discard the duplicate message.

(c) (3 points) Eunice decides to augment her RPC implementation by providing *exactly-once* semantics. She pays extra for a better network, `CMU_SUPER_SECURE`, which guarantees that messages are always delivered in-order and will never be dropped in transit. Is this sufficient to guarantee exactly-once semantics? Why or why not?

Solution: No. The server could still crash between the time the client sends the message and the time the server would have received it (*i.e.* while the message is in transit), so she cannot guarantee that the message will be delivered.

In-Person Paxos

6. Alice, a CMU student, and Bob, an MIT student, are part of a group of people trying to emulate Paxos in real life, where each member of the group acts as a node in a system.

During the emulation, Bob is confused about why the various steps of Paxos are needed. Help Alice convince Bob why each step is necessary!

(a) (3 points) Bob first asks "Why is it that Paxos requires that at most less than half of us can fail? What happens if exactly half (or more) of us fail?"

Solution: In order to make sure that there is a *single* group for the consensus, less than half of the nodes are allowed to fail. If half or more of the nodes were to fail, then it is possible that two separate groups would reach their own separate 'consensus'.

(b) (3 points) "Hmmmm," says Bob, "that makes a lot of sense actually!". Then Bob frowns and asks, "Why do we each need these proposal numbers? What would happen if we did not have them?"

Solution: Proposal numbers are necessary to ensure that there is always a strict priority of proposals. Without them, it would be impossible to know which one to accept when two different proposal are received.

(c) (3 points) "Wow, thank you for explaining all of this!" Bob says gratefully. "This reminds me how cool distributed systems are! One thing I don't think you can explain is why we need both *Prepare* and *Accept*. If I am the proposer, why do I first need to announce that I am preparing and then later announce that everyone should accept my proposal?"

Solution: In the *Prepare* stage, we are seeing if our proposal numbers are out of date and gauging whether we need to update the proposed value. In the *Accept* stage, we are actually broadcasting the value that we want other people (nodes) to accept.

(d) (3 points) Finally, Bob says "I have heard rumors that it is possible to have a livelock, but I do not believe them. Can you convince me that a livelock is indeed possible?"

Solution: Live lock is possible. If two 'dueling proposers' are competing with each other, then they could act in a way where they are constantly proposing and interfering with the other's proposals.

That's a Lot of Cache

7. In a distributed file system, two clients, C_0 and C_1, are accessing three files, X, Y, and Z stored on a remote server. All three files are initially empty. Each client maintains a separate cache, which implements *whole-file caching* with *check-on-use* and *open-close session* semantics. Assume that the client caches have infinite capacity.

   Each client can execute 4 types of file operations:

   1. `open(file, mode)` opens the file in the client's local cache and moves the file pointer to the beginning of the file.
   2. `read(file)` returns the contents of the entire file and advances the file pointer to the end of the file.
   3. `write(file, payload)` writes the payload at the file pointer's current position, and advances the pointer by the number of bytes written. The original file contents are overwritten (either fully or partially).
   4. `close(file)` will close the file in a client's cache.

   Each operation starts and finishes in one epoch. If the operation results in a change on the server, the change occurs in the same epoch. The actions of the two clients are depicted in the table below:

Epoch	C_0 Operations	C_1 Operations
0	open(X, "rw")	open(Y, "rw")
1	open(Y, "rw")	write(Y, "Welcome")
2	write(X, "Tartan")	close(Y)
3	close(X)	
4	write(Y, "Thank")	open(Y, "rw")
5	close(Y)	open(X, "rw")
6	open(X, "rw")	data = read(X)
7	read(X)	write(Y, data)
8		close(Y)
9	write(X, "Scholars")	write(X, "Proud")
10	close(X)	open(Z, "rw")
11	open(Y, "rw")	write(Z, "JoinUs")
12		close(X)
13	write(Y, "Thank")	close(Z)
14	write(Y, "You")	open(Y, "r")
15	close(Y)	read(Y)
16	open(Z, "rw")	close(Y)
17	write(Z, "Please")	
18	write(Z, "Like")	

(a) (3 points) How many versions of files X, Y, and Z have there been on the server, respectively? (Also include the initial, empty version of each file in your tally.)

File	#Versions
X	
Y	
Z	

Solution:

File	#Versions
X	4
Y	5
Z	2

(b) (6 points) What are the contents of files X, Y, and Z on the server after epoch 18?

File	Contents
X	
Y	
Z	

Solution:

File	Contents
X	TartanProud
Y	ThankYou
Z	JoinUs

(c) (5 points) At which epochs are the server and C_1 transferring file data? (Do not include epochs where they are communicating but no file data is transferred.)

Server to C_1	
C_1 to Server	

Solution:

Server to C_1	5
C_1 to Server	2, 8, 12, 13

Help, My Oven is Sentient!

8. Eunice has decided to bake some cookies with her new smart oven. Since her oven is very technologically advanced, it uses semaphores to control locking and unlocking so that the cookies bake for the perfect amount of time. However, Eunice likes to eat the cookies as soon as they bake (even if they aren't all done at the same time), so she'd like to be able to get in and out of the oven whenever possible. Consider the following code demonstrating the actions of Eunice and the oven:

```
func oven_main() {
  P(oven_lock);
  P(closed_oven);
  cookies_ready++;
  V(oven_lock);
  V(open_oven);
}

func eunice_main() {
  P(oven_lock);
  P(open_oven);
  cookies_ready--;
  V(oven_lock);
  V(closed_oven);
}
```

The oven is initially closed, and the following table depicts the initial state:

Variable	Value
oven_lock	1
open_oven	0
closed_oven	1
cookies_ready	0

(a) (3 points) Describe the difference between deadlock, livelock, and starvation.

> **Solution:** Deadlock occurs when no processes are making progress because each is waiting on access to a resource that another holds. Livelock occurs when each process is doing meaningless work that does not move the program forward. Starvation occurs when some processes are perpetually denied a critical resource necessary to make progress.

(b) (3 points) Which of the above issues (deadlock, livelock, and starvation) manifest in the given code? Explain how the problem(s) arise.

> **Solution:** The above program has an issue with deadlock. When the oven is closed, the `eunice_main` process could acquire *oven_lock* and then wait on

the *open_oven* semaphore; during this time, `oven_main` is blocked waiting on the *oven_lock* semaphore. Since both processes are blocked on each other and neither is performing any work, this situation constitutes a deadlock.

(c) (4 points) Rewrite the code snippet so as to fix the issue described in Part (b).

```
func oven_main() {









}

func eunice_main() {









}
```

Solution:

```
func oven_main() {
  P(closed_oven); // These lines
  P(oven_lock);   // are swapped.
  cookies_ready++;
  V(oven_lock);
  V(open_oven);
}

func eunice_main() {
  P(open_oven);   // These lines
  P(oven_lock);   // are swapped.
  cookies_ready--;
  V(oven_lock);
  V(closed_oven);
}
```

Anonymous Feedback

9. (1 point) Tear this sheet off to *receive points*. 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.

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