Problem Set 4

15-440/15-640 Distributed Systems Spring 2026

Assigned: Thursday, April 16, 2026

Due: Thursday, April 23, 2026 at 11.59 pm

Submission procedure:

- Create a **.pdf** of your answers and upload to `Gradescope`.
- Here are some tips to make your submission easy to read and to grade. Remember, the easier you make this, the less likely we are to make grading errors. Following these guidelines will help us to focus on the technical content of your answers rather than trying to understand what you have written.
    - Don't handwrite your answers. Use LaTeX or Google Docs or some similar input mechanism. If you use LaTeX, a template can be found on the course web page.
    - Put the answer to each question on a separate page.
    - Carefully tag your pdf pages to each question on `gradescope`. You can use the SHIFT key to select multiple pages and associate them with a single question.
- Assume SI notation
    - 1 KB = 10^3 bytes, 1 MB = 10^6 bytes, 1 GB = 10^9 bytes, 1 TB = 10^{12} bytes
    - 1 Kbps = 10^3 bits per second (bps), 1 Mbps = 10^6 bps, 1 Gbps = 10^9 bps
    - but a byte is still 8 bits (not 10 bits) :-)
- **Remember that you have a limit of 2 grace days totaled over all four problem sets.** You can use at most one of those grace days per problem set. Although Gradescope does not track grace days, we will. Exceeding your grace days will result in a zero grade for that problem set.

Question 1 (20 points)

A new startup, Luely, is building an international remittance service that lets you send money from your Luely account to friends anywhere in the world in their local currency. Luely has servers located around the world, each serving a local population with a single currency.

A. The lead developer decides to first tackle the easier case where the sender and receiver use the same currency and are on the same server. In this case, only local operations are needed for the transaction. The initial implementation of the local transaction looks like this:

```
// assume all arguments are valid, both accounts are local, etc.
def send_money_local(txn_id, sender, receiver, amount):
       log = get_log()
       // atomic append to persistent log on disk
       write_to_log(log, txn_id + " START")
       lock_account(sender)
       lock_account(receiver)

       account_s = get_account(sender)
       account_r = get_account(receiver)

       if (account_s.balance >= amount):
              account_s.balance -= amount
              account_r.balance += amount
              write_to_log(log, txn_id + " COMMIT")
              write_account_to_disk(sender, account_s)
              write_account_to_disk(receiver, account_r)
              send_confirmation(sender, receiver)
       else:
              write_to_log(log, txn_id + " FAILED")
              send_failure(sender, receiver)

       unlock_account(sender)
       unlock_account(receiver)
```

This code has bugs. For each of the ACID properties, state whether this code results in violations of that property. If so, provide a scenario where the property is violated. If not, explain why the property is preserved. Does this code also violate other properties (e.g., liveness)?

B. Fix the code in part (A) to prevent the violations you identified.

C. Unlike the special case in (A) and (B), in general, a transfer will require a distributed transaction composed of three operations:
    1. Debit amount in source currency from sender's account on their local server
    2. Convert amount to target currency at a promised exchange rate using a currency exchange service
    3. Deposit target currency amount in receiver's account on their local server

   Only if all three succeed can the transaction be committed. Luely, therefore, plans to implement distributed transactions using a 2-phase commit protocol.

   Note that any of the steps may fail. Step 1 may fail if the sender account does not have enough money. Likewise, step 2 may fail because exchange rates fluctuate continuously and the promised rate may no longer be available. Even step 3 may fail if the target account server is down or not accessible.

   Consider an implementation of 2-phase commit (as covered in lecture) for this application. The coordinator will start a distributed transaction by sending the transaction details to the source account server, the destination account server, and the currency exchange server. If all say yes, then the transaction commits. Otherwise, it fails and no changes should be made. The server ensures that all participants are made aware of the results of a transaction, retransmitting as necessary.

   What should the source account server do to act as a participant in the 2-phase commit? Fill in the skeleton code below. You do not have to write recovery code, but ensure the right information is persisted in case of failures.

```
// assume txn_info is a struct <txn_id, sender, receiver, amount, exchange_rate>
// assume msg is a struct <type, txn_info>
// assume helper functions:
//           lock_account(id), try_lock_account(id), unlock_account(id),
//           get_account(id), send(node, string), log.write(string)
def source_account_server():
       log = get_log()
       while (1):
              msg = get_message(coordinator);  // wait for message from coordinator
              if msg.type==PREPARE:
                     // FILL IN CODE HERE
                     continue
              if msg.type==COMMIT:
                     // FILL IN CODE HERE
                     continue
              if msg.type==ABORT:
                     // FILL IN CODE HERE
                     continue
```

D. What complexities do you foresee with the currency exchange service? Are these problematic for a 2-phase commit implementation?

Question 2 (12 points)

For parts A and B, fill in the blank lines with the correct message type and contents to ensure a valid Paxos sequence. Using the same syntax as described during lecture, recall that there are five messages that may be sent during the protocol:

- Prepare(***n***), where ***n*** is a unique round number.
- Promise(***n***, AlreadyAccepted(n_k, a_k)), where n_k and a_k are the last round number and value, respectively, that were accepted by Acceptor ***k*** and committed to stable storage. These values are NULL in the stable storage of all Acceptors at the start of the protocol.
- PleaseAccept(***n***, ***v***), where $v = a_k$ of highest n_k seen among the promise responses. If no promise response contains a past accepted proposal, ***v*** can be any value chosen by the coordinator.
- Accept_OK()
- NAK(N_p) where N_p is the highest round number that the rejecting node has seen.

"**I→J**" denotes a successfully delivered message from server **I** to server **J.** Note that you cannot use Prepare() to fill in any of the blanks unless it is the ONLY correct answer to make the sequence valid.

A. Consider five nodes (S1, S2, S3, S4, S5):
   1. S1 → S1: Prepare(500)
   2. S1 → S2: Prepare(500)
   3. S1 → S3: Prepare(500)
   4. S1 → S1: Promise(500, null)
   5. S2 → S1: Promise(500, null)
   6. S3 → S1: Promise(500, null)
   7. S1 → S1: PleaseAccept(500,v1)
   8. S1 → S2: PleaseAccept(500,v1)
   9. S1 → S3: PleaseAccept(500,v1)
   10. S1 → S1: Accept_OK()
   11. S3 → S1: Accept_OK()
   12. S5 → S5: Prepare(600)
   13. S5 → S5: Promise(600, null)
   14. S5 → S4: Prepare(600)
   15. S4 → S5: <BLANK_1>
   16. S5 → S3: Prepare(600)
   17. S3 → S5: <BLANK_2>
   18. S5 → S3: <BLANK_3>

15. <BLANK_1>	
17. <BLANK_2>	
18. <BLANK_3>	

B. Consider five nodes (S1, S2, S3, S4, S5):

1. S1 → S1: Prepare(500)
2. S1 → S2: Prepare(500)
3. S1 → S3: Prepare(500)
4. S1 → S1: Promise(500, null)
5. S2 → S1: Promise(500, null)
6. S3 → S1: Promise(500, null)
7. S1 → S2: PleaseAccept(500,v1)
8. S2 → S1: Accept_OK()
9. S5 → S2: Prepare(600)
10. S5 → S4: Prepare(600)
11. S5 → S5: Prepare(600)
12. S5 → S3: Prepare(600)
13. S4 → S5: Promise(600, null)
14. S3 → S5: Promise(600, null)
15. S5 → S5: Promise(600, null)
16. S2 → S5: <BLANK_1>
17. S1 → S3: PleaseAccept(500,v1)
18. S3 → S1: <BLANK_2>
19. S5 → S4: <BLANK_3>

16. <BLANK_1>	
18. <BLANK_2>	
19. <BLANK_3>	

For parts C and D, state whether the execution trace (using the same notation as in parts A and B) could be produced by a correct Paxos implementation. If it is correct, give an informal explanation. If it could not be produced by a correct Paxos implementation, identify the earliest step in the trace that is incorrect, and explain why.

C. Consider four nodes (S1, S2, S3, S4):

01. S2 → S1: Prepare(001)
02. S2 → S2: Prepare(001)
03. S2 → S2: Promise(001, null)
04. S2 → S3: Prepare(001)
05. S3 → S2: Promise(001, null)
06. S2 → S4: Prepare(001)
07. S4 → S2: Promise(001, null)
08. S4 → S1: Prepare(005)
09. S4 → S4: Prepare(005)
10. S4 → S4: Promise(005, null)
11. S3 → S1: Prepare(003)
12. S3 → S2: Prepare(003)
13. S4 → S3: Prepare(005)
14. S1 → S2: Promise(001, null)
15. S4 → S2: Prepare(005)
16. S1 → S4: Promise(005, null)
17. S2 → S1: PleaseAccept(001, v1)
18. S1 → S2: NAK(005)
19. S1 → S3: Promise(003, null)
20. S3 → S4: Promise(005, null)
21. S2 → S4: Promise(005, null)
22. S4 → S1: PleaseAccept(005, v1)
23. S4 → S2: PleaseAccept(005, v1)
24. S4 → S3: PleaseAccept(005, v1)
25. S4 → S4: PleaseAccept(005, v1)
26. S2 → S4: Accept_OK()
27. S3 → S4: Accept_OK()
28. S1 → S4: Accept_OK()
29. S4 → S4: Accept_OK()

D. Consider five nodes (S1, S2, S3, S4, S5):

01. S1 → S1: Prepare(100)
02. S1 → S2: Prepare(100)
03. S2 → S1: Promise(100, null)
04. S2 → S1: Prepare(101)
05. S1 → S1: Promise(100, null)
06. S1 → S3: Prepare(100)
07. S2 → S2: Prepare(101)
08. S2 → S2: Promise(101, null)
09. S3 → S1: Promise(100, null)
10. S4 → S1: Promise(100, null)
11. S1 → S4: Prepare(100)
12. S2 → S3: Prepare(101)
13. S3 → S2: Promise(101, null)
14. S2 → S4: Prepare(101)
15. S1 → S5: Prepare(100)
16. S2 → S5: Prepare(101)
17. S5 → S1: Promise(100, null)
18. S1 → S2: PleaseAccept(100, v1)
19. S2 → S1: NAK(101)
20. S2 → S1: PleaseAccept(101, v1)
21. S1 → S2: Accept_OK()
22. S2 → S2: PleaseAccept(101, v1)
23. S2 → S2: Accept_OK()
24. S2 → S3: PleaseAccept(101, v1)
25. S2 → S4: PleaseAccept(101, v1)
26. S3 → S2: Accept_OK()
27. S2 → S5: PleaseAccept(101, v1)
28. S5 → S2: Accept_OK()
29. S4 → S2: Accept_OK()

Question 3 (12 points)

The music provider LoFi has hired you as a distributed systems architect. The platform has thousands of globally-distributed servers, serves millions of users across 100+ countries, stores millions of songs, and must remain available 24/7. You are tasked with designing the backend infrastructure for several core subsystems.

A. For each of the following scenarios, pick a framework or abstraction from the provided list and justify briefly why it is the best fit for this scenario. You must choose from the following frameworks: **MapReduce**, **MPI, DHT (Distributed Hash Table), Blockchain**, **RPC**

   i. Every 24 hours, Lofi wants to compute the global play count for every song, broken down by country, age group, and device type. The input is a day's worth of raw event logs: roughly 800 GB of records of the form `(user_id, song_id, country, timestamp, device)`.

   ii. The platform stores metadata (title, artist, album, duration, codec info) for 100 million songs. Any server in a globally distributed cluster may need to look up metadata for an arbitrary song based on client activity. The lookup must be fast and respond quickly to the client.

   iii. Of the 100 million songs, only about 10 thousand are popular and frequently played. The rest are rarely accessed, but the system needs to be able to reliably store and retrieve these songs on demand if clients (at arbitrary servers) request them.

B. At the end of each month, LoFi computes artist royalties and distributes payments. The process involves multiple independent organizations (record labels, artists, licensing agencies), each maintaining their own internal records. In the past, discrepancies between these records have led to disputes, requiring time-consuming audits and reconciliation processes. LoFi is exploring ways to ensure that all parties have access to a consistent, verifiable history of streaming activity and payments, without relying on a single organization to act as the trusted authority.

   Joe wants to keep the current RPC-based microservices with a centralized database however your manager Ray proposes to switch over to blockchain.

   i. Identify **one advantage and one disadvantage** of using RPC-based microservices

   ii. Identify **one advantage and one disadvantage** of using blockchain

   iii. Briefly state **which approach you would lean toward and why**, given the requirements

Question 4 (20 points)

VroomVroom is a car rental company operating in **50 cities**. Each city office has a local server that logs every rental as a record with the form:

```
( rental_id, vehicle_class, vehicle_model, vehicle_color,
  rent_time, return_time, rent_city, return_city,
  renter_name, renter_gender, renter_age, payment_details )
```

Here, `vehicle_class` is one of five categories: `Economy`, `Compact`, `SUV`, `Luxury`, or `Van`. The VroomVroom fleet has 30 different car models and 10 different colors. Rental and return times are specified as Unix timestamps (integer seconds since Jan 1, 1970 UTC). Each office holds 1,000,000 such rental records.

VroomVroom uses MapReduce to perform analytics on this data. Each of the 50 city office servers acts as one mapper on its local data. Four additional servers in a Pittsburgh datacenter are used to perform reductions. The CEO of VroomVroom would like to determine if certain factors affect the average length of rentals. She wants to calculate the average rental length grouped by **vehicle class, vehicle model, vehicle color, one-way vs local rental, and renter age range** (young adult < 30 <= adult < 55 <= senior). For example, she wants an average for all blue cars and another for all red cars, and for all rentals by seniors, and all rentals by young adults. (She does not need combined factors, such as the average for blue cars rented by seniors).

Assume each machine processes records at a fixed rate: An **office machine** processes **1,000,000 records/sec** for map work, or **1,000,000 key-value pairs/sec** for reduction work. A **datacenter machine** processes **10,000,000 key-value pairs/sec** of reduction work (10x faster than an office machine). The wall-clock time of a phase is determined by when the last machine completes its work for that phase. You may ignore network and shuffle costs except where stated.

A. Write the **map function** applied to each record. What (key, value) pair(s) does it emit? Note: the value may be a tuple of multiple scalar values if needed.

B. How long do you expect the map task to take? Do you expect stragglers to be a major problem? Why or why not?

C. Describe the **shuffle / partition** step: how many keys are assigned to the 4 reducers?

D. Write the **reduce function** applied to all of the values for a given key. Is there any final step needed to compute the average for a given key?

E. Consider how much reduction work is performed at the 4 reduction nodes. Do you expect stragglers to be a major concern? Why or why not?

F. Assuming equal occurrence of each key, how long does the reduction phase take?

G. Suppose during the map phase, each mapper also performs a partial reduction on locally generated key-value pairs. How long will the map phase now take?

H. Given the modified map operation from (G), how long does the whole map-reduce algorithm now take? Do you expect stragglers to be a major problem? Why or why not?

Question 5 (18 points)

ScottyLabs is building a restaurant review platform for Carnegie Mellon University students, designed to combat review manipulation, but their existing servers are already overloaded, and there is no remaining budget to scale infrastructure this year. One member suggests using a P2P network architecture, utilizing a distributed hash table (DHT) for storage. All Computing Services-provisioned desktops are set up as DHT participants, while all mobile devices, such as laptops and smartphones, are lightweight clients. All of the desktop machines are on the campus Ethernet network, with 5ms ping times between nodes and 1Gbps bandwidth. To look up a review, a hash is generated from the restaurant's name and geo-coordinates and used as the key. Nodes in the network store reviews for the keys they are responsible for in the DHT, allowing storage and query load to be spread across participating clients rather than a centralized server.

When a student submits a review, it is hashed and routed to the appropriate node, which stores the data and replicates it to a small number of successive nodes in the ring. To accelerate lookups, each node maintains a finger table to allow for "jumping" across the ring.

A testing version of the system is deployed, using 10-bit keys, a replication factor of 2, with an initial set of 12 participating nodes with ids: 42, 93, 122, 212, 278, 335, 440, 571, 640, 776, 823, 937.

A. Show the finger table for the node 122.

B. Suppose a mobile phone connected to node 122 adds a new review that has a key of 793. What is the sequence of nodes visited (starting at 122) to perform this insert? What node(s) will store the new document?

C. How long will the operation in (B) take if the document is 1KB in size, and the mobile phone is on the WiFi network (15ms ping time to machines on the campus wired network, and 100Mbps bandwidth)? How does this compare against time to store the file to a well-provisioned centralized server? Show your work and state any assumptions you make.

D. Suppose a new desktop machine wishes to join the DHT network. How is the node's position in the ring determined? Suppose the node is given id 692. Outline the steps needed to add this node to the DHT. What changes to the DHT are required?

Question 6 (18 points)

A. The Starbucks app stores every user's order history on data center servers. Every time someone orders through the app, the order (drink, customizations, timestamp) is appended to a set of shared log files that aggregate every order from users in a region into one large, append-only data file per region. These logs grow very large (GBs per day globally). This data is later used for analytics (e.g., determining the "most popular drinks," "peak hours"), where the files are read in large sequential scans. Is GFS or Ceph more suitable here? Briefly justify your answer in 2 sentences.

B. The BeReal app sends a notification to all users at the same time, prompting them to spontaneously upload a photo of what they are doing at that moment. As a result, the system experiences a sudden surge of millions of uploads within a few minutes and most of the app's activity is concentrated in this short time window. Users expect their photos to be uploaded and visible quickly. Is GFS or Ceph more suitable here? Briefly justify your answer in 2 sentences.

C. Spotify creates an annual "Spotify Wrapped" (a summary of all the songs and genres a user has listened to over the year), personalized to each user. It does this by batch processing the listening history logs of all users. Thus, it needs to store an enormous amount of data representing the listening-history events for millions of users and efficiently run large batch jobs over it. Assuming that GFS is used, which of the following ways of organizing the data into files would be best and why? Briefly justify your answer in 2 sentences.

   - Option A: Store all listening events from all users in one very large append-only file.
   - Option B: Store one separate small file per user containing that user's listening history.
   - Option C: Store one file per day or per region, each containing many users' listening events grouped into large files.

D. On Black Friday, Amazon experiences a surge of orders. Order details are stored in a distributed set of log files and later processed by a batch map-reduce job to compute total sales per item. Assume the file system is GFS.

   During the map phase, each worker processes a chunk of the sales logs and appends key-value pairs to intermediate files. The output records use the name of the product as the key and the value is the quantity of the product in each order, like the following:

   ("Apple TV", 1)
   ("Holiday Sweater", 3)

   Suppose a worker processes its assigned chunk and writes its output, but crashes before confirming completion.

   The system reassigns the same chunk to another worker, which processes it again causing some sales records to be processed twice. During the subsequent reduction phase, is duplicate elimination necessary in this scenario? Why or why not? If so, what would you change about the system in order to eliminate the duplicates?