[Mock] Exam 1

[CSCI 1380 Spring 2025] February 31st, 2025

Paper notes allowed. No online resources, external help, or LLMs.

NAME: _____
BROWN ID: _____

Key points about this exam:

- *No need* to answer *all* questions — you can score up to 150 points, but *only 100 points are needed* for a perfect score. Anything beyond that is extra credit!
- Answer any combination of questions — irrespective of groups.
- If justified, multiple answers might be correct — i.e., no single correct answer.
- Coding preference: ideally JavaScript > pseudocode > English. Non-JavaScript implementations may lose points.
- You can use any notes, provided that they are printed and you bring them with you.
- No online resources, external help, or LLM support is allowed.
- There is **no penalty for guessing**. If you have time, attempt every question!
- You will have **60 minutes** for this exam. Good luck!

---

1. Warmup [25 pts]

Suppose you're employed by the National Weather Service, a service that ideally provides the following guarantees:

- [G1] Requests can be handled at any time, including when weather stations go offline.
- [G2] All weather requests are processed and eventually return a result.
- [G3] All air pressure data returned reflects the most up-to-date readings

Your boss tells you that G2 can be sacrificed in view of G1 and G3. She asks you to write a specification for how the system will handle network failures. For each of the following failure scenarios, specify:

Detection: How can the system detect this failure?
Client Response: What should the user's client do in response?
Correctness: How does this response ensure the system still respects R1 and R3?
Impact: How is the user affected in this specific case?

Question 1.1. The user's client cannot make a connection to a weather station.

Question 1.2. A weather station fails after receiving a user's request for air pressure, and before sending a response.

Question 1.3. The air pressure data sent from the weather station to the user is lost in transit.

Question 1.4. A few weeks later, as part of the same group, you're asked to implement an advanced serialization protocol for transmitting temperature readings from a weather balloon to a base station. Develop a function that implements the serialization logic. The function accepts an `Array` of readings, each of which is represented by a `number`, and returns a `string`.

Note: Do not implement any network communication as part of your solution.

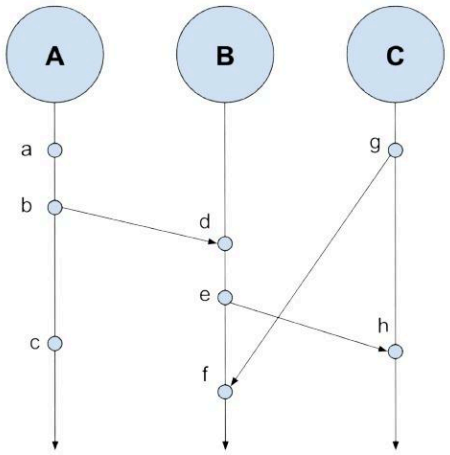
```
function serialize(readings: Array<number>): string {




}
```

Question 1.5. Over the summer, an intern proposes a method to guarantee that the system simultaneously meets requirements R1, R2, and R3: include a physical timestamp, generated by the weather balloon's onboard clock, directly in the serialized weather data. When the base station receives readings from different balloons, it will use these physical timestamps to immediately determine which reading is the most up-to-date.

The intern argues that this ensures all requests are processed (R2), up-to-date readings are always used (R3), even when stations go offline (R1). Explain how this approach actually *fails* to guarantee all three of R1, R2, and R3. In what situations does this approach break down?

2. Logical Time [25 pts]

Consider the following series of events in a three-node distributed system. The only nodes interacting in this system are these three nodes — i.e., there are no other, external nodes, with which any of these nodes is interacting. Time in this diagram flows downwards.



Note: diagram annotations are not required for 2.1 and 2.2; answer in the space provided below.

Question 2.1. Mark events b, f, and h with Lamport timestamps.

Question 2.2. Mark events b, f, and h with Vector timestamps.

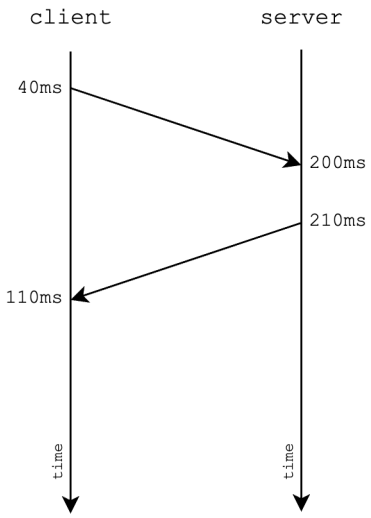
Question 2.3. List all the event(s) that occurred concurrently with e.

Question 2.4. List all the event(s) that "happened before" d.

Question 2.5. If event x has an earlier *Lamport* timestamp than event y, is it guaranteed that event x occurred *before* event y? If not, provide a brief counterexample from the given figure.

3. Physical Time [25 pts]

Imagine you are employed at *1380Games*, a game development company that is working on a new strategy game. While debugging some new features of the game, you notice a few things related to a client-server scenario:



Question 3.1. When observing a specific packet, you see that the client sent it to the server at a local physical time of `40ms`, and received it at `110ms`. The server received this packet at a local physical time of `200ms`, and responded at `210ms`. Calculate the round-trip delay and the clock skew.

Question 3.2. What should the client do in this situation to align with the server clock?

Question 3.3. These solutions work, but you envision a great future of this game with thousands of clients and multiple servers all working in tandem. You would like for all of these machines to have roughly the same physical time. What system could you use to implement this functionality, and how would it help?

Question 3.4. Your solution in 3.3 works and scales well, but you notice that sometimes clients will randomly disconnect. This is completely out of your control, but upper management at *1380Games* wants you to ensure that

- all clients have roughly synchronized clocks (when online) *and*
- clients can always continue playing the game, even if they temporarily lose connection.

Your manager asks you for a plan to solve this problem. What do you tell them?

Question 3.5. One of your coworkers wants to add an “action history” feature that shows the past actions of all the players in a lobby, and the order in which they were performed. It is critical that this order is accurate, as it may affect the strategies players use.

In the current implementation of your game, clients send *their local physical timestamps* to the server when communicating their actions. Is this information enough to establish the order of the actions taken? Why or why not? If not, what extra information or setup will you need to support this feature?

4. Function Properties [25 pts]

Consider the following shell program, which combines several software components:

```
cat f1.txt |           # read and output the contents of f1.txt
  shuf |               # randomly shuffle lines (line contents preserved)
  grep -E '[0-9]+' |   # filter for lines that contain at least one digit
  sort -n |            # sort in ascending numeric order
  tac |                # reverse the order of the lines
  head -n 1            # output the first line
```

The questions below ask you to pick at least one component invocation in the pipeline that has a certain property, write it down, and explain why it has that property. **If *no* component has that property, say so.**

4.1. Injectivity

A component invocation that is *non-injective,* and why:

A component invocation that is *injective (one-to-one),* and why:

4.2. Idempotence

A component invocation that is *idempotent,* and why:

A component invocation that is non-*idempotent,* and why:

4.3. Determinism

A component invocation that is *deterministic,* and why:

A component invocation that is *non-deterministic,* and why:

4.4. Range-Uniformity

A component invocation that is *range-uniform,* and why:

A component invocation that is not *range-uniform,* and why:

4.5. Streamability

A component invocation that is *streaming,* and why:

A component invocation that is *not streaming,* and why:

5. Sharding & Partitioning [25 pts]

Suppose you have a distributed system of four nodes: `10.0.0.1`, `10.0.0.2`, `10.0.0.3`, and `10.0.0.4`. You want to balance load across these nodes by having different objects stored on different nodes. You have four different hash functions which are used to generate node identifiers: `h1`, `h2`, `h3`, and `h4`. The output of each function for your nodes is shown below.

	h1	h2	h3	h4
a @ 10.0.0.1	0	0	0	0
b @ 10.0.0.2	30	1	1	1
c @ 10.0.0.3	60	12	1	2
d @ 10.0.0.4	90	3000	3	3

You want to distribute the product data across several nodes, which might increase as load increases. Each product has a unique `product_id`, and exists as an object with property names `product_id`, `name`, `category`, `price`, and `size` — for example:

```
{
  "product_id": 12,
  "name": "Soft Cotton Tee",
  "category": "Graphic",
  "price": 15.99,
  "size": "L"
}
```

Question 5.1. For each hash function (`h1`, `h2`, `h3`, `h4`), assess whether the resulting node identifiers would be suitable for load balancing in a distributed system. Explain why.

Question 5.2. Suppose you have six `product_ids: 4, 12, 23, 48, 63, 94`. Using (a, left column) **`h4`** for node identification and a simple modulo-based sharding, and (b, right column) **`h1`** for node identification and consistent hashing, where would you place these objects?

4 →	**4** →
12 →	**12** →
23 →	**23** →
48 →	**48** →
63 →	**63** →
94 →	**94** →

5.3. Assume a fifth node `e` joins, with **`h1(e) = 25`** and **`h4(e) = 5`**. According to **both** load-balance schemes in 5.2, how many products need to be relocated, and why?

5.4. Between (1) hyperspace hashing, (2) consistent hashing, (3) modulo-based hashing, and (4) centralized coordination, **which** would be most appropriate for this scenario and **why**?

5.5. You are asked to build a feature that allows users to search for products based on their category and size (e.g., "find all L sized products in the Graphic category"). This search will be used frequently in the application. The structure of the product data is *fixed* or *static* as defined above. Between the four schemes in 5.4, **which** would be most appropriate and **why**?

6. Gossip Protocols and more [25 pts]

You are designing a gossip protocol to share weather information across a network of 1000 nodes. Assume the following:

1. Only *one* node, the leader node, starts a rumor — i.e., new weather information is only relayed from the leader node, and everyone else hears about it via the gossip protocol.
2. Only one rumor is spread at a time — i.e., until the current rumor is heard by all nodes, the leader will not start any new ones.
3. Each message takes, on average, `12ms` to be sent, received, and processed.
4. A node can send an unlimited number of messages concurrently.
5. Each node is only aware of the *existence* of other nodes, and the information in the packets *it* receives.

Question 6.1. At a high level, show how this gossip protocol operates from the perspective of a node. In detail, include what nodes should do when they are sent weather information, and who they should relay this information to in various cases. Use a variable `fanout` (do not set its value, later questions ask you that) to to represent the number of nodes each node gossips to. Your node's behaviors should guarantee that every node in the network *eventually* receives the weather information, while minimizing the number of messages sent across the network (with respect to `fanout`, of course).

Do not violate any of the assumptions above, and pay particular attention to *Assumption 5*.

Question 6.2. With your protocol, if you want to propagate an entire rumor across the network in `12ms` on average, what should you set `fanout` to? How does this satisfy the constraint?

Question 6.3. If you wanted your protocol to send at most 1 message per node, how long could it take for the system to converge?

Question 6.4. Which of the broadcast guarantees discussed in class does your network violate when gossiping? Justify your answers, possibly including your assumptions.

Question 6.5. In `v2` of your protocol, you are forced to get rid of *Assumption 2*. This means that the leader could start a rumor before another has finished propagating. Assuming your protocol operates similarly, which of the broadcast guarantees discussed in class does your network violate when gossiping? Justify your answers.