HW4 Solutions: CS425 FA17

Proceed to next page
Question 1. (Solution and Grading by: Faria)

a. They are not serially equivalent. Conflicting operations on b and c are in the order (T2, T1), but conflicting reads and writes on a are in the order (T1, T2).
   i. Object a: read(a, T1); write(a, baz, T2)
   ii. Object b: read(b, T2); write(b, bar, T2); write(b, caz, T1); read(b, T1)
   iii. Object c: read(c, T2); write(c, foo, T1)

b. Serially equivalent. All conflicting operations are in the order (T1, T2).
   i. Object a: read(a, T1); write(a, baz, T2);
   ii. Object b: write(b, caz, T1); read(b, T2); read(b, T1); write(b, bar, T2);
   iii. Object c: write(c, foo, T1); read(c, T2);

c. They are not serially equivalent. Conflicting operations on a and b are in the order (T2, T1), whereas conflicting operations on c are in the order (T1, T2).
   i. Object a: write(a, baz, T2); read(a, T1);
   ii. Object b: read(b, T2); write(b, bar, T2); write(b, caz, T1); read(b, T1);
   iii. Object c: write(c, foo, T1); read(c, T2);

d. They are not serially equivalent. Operations on a and c are in the order (T1, T2) but operations on b are not.
   i. Object a: read(a, T1); write(a, baz, T2);
   ii. Object b: read(b, T2); write(b, caz, T1); read(b, T1); write(b, bar, T2);
   iii. Object c: write(c, foo, T1); read(c, T2);
Question 2. (Solution and Grading by: Faria)

a. Consider two objects A and B that have concurrent timestamps. Transaction T1 locks A first and transaction T2 locks object B first because picking in random order is allowed. Then T1 tries to lock B and T2 tries to lock A -- both of these will block. This deadlocks T1 and T2 (they are stuck waiting for each other).

b. To avoid deadlocks, we must come up with a policy that ensures that all processes will try to acquire and release locks on concurrent objects in the same order. An example policy may be that we consider the vector timestamps to be strings and simply order the strings lexicographically. The object with the smaller (lexicographical) timestamp is locked first.

For the proof, we assume that we build a wait-for graph for all the transactions in the system. According to the rules we have specified, objects with smaller IDs are locked first.

Proof by contradiction.

There is a deadlock in the system.

This means that there must be a cycle in the wait-for graph.

We traverse the wait-for graph. If all transactions are following the given rules, then each transaction is waiting to acquire a lock on an object with a higher ID than the object it has already acquired i.e., objects on which locks have to be acquired are in the order: A < B < C …

However, if there is a deadlock, then that means there is a transaction that is waiting to acquire a lock on an object with a lower ID i.e., we find a cycle in the graph that looks like A < B < C < A.

This is not possible (if all transactions follow the rules given).

Thus, there cannot be a cycle in the graph (or a deadlock in the system).
Question 3. (Solution and Grading by: Akshun)

a. Availability is violated as writes or reads will not be propagated. Consistency is satisfied as no reads are possible.
b. Availability is violated as writes are not allowed. Consistency is satisfied as all reads will return the same value.
c. Availability is violated as the partition which does not have quorum will not be able to write. Consistency is violated as partition 1 and partition 2 will then have inconsistent values.
d. Availability is satisfied as all writes and reads are processed. Consistency is violated as the two partitions can have inconsistent values of the same file (Write hello to F1 in NJ partition. Write bye to F1 in DC partition).
e. Consistency and availability are both violated because write results on the partition without a quorum will not be visible to reads in the other partition, and no reads are allowed on the partition without a quorum.
f. Consistency and Availability are both violated as writes are not allowed by the NJ partition and reads in the other partition return stale values because values are not from the DC datacenter to the NJ datacenter.
g. Consistency and Availability are violated as writes performed in DC datacenter will not reflect in NJ datacenter and no writes are performed the NJ datacenter.

Also accepted:
a. Availability **violated**  Consistency **satisfied**

b. Availability **violated**  Consistency **satisfied**

C. Availability **violated**  Consistency **violated**

D. Availability **satisfied**  Consistency **violated**

E. Availability **violated**  Consistency **violated**

F. Availability **violated**  Consistency **violated**

G. Availability **violated**  Consistency **violated**
Question 4. (Solution and Grading by: Rui)
Solution: A publish subscribe system has publishers which publish data (e.g., servers publishing sports scores) and subscribers receiving the data (e.g., clients receiving sport scores). There is little to no processing of data in publish subscribe systems. Kafka is a pub-sub system, but also has some additional functionalities.

(Optionally: Pub-sub systems may be topic-based or content-based. Examples are: topic-based -- get me all scores for match “Nadal vs. Federer”, and for content-based -- get me all stocks whose values are more than $50.30).

(Optionally: Kafka is a pub-sub system that can be connected to data sources. Latest version of Kafka also support some stream processing and stateful functionality. More typically though, Kafka is used to feed data into stream processing engines like Storm, etc.)

https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern
https://kafka.apache.org/documentation/#intro_topics.
Question 5. (Solution and Grading by: Rui)
At an abstract level Spark streaming and Storm/Heron are equivalent in that they process streaming data. However Spark streaming processes data in (micro-)batches while Storm/Heron process one tuple at a time. Spark runs on RDD (Resilient Distributed Datasets), which is why it uses micro-batches. Information link https://spark.apache.org/streaming/

(We also appreciate more detailed answers, as long as they are not too long).
Question 6. (Solution and Grading by: Akshun)

a. Can’t tell
b. Log - Changing the scale of the x axis to log scale will make the curve resemble log curve
c. Can’t tell
d. Zipf - this is a canonical example of a zipf distribution
Question 7. (Solution and Grading by: Ashwini)

a. Since the radio transmission of motes is very energy intensive, transmitting and relaying all the values over the sensor network will not be energy efficient.

In option-2, imagine that the sensor network is organized as a tree with child nodes relaying the measurements only to their parent nodes. Parent node computes the average of its local measurements and the measurements received from its children, and transmits the computed average value to its parent. This will result in fewer message transmissions.

b. Average is an algebraic measure. Each mote needs to relay: Local average computed on the measurements observed by the mote and its children, along with the total count. Given a new reading from a child \((\text{avg}_1, \text{count}_1)\) and current aggregated, \((\text{avg}_2 \text{ count}_2)\), the new output is calculated as \(\frac{\text{avg}_1 \times \text{count}_1 + \text{avg}_2 \times \text{count}_2}{\text{count}_1 + \text{count}_2}\). (Sum can be used instead of average, though the equations change.)

c. Median is not amenable to in-network aggregation, so each mote needs to relay all the individual measurement values to the base station.
Question 8. (Solution and Grading by: Mayank)

a) This state cannot happen. When a page is in ‘W’ state, only the owner has a copy. Hence, if P4 is holding it in Write mode, P3 can neither have it in Read mode nor be the owner.

b) P3 will multicast asking all other processes to invalidate their copies of the page (here, P4), then fetch all copies and use the latest one (from P4), becoming the owner. Then it can write to the page.

c) This state cannot happen. Two processes cannot hold a page in Write mode at the same time. In addition, the owner (here, P4) should be the only one holding the page in Write mode.

d) P3 will multicast and ask all other processes to invalidate their copies of the page (here, P4 and P5), then fetch all copies (or any copy) and use the latest one, becoming the owner. Then it can write to the page.

Grading Rubric -

- -2 points for an incorrect part
- -1 if identified wrong setup but gave incorrect reasoning
Question 9. (Solution and Grading by: Ashwini)

a. Incorrect, in order to verify the digital signature, the receiver will have to encrypt M with Ka_priv and then Hash it. Since only Alice has access to Ka_priv, the digital signature cannot be verified by the receiver. Decrypting with Alice’s public key doesn’t work either since the Hash is irreversible.

b. Incorrect, since Ka_pub is publicly known, anyone can generate the signature and pretend to be Alice.

c. Incorrect, since Ka_pub is publicly known, anyone can generate the signature and pretend to be Alice.

d. Correct, encrypted Hash(M) message can be decrypted using Ka_pub to recover the hashed message. This hashed message can be compared with the Hash(M).

e. Correct, M can be decrypted using Ka_pub

f. Incorrect, since Ka_pub is publicly known, anyone can generate the signature and pretend to be Alice.

g. Best option is d since Hash(M) reduces the size of the message that needs to be encrypted, transmitted and decrypted.
Question 10. (Solution and Grading by: Jayasi)

a) No. Here are two problems:
   i) Auto advancing read-write pointers: to satisfy at-least once invocation semantics, you can’t have auto advancing read-write pointers because it is not idempotent. With at-least once invocation semantics, suppose the write operation worked correctly, but the ACK reply failed. There will be another write operation, but because of the auto-advancing, it’ll create an incorrect state for data.
   ii) Being stateless: the servers shouldn’t maintain per file/session info, which is not the case in UNIX. So, if OS crashes, all the processes have also crashed. This is not expected of a distributed file system, where you can have fault tolerance, depending on the application use-case.

b) Move the read-write pointer to the client side, where:
   i) You can ensure that the read/write functions use absolute positions and remove access of file descriptors
   ii) As the clients will specify absolute position for their read/write operations, there won’t be any need for server to maintain any information. So, if a server fails, another can take over.

c) Any answer is acceptable. But what we are really hoping is you’ll say “none of these, I want to start my own company!” :)

d) Woah, that’s so cool! :)