Question 1:

a) Show an example of causal ordering violation involving point-to-point messages (when no causal delivery protocol is in place). Explain your example.

Answer:

m3 causally depends on m1 since the latter’s send event happens before the send event of the former, but due to network delays, it arrives before it, violating causality.

b) Expand the vector timestamp protocol for causal broadcast that we have seen in class to a matrix based protocol that can ensure causal ordering among point-to-point messages. Explain why it works. No need for a formal proof (but of course a correct formal proof is always acceptable 😊).

Answer: Here, a vector can no longer capture all the causal dependencies and we need to extend it to a matrix. Specifically, each node $p_i$ maintains a matrix $M_i$ such that $M_i[j,k]$ is the number of messages $p_i$ knows that $j$ has sent to $k$ (not delivered!).

The protocol:

When $p_i$ wishes to multicast a message $m$ to a subset $S$, it increments $M_i[i,k]$ for every $k \in S$ and attaches a copy of $M_i$ to $m$.

Whenever a process $p_k$ receives a message $m$ from $p_i$ with matrix $M_i$, then

- If for each $i \neq j$ $M_i[j,k] \leq M_i[j,k]$ and $M_i[i,k] = M_i[i,k]+1$ then
  - Pass $m$ to the application
  - For all $0<j,i\leq n$, $M_i[j,k] := \max(M_i[i,k], i,k,M_i[i,k])$
  - Check if any other message is now deliverable by the above rule

- Else
  - Buffer $M$

We only accept a message if we know about all other messages destined to us that were known to the sender when it sent the message. If we accept a message, we need to update the matrix to capture the most advanced causal dependencies.
**Question 2:**

a) Describe at least three differences between Kademlia's lookup/routing process and the one of Chord.

**Answer:** (i) In Chord the lookup process is recursive (passed from one node to the other) while in Kademlia it is iterative (the searching node drives each step of the search) (ii) In chord the process is sequential while in Kademlia it is parallel (up to alpha nodes are contacted concurrently at each step), (iii) in Kademlia nodes update their routing table during the search process, (iv) the metric used in Kademlia is XOR which is symmetric, (v) Kademlia’s lookup process might terminate even before reaching the successor if all K nearest known nodes have been contacted and they do not return any closer node.

b) As was stated in class and in the slides, for a random overlay in which nodes pick d neighbors in an I.I.D. uniform random manner and links become symmetric it is enough to have d=2 to ensure connectivity with high probability. Why does the Bitcoin network require d=10?

**Answer:** There are several reasons: (i) to reduce the risk of eclipsing attacks, (ii) to expedite the delivery of messages (larger fan-out means smaller network diameter), (iii) to increase resilience, (iv) in bitcoin neighbors are not chosen in a truly I.I.D. uniform random manner while d=2 only guarantees connectivity w.h.p. when the above is true – unfortunately, despite the hint in the questions’ text, no one mentioned this (which to you was fortunate since I could not reduce points for this omission).

**Question 3:**

a) What architectural feature of BigTable enables it to support atomic updates at a line border but prevents it from supporting atomic updates of multiple (fields in multiple) lines? Explain.

**Answer:** In BigTable each row is inside the same tablet, and all accesses to each tablet are handled by the same single tablet server. Hence, this server orders all accesses to each row and ensures atomicity. However, different lines might reside in different tablets and therefore managed by different tablet servers and since there is no synchronization between tablet servers, atomicity of such updates might be violated by interleaved updates to other rows.
b) What architectural feature of Spanner allows it to support this missing functionality in BigTable? Explain.

Answer: This is enabled by a combination of mechanisms: Data is sharded in Spanner across multiple replication groups. Each replication group has a leader that maintains a locking table using 2PL to ensure serializability of transactions in terms of accesses to objects inside the corresponding shard, and uses Paxos to coordinate and order all changes. In particular, locks are released only after commit has been decided by Paxos. When a transaction touches objects in multiple shards, the leaders of the respective replication groups agree on committing or aborting by running a 2PC protocol. Finally, TrueTime, a very accurate time service is used to associate a timestamp with each transaction that reflects its commit real time ordering. (They managed to combine 70% of the course material in one system ;)

Question 4:

a) Explain why waiting for a timeout even after receiving more than 2/3 pre-votes that are not all the same is required for Tendermint's liveness, whereas if all these votes are the same, there is no need to wait for the timeout.

Answer: Without this timeout, the Byzantine nodes can run the following liveness prevention attack. In each round they would immediately send pre-vote for nil. Since they need not wait for the proposer, their pre-votes would be likely to be included in the first 2/3+ that each correct node gathers. Hence, without the timeout no correct node would ever collect more than 2/3 pre-votes for any block. On the other hand, if 2/3+ votes for a given block or for nil have already been collected, waiting for additional votes would not change the result of this step (we already have a Byzantine quorum all supporting to do something) so there is no reason to keep waiting.

b) In the fast probabilistic shared coin Byzantine binary consensus protocol we have seen (Friedman, Mostefaoui, Raynal), show an example where the protocol would not provide safety if executed with n=3f+1 nodes (f the total number of faulty nodes of any kind).

Answer: With n=3f+1 there might be a violation of agreement. Here n-2f = f+1. Suppose f+1 correct nodes start with v while f correct nodes start with v' (v≠v'). In the first round (without the fast termination), suppose the coin falls on v. The Byzantine nodes send v to one of the nodes that started with v and v' to the others. Since this correct node that started with v found f+1 votes for v which is also the coin it then decides v. On the other hand, all other correct nodes might find f+1 votes for v' and therefore adopt it and ignore the coin. In the next round, all correct nodes that did not decide in the first round receive 2f+1 votes for v' (from themselves and the Byzantine nodes) and as this is also the value of the coin they decide v'.