**Question 1 (26 points):**

a. In class we saw 3 total ordering protocols, which we nicknames Sequencer, Orderer, and Rotating Token. For each such protocol, give one scenario in which it is likely to be better than the other two. Explain.

**Answer:**

**Sequencer** - whenever there is a single sender, or a small subset of senders and the messages are relatively small. In the single server case, this enables the sender to send without delay and requires a single protocol message per application message, which is optimal. When there is a small subset of senders and messages are small, the cost of the extra message incurred by the Orderer can be higher and just adds latency.

**Orderer** - there is a relatively small subset of servers and messages are large, ideally also when either there is laxity in the latency requirements, or the senders generate frequent messages. As above, in this case, the rotating token adds a significant latency for the rotation of the token. The sequencer is less efficient since it requires double transmission of large messages.

**Rotating Token** - when (almost) all nodes generate messages all (or most) of the time, and especially when the load is high or when using a shared medium like WiFi. When all nodes constantly generate messages, this implies one protocol message per application message. As we discussed, when the load is high or we have a shared medium, the algorithm provides a self-clocking congestion control, as it enables processes to send new messages only as fast as the network is able to deliver them and reduces collisions in the case of shared medium.

b. Extend the vector timestamp based causal broadcast protocol that we studied in class (in which each message is always broadcast to all group members) to enable causal ordering when each message can be sent to any subset of nodes, including point-to-point. Explain your solution.

**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 as many wrote!).

**The protocol:**

When $p_i$ wishes to multicast a message $m$ to a subset $S$, it increments $M_i[j,k]$ 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 $0 \leq j \leq n$, $M_k[j,k] = \max\{ M_i[j,k], M_k[j,k] \}$
  - 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 (30 points):

a. Describe a simple generic deterministic construction of a dissemination graph (overlay) in which the degree of each node as well as the diameter of the graph are both log(n). The construction should work for any value of n (and not just powers of 2 like in a hypercube). Hint: start with a logical ring.

Answer: Assume the nodes’ IDs are from 0 to n-1. We place them on a logical ring based on their ID. We then connect each node i to all nodes whose ID is \((i + 2^k \mod n)\) for \(k \in [0, \ldots, \log(n)-1]\). Here, each node connects itself to \(\log(n)\) other nodes while \(\sim \log(n)\) other nodes connect themselves to it. Hence, the degree is \(O(\log(n))\).

The diameter is also \(\log(n)\) since when routing from node A to B, in each hop we can traverse at list half of the remaining distance.

b. List 2 advantages and 1 disadvantage of a randomly constructed overlay like Araneola compared to an overly like the one you were requested to construct in the previous question (2.a.).

Answer:

Advantages of Araneola: (1) Simple local maintenance - when a node joins or fails/leaves, there is no need to rearrange the entire overlay. (2) a constant degree, which is smaller (for most values of n) then logarithmic degree => lower fan-out/in translates to lower load on nodes and lower bandwidth utilization.

Disadvantages of Araneola: (1) Diameter is slightly longer than \(O(\log(n))\). (2) Only probabilistic guarantees.

Question 3 (14 points):
Consider the following caching scheme that is proposed for DHTs: Whenever a node \(p\) that is responsible for a given key according to the DHT answers a request initiated by some other node \(q\), cache the reply of \(p\) in all nodes on the search path from \(q\) to \(p\). In which DHT do you expect this scheme to offer a more effective caching: Chord or Kademlia? Explain.

Answer: Chord is likely to benefit more from this type of Caching. Chord has a rigid structure where each entry of the finger-table has a single entry. Hence, paths to a given object are static (as long as there is not churn) and also likely to converge quickly, meaning that such caching is beneficial.

On the other hand, Kademlia has a pseudo parallel search process, and more choices due to the k-buckets. Hence, there are more search paths and therefore search paths are expected to join only closer to the holders of the object. Further, Kademlia’s overlay can adapt dynamically as nodes learn about each other from searches they perform and take part in. Finally, some of you noticed that in Kademlia some other form of caching was already proposed. When that optimization is activated, yet another caching is likely to have a limited impact.
Question 4 (30 points):

a. In the quorum replication protocol we've studied, as well as in Dynamo, in order to survive $f$ failures, each write is being sent to $n=2f+1$ replicas. On the other hand, in GFS, as we talked in class, each write can be stored only on $f+1$ replicas. Explain this difference? Which design goal of GFS vs. Dynamo justifies one option vs. the other?

Answer: Notice that in both cases, an update is not complete until the data is stored on $f+1$ nodes, in order to ensure that even if $f$ fail, at least one copy of the most updated version will remain. Also, notice that quorum replication provides strong consistency, so the difference does not relate directly to the choice of consistency level.

Rather, in GFS, the write is managed by a single server that orders all updates. If there is a failure, the system stalls until $f+1$ chunk servers become available once again. The tradeoff here is lower availability of specific chunks in exchange for bandwidth preservation and lower latency compared to the quorum replication protocol. Notice that chunk servers’ allocation is controlled by a master server, which is replicated using primary backup, whose split brain avoidance is ensured using a consensus service, which internally requires $2f+1$ nodes. So at the end, the crux of the issue here is that we decoupled the versioning task from the data storage task.

In contrast, quorum based protocols are decentralized and more highly available - as long as fewer than $f+1$ concurrent failures occur, they continue executing with no stalls. Compared to quorum replication, Dynamo offers weaker consistency in order to reduce the latency (no need to wait for the read repair to be acked by a quorum) and even greater availability (hinted handoffs and sloppy quorums enable availability even when we cannot get $f+1$ replies from the preference list). Unlike GFS, here the versioning task and the storage task are intertwined, so we must interact with $2f+1$ nodes.

b. BigTable shards (partitions) tables into smaller tablets. From a distributed systems perspective, what is the main benefit of doing this? What does BigTable sacrifice because of this? Explain.

Answer: The division into tablets enables scalability, as each tablet can be managed by a different tablet server. This way, the system enjoys greater parallelism and better utilization of its resources. The availability per row is also improved as a failure by a single tablet server only impacts a small portion of the entire table.

The main sacrifice is the inability to support serializability (or transactional semantics) across multiple table rows.
Submission instructions:
You must solve this exercise alone – submissions are individual. Solutions must be submitted through the course web site – either printed or a high-resolution scan. Solutions must be written in Hebrew unless you get an authorization from Prof. Friedman to submit in English.

The submission date is Monday 23/01/2017 before midnight.

Good luck!