CPS 512 midterm exam #2, 11/18/16

/50
/75
/45
/30
/200

Your name please: ____ NetID:____________

Answer all questions. Many of the questions are **true/false**: indicate whether each statement is true or false by placing a T or F to the **left** of the statement. You can place additional notes or conditions to the right. T/Fs are 5 points each. Other written questions are 10 points each.

Many of the questions pertain to systems for updating replicated data, e.g., Bayou, Dynamo, Lamport's replicated mutex service, RSM **consensus** (VR/Viewstamped Replication, Paxos, Raft), or Practical Byzantine Fault Tolerance (PBFT). For these questions, i and j refer to replicas, and u and w are updates.

P1. Keep it causal. Consider a system in which nodes maintain logical clocks and vector clocks in the typical fashion: each element v[i] of a vector clock v[] is a logical clock value of the corresponding node i. Consider two events e1 and e2 that are timestamped with vector clocks v1[] and v2[]. T/F:

If v1[i] < v2[i], then e1 happened-before e2.
False. For example, e1 and e2 could have happened concurrently on different nodes other than i, and which had their last transitive contact with i ("heard from" i) at different times in the past.

If e1 occurred on node i, and v1[i] < v2[i], then e1 happened-before e2.
True: the node where e2 occurred had "seen" e1, which occurred on node i at logical clock v1[i].

If e2 occurred on node i, and v1[i] < v2[i], then e1 happened-before e2.
False. For example, e1 and e2 could have happened concurrently, and the node where e1 occurred last "heard from" node i at some time in the past.

If e1 occurred on node i, and e1 happened-before e2, then v1[i] < v2[i].
False. This is a little tricky. It cannot be that v2[i] < v1[i], but it is possible that v1[i] == v2[i], e.g., if e2 happened on a different node, and it last heard from i immediately after e1.

If e2 occurred on node i, and e1 happened-before e2, then v1[i] < v2[i].
True. Since i's logical clock is incremented upon event e2, no event e1 that happened-before e2 could have seen the state of i after e2, at logical clock v2[i], so it must have a lower/earlier logical clock value in v1[i].

P2. An epidemic of updates. Consider an asynchronous replication system with propagation of updates by randomized pairwise anti-entropy exchanges, as in Bayou. T/F:

If replica i learns of update u before i learns of update w, then u happened-before w.
False: u and w could have happened concurrently.

If u happened-before w, then i learns of u before learning of w.
True: Bayou preserves a causal order.

If u happened-before w, and i knows of both u and w, and i passes u to j in an exchange, then i also passes w to j if j does not already know of w.
True: this ordered flooding of updates is the key trick that enables Bayou to preserve a causal order.

If u happened-before w, and i knows of both u and w, and i passes u to j in an exchange, then it is not possible that j already knows of w.
True: if j knows of w, then it has learned of updates in an order that violates causal order.

If u is known to i but is not known to j, and w is known to j but is not known to i, then u and w are concurrent.
True. If either had happened-before the other, then a replica that knows of the later event must also know of the earlier event, or else it has learned of updates in an order that violates causal order.

P3. Commitment to learning. Consider an asynchronous replication system with propagation of updates by randomized pairwise anti-entropy exchanges, as in Bayou. Updates commit in the order they are received by a designated primary replica. Replicas learn of commitment via later anti-entropy exchanges (transitively) with the primary. T/F:

[You were asked to presume that "learned before" could mean "learned in an earlier exchange", although the answers don't change if it is the same exchange. You were asked to presume that faulty nodes eventually recover and partitions eventually heal.]

If i learns that update u committed before it learns that update w committed, then u happened-before w.
False. Update u is ordered before w now and forever, but this ordering is arbitrary: they could have been concurrent.

If i learns that u committed before i learns that w committed, then all replicas eventually apply u before w.
True. Any replica that applies both u and w (which all will do eventually) applies u before w.

If i knows of u and w and then learns that u and w have committed, then it may be necessary for i to reorder u and w in its log. [relative to one another]
True. If they were concurrent, the primary could have learned of them in a different order than i learned of them, and so could have committed them in a different order than i learned of them. But it would have had to commit them out of timestamp order.

If i knows of a committed update u and then learns of an uncommitted update w with a lower accept stamp than u, then u and w are concurrent.
True. Certainly u could not have happened-before w, if w has a lower accept stamp (logical clock) than u. And w could not have happened-before u, or else any node that knows of u would also know of w (causal order).

If i knows of a committed update u and then learns of an uncommitted update w with a lower accept stamp than u, then i orders w before u in its log.
False. All committed updates are ordered before all uncommitted updates, regardless of their accept stamps.

P4. Agreement of orders. Replication systems deliver/apply updates in various orders. Presume there are no network partitions unless explicitly stated. T/F:

In Bayou: all non-faulty replicas eventually agree on a total order of committed updates.
True. [Presuming no partitions or that any partitions eventually heal.]

In **consensus**: all non-faulty replicas eventually agree on a total order of committed updates.
True. [Presuming no partitions or that any partitions eventually heal.]

In Bayou: application code at a replica i can execute an update u before i learns that u has committed.
True. Application code executes even uncommitted updates, but they are tentative and may be rolled back later.

In **consensus**: application code at a replica i executes an update u only after i learns that u has committed.
True. Just because.

In Bayou: a connected set of non-faulty replicas eventually agrees on a total order of all updates that they know, even if they are disconnected from the primary by a network partition.
True. If connected, they eventually exchange all their updates and any commits, and the updates are ordered by accept stamp and/or by commit order, which everyone knows.

In Bayou: a replica that is disconnected from the primary by a network partition may be forced to reorder some pair of updates u and w in its log when the partition heals.
True. These updates could be concurrent and the primary may have learned of them (and committed them) out of accept stamp order. Any node that knows of both, but not of their commitment, would order them by accept stamp.

In **consensus**: no update can commit on the minority side of a network partition.
True: committing requires a majority vote.

In Bayou: no update can commit on the minority side of a network partition.
False: the primary may reside on the minority side of a network partition, and it will commit: nobody can stop it.

In Bayou: a replica receiving two updates u and w knows if they are concurrent.
False. It only knows their logical clocks, and logical clocks can't tell us if two events are concurrent.

In Dynamo: a replica receiving two updates u and w to a given data item knows if they are concurrent.
True. Updates to a given item have vector clocks that can be compared to determine concurrency.

A bonus question: In Akka: message delivery preserves causal ordering (e.g., as demonstrated in Lab #1).
False. Akka preserves "program order", but not causality for cases involving three or more nodes.

P5. Quorum rules. These questions apply to **quorum** sets in systems that use voting among a fixed configuration of N replicas. A quorum is a set of replicas that vote for some outcome, and for which the set is large enough to cause that outcome to occur. They include "fuzzy quorum" sets in Dynamo, in which a read quorum of R voting replicas completes a read and a write quorum of W voting replicas completes a write. T/F:

[You were asked to presume that these protocols are operating normally, within accepted parameters (e.g., no more than f failures).]

In Dynamo: the intersection of a read quorum and another read quorum contains at least one replica.
False. There is no requirement for this. For example, R could be 1 and N could be 3 or more.

In Dynamo: the intersection of a read quorum and a write quorum contains at least one replica.
False. It is true of R+W>N, but there is no requirement for this in Dynamo.

In **consensus**: for a committed update u, at least one replica that voted for the current primary also voted for u.
True. Primary election and commitment both require a majority vote (including the primary), and every majority quorum overlaps every other in at least one replica.

In PBFT: for committed update u, at least one non-faulty (honest) replica voted for the primary and also voted for u.
True. If N = 3f+1 then primary election and commitment both require a quorum of 2f+1 replicas: f of them could be faulty, but there are at least f+1 honest nodes in any quorum. A 2f+1 quorum could be missing at most f of the N=3f+1 nodes, so even if two quorum sets are missing a different f nodes, there must be at least one node in common. And in this case there are at most f nodes in one quorum that are missing from the other: even if all the missing nodes were honest nodes, there is at least one remaining honest node in common.

In **consensus**: if two views have primaries P and Q that were duly elected, then at least one replica voted for both.
True. Primary election requires a majority vote, and every majority quorum overlaps every other in at least one replica.

P6. Fragile foundations. In 2011, a company called DigiNotar reported that a private key had been stolen. This incident generated worldwide concern given DigiNotar's role as a Certifying Authority (CA) trusted by the vendors of major web browsers. [20 points]

Outline briefly how an adversary in possession of DigiNotar's private key could use it to mount a Man-in-the-Middle attack on any secure web (HTTPS) connection, e.g., a user's session with gmail. What other powers must the adversary possess?

The adversary (M) can forge a certificate for any target domain T, containing the public key of a keypair that M controls. If M can intercept a client C's communication to and from T (e.g., by DNS spoofing or by controlling a network element on the path to T), then M can intercept a connect request from C to T and return its forged certificate to identify itself as T. The browser trusts DigiNotar, and so it will accept the certificate. C opens an SSL/TLS connection to M believing that M is T. M may connect to T and represent itself as C. M can now mount a man-in-the-middle attack, and can read or modify any of the communication between C and T.

To defend against hacking from the DigiNotar leak, users were urged to upgrade their web browsers and change the passwords they use for Web services. Why are these steps necessary?

DigiNotar's public key is baked into the browsers, so the only way to revoke a browser's trust in the stolen keypair is to upgrade the browser.

If a client C had been targeted for a man-in-the-middle attack to domain T, then M had access to C's password at T, and could have stolen it.

P7. More crypto [30 points]

1. Alice and Bob are back together. Today Alice wants to send Bob a message that is secret and also authenticated, so that Bob "knows" the message came from Alice. Alice and Bob have keypairs and each knows the other's public key.

How should Alice send the message? How should Bob validate the message? Briefly explain why your scheme works, and note any additional assumptions.

The trick here is for A to encrypt the message with A's private key (so Bob knows it came from A) and again with Bob's public key (so nobody else can read it). The order does not matter, but A and B must agree on the protocol. A could make things more efficient by encrypting a hash of the message with her private key instead of the entire message (a digital signature), and/or by encrypting the message with a symmetric key added to the message encrypted with B's public key (like SSL/TLS). We assume that nobody had their private key stolen, or all bets are off.

2. **Secure hash** functions are fast and cheap. Briefly describe how Practical Byzantine Fault Tolerance (PBFT) uses hashes. What could go wrong in the view change protocol without them?

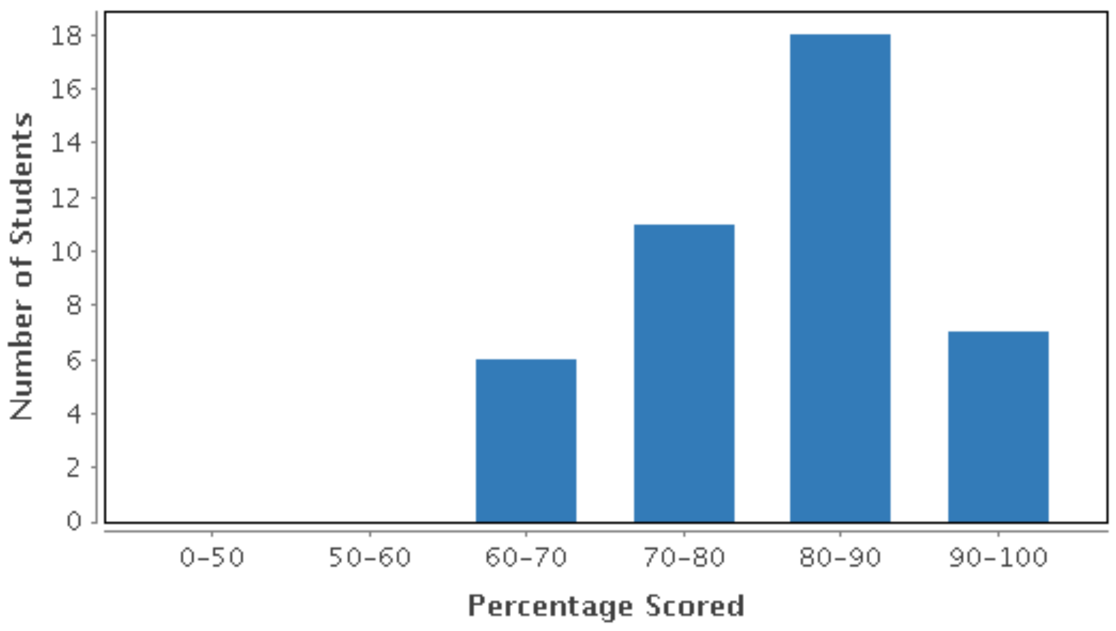
PBFT uses digital signatures on all messages, so that an honest node can prove what other nodes said to it, and a dishonest node cannot lie about what other nodes said to it. Without this the leader of a new view cannot distinguish whether a node claiming that some update u committed in a previous view is lying or not, since there may be only one honest surviving from the previous view.

A digital signature of a message M is a hash over M encrypted with the private key of the signer/issuer.

3. Often a server returns a **certificate** in response to a connect request. The certificate contains the server's public key. Presuming that the certificate is valid, what does the client do with the server's public key?

The question refers to SSL/TLS. The client makes up a session key for the connection and passes the session key to the server, encrypted with the server's public key.

Grade Statistics for Midterm #2



Average (Mean) Score	159.74 / 200 (79.87%)
Median Score	160 / 200 (80%)
Standard Deviation	16.49
Lowest Score	130 / 200 (65%)
Highest Score	190 / 200 (95%)
Total Graded Scores	42

Scores were a little higher this time around. In particular, some very strong students who inexplicably did “less well” on the previous midterm obviously put in some extra effort. (Thanks!) The tail of the distribution also came up by a good margin. The median is up by about 10 points.