CMPS128: Final Examination review topics

Peter Alvaro

May 28, 2016

Here is a short document that covers some topics that I think are important to review for the final examination.

It is not intended to be comprehensive in any way. After all, if we could effectively condense course material into a document, why bother making you attend a course?

My hope is that this document will create questions on Piazza and in our review session, as well as drive discussion in study groups.

This is a first draft of a dynamic document.

1 Definitions and intuitions

1.1 Agreement protocols

Agreement protocols attempt to satisfy three properties:

1. **Agreement:** if any processes decide some value V, then all processes decide V.
2. **Validity:** if a process decides a values V, then V was proposed by some process.
3. **Termination:** if a process proposal a value, eventually some value is chosen.

In theory, no protocol satisfies all three in all executions. In practice, consensus protocols such as Paxos and Raft come very close.

Two-phase commit (2PC) guarantees Agreement. As we saw in class, it does not guarantee termination. In the event that the coordinator and a single participant crash, the other participants (assuming none of them has received a commit or abort message) cannot safely decide whether to commit or abort. In practice, 2PC does not guarantee Validity either—even if all participants "want" to commit, the failure of a node or loss of messages could cause the system to decide to abort. This violates Validitity because `Abort` was not "proposed" by any process.

Leader election protocols typically guarantee Termination and Validity, but not Agreement. This is because in the event of partitions and delays they can choose multiple leaders.

Paxos (and consensus protocols in general) guarantee Agreement and Validity but not Termination. We saw how the "duelling proposers" problem can lead to infinite executions that do not decide. We also saw how techniques such as leader election can be used in concert with basic Paxos to make these nonterminating executions arbitrarily unlikely.

1.2 Partitioning

We discussed in class how ideally, we would like a partitioning algorithm with the following qualities:

1. **Uniformity**: Data and/or computation is spread evenly among the nodes.
2. **Locality**: The decision about where to place data and/or computation can be made without communicating with other nodes.
3. **Cheap Reads**: Reads should be efficient.
4. **Cheap Writes**: Writes should be efficient.
5. **Stability**: When nodes are added or removed, data should be "shuffled" as little as possible.

We discussed how random and round-robin partitioning strategies fail to achieve Cheap Reads, how directory services fail to achieve locality, and how hashing fails to achieve stability. We then discussed how consistent hashing achieves stability (but sacrifices some locality, as in the case of Chord).

2 Protocols

2.1 Broadcast

There are a variety of algorithms that can provide broadcast guarantees. Here we sketch some.

2.1.1 FIFO Broadcast

State: A local sequence number N, a sequence number S_i for each sender i (initialized to 1), and a message buffer B.

1. **On Send**(M) **at host** H**:** Increment N, and include N in the message (M, N, H).
2. **On Receive**(M, S, H), where M is a message payload, S is its sequence number, and H is the identity of the sender: Buffer (M, S, H) in B.
   (a) If there is a message $m = (M, S, H)$ in B such that $S = S_H$, deliver M and increment S_H. Remove m from B, and repeat.

2.1.2 Causal Broadcast

State: A set D of previously-delivered messages, initially empty, and a buffer B.

1. **On Send**(M) **at host** H**:** include D in the message (M, D).
2. **On Receive**(M, T), where M is a message payload and T is a message set: Buffer (M, T) in B.
   (a) If there is a message $m = (M, T)$ in B such that for all m in T, m is in D, deliver M and add m to D. Repeat.

2.1.3 Reliable Delivery

State: A buffer B for outgoing messages and a buffer A of acknowledgements.

1. **On Send**(M) at host H: place (M, H) in B and send (M, H).
2. **On Receive**(M, H)**:** deliver M and send a message `Ack` to H.
3. **On Ack**: remove (M, H) from B
4. **On Timeout:** If there is a message m in B, resend m.

2.1.4 Reliable Broadcast

Assume reliable delivery State: A buffer B for incoming messages.

1. **On Send**(M): send M.
2. **On Receive**(M, H)**:**
   (a) Reliably deliver (M, H) to every other node.
   (b) Deliver (M, H).

2.2 Commit

Note that these protocols are sketches: real implementation involve additional details including logging to stable storage.

2.2.1 Two-phase commit

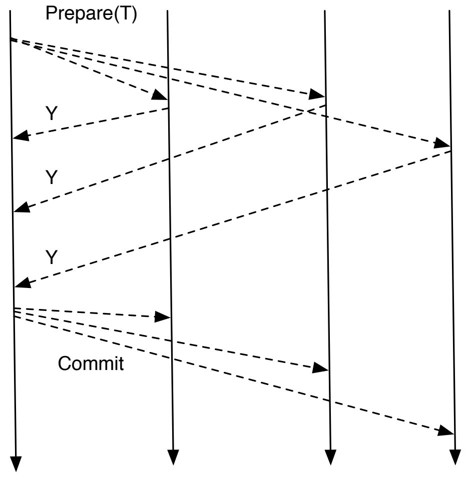
Roles: A *Coordinatior* and a set of *Participants*.

When the coordinator is ready to commit a transaction T, it sends a message `Prepare(T)` to all participants.

When a participant receives a `Prepare(T)` message, it checks if it can commit transaction T. If it can, it tentatively commits T and sends a `Vote(Yes)` message to the coordinator. Otherwise, it sends `Vote(No)`. (Note that if a participant votes `No`, it knows the transaction will be aborted and can throw away its tentative work.)

If the coordinator receives a single `No` vote, or if it times out before receiving a vote from all participants, it sends an `Abort` message to all participants.

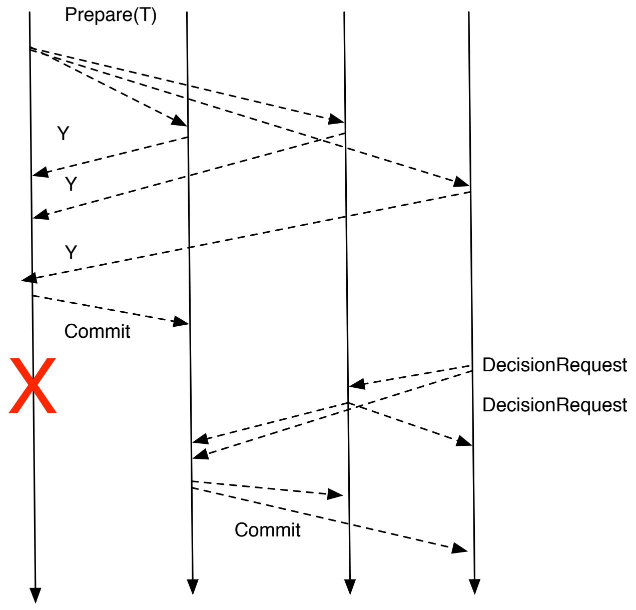
Otherwise, the coordinator has received a `Yes` vote from *all* participants. It then sends a `Commit` message to all participants. If a participant receives a `Commit` message, it can commit its tentative work.



2.2.2 Collaborative Termination Protocol

If a participant times out after sending a `Yes` vote to the coordinator, it sends a `DecisionRequest` message to the other participants.

Upon receipt of a `DecisionRequest` message, if a participant has received a `Commit` (respectively, `Abort`) message, then it responds with `Commit` (`Abort`).



2.3 Consensus

2.3.1 Basic Paxos

Roles: A set of *Proposers* and a set of *Acceptors*.

State:

Phase 1 A proposer that wishes to obtain consensus on a value `Val` chooses a unique and monotonically-increasing round number N and sends a message `Prepare(N)` to a majority of acceptors.

In Lamport's own words: " If an acceptor receives a prepare request with number N greater than that of any prepare request to which it has already responded, then it responds to the request with a promise not to accept any more proposals numbered less than N and with the highest-numbered proposal (if any) that it has accepted."

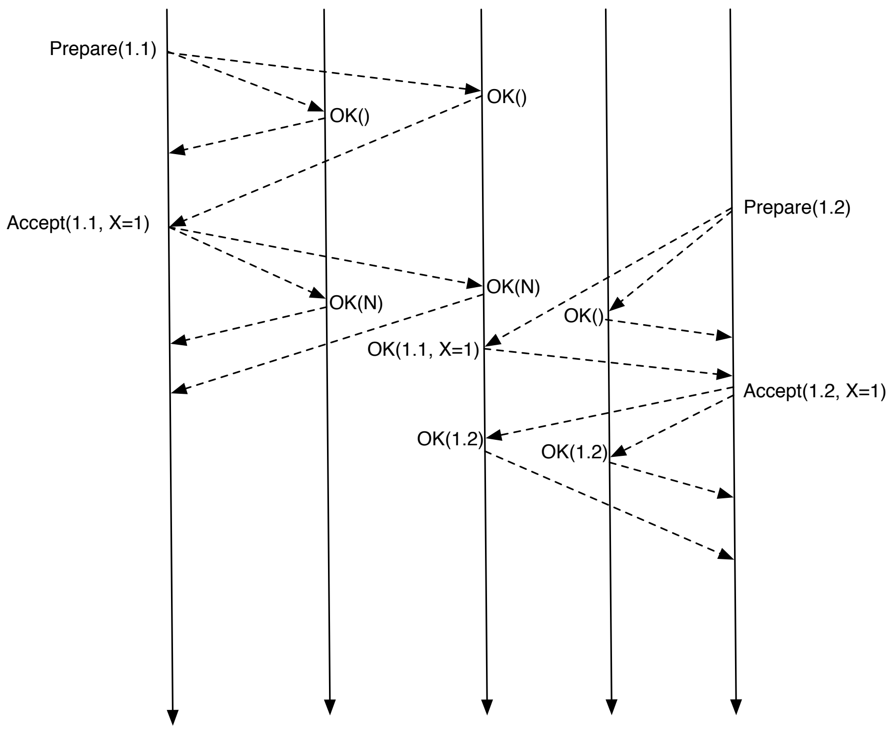
Note that the acceptors must return both the proposal number and the value of the previously-accepted proposal, if one exists.

Phase 2 When a proposed receives a response to its `Prepare(N)` message from a majority of acceptors, it sends an `Accept(N, Val)` message to those acceptors. N must be the same number it used in its `Prepare` message. Val must be either

1. The value of the highest-numbered proposal returned by an acceptor in response to the `Prepare` message, or

2. If no such value exists, the proposer may choose its own value.

When an acceptor receives an `Accept(N, Val)` message, it accepts the value if has not responded to a `Prepare(N')` request for some $N' > N$.



If we look at the three rightmost process lines, we see the common (interference-free) case of a basic paxos run. The prepare (for round number 1.1) succeeds (neither of the acceptors have responded to a higher-numbered prepare message). The proposer then sends an `Accept` that is *unconstrained*, because neither acceptor had already accepted some value. Both acceptors accept the proposer's value (`X=1`)—when the proposer receives their `OK` messages (which indicate that they have accepted the proposal `1.1`), it knows that its value has been chosen.

Then along comes proposer 2 (the rightmost process line). It starts a proposal numbered 1.2. The prepare also succeeds, because one acceptor has never responded to another prepare, while the other has responded to 1.1, which is lower. However, this acceptor responds that it has already accepted the value `X=1` with round number 1.1. Now proposer 2 may ask its quorum of acceptors to accept a value, but its choice of value is constrained: it *must* use `X=1`.