LECTURE 14

Speculative Paxos

Designing Distributed Systems Using Approximate Synchrony in Data Center Networks

Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma,
and Arvind Krishnamurthy, *University of Washington*

https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/ports

Premise

- Many distributed systems are designed independently from the underlying network.
- Thus, they assume worst case situations
  - Completely asynchronous
- Many distributed applications today are deployed on data centers
  - Network predictable and reliable
- Can we synergize the distributed system design with the network to improve performance?

Key contributions

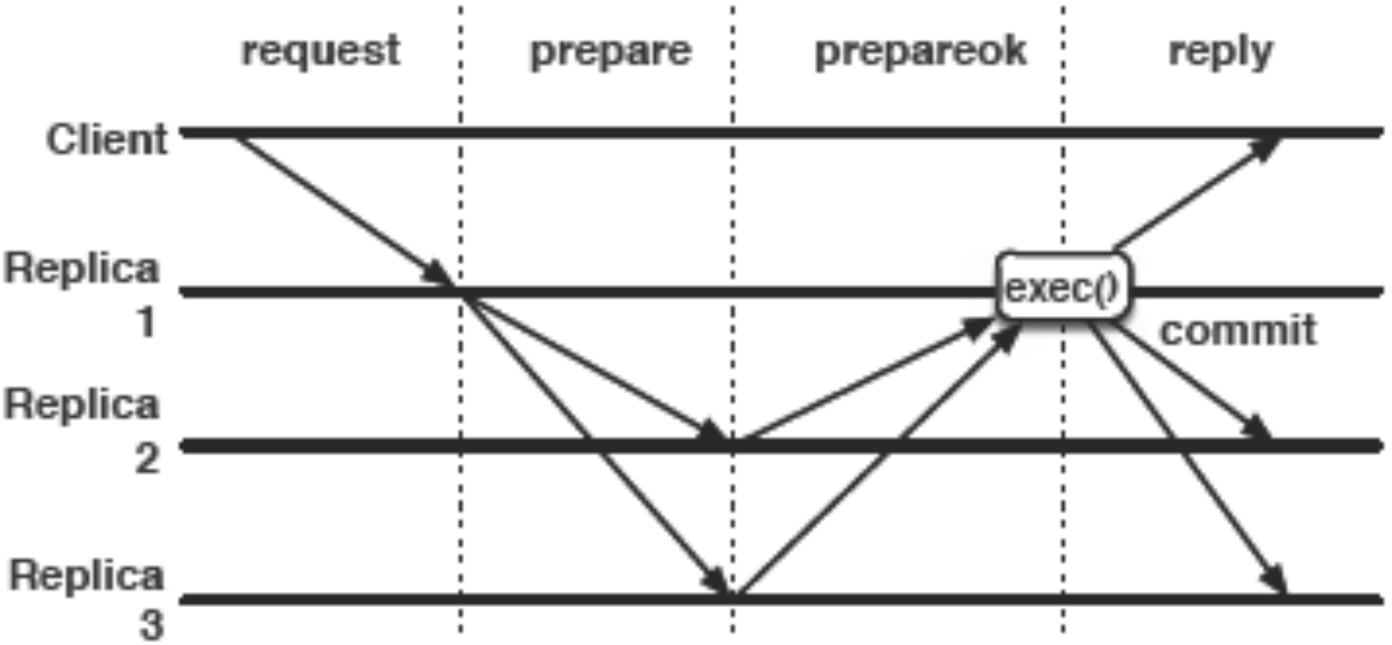
- MOM: Mostly ordered multicast
  - Tries to redesign multicast so that notifications arrive at recipients in order
    - Remember that multicasting is a key primitive in many distributed systems (e.g., communications among replicas)
  - Gives almost total ordering.
- Based on this – one can realize speculative Paxos
  - Commit before full consensus (as with Paxos or leader-based Paxos)
  - Because of MOM – very few cases require reconciliation or repair

Data center attributes

- Predictable: Structured topologies → easier to understand routes and latencies.
- Reliable : Packet losses are rare.
- Extensible
  - Use of technologies such as software-defined networking (SDN) allows flexibility in routing and in-network processing (using VMs or containers) of packets.

- Why do we care? → Can help choose routes so as to obtain ordered multicast.

Recall : Leader based Paxos



- Throughput suffers because of the load on the leader.
- If leader fails, a replica needs to take over as leader.

Network structure

- A single administrative domain and an OpenFlow type SDN controller
    - Allows implementation of customized forwarding rules.
- Organized structure of multi-rooted trees of switches
    - Leaves are top-of-rack switches (to which all machines on the rack connect to)
    - Top of rack switches connect to an intermediate tier of switches called “aggregation tier”
    - These in turn connect to a “core tier” at the top level.
- Control messages can be prioritized (both for transmissions and avoiding drops)
    - So latencies and losses of coordination messages can be drastically reduced.

Where are the replicas?

- Google's spanner and similar systems use one replica group per shard.
- Hundreds of thousands of shards per data center.
- Replicas that belong to a group may be on different racks but typically belong to the same cluster
  - To simplify cluster management and scheduling.

Mostly-ordered Multicast Goal

- We don't want to go with a dedicated leader.
  - As discussed, it leads to bottlenecks and delays among other things.
- How to provide ordered commits (even if in some "rare" cases, we need to fix things) ?
- Mostly-ordered multicasts!
  - To set the tone we first see what we mean by "totally ordered multicast"

Totally ordered multicast

- Clients communicate simultaneously with a group of N receivers (here, replicas)
- If process n_i processes message m before m', then any other node n_j that receives m' must process m before m'.
  - Ensuring this property holds during failures – is a problem equivalent to that of consensus.
- MOM considers a relaxed version – i.e., the above requirement does not have to hold in every case.

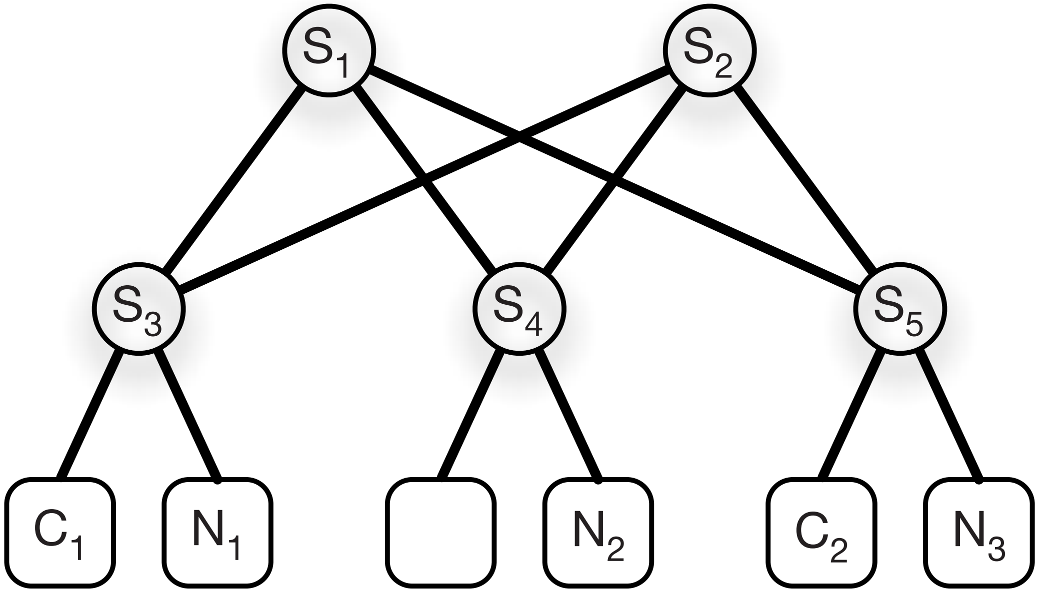
Mostly-ordered multicast property

- The requirement for total ordered multicast is satisfied with high frequency
- Occasionally the following ordering violations can occur:
  - n_i processes m after m’
  - n_i does not process m at all (because it was lost)
- While it can be implemented using a best-effort network primitive, takes advantage of network properties.
  - However, application code must handle ordering violations due to failures.

Principles of MOM

- Topology awareness → ensure that all multicast messages traverse same number of links.
  - Eliminates reordering due to path dilation
- High prioritization → Highest QoS class
  - Eliminate reordering due to queuing delays
  - Packet drops due to congestion
- In network serialization → route through a single root switch
  - Eliminate non-failure related reordering

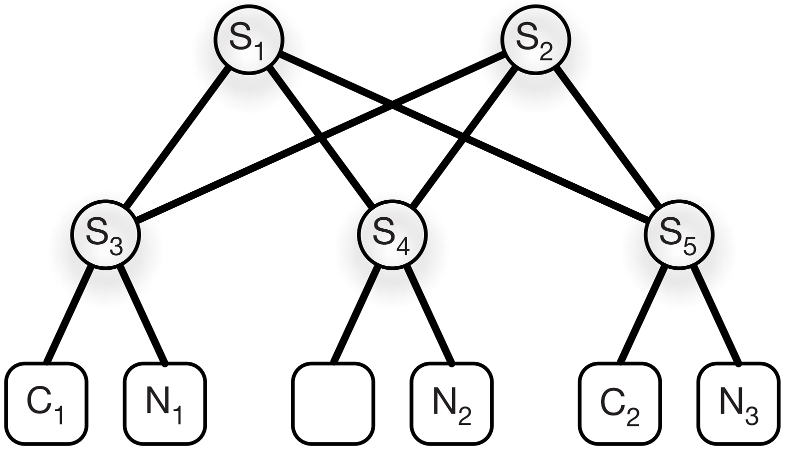
Example



- Clients C_1 and C_2 communicating with a multicast group N_1, N_2 and N_3. The three receivers share a multicast IP address.
  - This address is not shared by sub-nets.

Example continued

- Message from the client sent to root switch S1 or S2.
- The root switch makes 3 copies and sends it down
  - All the three copies traverse same number of links.
  - The flexible routing can be realized using SDN
  - Fault tolerance by routing around failure (if enough redundant paths) or changing root if needed (see paper)



Other attributes

- Even if one of the root switches is more congested than the other, the prioritization comes to the rescue.
  - If S2 is more congested than S1, it is ok since both prioritize the MOM messages.
- But still some variations can occur. So this is handled using in-network serialization i.e., use same top level switch.
  - Acts as a serialization point – the order in which this switch sees the messages is the order in which they are sent.

MOM and consensus

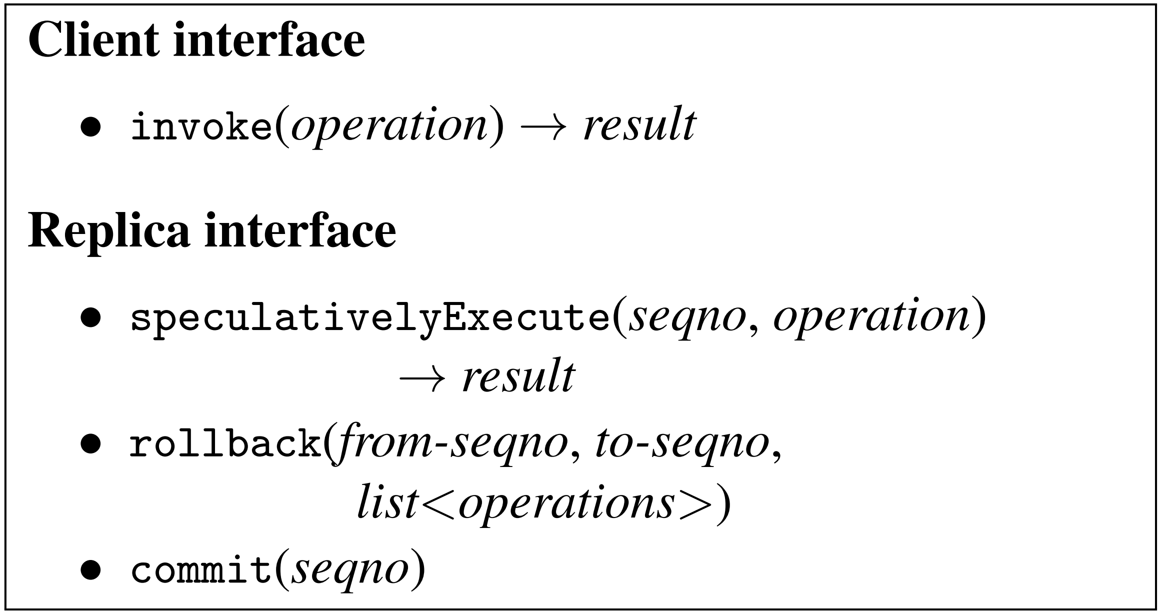
- How should MOM's properties affect how consensus is achieved ?
- What does it give us?
  - Approximate synchrony → strong ordering during the common case, but this strong ordering property can be violated during occasional failures.
- How do we take advantage of this ?
  - Speculative Paxos

Basics

- Speculative Paxos is a state machine replication protocol.
- Common case :
  - Rely on MOM
  - Speculatively execute requests (before agreement is reached).
    - Minimum latency (only two message delays)
    - High throughput → no need to communicate between replicas on each request.
  - Reconciliation via rollback when there are inconsistent operations.

What does it guarantee ?

- Linearizability if there are no more than *f* failures.
- Operations appear in consistent sequential order.
  - Each operation sees the effect of the operation before it.



Client interface

- `invoke`(*operation*) → *result*

Replica interface

- `speculativelyExecute`(*seqno*, *operation*) → *result*
- `rollback`(*from-seqno*, *to-seqno*, *list*<*operations*>)
- `commit`(*seqno*)

Speculative execs

- Replicas execute operations before agreement is reached.
- Upon receiving the speculative Paxos library makes the speculative execution upcall to the application.
  - Attributes are the operation and a sequence number.
- When there is a failed speculation a rollback is invoked to undo the most recent operations and return to a known state
  - Informs application about all operations/sequence numbers to be rolled back.
- Commit upcalls for those previous speculative operations that are never going to be rolled back.

Failure model

- Crash failures.
- Remains correct under the same assumptions as Paxos or Viewstamped replication
  - $2f+1$ replicas will provide safety as long there are no more than f replica failures.
- Liveness as long as messages are repeatedly sent and eventually delivered before the recipients time out.
  - Same as Paxos
  - This requirement is necessary because of FLT theorem (impossibility of consensus in an asynchronous system).

Protocols

- Speculative Paxos consists of three protocols.
- Normal execution : Speculative processing commits requests efficiently.
  - Messages are ordered
  - Less than $f/2$ replicas have failed
- Synchronization : Verify that replicas have speculatively executed the same requests in the same order.
- Reconciliation: Ensures progress when requests are delivered out of order or when between $f/2$ and f replicas have failed.

Replica states

- NORMAL : Allow speculative processing of new operations.
- RECONCILIATION: The reconciliation protocol is being applied (more later).
- RECOVERY : Failed replica is reconstructing state.
- RECONFIGURATION: Updates to replica memberships.

Replica log

- Log is a sequence of operations executed by the replica
  - Each has a sequence number.
  - State is either SPECULATIVE or COMMITTED
    - All COMMITTED operations precede SPECULATIVE operations.
- Each log entry has a summary hash
  - A hash of the previous summary and current operation
    - Why ?

Two replicas that agree on a summary for an entry *n*, have the same ordering of operations up to that entry.

Standard checkpointing can help truncate logs

View service

- System moves through a series of views
  - Each has a view number and leader
  - Leader can be selected using round robin ordering (recall election)
  - Leader's role is to mainly co-ordinate synchronization and reconciliation.

Speculative Processing

- Client requests an operation
- Speculative Paxos library sends request to all replicas
  - client ID, operation, request identifier
  - Using the MOM primitive --so mostly operations are ordered.
- Replicas speculatively execute request and send “SPECULATIVE-REPLY” messages to the clients.
- Client checks to see if $f + \lceil f/2 \rceil + 1$ (a superquorum) responds and if they do, commit transaction.
  - The operation needs to be consistent across replicas in terms of the current operation and a summary hash (which ensures that the order upto this transaction) are identical across replicas.

Failure or Success

- If responses don't match (i.e., there is no superquorum) or if there aren't sufficient responses before a time out, the client initiates a reconciliation.
- Note if there is a success, the replicas don't know that the operation has committed
  - This is taken care of later in reconciliation
    - Even if there are failures, the operation is not rolled back.

Why not just a quorum ?

- This is because of speculative execution.
- Let us say instead of a superquorum we only needed quorum (*f+1*).
- Let us say the client only got exactly *f+1* responses.
  - Let the other f speculatively applied some other transaction.
- The client commits the transaction, but the replicas have only done it speculatively.
  - So if even one fails, the recovery protocol cannot distinguish the order.

How does superquorum help?

- Now we can have only < *(f/2)* replicas apply some other transaction.
- So even if another *(f/2)* fail from those in the superquorum we have *(f+1)* replicas applying the operation correctly.
  - We can still tolerate *f* failures.
- This helps establish correct order during reconciliation.

Synchronization

- Since during the speculative processing the replicas do not know that a superquorum has committed to the operation, they need to do so.
- They do this via a leader initiated -- periodic synchronization – say every t seconds.
- A synchronization message has the current view number, the highest sequence number in the log (including speculative transactions) and the hash associated with the summary log.
- Leader commits if there are more than a superquorum of replicas that agree on these.
  - Confirms to all replicas (learning in Paxos – so that everyone can commit)
    - If no superquorum, go to reconciliation.

Divergence

- Reconciliation is invoked whenever there is a divergence across replicas
  - Sequence number of operations, or hash summary.
- Every replica stops processing new client requests and enters reconciliation phase.
- They send reconciliation messages to each other and to the leader.
- The leader will now need to take up the complex process of reconciling inconsistent replica logs.

Reconciliation messages

- Contain the view “v” for which the reconciliation is needed.
- The view “v_l” (prior) for which the status was normal.
- The logs of that replica

Merging of logs – step 1

- The leader needs to get messages from “f” replicas.
- It considers the logs with the highest v_l and retains all the entries upto that point.
  - These entries are viewed as normal and thus need to be maintained (previous reconciliations are respected).

Merging of logs – Steps 2, 3, 4

- Selects the log with the most COMMITTED entries.
  - These operations are known to have succeeded, so they are added to the combined log in COMMITTED state.
- Starting with next sequence number, check if the majority have the same summary hash for that number.
  - If yes, add to log in SPECULATIVE state
  - Repeat for each sequence number
- Gather other entries (sequence numbers), choose an order add to log in speculative state.

Final step

- Leader starts a new view and sends this log with that new view number to all replicas.
- When replicas receive, they install the new log
  - Roll back operations that are not in the new log
  - Execute operations in ascending order according to the log.
- Now the new state is set to normal and the replica starts speculative processing of new requests.
  - See paper for discussion on ensuring progress (liveness).