High Availability

Dominic Duggan
Stevens Institute of Technology
Based in part on material by Ken Birman

Types of reliability

- Recoverability
  - Restart in a sensible state
- High availability
  - Operational during failure
  - **Replicate critical data**

Replication for High Availability

- Active replication (State machine)
  - Peer-to-peer replicas
  - Each replica is **deterministic** state machine
  - **Operations** executed in same order on all replicas
  - All updates are totally ordered

Replication for High Availability

- Passive replication (Primary-backup)
  - Primary replica with pool of backups
  - **Operation** executed on the primary
  - **Updates** performed in same order on all replicas
- Hybrid
  - Ex: Primary-backup where operation executed on all replicas

Uses of replication

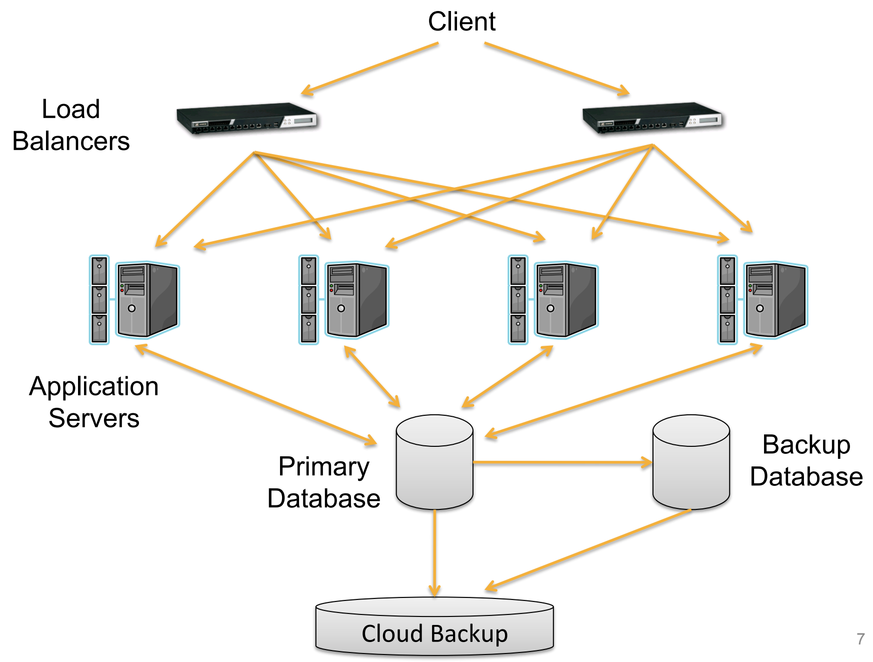
- High availability
- Share loads and improve scalability
- Replicate locking or synchronization state
- Replicate membership information in a data center (routing)
- Replicate management information to tune performance

5

Transactional Replication

- One-Copy Serializability (1SR)
  - Effect of transactions on *replicated* data items are same as if performed serially on *single* data items
  - Key: Failures and recoveries must be serialized with respect to transactions
  - Reason: Updates only performed on *available copies*

6



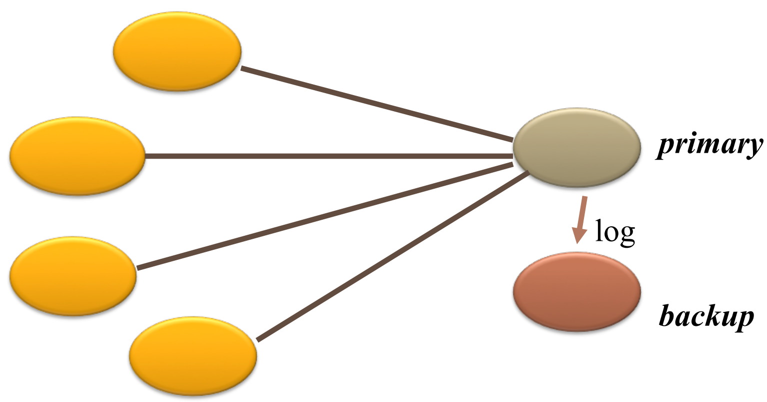
7

Server replication

- Primary sends log to backup
- Backup replays the log
  – applies committed transactions to its state
- If primary crashes, backup can take over

8

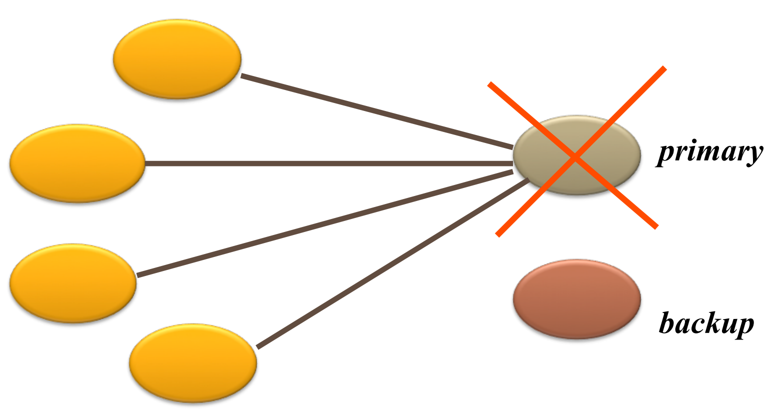
Primary/backup



Clients initially connected to primary, which keeps backup up to date. Backup tracks log

9

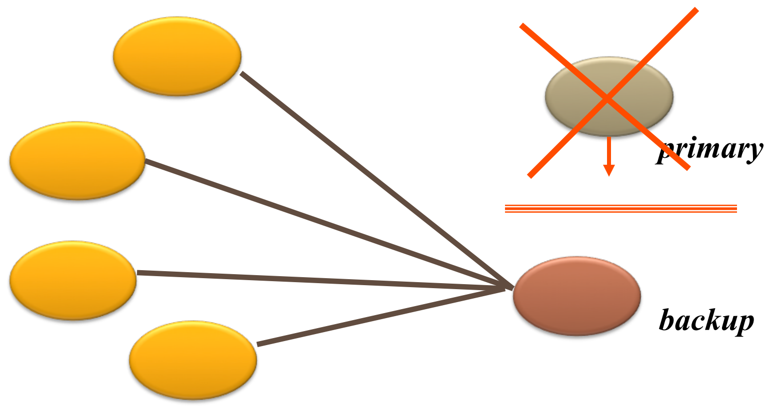
Primary/backup



Primary crashes. Backup sees the channel break.

10

Primary/backup



Clients detect the failure and reconnect to backup.

11

SERVER REPLICATION ISSUES

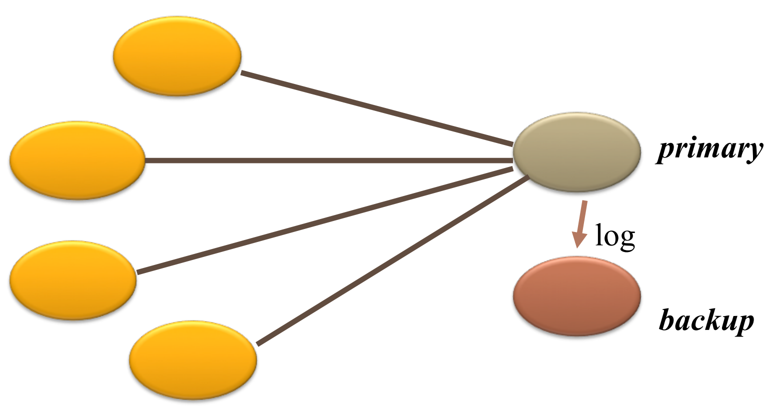
12

Issues

- Under what conditions should backup take over?
  - "Split brain" problem
- Theoretically needs 2PC to ensure that primary and backup stay in same states!

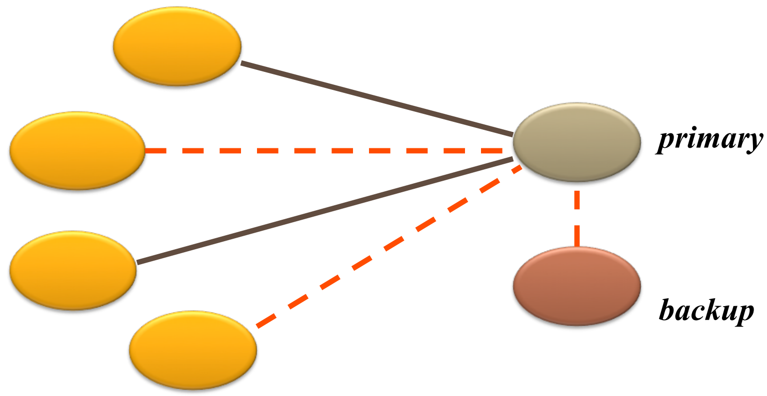
13

Split brain



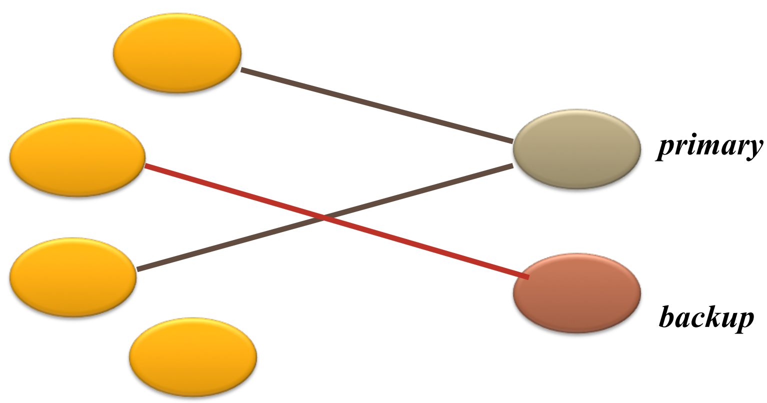
14

Split brain



15

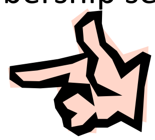
Split brain



16

Solutions

- Single server with restart
- Allow backup to "kill" the primary
  - Process groups membership service
- "Majority vote"
  - Quorum consensus



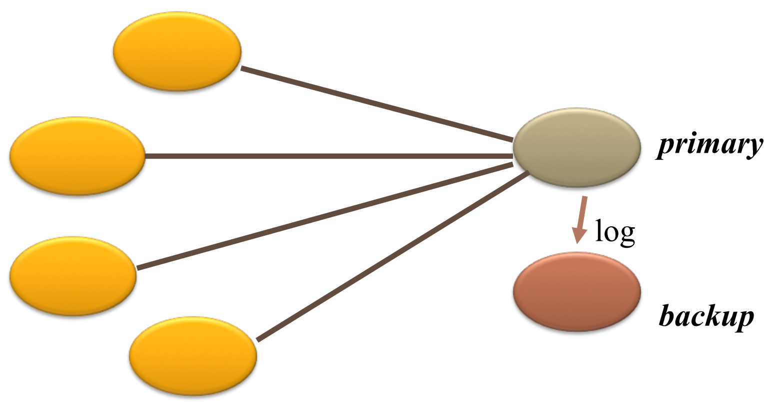
17

Issues

- Under what conditions should backup take over?
  - "Split brain" problem
- Theoretically needs 2PC to ensure that primary and backup stay in same states!

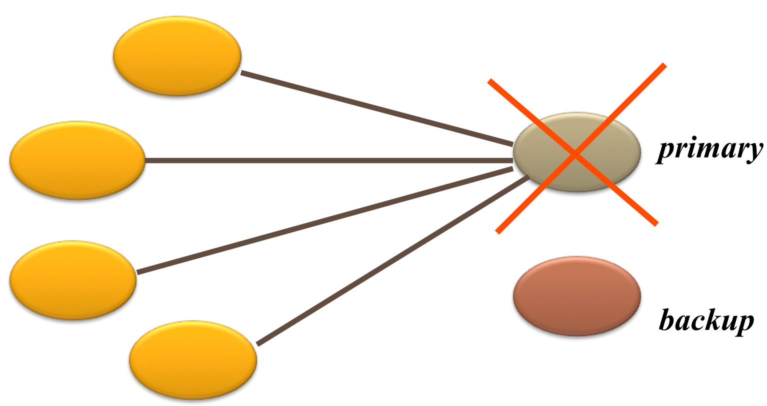
18

Primary/backup



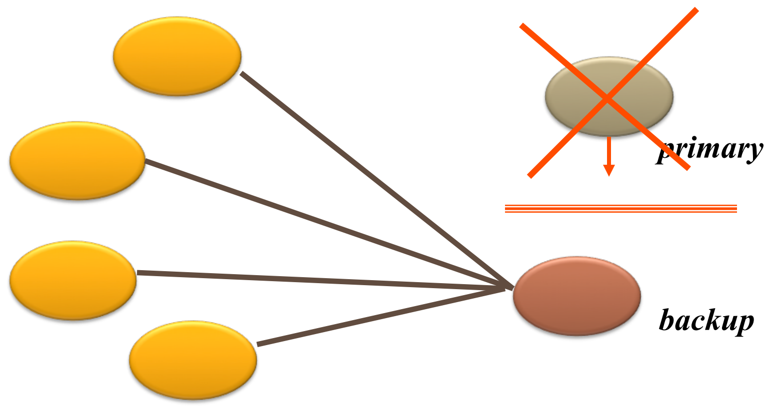
19

Primary/backup



20

Primary/backup



Backup state could be stale

21

Real systems

- Primary-backup with logging
- Omit the 2PC
  - Backup may lag state of primary
  - Hardware solutions?
    - Shared disk
    - Only one can write the disk – "token"

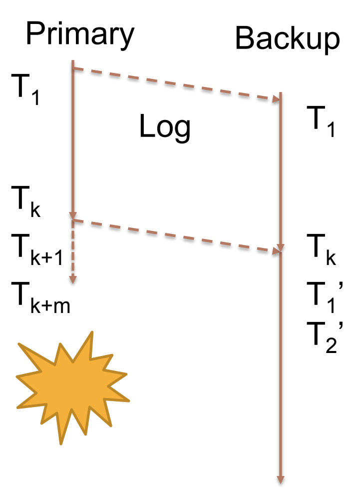
22

Reconciliation

- Fix transactions impacted by loss of tail of log
  - Apply the missing updates
  - Cascaded rollback
  - Worst case: human intervention
- Similar to compensations in long-lived transactions

23

Reconciliation



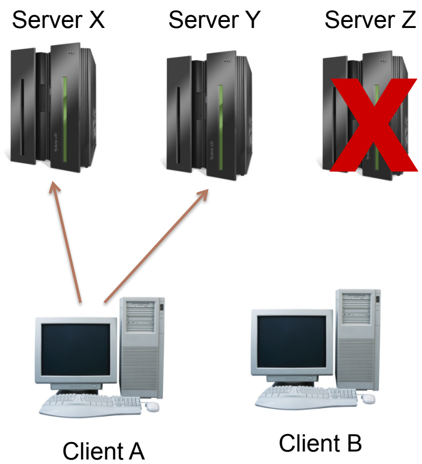
Reconciliation: Merge $T_{k+1},\ldots,T_{k+m}$ into $T_1',T_2',\ldots$ while preserving data consistency

24

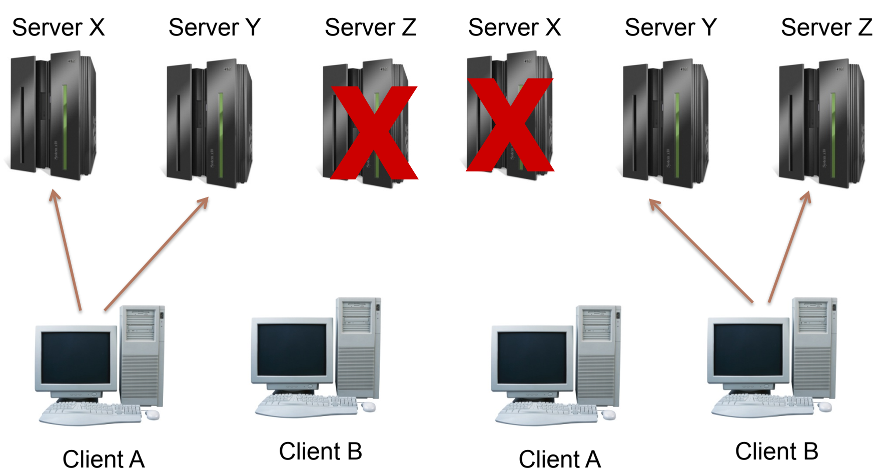
STATE MACHINE: QUORUM CONSENSUS

Issues

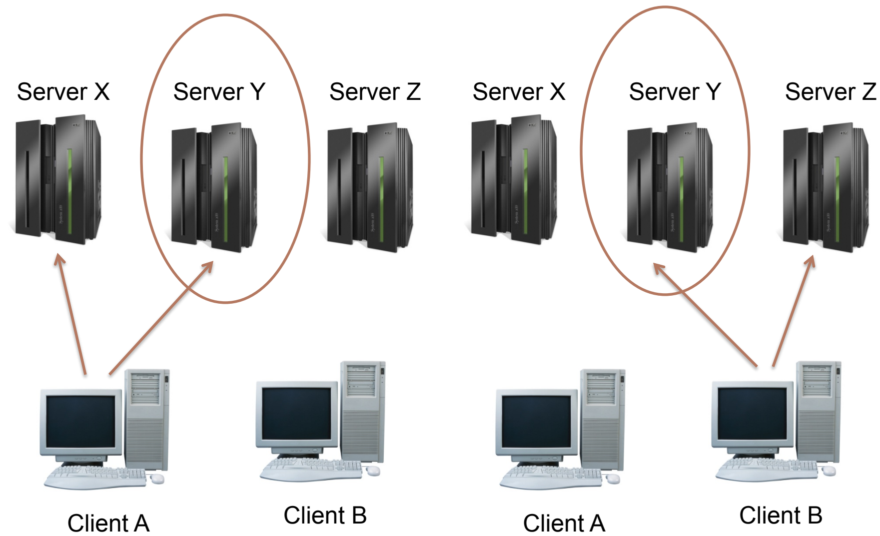
- How do we avoid split brain?
- How do we ensure agreement on order of updates?


Server X
Server Y
Server Z
Client A
Client B

27


Server X
Server Y
Server Z
Server X
Server Y
Server Z
Client A
Client B
Client A
Client B

28



Quorum Consensus

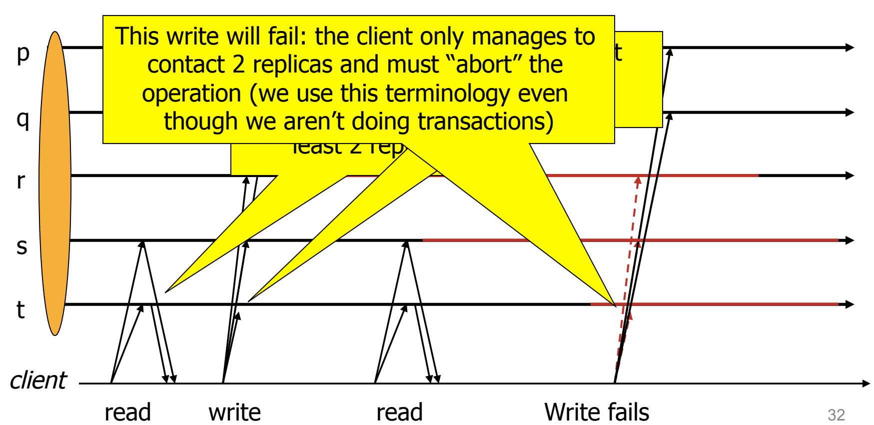
- Each replicated object has an update and a read quorum
- Rules
  - A quorum read should "intersect" any prior quorum write at ≥ 1 processes
  - A quorum write should also intersect any other quorum write
- So, in a group of size N:
  - $Q_r + Q_w > N$, and
  - $Q_w + Q_w > N$

Quorum example

- X is replicated at {a,b,c,d,e}
- Possible values?
  - $Q_w = 1, Q_r = 5$ (violates $Q_w+Q_w > 5$)
  - $Q_w = 2, Q_r = 4$ (same issue)
  - $Q_w = 3, Q_r = 3$
  - $Q_w = 4, Q_r = 2$
  - $Q_w = 5, Q_r = 1$ (violates availability)
- Probably prefer Q_w=4, Q_r=2

Static membership example

Q_{read} = 2, Q_{write} = 4



Issues

- How do we avoid split brain?
- How do we ensure agreement on order of updates?

33

STATE MACHINE: ORDERING UPDATES

34

Versions of replicated data

- Replicated data items have "versions"
  - I.e. can't just say "X_p=3".
    - X_p has *timestamp* [7,q]
    - X_p has *value* 3
  - Timestamp
    - must increase monotonically
    - includes a process id to break ties

35

Read

- Wait until Q_R processes reply
- Use value with largest timestamp
  - Break ties by looking at the pid
  - For example
    - [6,x] < [9,a]
    - [7,p] < [7,q]
  - *Even if a process owns a replica, it can't just trust it's own data*

36

Write

- Can't support incremental updates
  - x=x+1
  - Insert into a queue
- Quorum
  - Use a commit protocol
- How to determine the version number
  - Voting protocol

37

Protocol

1. Propose the write: "I would like to set X=3"
2. Members "lock" the variable against reads*, put the request into a queue of pending writes,* and send back:

   "I propose time [t,pid]"

   Time is a logical clock.
3. Initiator collects replies, hoping to receive Q_W

$\geq Q_w$ OKs

$< Q_w$ OKs

Compute maximum of proposed [t,pid] pairs.

Commit at that time

Abort

38

Voting based on logical time

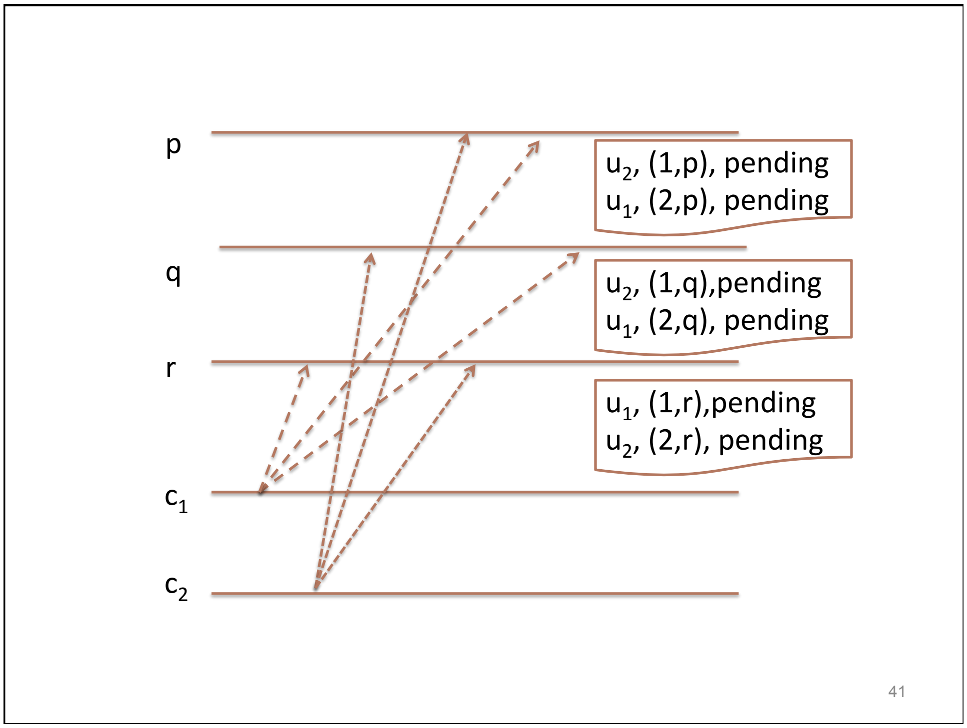
- Logical clocks
  - See mutual exclusion algorithm with logical time
- Update source takes the maximum
  - Commit message: “commit at [t,pid]”
  - Group member: if vote considered:
    - deliver committed updates in timestamp order
  - Group members: if vote not considered:
    - discard the update

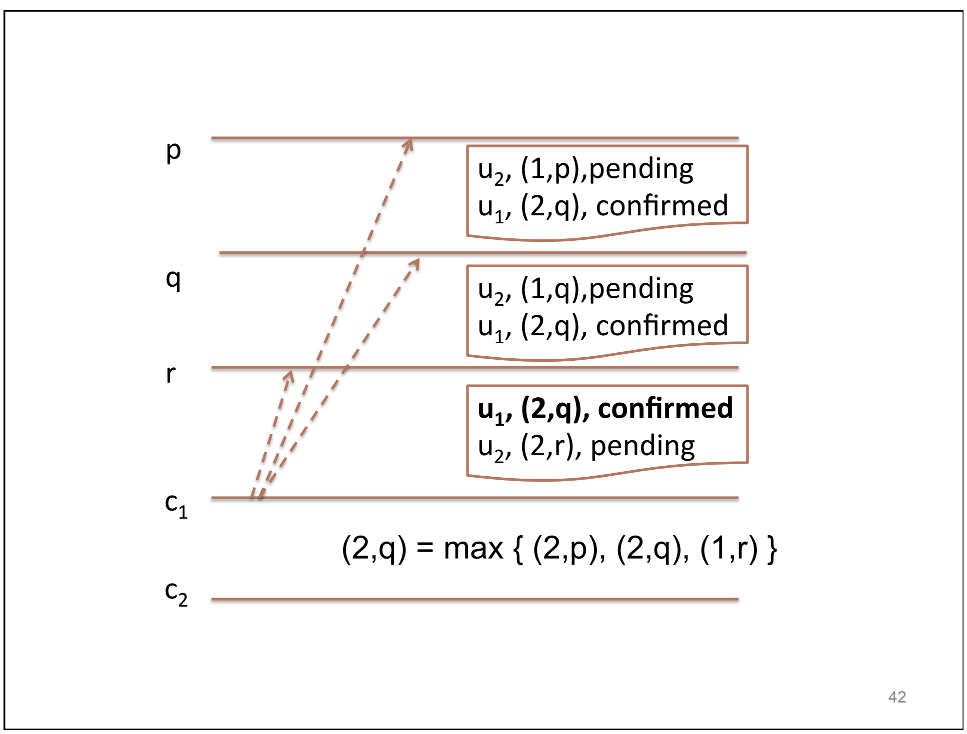
39

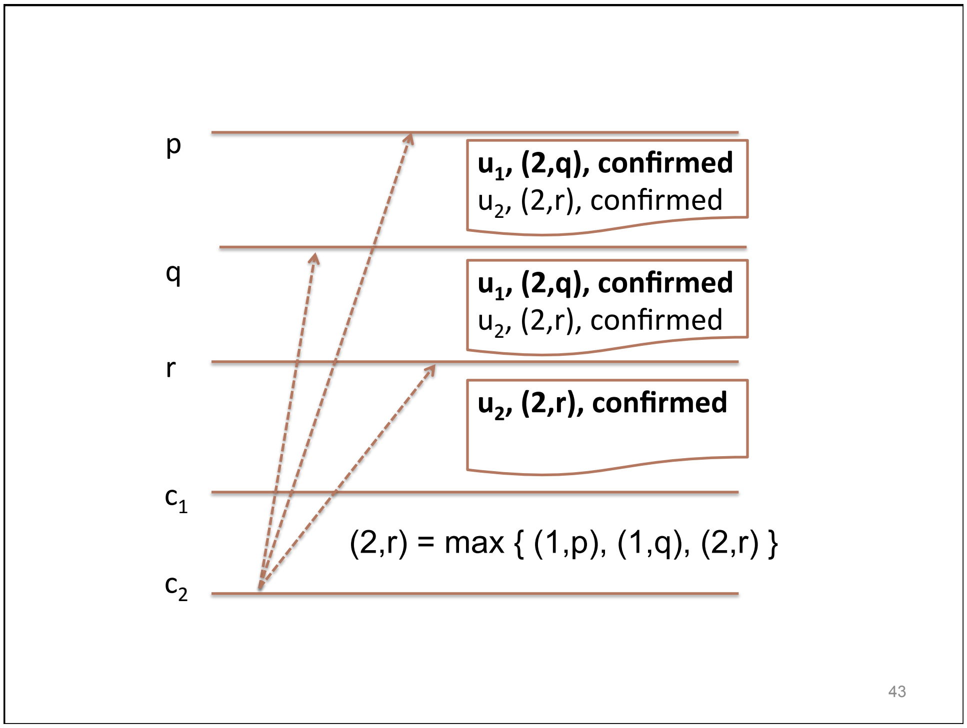
Where are the updates?

- Each member: queue of uncommitted updates
  - Survives crash and restart
- Example: Process p
  - (u_2: [1,p] pending), (u_1: [2,p] pending)
  - Neither can be delivered

40


p
q
r
c_1
c_2
u_2, (1,p), pending
u_1, (2,p), pending
u_2, (1,q),pending
u_1, (2,q), pending
u_1, (1,r),pending
u_2, (2,r), pending
41


p
q
r
c_1
c_2
u_2, (1,p),pending
u_1, (2,q), confirmed
u_2, (1,q),pending
u_1, (2,q), confirmed
u_1, (2,q), confirmed
u_2, (2,r), pending
(2,q) = max { (2,p), (2,q), (1,r) }
42



STATE MACHINE: PROTOCOL ANALYSIS

44

What if “my vote wasn’t used?”

- Process
  - had a pending update
  - discovers it wasn’t used
- Discard the request
  - Otherwise block forever (why?)
  - Ignoring the request won’t hurt (why?

45

Which votes got counted?

- Need to know which votes were “counted”
  - E.g. suppose A,B,C,D,E and they vote:
    - {[17,A] [19,B] [20,C] [200,D] [21,E]}
  - Vote from D is lost
    - the maximum is picked as [21,E]
  - Remember that the votes used to make this decision were from {A,B,C,E}

46

Recovery

- First recover queue of pending updates
- Next, learn the outcome of the operation
  - Contact Q_R other replicas
- Check if own vote counted (if committed)
  - If so, apply update
  - If not, discard update

47

Read requests while updates pending…

- Suppose a read while updates pending
  - Wait until those commit, abort, or are discarded
  - Otherwise process might not see its own updated value

48

Why is this "safe"?

- Commit: only move pending update to later
  - Discard pending update if vote not counted
  - Result: inconsistent replica
    - *but we always look at Q_R replicas*
  - Why we can't support incremental operations (insert, etc)

49

Why is this "safe"?

- Commit: moves pending update to front of Q
- Once a committed update reaches front of Q:
  - …no update can be committed at earlier time!
- Any "future" update gets later time

50

Why this works

- Everyone uses same commit time for an update
  - Can't deliver update unless [t,pid] is smallest
    - and is committed
  - Hence updates in same order at all replicas

51

Observations

- The protocol requires many messages
  - Could use IP multicast for first and last round
  - Need reliability
- Commit messages must be reliably delivered
  - Otherwise uncommitted updates on front of Q…
- 2PC and 3PC may block
  - FLP: *any* quorum write protocol can block

52

Risk of blocking

53

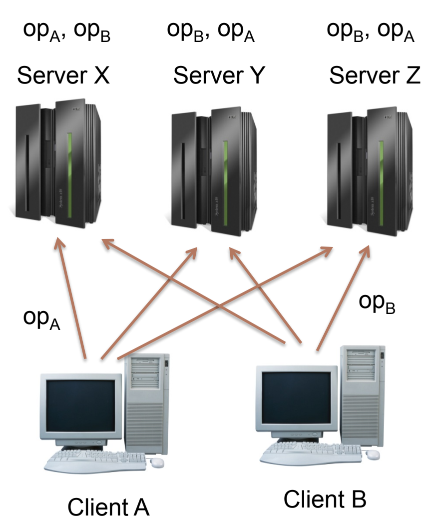
PRIMARY-BACKUP: VIEWSTAMP REPLICATION

54

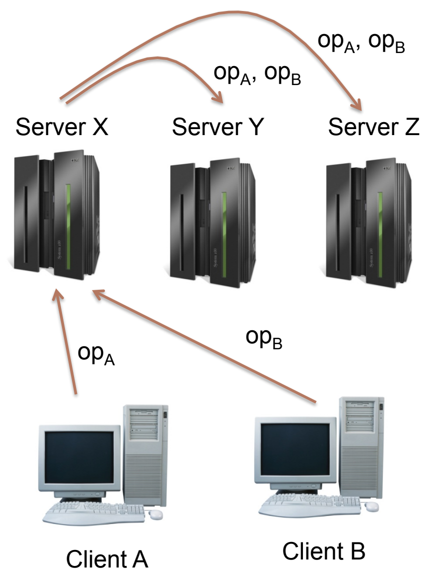
Quorum Consensus

- Crash-stop failures
- Requires 2f+1 replicas
  - Operations must intersect for at least one replica
  - Want availability for both reads and writes
  - Read and write quorums of f+1 nodes

55



56



57

Viewstamp Replication

- Primary-backup
- System moves through a sequence of views
  - Primary runs the protocol
  - Replicas do a view change if it fails

58

Replica state

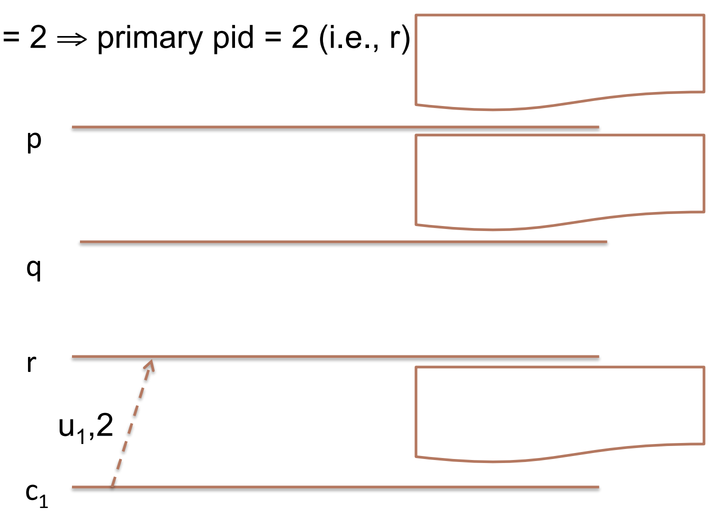
- A replica id i (between 0 and N-1)
  - Replica 0, replica 1, …
- A view number v#, initially 0
- Primary is the replica with id

  i = v# mod N
- A log of <op, op#, status> entries
  - Status = prepared or committed

59

Client knows current view #

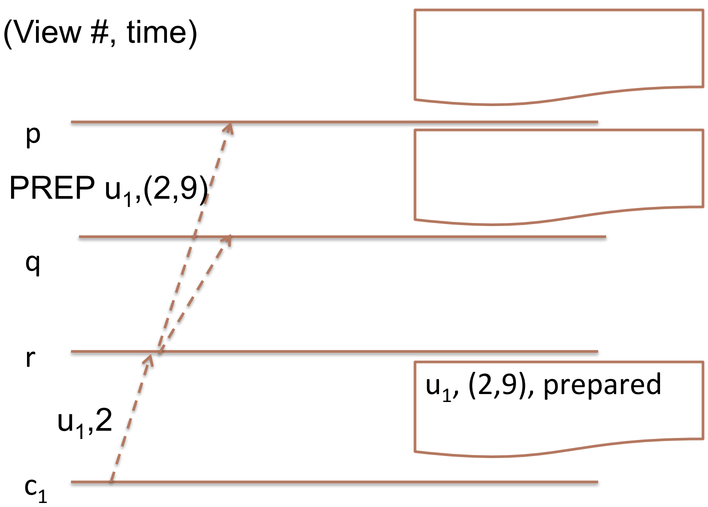
View # = 2 ⇒ primary pid = 2 (i.e., r)



60

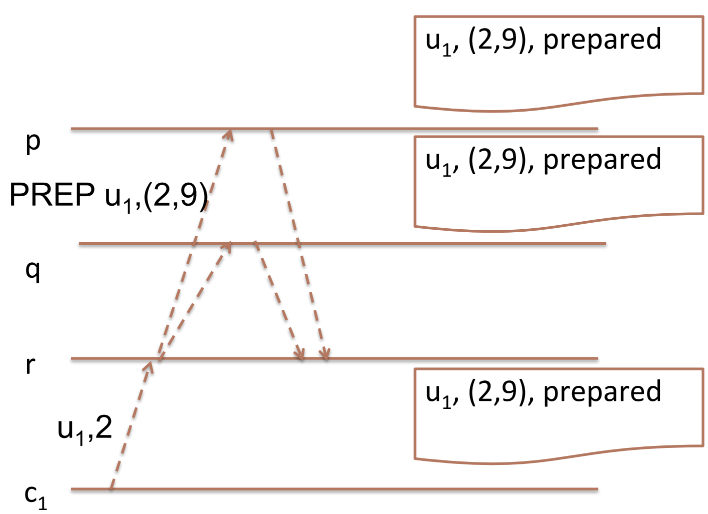
Primary chooses logical timestamp for update, e.g. LT(u_1)=9

Op # = (View #, time)



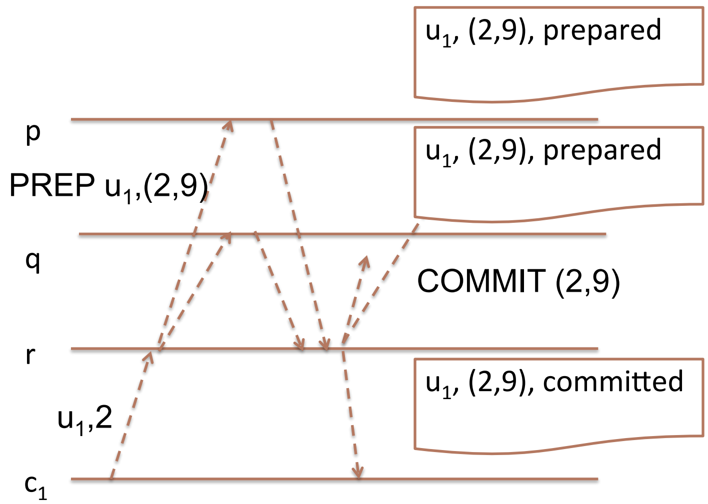
61

Primary waits for ≥ f replicas to respond (≥ f+1 total)



62

Client notified immediately after acks



63

PRIMARY-BACKUP: VIEW CHANGE

64

View Changes

- Used to mask primary failures
- Replicas monitor the primary
- Replica requests next primary to do a view change

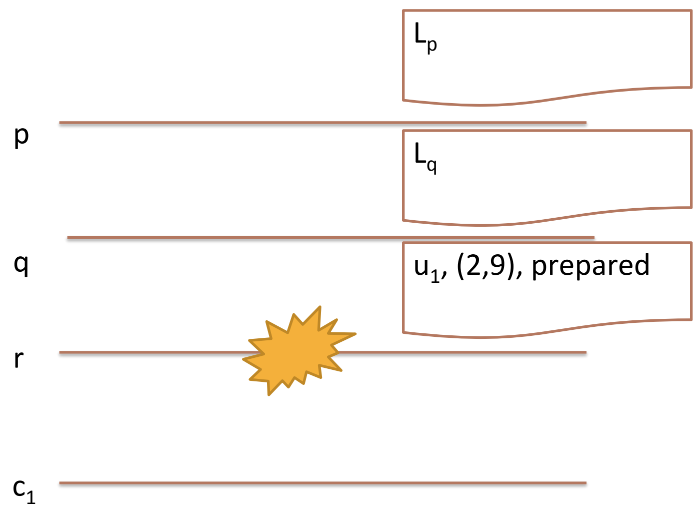
65

Correctness Requirement

- Operation order must be preserved by a view change
- For operations that are visible
  - executed by server
  - client received result
- An operation can be visible if it prepared at f+1 replicas
  - this is the commit point

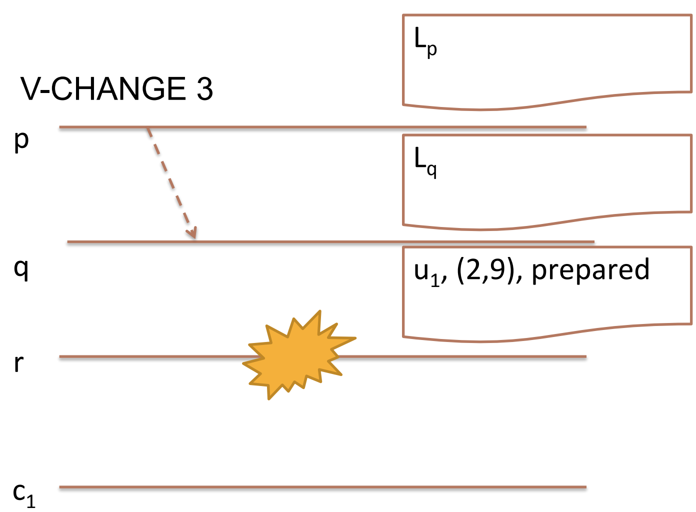
66

Primary r has crashed after assigning $LT(u_1)=9$



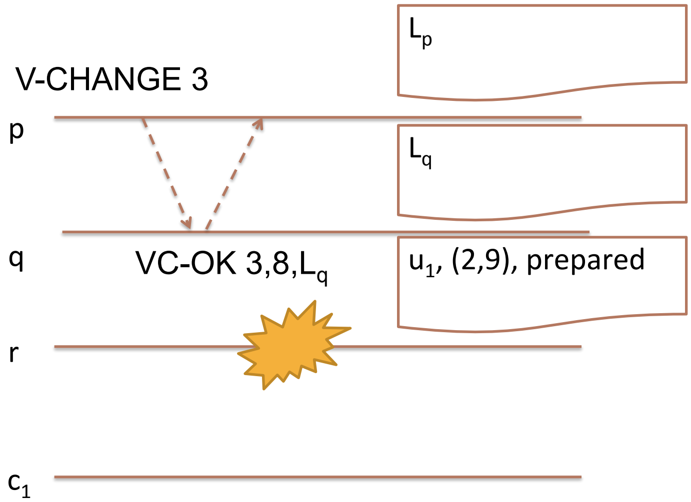
67

View # increments (to 3), new primary is p, start view change



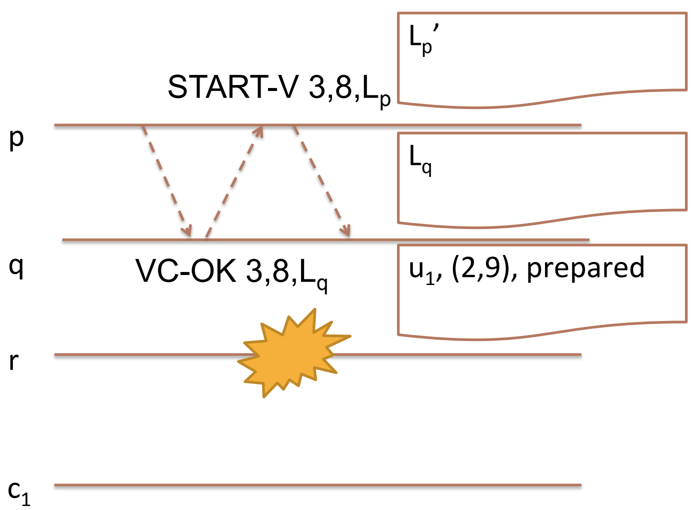
68

Replica q acks view change, returns last op# it knew of and its log (L_q contains any operations that p never heard of)



69

Primary p lets replicas know (via its log L_p') of ops that they were never informed of before old primary crashed



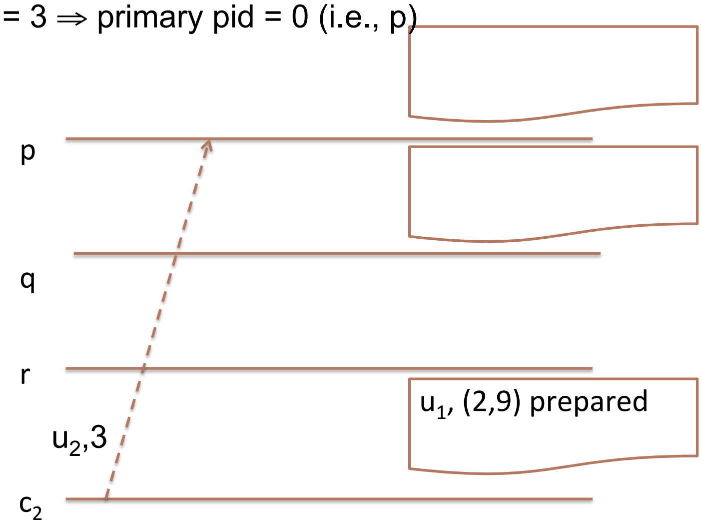
70

View Change

- New primary may not know of all updates from old primary
- New primary asks replicas for their logs
- Any committed operation was acked by a quorum, so must be in log of a surviving replica
  - Primary takes the max of the logs returned
  - That log has most recent updates

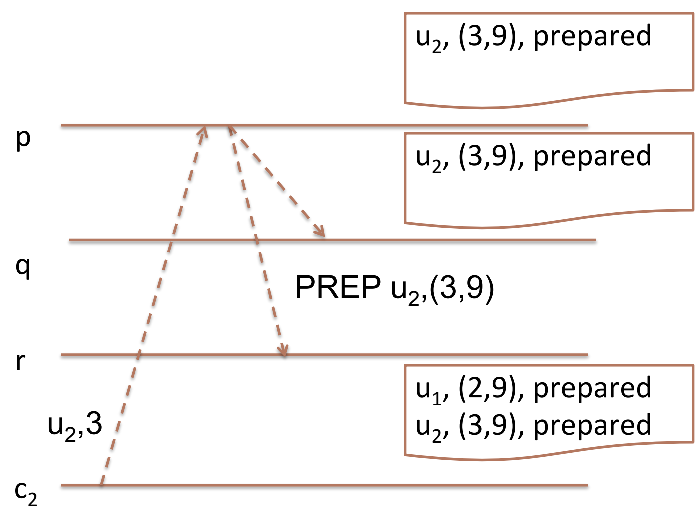
Second client c_2 knows current view #

View# = 3 ⇒ primary pid = 0 (i.e., p)



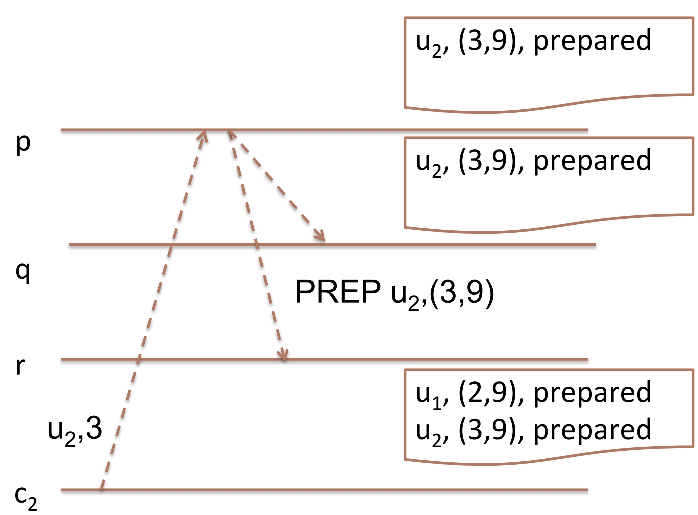
72

Primary p gets the other replicas to prepare to commit u_2



73

Viewstamp (3,9) avoids confusion between u_1 and u_2 at r
(Replica r will not think p is committing u_1)



74

Persistent State

- Voting protocol: votes must survive failure
  - Save queue of pending updates on disk
- Viewstamp: primary can respond to client without recording commit on disk
  - View change: recover commitment from logs of surviving replicas
- Only need to persist state after view change
  - So if we crash and recover, we know view# when we crashed
  - Even that unnecessary with more expensive recovery protocl