Robustness in the Salus scalable block store

Yang Wang, Manos Kapritsos, Zuocheng Ren, Prince Mahajan, Jeevitha Kirubanandam, Lorenzo Alvisi and Mike Dahlin

Summary created by Rafał Kołtun

Introduction

- Large, multi server datastore built from low-performance disks
- Similar to Amazon's Elastic Block Store
- User can request storage space from the service provider, mount it like a local disk, and run applications upon it
- Service provider replicates data for durability and availability

Introduction

- Append only datastore
- Only two available operations:
  - PUT – save object under given key
  - GET – read data associated with the key

Problem

- System may fail in many unpredictable ways:
  - Disk failures (3.45% of disks developed sector errors in 32 months)
  - CPU and memory errors
  - Power shortages
- Scalable distributed storage systems typically have single points of failure
- Other systems have provided end-to-end correctness guarantees (BFT), but they require each correct node to process at least a majority of updates (not scalable)

Salus

- (name of the Roman goddess of safety and welfare)
- Block store focusing on *scalability* and *robustness*
- That requires parallelism
- Extension of HBase

Background – HBase

- Distributed key-value store with PUT/GET interface
- Each table is split into multiple regions of non-overlapping key-ranges
- Uses HDFS as a storage layer to ensure that data is replicated persistently
- HBase uses a Master node to manage the assignment of key-ranges to regions
- Problem – computations are not replicated (memory error could corrupt data)

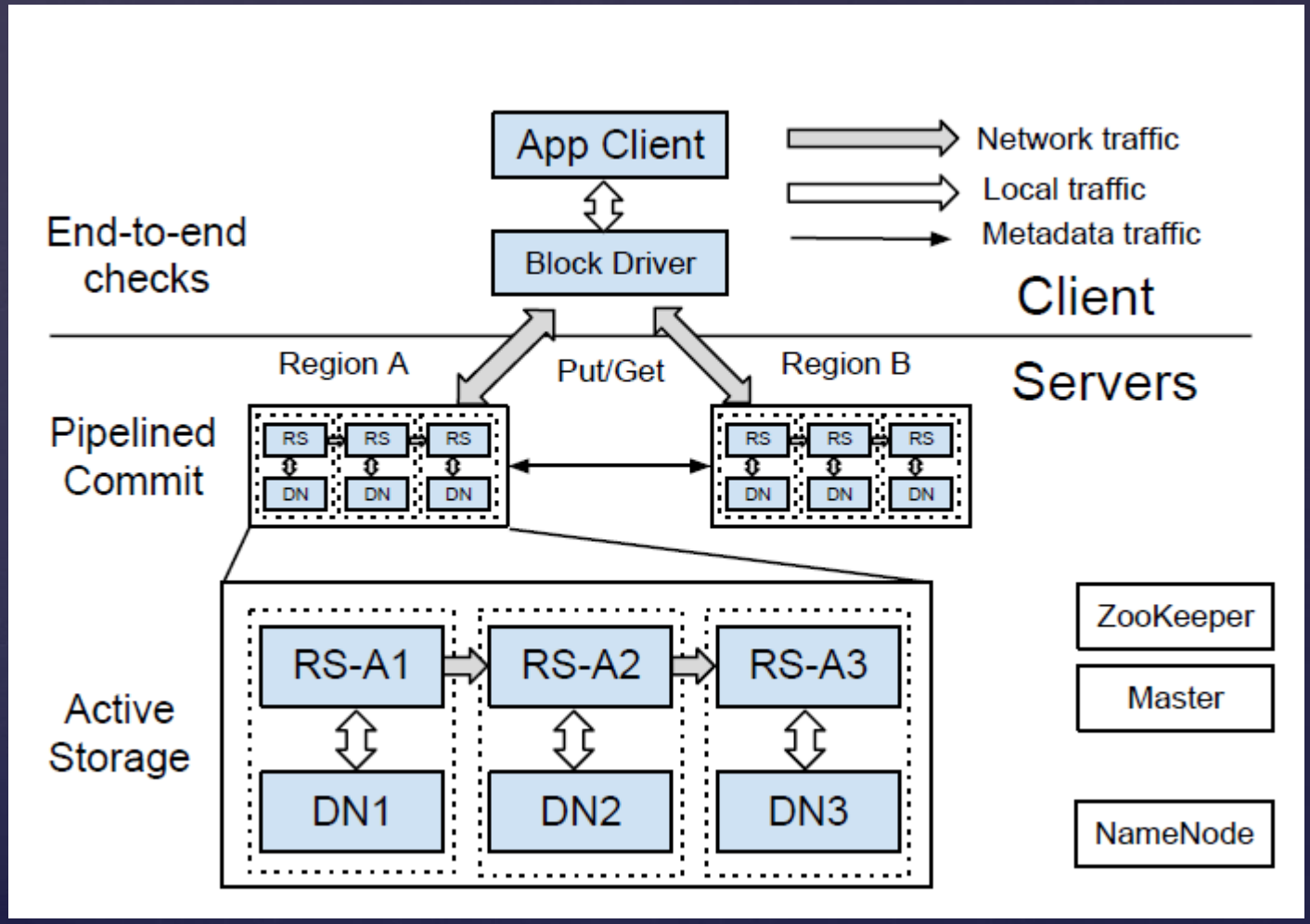
Background – HBase

- On receiving a PUT, a region server logs the request to a write-ahead-log stored on HDFS
- PUT updates in-memory map (called memstore)
- Memstore is flushed to checkpoint on HDFS
- Checkpoints are periodically merged (compaction)

Background – HDFS and ZooKeeper

- ZooKeeper is a replicated coordination service, used to ensure that each keyrange is assigned to at most one region server
- HDFS is an append-only distributed file system.
  - It stores the system metadata in a *NameNode*
  - Replicates the data over a set of *datanodes*
  - It ensures that each block of data is replicated on multiple datanodes (default is three)

Salus – structure



Salus – structure

- Core of Salus' active storage is a three-way replicated region server (RRS)
- Client requests are mediated by the block driver, which exports a virtual disk interface by converting the application's API calls into Salus GET and PUT requests.
- The block driver is also a component of end-to-end verification.
- To issue a request block driver contacts the Master, which identifies the RRS responsible for servicing the block that the client wants to access.

Salus – structure

- RRS ensures that the request commits in the order specified by the client
- If the request is a PUT, the RRS also needs to ensure that the data is made persistent, despite possible failures
  - the responsibility of processing PUTs is no longer assigned to a single region server, but it is based on unanimous consent of multiple servers
- GET requests can be safely carried out by a single region server (additional verification can be done by client)

Salus – improvements

- Pipelined commit
  - allows writes to proceed in parallel at multiple disks
  - guarantees that system will be left with in a state consistent with the ordering of writes specified by the client (even if failure occurs)
- Active storage
  - Salus replicates both the storage and the computation layer. It updates persistent state only if the update is agreed upon by all of the replicated computation nodes
  - It remains safe despite two commission failures with three-way replication
  - Faulty set of computation nodes can be replaced with a new set that can use the storage layer to recover the state required to resume processing requests

Salus – improvements

- Scalable end-to-end verification
  - Salus maintains a Merkle tree for each volume so that a client can validate that each GET request returns consistent and correct data
  - if not, the client can reissue the request to another replica
  - Reads can be safely responded by a single server
  - Salus' Merkle tree is scalable
    - each server only needs to keep the sub-tree corresponding to its own data
    - client can rebuild the whole tree even after failing and restarting from an empty state.

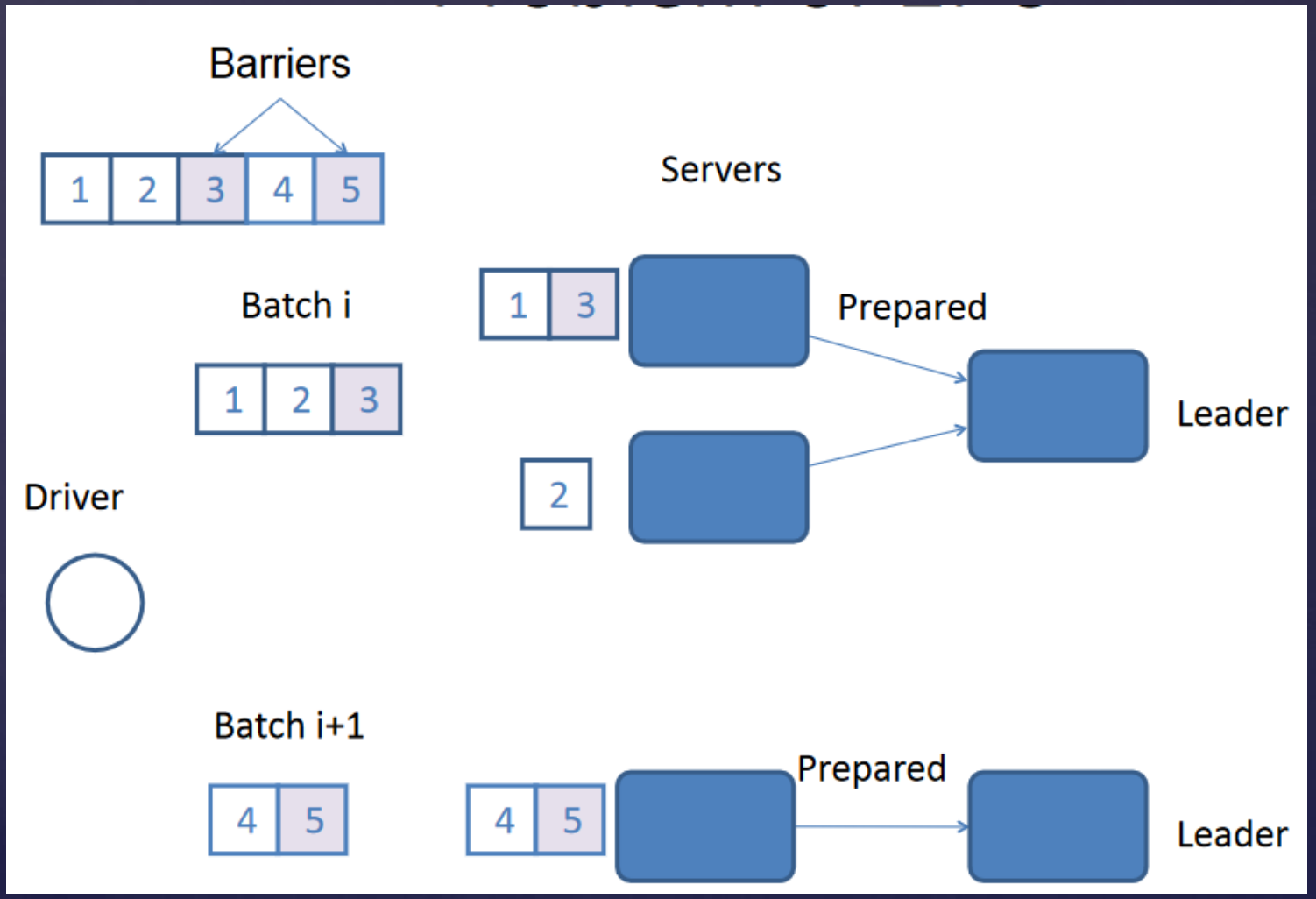
Goals and model

- Salus supports only single writer per volume
- Design seeks to minimize network bandwidth consumption
- Failures
  - Servers can crash and restart (indistinguishable from slow node)
  - Servers and disks can fail permanently losing data
  - Outer software can fail or corrupt data
- No attempts to stop malicious parties
  - Assumption: it is impossible to produce checksum for given node by other node.

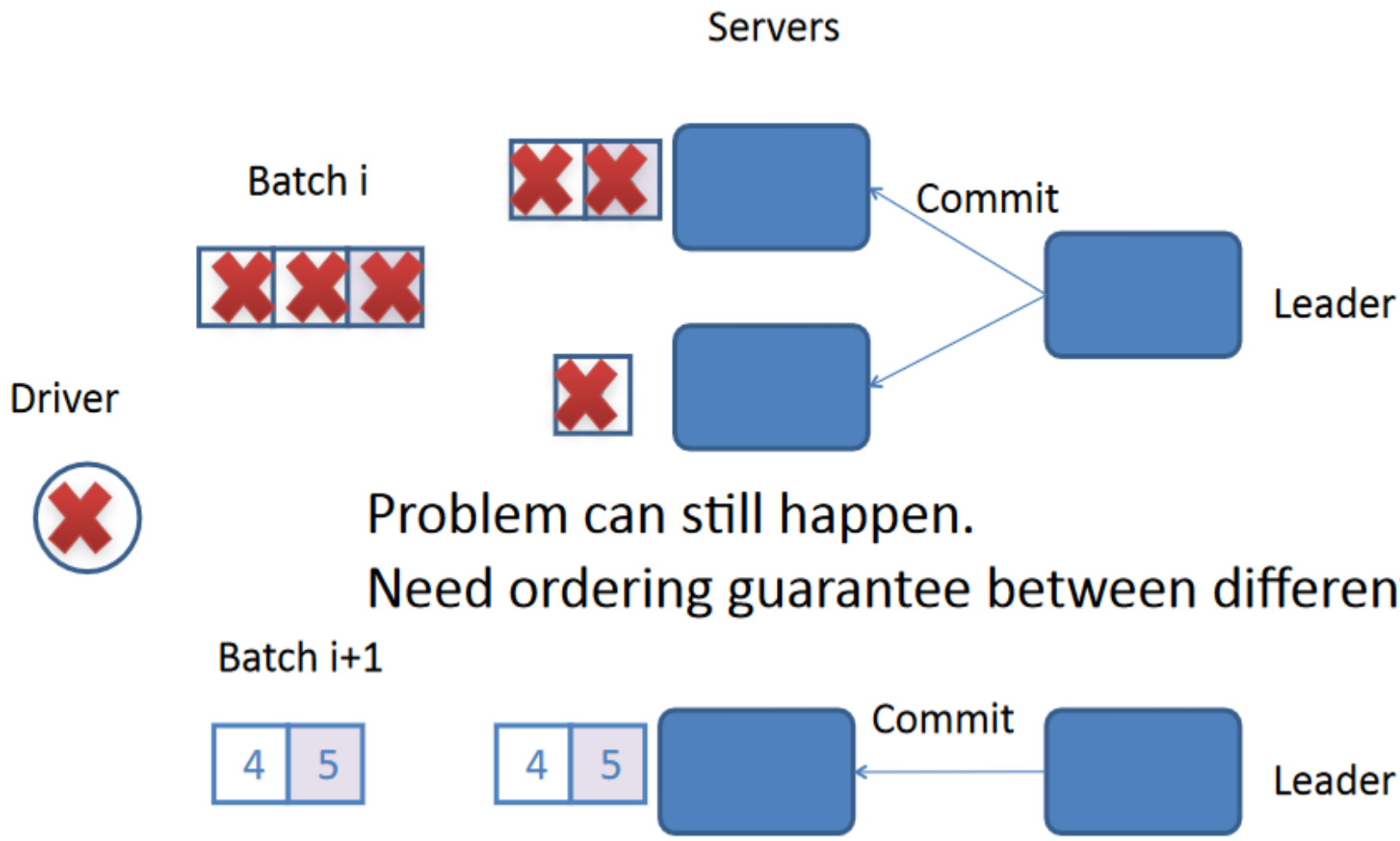
Goals and model

- Consistency
  - Standard disk semantics
    - All requests received before barrier are committed before barrier
    - Freshness – read to a block returns latest committed value
  - Ordered commit – before request R is committed, all previous requests are committed
  - Under severe failures
    - weaker prefix semantics - client that crashes and restarts may observe only a prefix of the committed writes

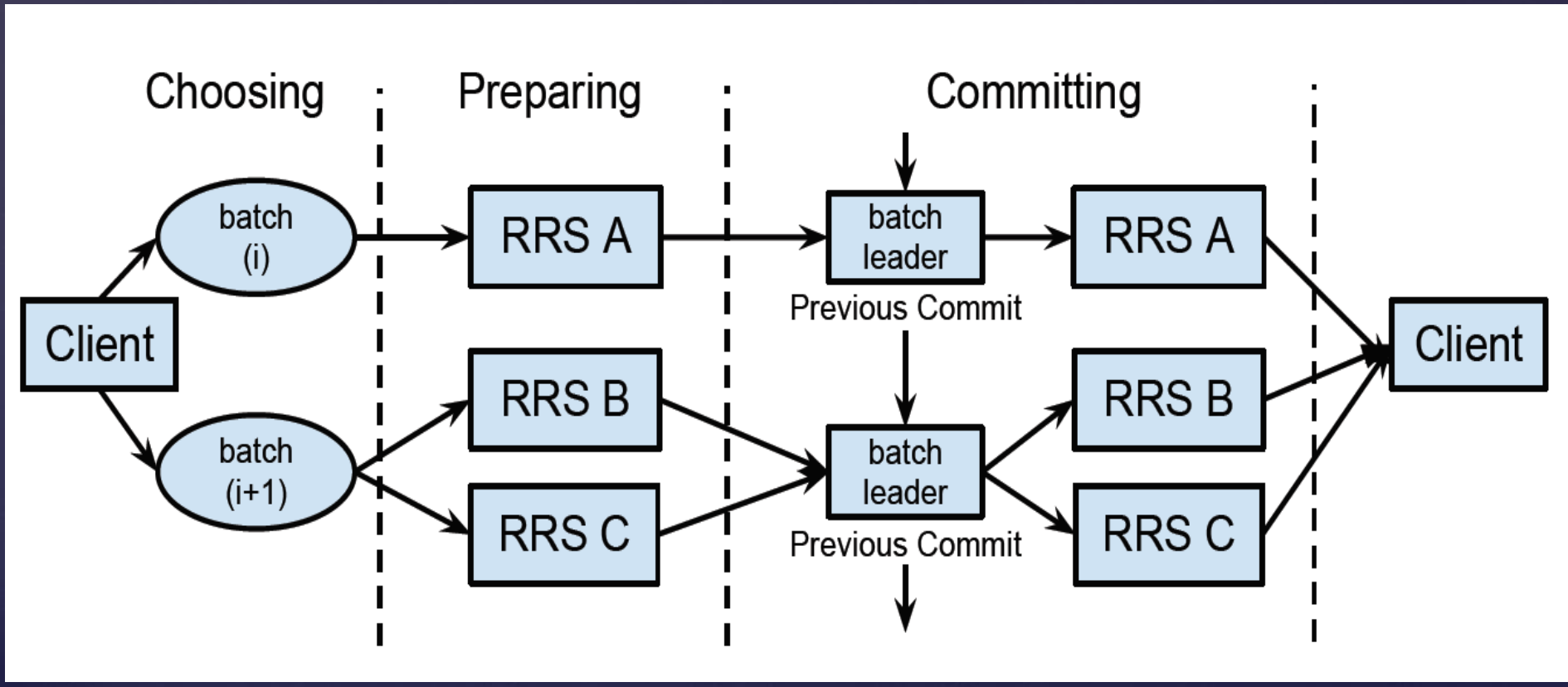
2 phase commit (2PC)



2 phase commit (2PC)



Pipelined commit



Pipelined commit

- Idea – clients assign sequence number to each requests
  - Region servers use it to guarantee that a request does not commit unless the previous request is guaranteed to eventually commit
  - During recovery, these numbers are used to ensure that a consistent prefix is recovered
- GET requests
  - Carry prevNumber of the sequence number of the last PUT executed on given region
  - region servers do not execute a GET until they have committed a PUT with the prevNum sequence number
  - Later PUTs may be blocked (by clients) before GETs evaluate

Pipelined commit

- PUT requests
  - PC1. Choosing the batch leader and participants
    - Client divides its PUTs into various subbatches (per region server)
    - prevNum field to identify the last PUT request issued to that region and batch leader identity
  - PC2. Preparing
    - A region server validates all requests (check prevNum)
    - If validation succeeds server logs the request (which is now prepared) and sends YES to the batch's leader; otherwise it votes NO.

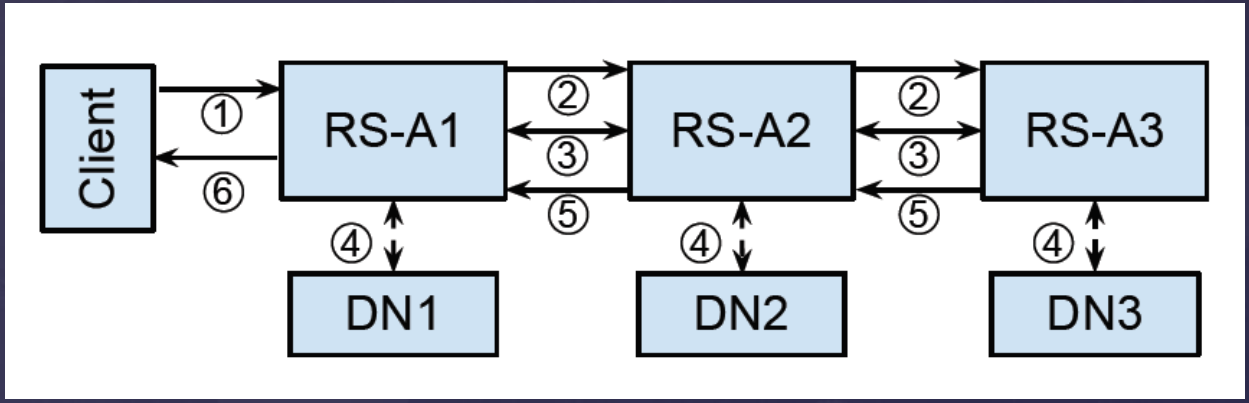
Pipelined commit

- PC3. Deciding.
  - COMMIT only if
    - all voted YES
    - COMMIT-CONFIRMATION from the leader of the previous batch
  - ABORT otherwise
  - Leader notifies the participants of its decision
    - COMMIT
      - each server update memstore
      - sends PUT_SUCCESS to the client
      - asynchronously marks as committed on persistent storage
    - ABORT
      - discards the state associated with that PUT
      - sends PUT_FAILURE

Active storage

- Replicating computation layer
- System works even though n – 1 servers failed
- Idea – use quorum of nodes in given region
- Implementation by changing the NameNode API for allocating blocks
  - To create a block a region server sends a request to the NameNode, with preferred datanodes
  - New location is on the same datanode that also hosts server that will access the data block
- Soft state – if region servers are not responding then Master may threw out all of them and replace with new ones

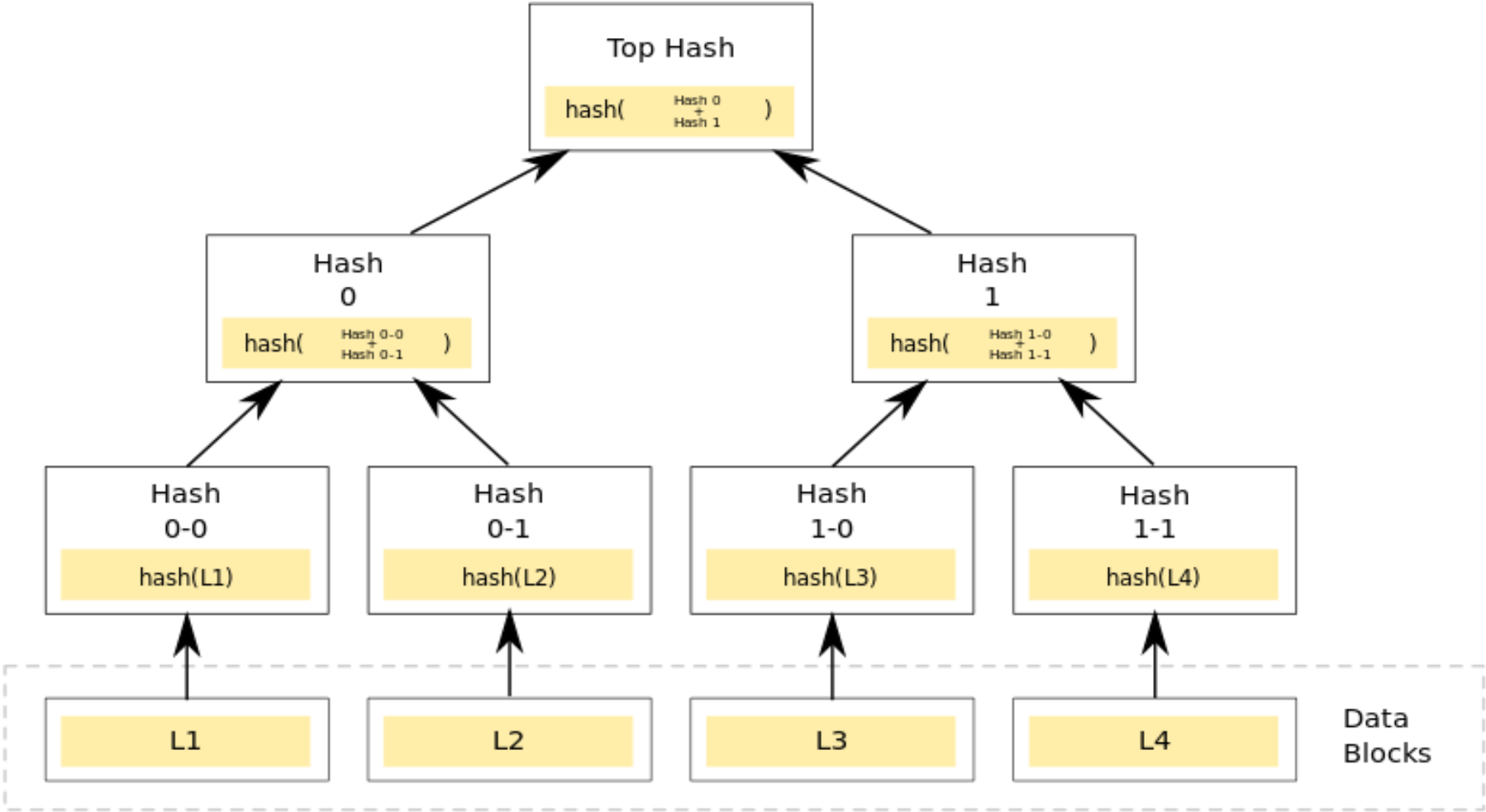
Active storage



PUT requests

1. Receive sub-batch
2. Forward to replicas
3. Servers agree on location and create PUT-log
4. Send PUT to datanode, wait for confirmation and contact batch-leader.
5. On COMMIT send PUT_SUCCESS certificate, on ABORT PUT_ABORT
6. Primary forwards certificate to client

Merkle tree



End-to-end verification

- Client maintains Merkle tree on the volume's block
  - Update on every PUT (client recalculate by himself)
  - Validate on every GET
- Copy of the region subtree on each region server (easy to rebuild)
- Each region server stores latest known hash for the root of the full volume tree and last PUT sequence number
- If client's copy of the tree is lost then
  - Client identifies most recent root hash
  - Servers commits all possible PUTs
  - Then client tries to compute the whole tree by using region trees
  - If the result does not match ask Master for replacing potentially broken servers

Recovery

- System wants to recover longest available prefix of PUTs
  - PUT logs may not be available (or some can be missing)
  - Prefix should be on whole volume, not only on single region

Recovery

- Recovery algorithm

  1. Remap – Master swaps all broken servers with new ones

  2. Recover region log – each server validates PUT-logs. Then all servers in RSS agree on the longest prefix.

  3. Find longest volume prefix – if client is avaiable then it chooses, otherwise ZooKeeper coordinates RSSs, and chooses longest prefix

  4. Rebuild volume – Master orders all servers to replay logs (ie. commit available PUTs) and rebuild region trees. Then checks root tree.

- If anything fails replay again (with shorter prefix). Recovery may be impossible if all servers failed. However, if there is at least one correct server it will eventually be found.

Evaluation - robustness

- Not full prototype – no BFT-replicated Master, NameNode, ZooKeeper
- Test by injecting failures into region servers
  - Crash
  - Corrupt data in memory
  - Write corrupted data to HDFS
  - Refuse to process requests
  - Delete files
- Comparison with HBase

Evaluation - robustness

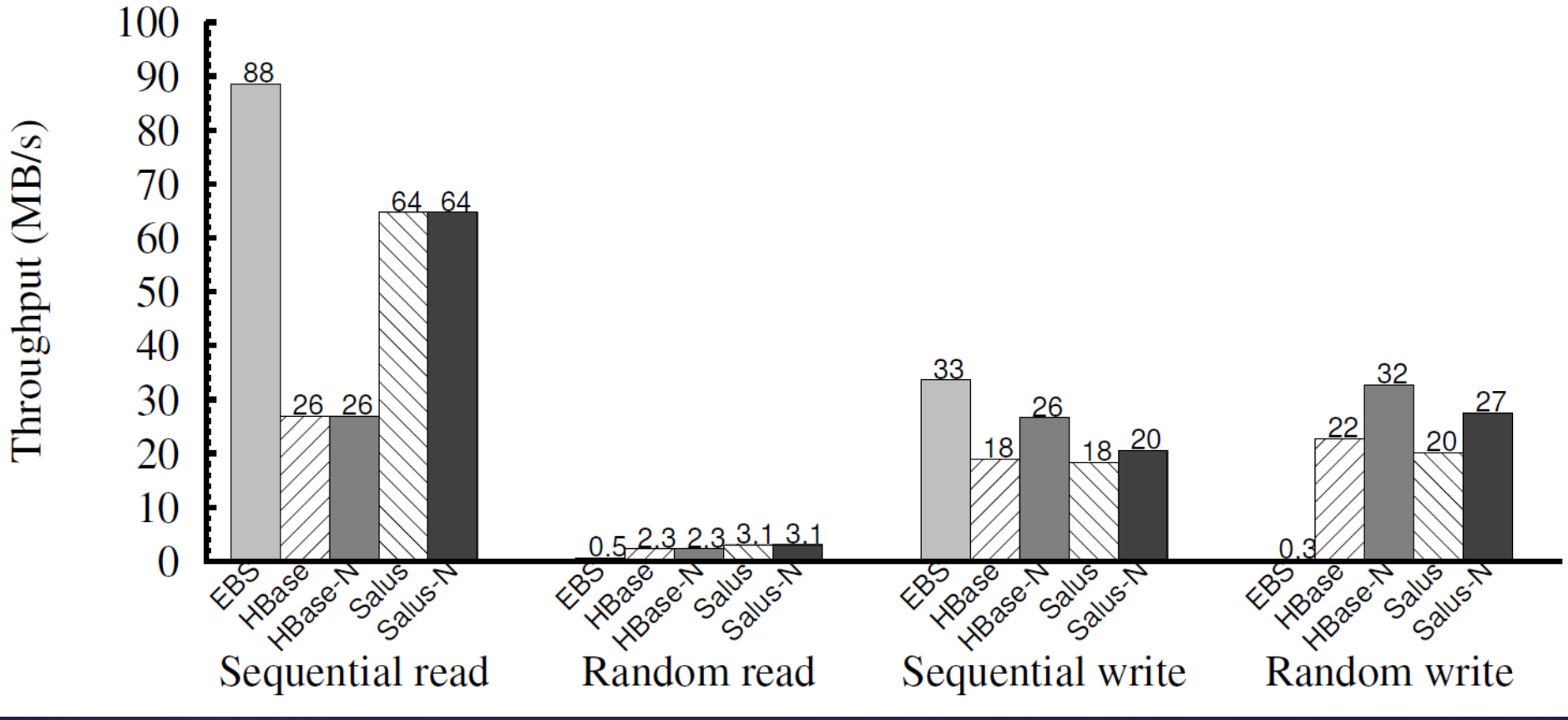
Affected nodes	Faults	HBase		Salus	
		GET	PUT	GET	PUT
Client	Crash and restart	Fresh	Not ordered	Fresh	Ordered
DataNode	1 or 2 permanent crashes	Fresh	Ordered	Fresh	Ordered
	Corruption of 1 or 2 replicas of log or checkpoint	Fresh	Ordered	Fresh	Ordered
	3 arbitrary failures	Fresh*	Lost	Fresh*	Lost
Region server+DataNode	1 (for HBase) or 3 (for Salus) region server permanent crashes	Fresh	Ordered	Fresh	Ordered
	1 (for HBase) or 2 (for Salus) region server arbitrary failures that potentially affect datanodes	Corrupted	Lost	Fresh	Ordered
	3 (for Salus) region server arbitrary failures that potentially affect datanodes	-	-	Fresh*	Lost
Client+Region server+DataNode	Client crashes and restarts, 1 (for HBase) or 2 (for Salus) region server arbitrary failures causing the corresponding datanodes to not receive a suffix of data	Corrupted	Lost	Fresh	Ordered

* - potentially not live

Evaluation - performance

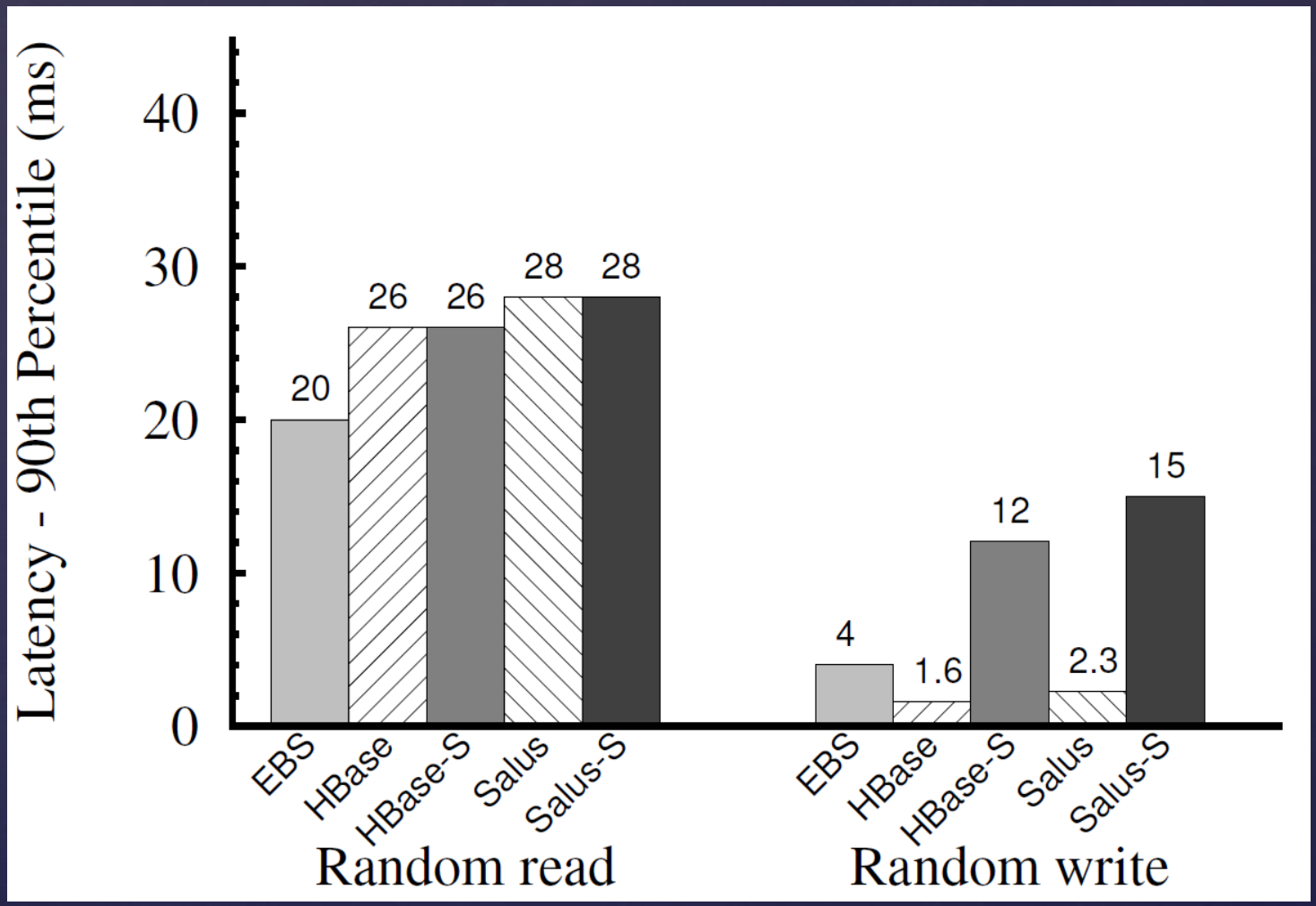
- 9 nodes as regions servers and database
- Up to 4 clients
- Each batch has 250 requests (~1MB)

Evaluation - performance



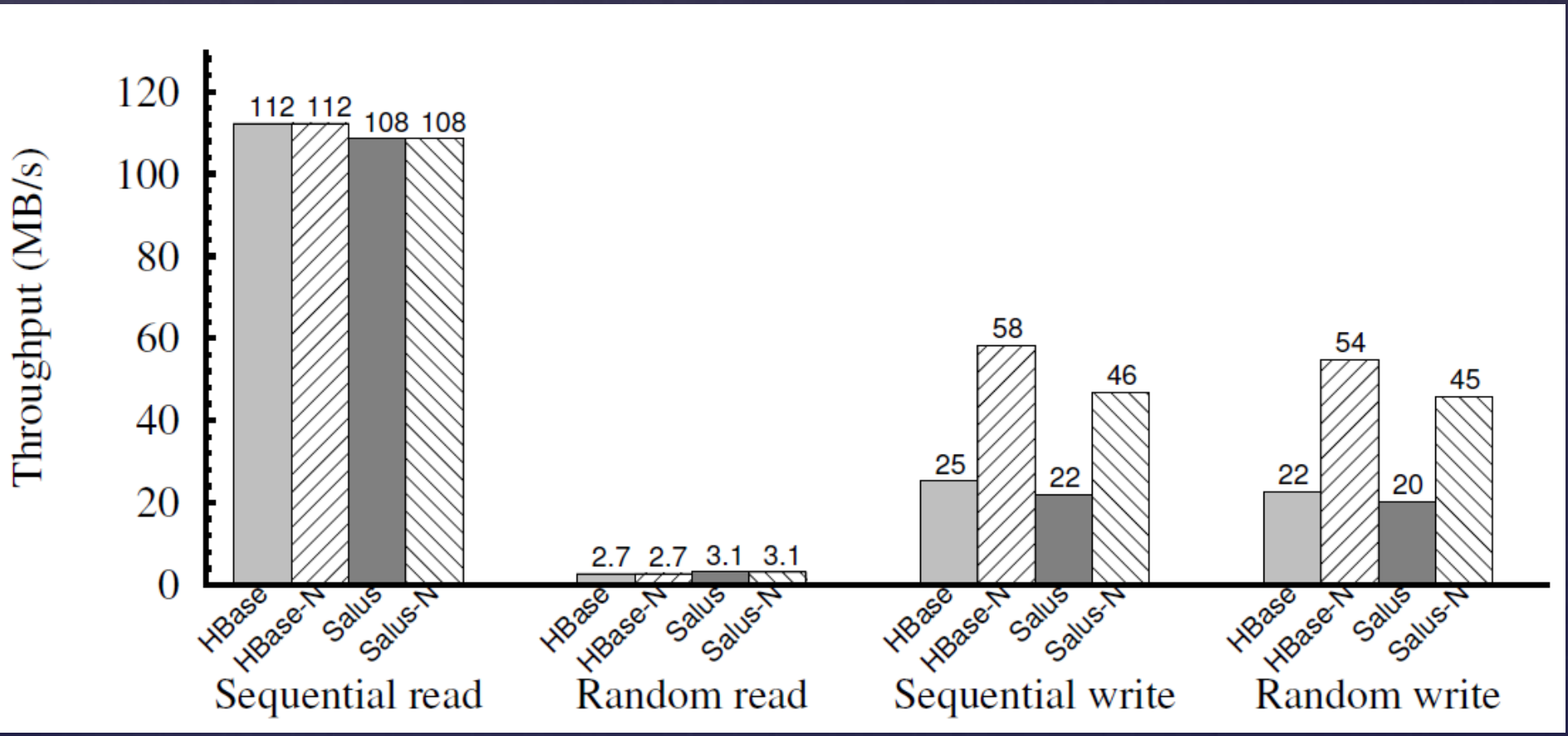
Single client only.
N – no compaction

Evaluation - performance



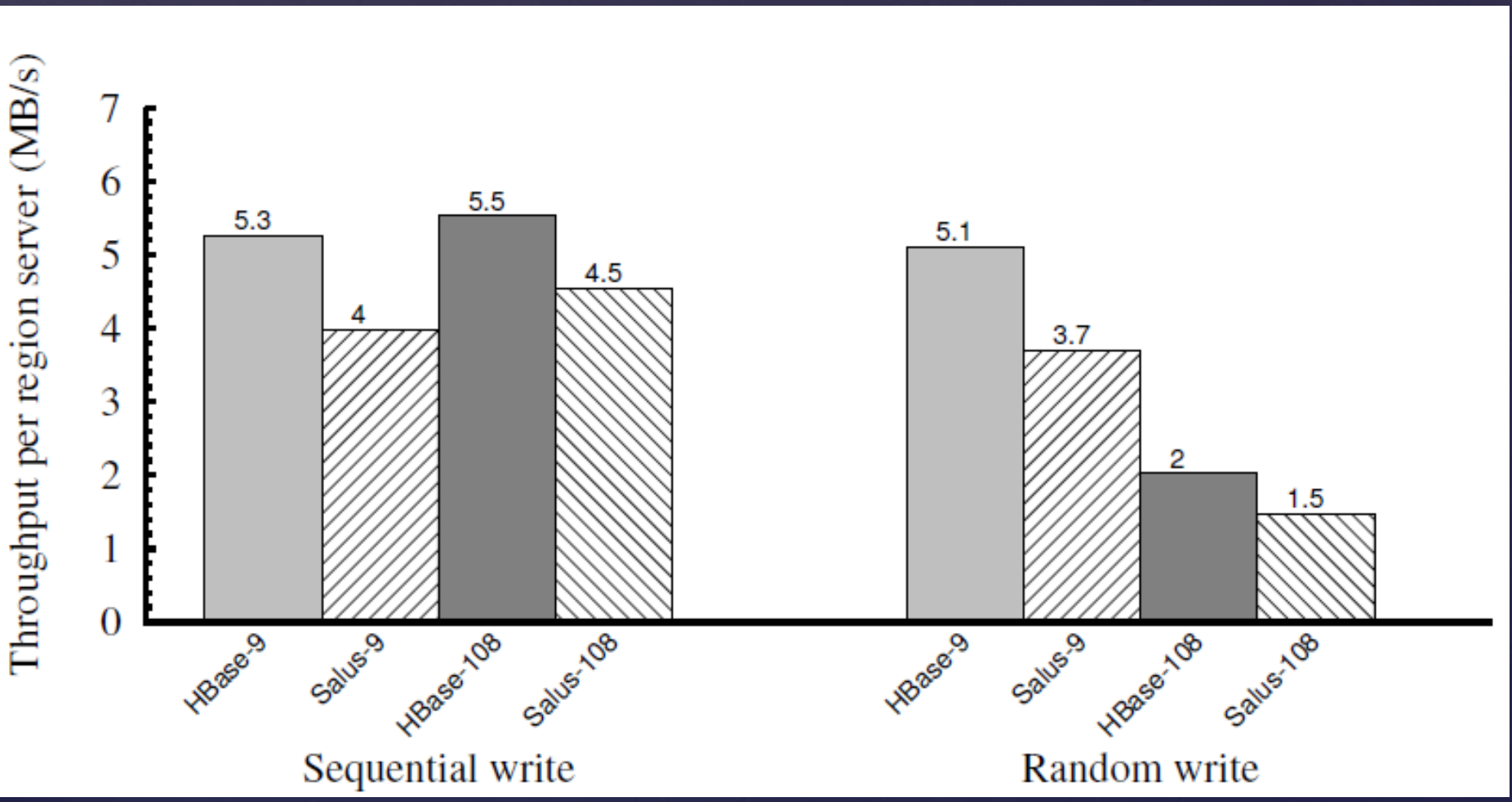
Single client only.
S – enabled sync when performing disk write

Evaluation - performance



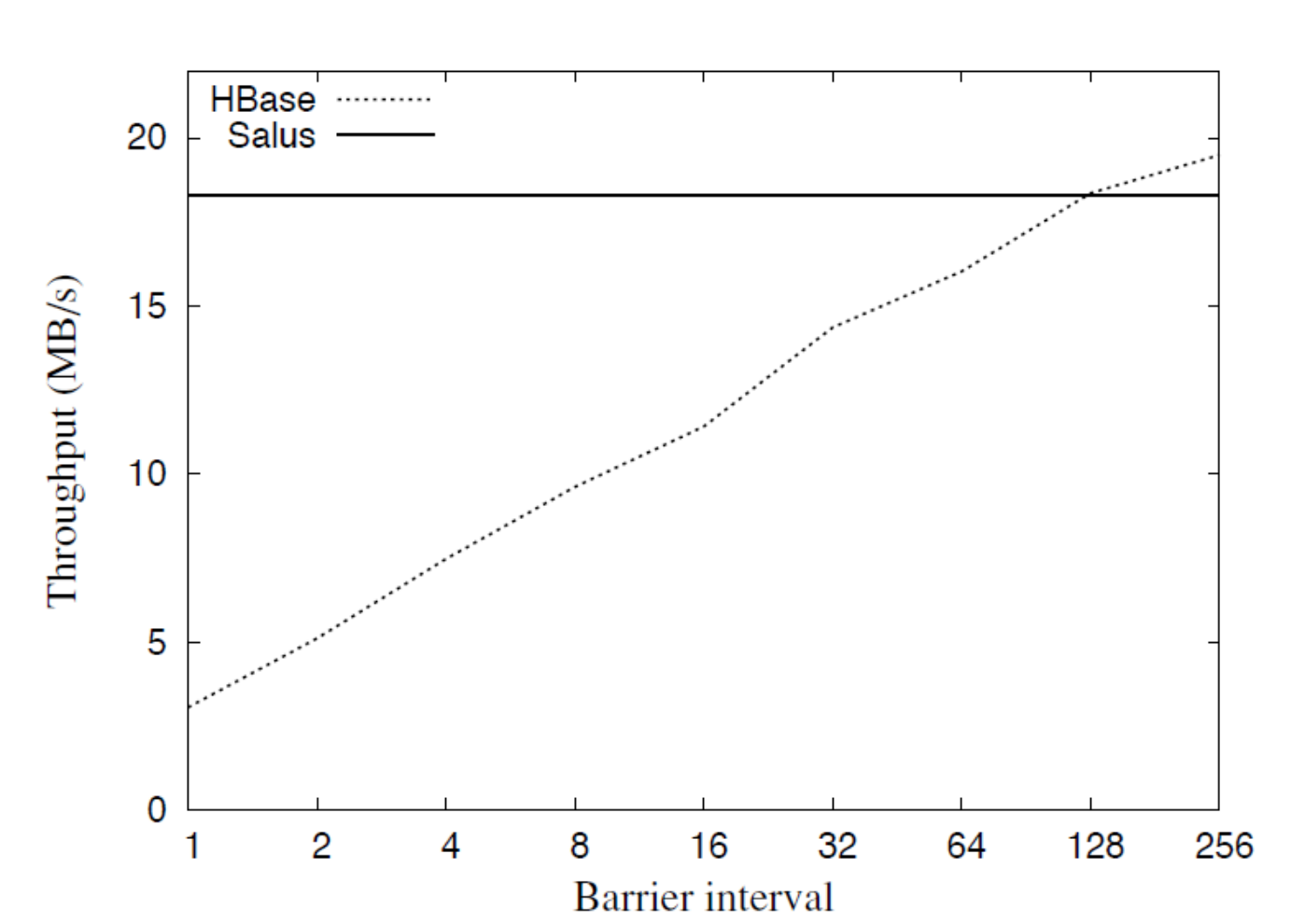
Increasing number of clients until throughput does not increase.
N – no compaction

Evaluation - scalability



It was run on Amazon EC2. Random writes scales worse because subbatch size decreases with number of the available servers.

Evaluation – pipelined commit



Salus is not affected by barriers (because it performs ordered-commit).

Further reading

- **The article and presentation** https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/wang_yang
- **Amazon's EBS** https://aws.amazon.com/ebs/details/
- **Merkle trees** https://en.wikipedia.org/wiki/Merkle_tree
- **About HBase** https://hortonworks.com/apache/hbase/#section_1
- HBase code https://github.com/apache/hbase

Q&A