# Feb 25
# Chain Replication and Remus

Vijay Chidambaram

# The problem

- We want a build a storage system over a group of servers
- Each server may fail at any time
- How do we ensure that the data is consistent when reading and writing to this distributed storage system?
- Pre-cursor to the Amazon S3 we have today

# Goals

- Availability:
  - Loss of one server should not render data unavailable
- Consistency:
  - Read of data item X should return the last written value V
  - Write and update operations are serialized in some order
- Performance:
  - High read and write throughput

**State is:**

$Hist_{objID}$ : **update request sequence**

$Pending_{objID}$ : **request set**

**Transitions are:**

T1: Client request $r$ arrives:

$$Pending_{objID} := Pending_{objID} \cup \{r\}$$

T2: Client request $r \in Pending_{objID}$ ignored:

$$Pending_{objID} := Pending_{objID} - \{r\}$$

T3: Client request $r \in Pending_{objID}$ processed:

$$Pending_{objID} := Pending_{objID} - \{r\}$$

**if** $r = \mathsf{query}(objId, opts)$ **then**

**reply** according options $opts$ based on $Hist_{objID}$

**else if** $r = \mathsf{update}(objId, newVal, opts)$ **then**

$Hist_{objID} := Hist_{objID} \cdot r$

**reply** according options $opts$ based on $Hist_{objID}$

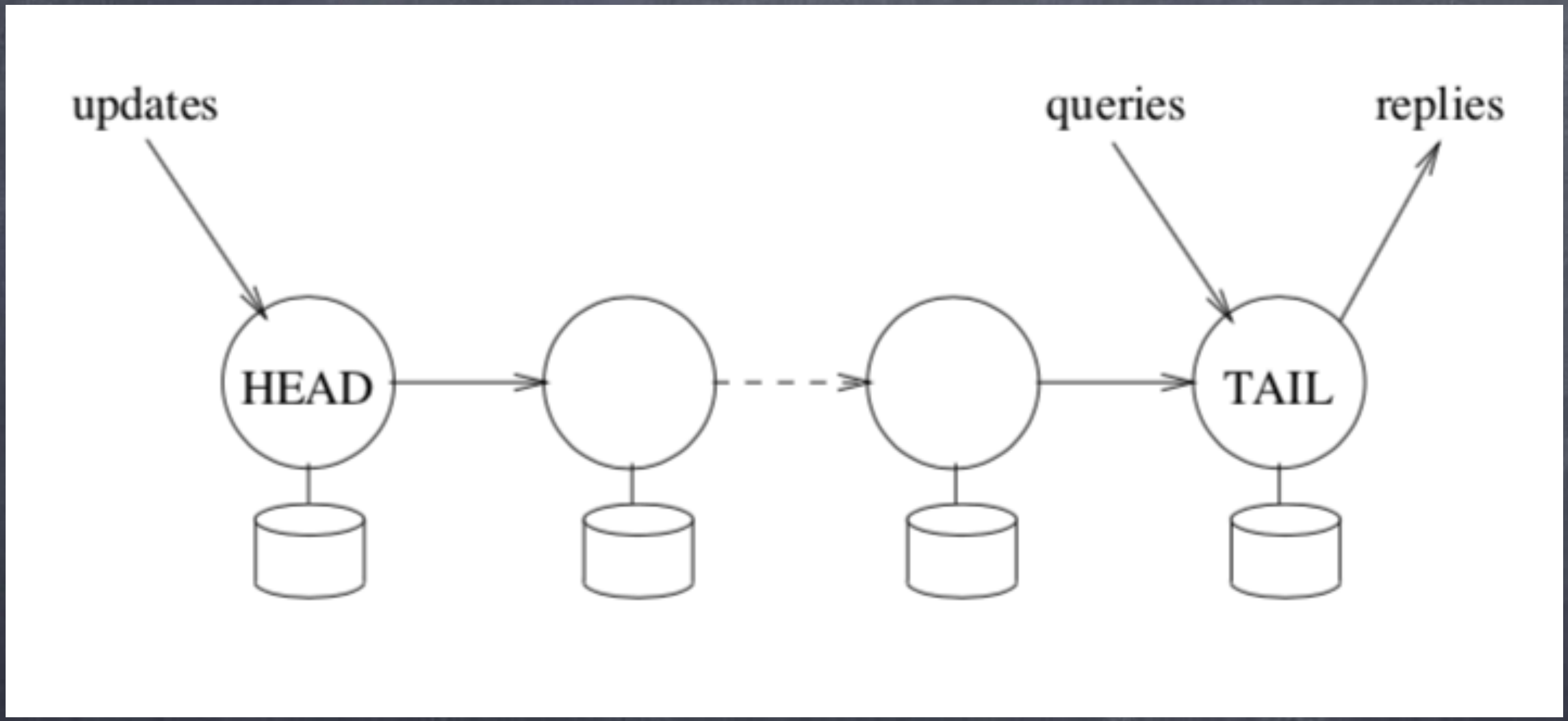
Figure 1: Client's View of an Object.

# Correctness

- Correctness depends on maintaining Pending and History
- Pending = set of requests received by any server in the chain and not yet processed by the tail
- History = set of updates done on an object by the chain (by the tail)

# Assumptions

- Servers fail in a fail-stop manner
- Server failures can be detected
- Servers communicate via reliable, FIFO channels
- At most n-1 servers (out of n servers) fail concurrently at any time


updates
queries
replies
HEAD
TAIL

# Chain Replication

- Properties:
  - Availability: data is available as long as at least one server is up
  - Strong consistency: all read and write requests handled serially by the tail server

# Chain Replication

- Servers are arranged in a chain
- Reply for every request is generated by the tail
- Read requests redirected directly to the tail
- Write/update requests arrive at the head, and are atomically processed by each node in the chain

# State Forwarding

- The write/update operation is performed first at the head
- The head server performs the operation and produces the resulting low-level update to data
- Every other server in the chain only applies this low-level update; it does not perform the write/update operation
- Thus, if the update operation is non-deterministic, the head operation performs it and changes it into a deterministic low-level update

# Dealing with server failures

- A master server:
  - detects failures
  - informs nodes in chain about failed node
  - informs clients of new head or tail if required
- Master implemented using Paxos on a group of nodes

# Correctness

**Update Propagation Invariant.** For servers labeled $i$ and $j$ such that $i \leq j$ holds (i.e., $i$ is a predecessor of $j$ in the chain) then:

$$Hist^{j}_{objID} \preceq Hist^{i}_{objID}.$$

- This invariant is maintained because i is updated before j in the chain. Thus, the history of i = history of j + some updates

# Failure of the head

- Clients need to be told of new head
- Pending needs to be updated, whatever was pending at previous head is lost
- Client sees this as a message lost/ignored by the chain

# Failure of the tail

- Clients need to be told of new tail
- History needs to be updated, any updates performed by the new tail are considered done by the chain
- Pending needs to be updated, to remove items that have been processed by the new tail
- This works because of the Update Propagation Invariant: new tail is guaranteed to have processed anything processed by old tail

# Failure of other servers

- If Server F fails:
  - Master informs F's successor S+
  - Masters informs F's predecessor S-
  - S- forwards to S+ requests that were lost when F failed
  - Each server maintains Sent variable
    - Sent = Sent U r when r is forwarded
    - Sent = Sent - r when r's ack is received
    - Requests are forwarded head to tail
    - Acks are forwarded tail to head

# Adding new servers

- New servers are added to the tail
- Old tail forwards History to new tail
  - This can be done incrementally since we only need to ensure new tail history is a prefix of old tail history
- Old tail is informed its not the tail
- Old tail forwards requests to new tail
- Master is notified that new tail is functioning
- Clients are notified of new tail

# Primary/backup

- Chain replication is an instance of the primary/backup approach
- In primary/backup approach:
  - the primary sequences all requests
  - primary has to wait for acks from all other nodes before responding to client
- In chain replication, sequencing is split between head and tail
  - Head sequences updates
  - Tail sequences queries
- In chain replication, queries are never blocked by updates
  - Can always happen at the tail
- Chain replication has higher latency for writes (must wait until chain processes write in a serial fashion)
  - Primary/backup approach can write in parallel to replicas

# Unavailability during failure

- Failure of head/tail:
  - Processing queries blocked for 2 messages while new head is being setup
  - Message 1: master broadcasts new head info to everyone
  - Message 2: master tells all clients about new head
- Middle server failure:
  - Query processing not interrupted
  - Update processing delayed while chain is reconfigured
  - Updates not lost as at least head has update
- Primary/backup failure:
  - Primary failure: 5 message delay
  - Backup failure: 1 message delay

# Remus: the problem

- Bring high availability to the masses!
- Goals:
  - Generality and transparency (should work with unmodified apps and unmodified operating systems)
  - Shouldn't require custom hardware
  - No externally visible state should ever be lost
  - Quick failure recovery: should seem like packet loss
  - Crash should leave data in a consistent state

# Approach

- VM-Based Whole-system replication
- Speculative Execution
- Async replication

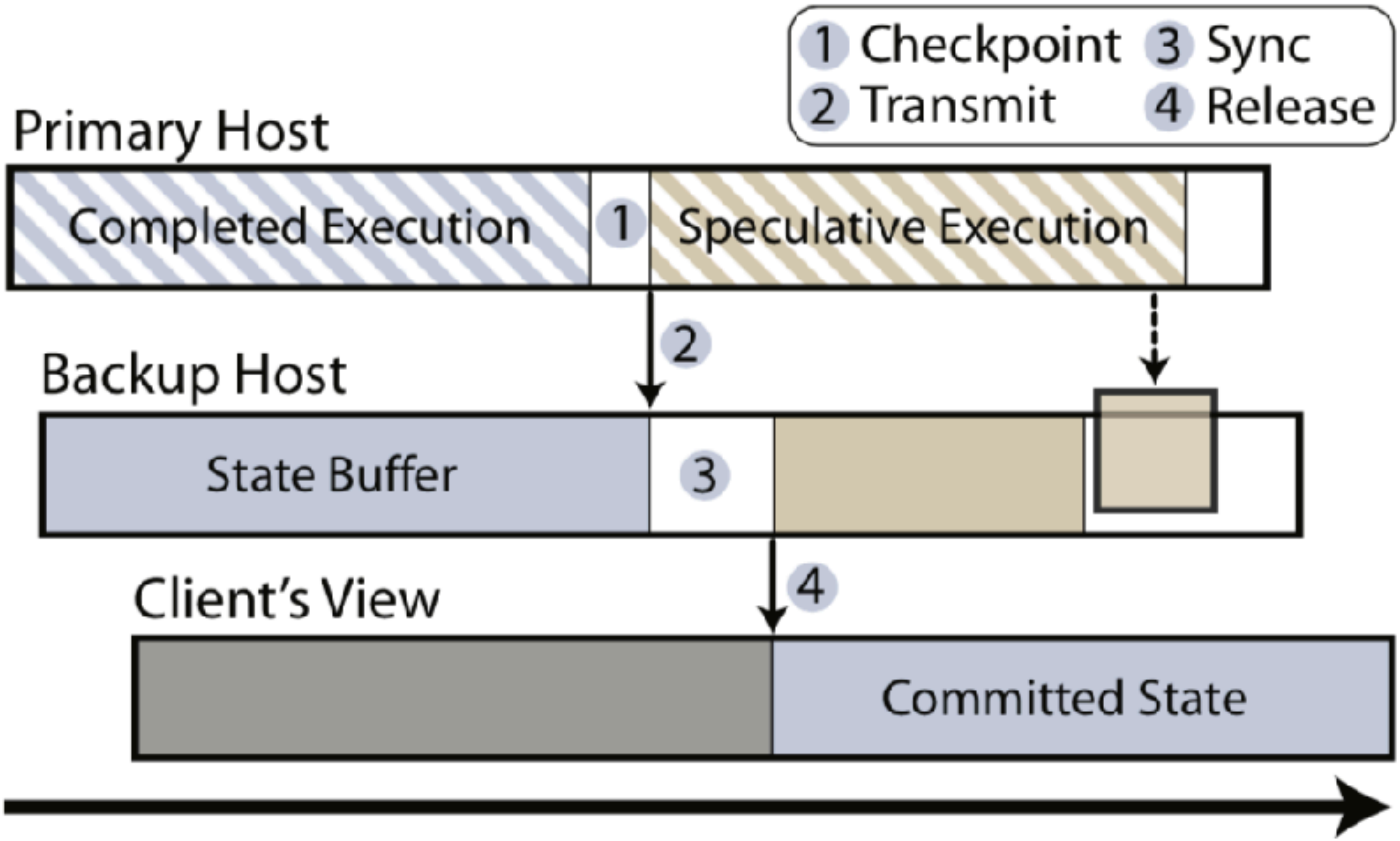



Figure 1: Speculative execution and asynchronous replication in Remus.

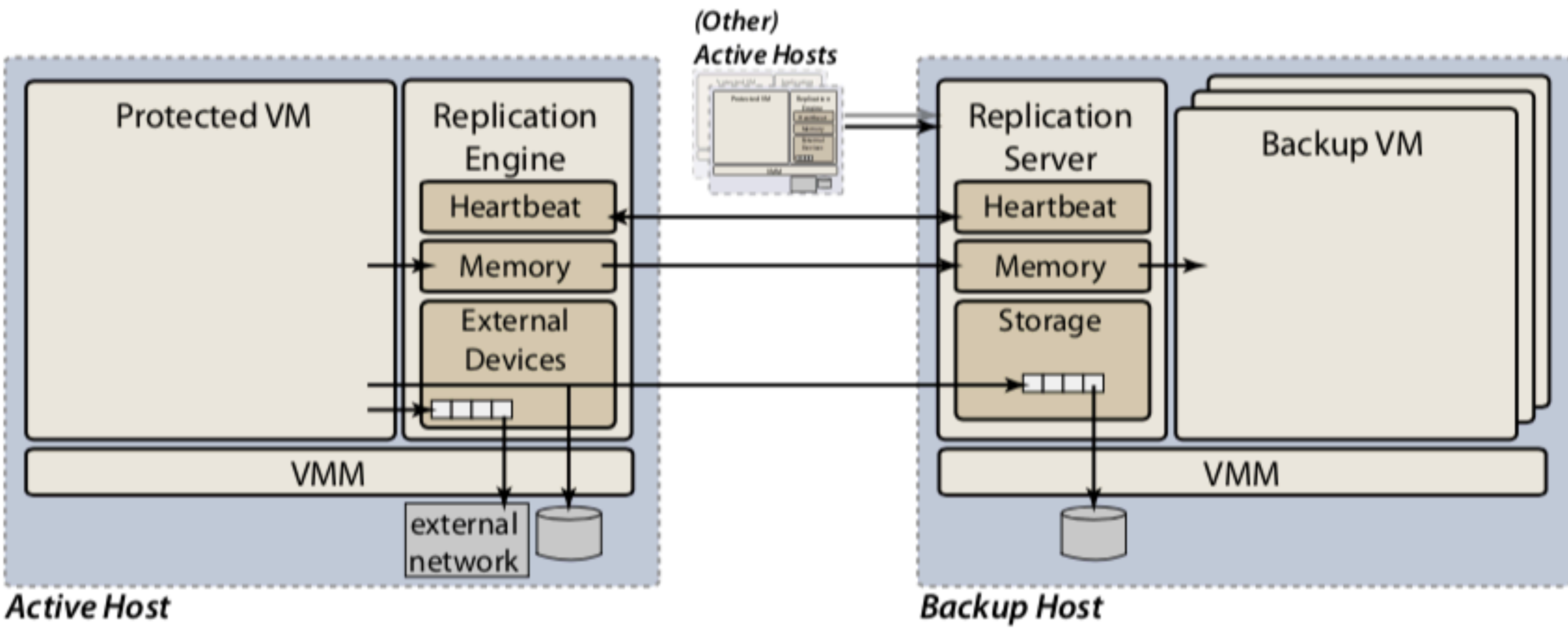



Figure 2: Remus: High-Level Architecture

# Questions to think about

- How is snapshotting done efficiently?
  - Changes to xenstore
  - Making the copy fast
  - Reducing inter-process communication
- How is network traffic handled?
  - Inbound traffic delivered directly
  - Outbound traffic buffered until checkpoints
  - Use an intermediate queueing device

# Handling disk writes

- Writes are buffered on backup node
- Writes are applied to backup storage at the end of the checkpoint
- This is done to ensure that the backup storage is consistent if there is a crash
- Only one of the two disk mirrors managed by Remus is valid at any given time; this is identified using an activation record