All about Eve: Execute-Verify Replication for Multi-Core Servers

Authors: Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, Mike Dahlin

Presenter: Xuerui Li

Table of Contents

Comparison with Zyzzyva

- In Zyzzyva the agreement on inputs guarantees agreement on outputs: hence, a replica need only send (a hash of) the sequence of requests it has executed to convey its state to a client.
- In contrast, in Eve there is no guarantee that correct replicas will be in the same state, as the mixer may have incorrectly placed conflicting requests in the same parallelBatch.
- Verification stage is moved to the clients to reduce cost
- Shortcomings: It may introduce corner cases

Execution Stage

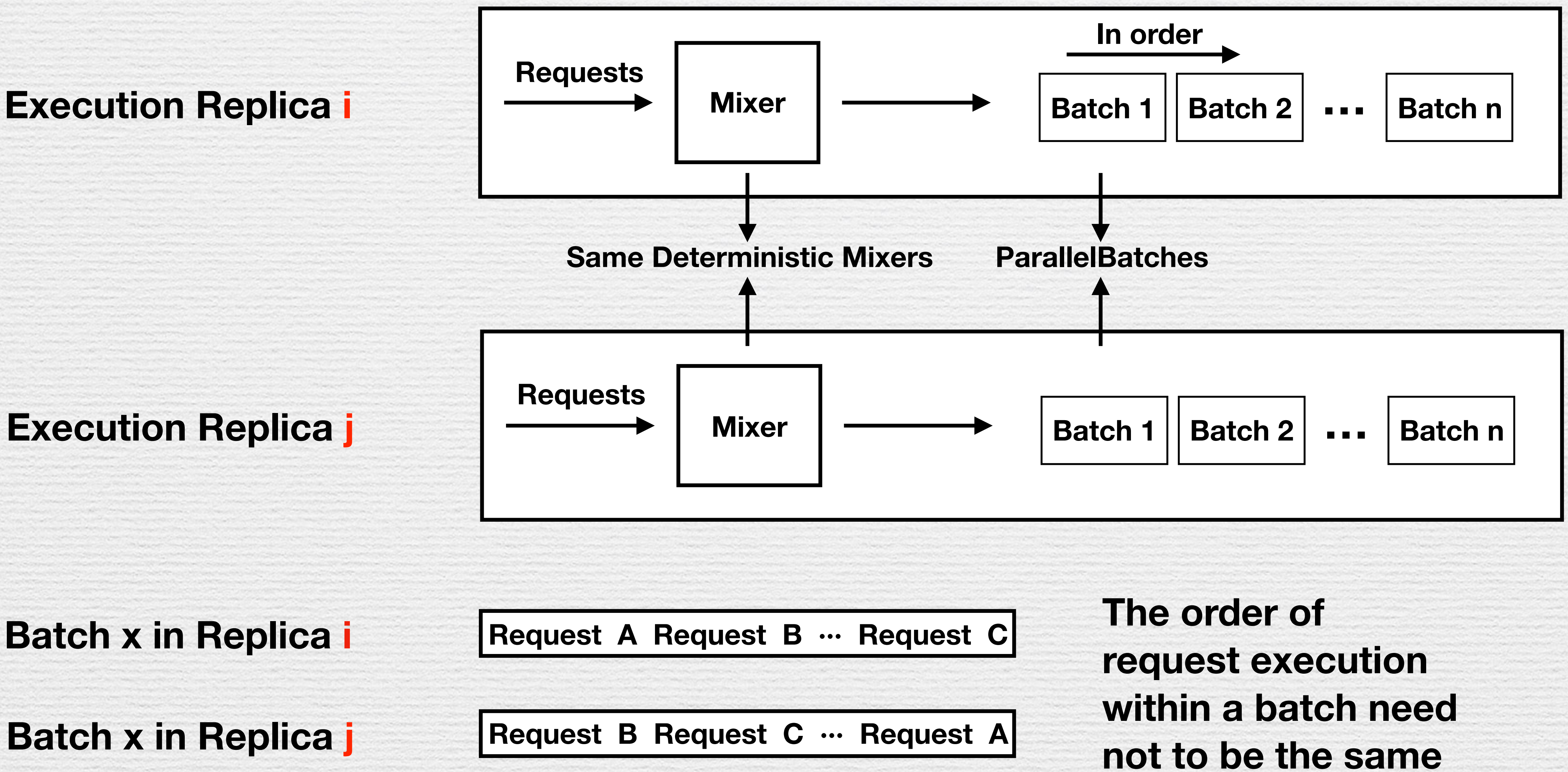


Fig. 2 ParallelBatches

Execution Stage

- Divergence: When one object is written by a request, at the exactly same time it is also used by another request.

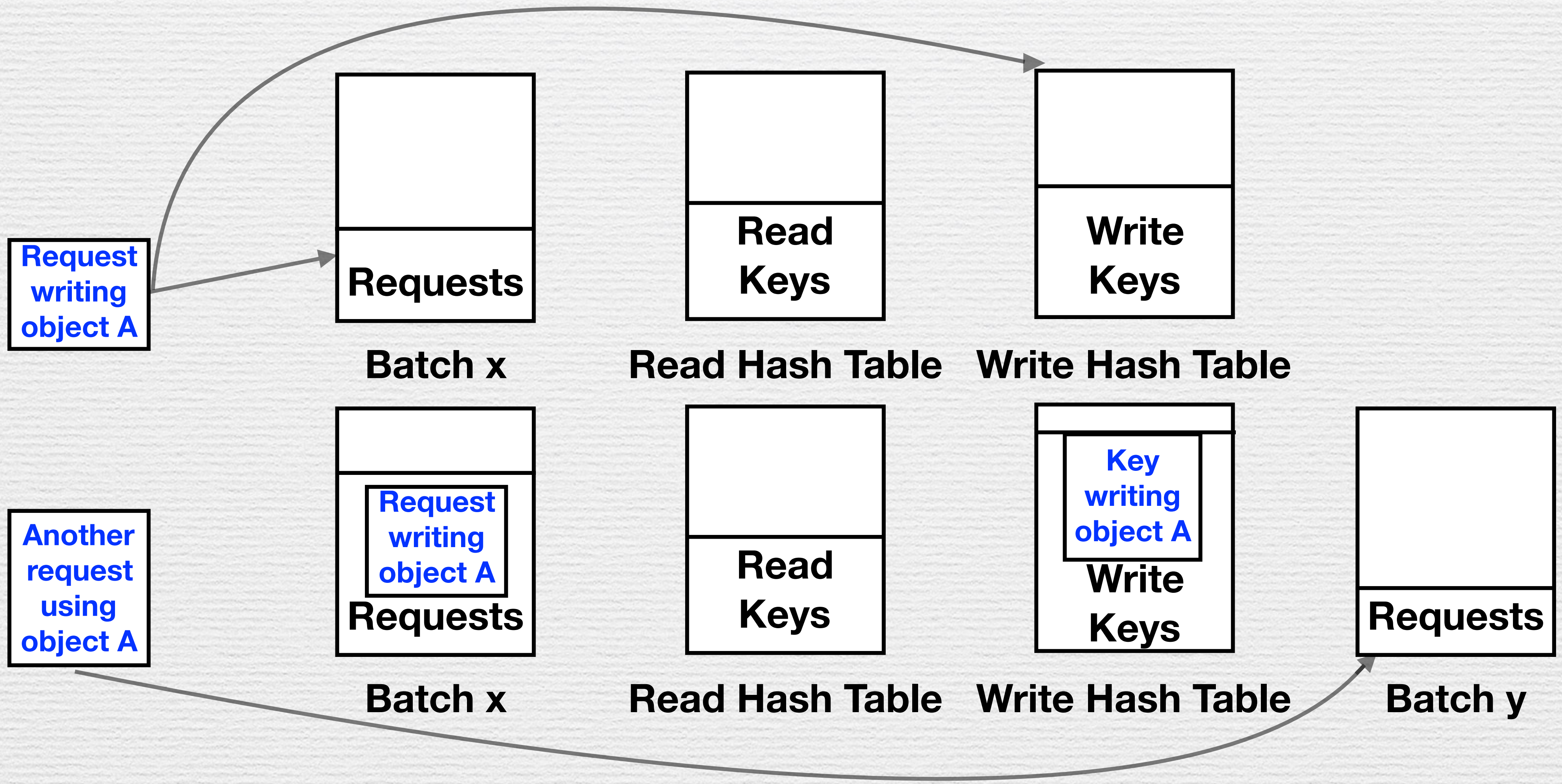


Fig. 3 Mixer

Execution Stage

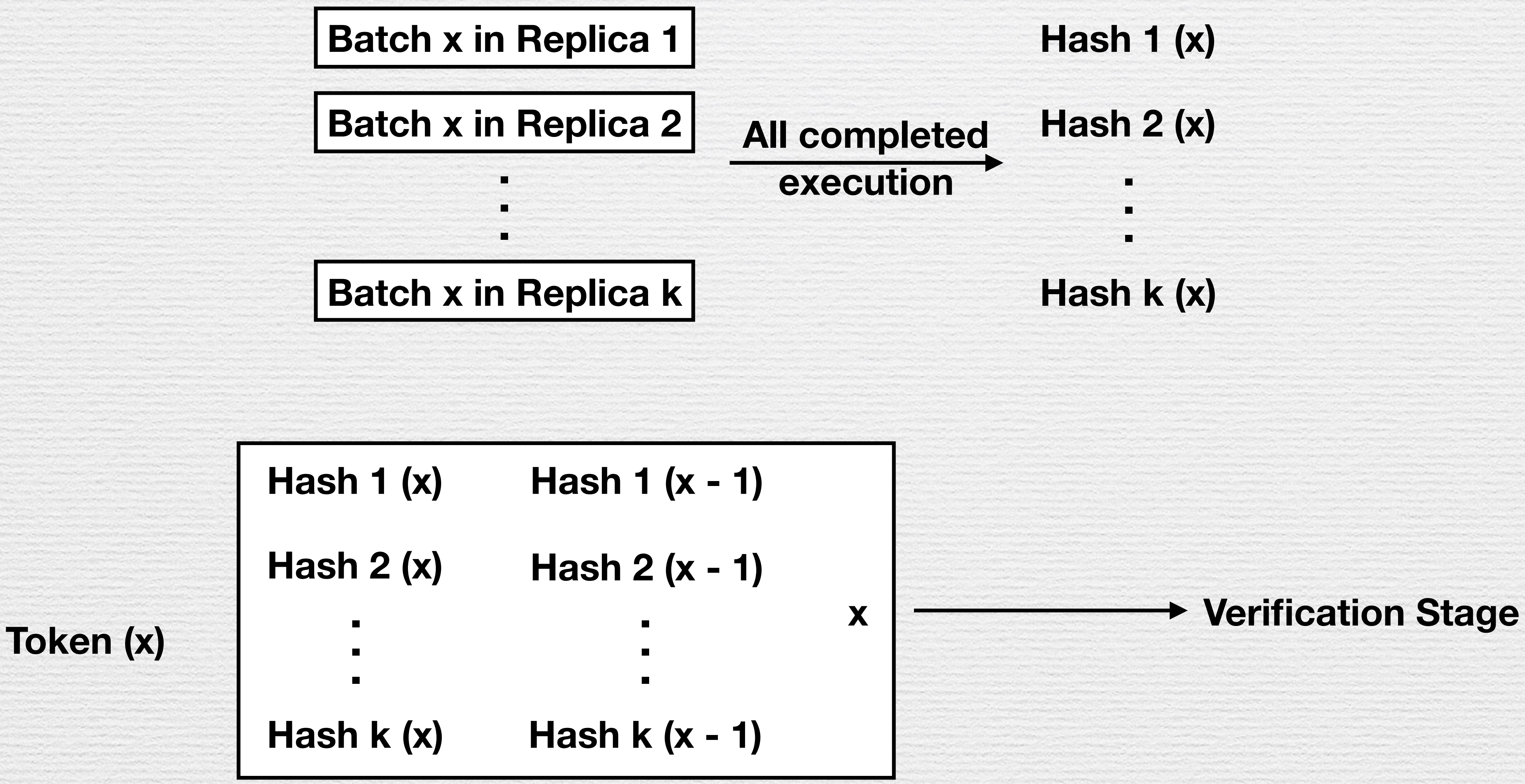


Fig. 4 Token for Verification

Verification Stage

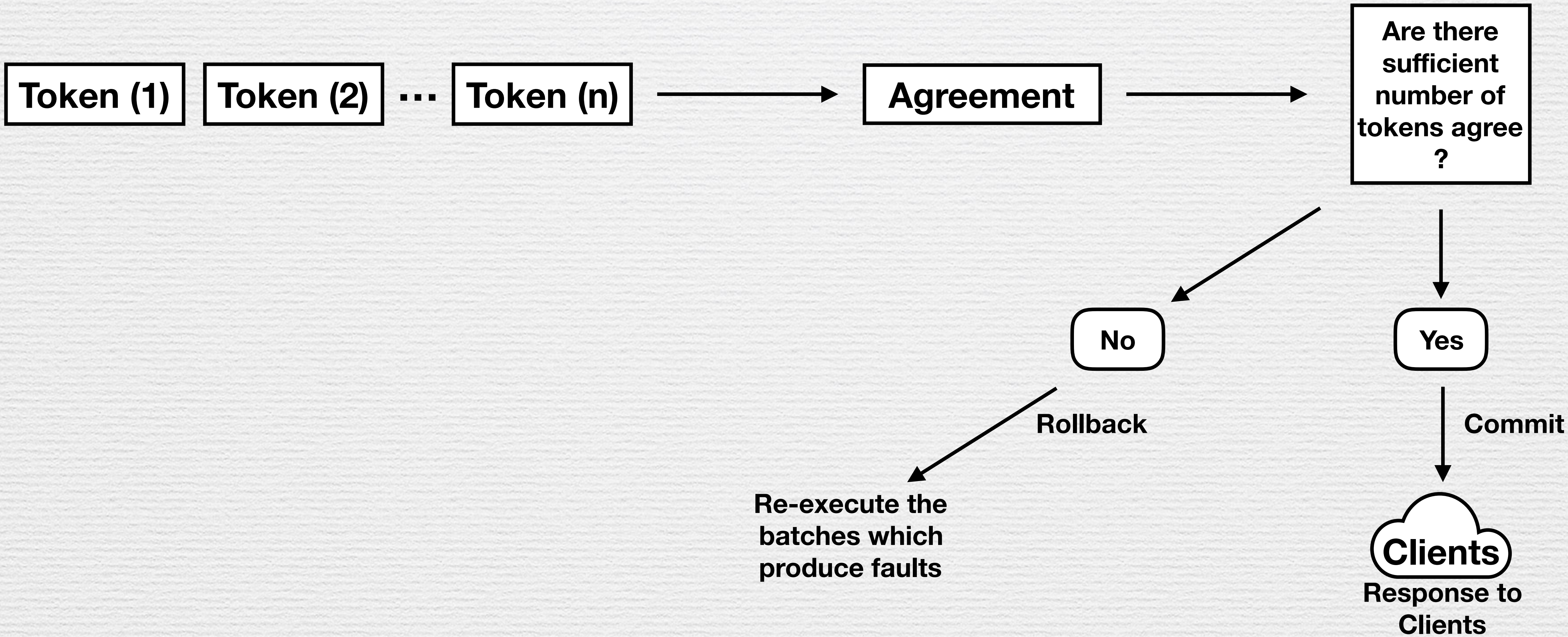
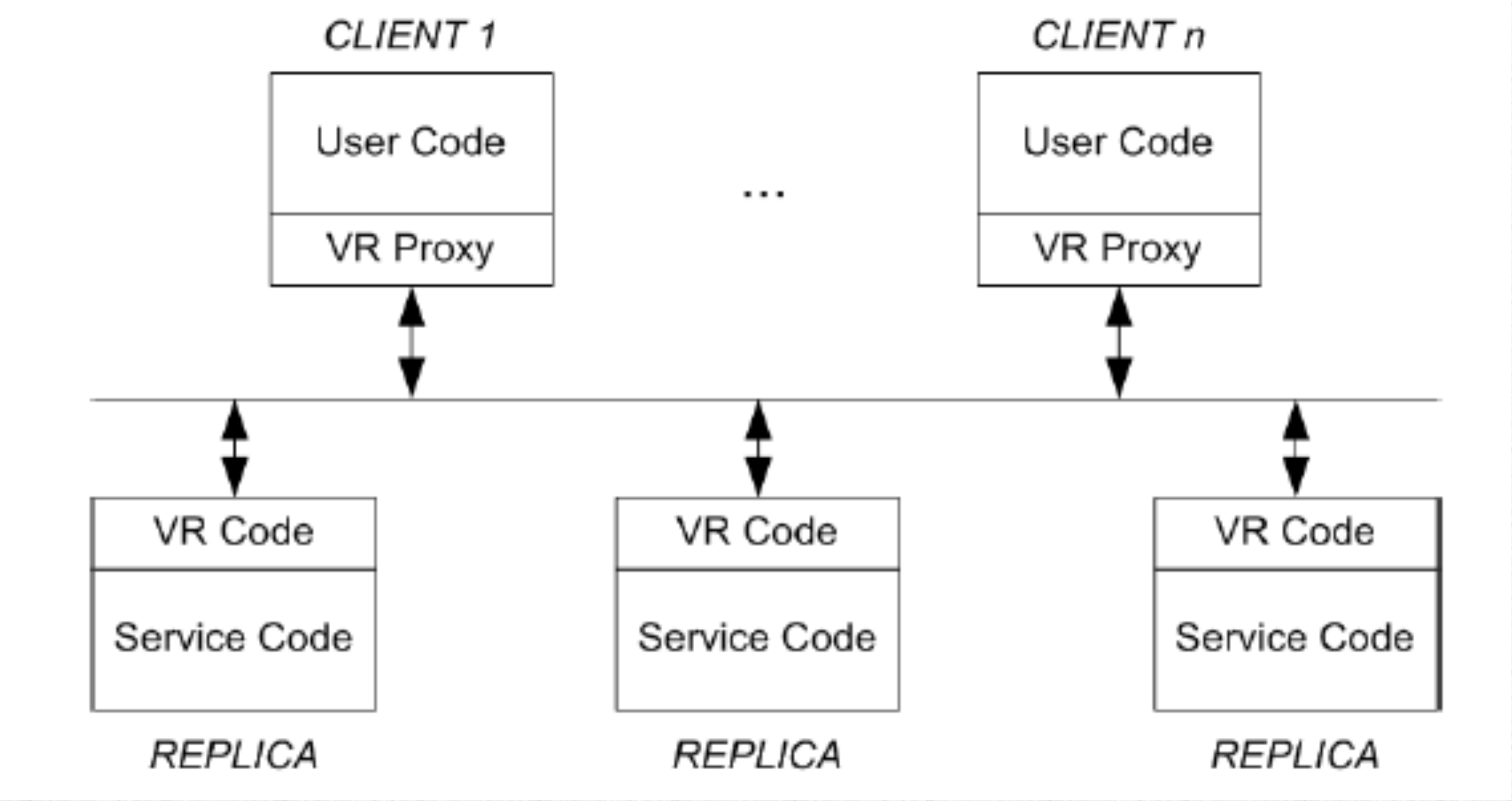


Fig. 5 Verification Stage

Background

- State machine
  - An abstract machine concerning states
- State machine replication
  - Replication: keeping the same state among distinct nodes through placing copies of the State Machine on distinct servers
  - Provides multiple equivalent execution for the same program
    - Enables implementation of analysis tools on some of the replica nodes while not disturbing the work of other replica nodes instead of slow them down.
  - Steps:
    - Receive client requests as Inputs
    - Choose an ordering for the Inputs and execute them in this order on each server
    - Respond to clients with the Output from the State Machine

Background



CLIENT 1
User Code
VR Proxy
...
CLIENT n
User Code
VR Proxy
VR Code
Service Code
REPLICA
VR Code
Service Code
REPLICA
VR Code
Service Code
REPLICA

Overall Model

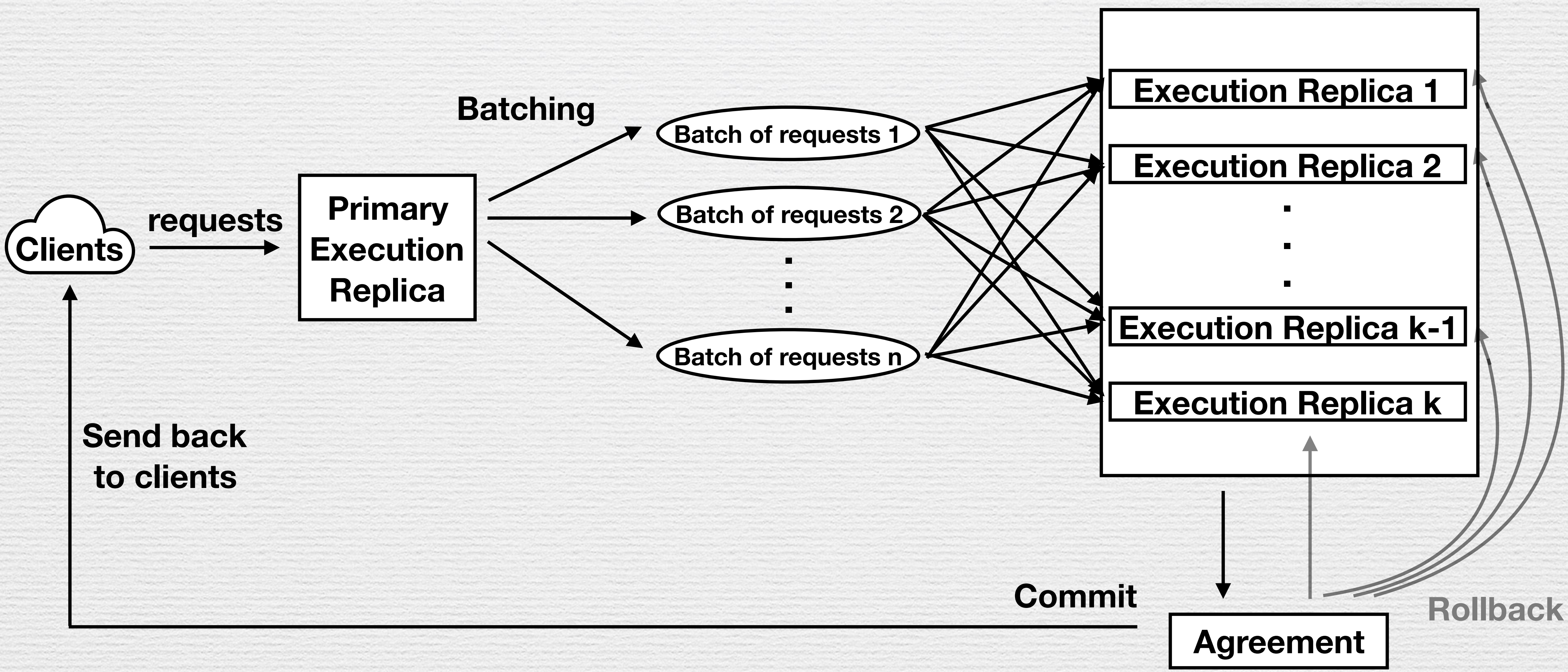


Fig. 1 Overall Model

Characteristics

- A new Execute-Verify architecture that allows state machine replication to scale to multi-core servers
- Can be applied to both synchronous and asynchronous systems
- Guarantees of robustness:
  - Attempting to run only unlikely-to-interfere requests in parallel
  - Ability to detect and recover when concurrency causes executions to diverge

Characteristics

- Nondeterministic interleaving of requests ensures high-performance replication for multi-core servers.
    - It avoids the overhead of enforcing determinism

- Independence enables tolerance of a wide range of faults
    - It helps tolerating crash, omission, or Byzantine faults

Characteristics

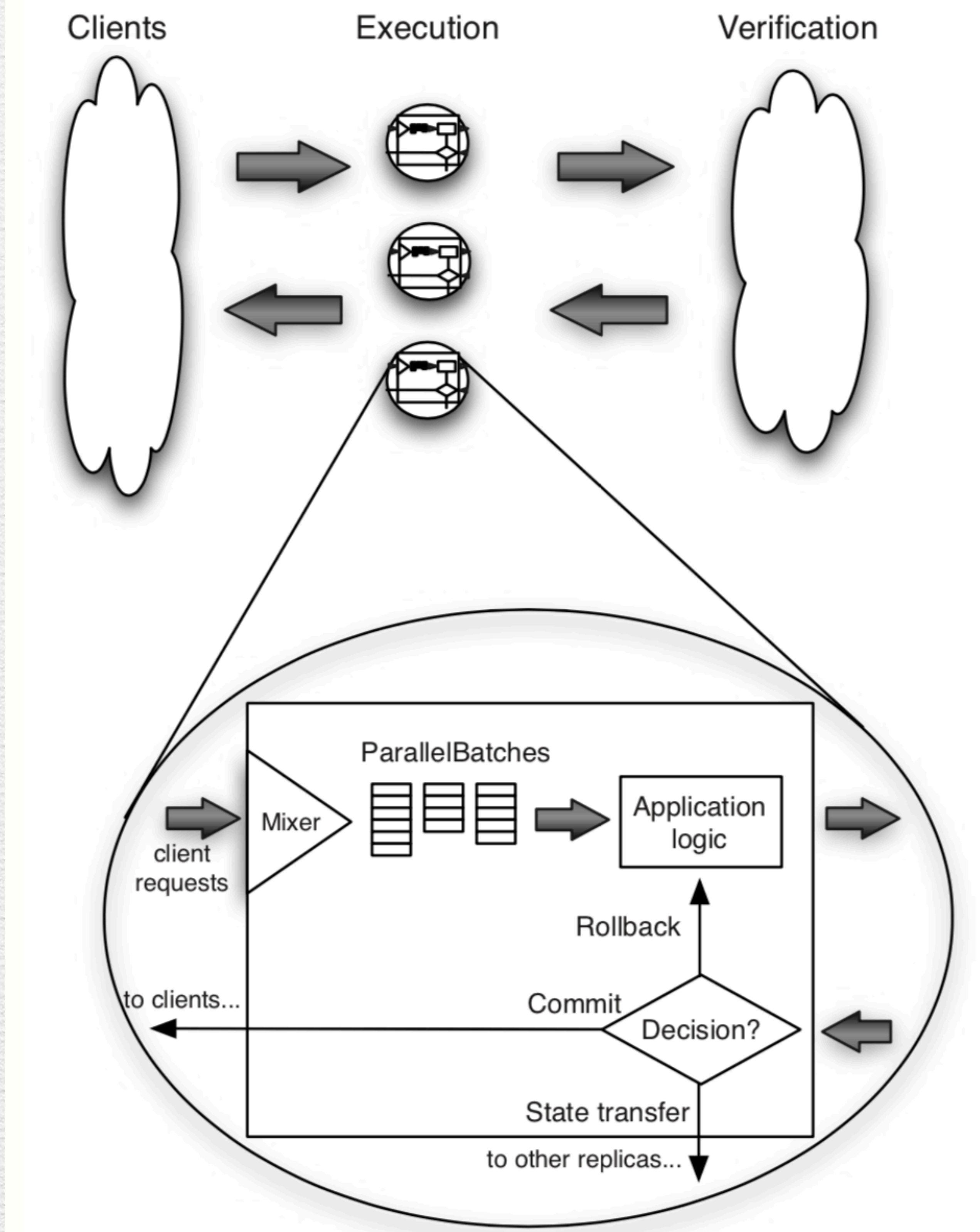


Figure 1: Overview of Eve.

Challenges

- Traditional idea for replicas: hard to implement on multi-core servers
  - Processing the same sequence of requests
- Modern servers: parallel execution can cause divergence of correct states and outputs
  - Most SMR systems require servers to process requests sequentially: a replica finishes executing one request before beginning to execute the next
- Traditional agree-execute method can cause violation of the safety requirements (correct replicas agree on the same state and output)
  - First agree on the order in which requests are to be executed and then execute them

Challenges

- Recent efforts to enforce deterministic parallel execution do not work well
  - Reason 1: the practical limitations of current implementations (e.g. high overhead)
  - Reason 2: many modern replication algorithms do not actually execute operations in the same order at every replica to achieve better performance

- Problems of deterministic execution:
  - Current techniques for deterministic multithreading either require hardware support or are too slow
  - Semantic gap exists between modern SMR protocols and the techniques used to achieve deterministic multithreading.

Challenges

- Current methods to discover the semantics of the requests do not work well
  - Current methods:
    - Aims to achieve replica coordination without forcing all replicas to process identical sequences of inputs
    - Read-only optimization: many modern SMR systems no longer insist that read requests be performed in the same order at all replicas
    - Preferred quorum optimization: allows read requests to be executed only at a preferred quorum of replicas, rather than at all replicas
  - Problems:
    - Deterministic multithreading techniques know nothing of the semantics of the operations they perform
    - Read-only optimizations and preferred quorum operations violate the assumption that all replicas receive identical sequences of inputs and hence lead correct replicas to diverge
    - Read-only requests advance a replica's instruction counter and may cause the replica to acquire additional read locks

Improvement Based on Previous Techniques

- Replicas do not have to execute requests in the same order
- Requests are partitioned in batches
- Allowing different replicas to execute requests within their own batches in parallel
  - Ensuring that the system's important state and output at each replica will match across enough replicas.
- Execute requests concurrently before verify that they have agreed on the state and the output produced by a correct replica
  - If too many replicas diverge so that a correct state/output cannot be identified, it guarantees safety and liveness by rolling back and sequentially and deterministically re-executing the requests

Improvement Based on Previous Techniques

- Minimizes divergence through a mixer stage
  - Applying application-specific criteria to produce groups of requests that are unlikely to interfere
  - Making repair efficient through incremental state transfer and fine-grained rollbacks
- Taking methods to guarantee that replication remains safe and live
  - The agreement is delayed until after execution, even if the program is correct under un-replicated parallel execution
  - When necessary, falling back to sequential re-execution even if the mixer allows interfering requests in the same group.

Basic Steps of Execution Stage

- Eve's execution stage tries to make divergence unlikely
- Batching
  - Clients send their requests to the current primary execution replica
  - The primary execution replica divide clients' requests into batches, assigns each batch a sequence number, and sends them back to all other execution replicas.
  - The primary execution replica sends other data needed to process the requests along with the sequence number
  - Ensuring that distinct batches are processed in order.

Basic Steps of Execution Stage

- Mixing
  - ParallelBatches: groups of requests that can be executed in parallel
  - Each replica runs the same deterministic mixer to partition each batch into these ParallelBatches
  - The mixer ensures that different interleavings will not produce diverging results at distinct replicas.

Basic Steps of Execution Stage

- Parallel Execution
  - Each replica executes the parallelBatches in the order specified by the deterministic mixer.
  - Then a replica computes a hash of its application state and of the outputs corresponding to requests in that batch.
  - This hash, along with its sequenceNumber and the hash for previous batch, constitute a token that is sent to the verification stage in order to discern whether the replicas have diverged

Mixer Design

- The mission of the mixer: to identify requests that may be executed in parallel with low false negative and false positive rates
  - False negatives will cause conflicting requests to be executed in parallel, creating the potential for divergence and rollback
  - False positives will cause requests that could have been successfully executed in parallel to be serialized, reducing the parallelism of the execution

- The mixer parses each request and tries to predict which state it will access
  - It may cause two requests to conflict when they access the same object in a read/write or write/write manner.

Mixer Design

- Methods to avoid putting together conflicting requests:
  - The mixer starts with an empty parallelBatch and two (initially empty) hash tables, one for objects being read, the other for objects being written
  - The mixer then scans in turn each request, mapping the objects accessed in the request to a read or write key
  - Before adding a request to a parallelBatch, the mixer checks whether that request's keys have read/write or write/write conflicts with the keys already present in the two hash tables
  - If not, the mixer adds the request to the parallelBatch and adds its keys to the appropriate hash table
  - When a conflict occurs, the mixer tries to add the request to a different parallelBatch—or creates a new parallelBatch, if the request conflicts with all existing parallelBatches
- Experiment Results:
  - This simple mixer achieves good parallelism (acceptably few false positives), and has not presented any rollbacks (few or no false negatives)

Mixer Design

- Using the names of the tables accessed in read or write mode as read and write keys for each transaction

Transaction	Read and write keys
getBestSellers	**read**: item, author, order_line
getRelated	**read**: item
getMostRecentOrder	**read**: customer, cc_xacts, address, country, order_line
doCart	**read**: item **write**: shopping_cart_line, shopping_cart
doBuyConfirm	**read**: customer, address **write**: order_line, item, cc_xacts, shopping_cart_line

Figure 2: The keys used for the 5 most frequent transactions of the TPC-W workload.

State Management

- Problems
  - Moving from an agree-execute to an execute-verify architecture puts pressure on the implementation of state checkpointing, comparison, rollback, and transfer.
- Eve stores the state using a copy-on-write Merkle tree, whose root is a concise representation of the entire state
  - To achieve efficient state comparison and fine-grained checkpointing and rollback
  - First, it includes only the subset of state that determines the operation of the state machine, omitting other state that has no semantic effect on the state and output produced by the application
  - Second, it provides an abstraction wrapper on some objects to mask variations across different replicas
  - Eve manually annotates the application code to denote the objects that should be added to the Merkle tree and to mark them as tainted when they get modified.

State Management

- Problems of determinism Merkle trees:
  - It is challenging to maintain a deterministic Merkle tree structure under parallel multithread execution and parallel hash generation
  - First current solution: making memory allocation synchronized and deterministic
    - Ignores efforts paid in concurrent memory allocation
  - Second current solution: generating an ID based on object content and to use it to determine an object's location in the tree
    - Fails to work because many objects have the same content

- Solution by Eve
  - Postponing adding newly created objects to the Merkle tree until the end of the batch.
    - Eve scans existing modified objects, and if one contains a reference to an object not yet in the tree, Eve adds that object into the tree's next empty slot and iteratively repeats the process for all newly added objects
  - Reason
    - First, existing objects are already put at deterministic locations in the tree
    - Second, for a single object, Eve can iterate all its references in a deterministic order

State Management

- Challenges when implementing on Java:
  - Objects to which the Merkle tree holds a reference to are not eligible for Java's automatic garbage collection
  - Several standard set-like data structures in Java are not oblivious to the order in which they are populated
  - Two set-like data structures at different replicas may contain the same elements but may generate different checksums when added to a Merkle tree, which causes divergence

- Solution by Eve
  - Creating wrappers that abstract away semantically irrelevant differences between instances of set-like classes kept at different replicas
    - For each set-like data structure, the wrappers generate a deterministic list of all the elements it contains and a corresponding iterator
  - Two wrappers are needed for two Java interfaces (Set and Map) respectively

Verification Stage

- Problems of Execution Stage
  - There are still some divergencies that cannot vanish after execution stage
- Mission of Verification Stage
  - Determining whether enough execution replicas have same state and responses after executing a batch of requests
  - Ensuring that divergences such as conflicting requests in the same parallelBatch cannot affect safety
  - Ensuring that all correct replicas that have executed the ith batch of requests are guaranteed to have reached the same final state and produced the same outputs

Verification Stage

- Concurrency Bugs: deviations from an application's intended behavior that is caused by particular thread interleaving
- Reasons for Eve's ability to mask replica divergences caused by concurrency bugs
  - The mixer makes concurrency bugs less likely to occur by trying to avoid parallelizing requests that interfere
  - Concurrency bugs may manifest differently on different replicas

Basic Operations

- Agreement
  - The verification stage runs an agreement protocol to determine the final decision, which is either commit (if enough tokens match) or rollback (if too many tokens differ)
- Verification on whether replicas have diverged
  - If all tokens agree, the replicas' common final state and outputs are committed
  - If there is divergence, the agreement protocol tries to find a pair of final state and outputs that leads to a correct replica, and ensures that all correct replicas commit to that state and outputs.
  - Otherwise, it rolls back.

Basic Operations

- Commit
  - The execution replicas mark the corresponding sequence number as committed and send the responses to corresponding clients
- Rollback
  - The execution replicas roll back their state to the latest committed sequence number and re-execute the batch sequentially to guarantee progress

Asynchronous Byzantine Fault Tolerant

- Model
  - `u + max(u, r) + 1` execution replicas and `2u + r + 1` verification replicas, which allows the system to remain live despite `u` failures (whether of omission or commission), and safe despite `r` commission failures and any number of omission failures
- Similarities between Eve and PBFT
  - Both protocols attempt to perform agreement among `2u + r + 1` replicas (`3f + 1` in PBFT terminology)
- Differences between Eve and PBFT
  - In PBFT the replicas try to agree on the output of a single node—the primary, while in Eve the object of agreement is the behavior of a collection of execution replicas
  - In PBFT the replicas try to agree on the inputs to the state machine, while in Eve replicas try to agree on the outputs of the state machine

Asynchronous Byzantine Fault Tolerant

- Steps:
  - 1 When an execution replica executes a batch of requests, it sends a ⟨VERIFY, υ, n, T, e⟩ message to all verification replicas, where υ is the current view number, n is the batch sequence number, T is the computed token for that batch, and e is the sending execution replica.
  - 2 When a verification replica receives sufficient VERIFY messages with matching tokens, it marks this sequence number as preprepared and sends a ⟨PREPARE, υ, n, T, v⟩ message to all other verification replicas.
  - 3 When a verification replica receives sufficient matching PREPARE messages, it marks this sequence number as prepared and sends a ⟨COMMIT, υ, n, T, v⟩ to all other verification replicas.

Asynchronous Byzantine Fault Tolerant

- Steps:
  - 4 When a verification replica receives sufficient matching COMMIT messages, it marks this sequence number as committed and sends a ⟨VERIFY-RESPONSE, υ, n, T, v⟩ message to all execution replicas.
  - 5 If agreement can not be reached, it sends a ⟨VERIFY-RESPONSE, υ, n, T, v, f⟩ message to all execution replicas, where f is a flag that indicates that the next batch should be executed sequentially to ensure progress
  - 6 Upon receipt of r + 1 matching VERIFY-RESPONSE messages, an execution replica e judges whether the view number has increased and whether the agreed-upon token matches the one e previously sent, before choosing to commit, transfer its state or rollback.

Asynchronous Byzantine Fault Tolerant

- Commit
  - View number has not increased, and the token matches the execution replica previously sent
  - Releasing the responses to the corresponding clients.
- State Transfer
  - View number has not increased, but the token does not match the execution replica previously sent
  - This replica has diverged from the agreed-upon state and thus issues a state transfer request to other replicas.
- Rollback
  - View number has increased
  - Agreement could not be reached and the primary is changed

Synchronous Primary-backup

- The primary receives client requests and groups them into batches. When a batch `B` is formed, it sends a ⟨`EXECUTE-BATCH, n, B, ND`⟩ message to the backup, where `n` is the batch sequence number and `ND` is the data needed for consistent execution of nondeterministic calls.

- The backup sends its token to the primary, which compares it to its own token.

- If the tokens match, the primary marks this sequence number as stable and releases the responses to the clients.

- If the tokens differ, the primary rolls back its state to the latest stable sequence number and notifies the backup to do the same.

Protection Against Concurrency Bugs

- Concurrency bugs
  - It can lead to both omission faults and commission faults
  - It's easy to repair compared with other Byzantine faults
- Asynchronous case
  - When configured with $n = 2u + 1$ and $r = 0$, Eve guarantees the safety and liveness corresponding to the requirements of state machine replication, and correctness of the state machine itself despite up to u concurrency or omission faults.
  - It ensures that the committed state and outputs at correct replicas match and that requests eventually commit

Protection Against Concurrency Bugs

- Synchronous case
  - When configured with just `u + 1` execution replicas, Eve can continue to operate with one replica if `u` replicas fail by omission.
  - Eve does not have spare redundancy and can not mask concurrency faults at the one remaining replica.
  - In both the synchronous and asynchronous case, when configured for $r = 0$, Eve enters extra protection mode after `k` consecutive batches for which all execution replicas provided matching, timely responses.
- Extra protection during good intervals
  - Good intervals: when there are no other replica faults or time-outs except than those caused by concurrency bugs
  - Eve uses spare redundancy to boost its best-effort protection against concurrency bugs in both the synchronous and asynchronous cases.

Results

- Eve provides gain on execution speed compared to traditional sequential execution approaches
- The mixer never parallelized requests it should have serialized
- The mixer could completely mask concurrency bugs
  - When the bug manifests only in one replica, Eve detects that the replicas have diverged and repairs the damage by rolling back and re-executing sequentially.
- Eve has better performance under heavier workloads
- The scalability decreases for larger objects, which is an artifact of the hashing library.

Results

- The effect of false negatives on throughput
  - Even for 0% false negatives, the throughput drops as the pairwise conflict chance increases due to the decrease of available parallelism.
  - With a 1% conflict rate, the batch will be divided into only a few parallelBatches, so there is a good chance that conflicting requests will land in the same parallelBatch.
  - Increased false positive ratios can lead to lower throughput.

- Eve uses significantly less bandwidth, achieves higher throughput, and provides stronger guarantees compared to passive replication approaches.

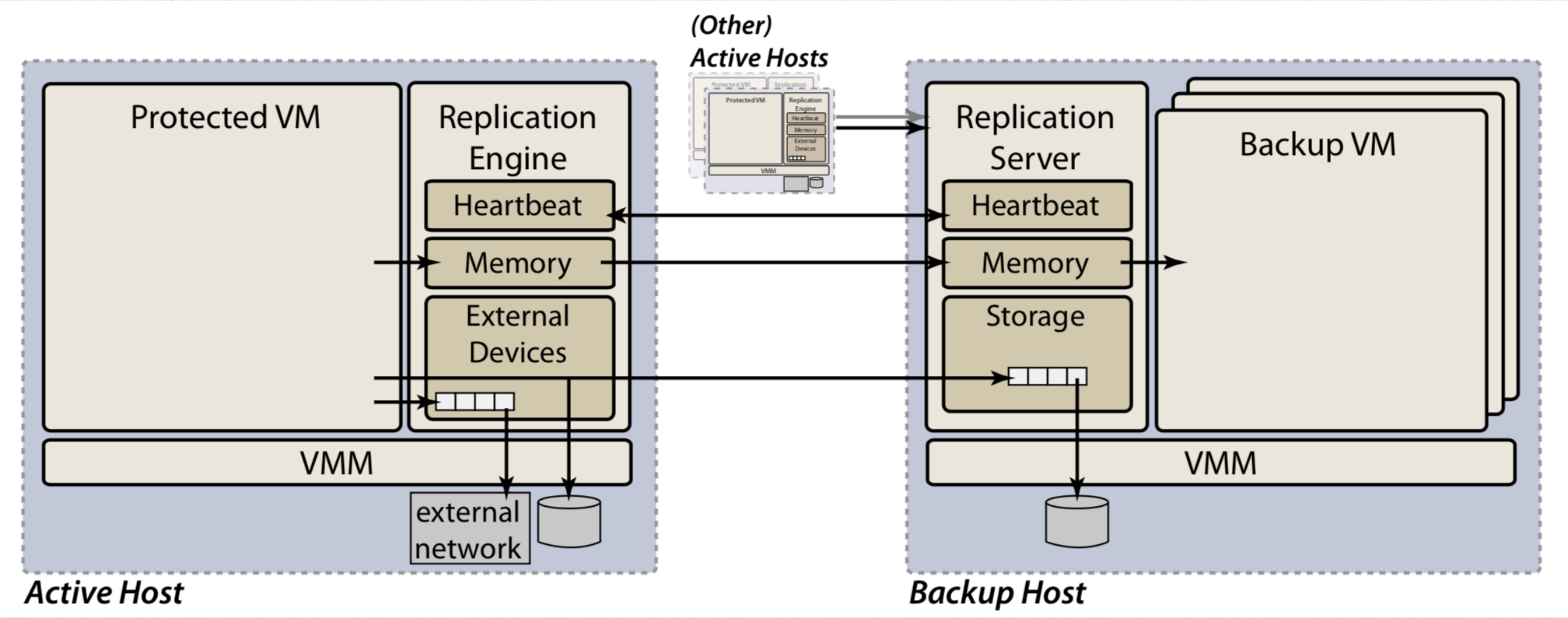
- Eve uses a dynamic batching scheme:
  - The batch size decreases when the demand is low (providing good latency), and increases when the system starts becoming saturated, in order to leverage as much parallelism as possible.
  - It could keep its latency low while maintaining a high peak throughput

Comparison with other Works

- Semi-active Replication
  - It weakens state machine replication both in determinism and in execution independence
    - The primary executes nondeterministically and logs all the nondeterministic actions it performs
    - All other replicas then execute by deterministically reproducing the primary's choices
  - Relevant Works
    - Vandiver et al. : a Byzantine-tolerant semi-active replication scheme for transaction processing systems
    - Kim et al. : applying it to a transactional operating system.
  - Shortcomings
    - Relaxing the requirement of independent execution makes these systems vulnerable to commission failures
    - That the same input is given to all replicas is violated in modern replication systems

Comparison with other Works

- Remus primary-backup system
  - The backup does not execute requests, but instead passively absorbs state updates from the primary
  - Shortcomings: it cannot tolerate any commission failures



Conclusion

- Goal of Eve:
  - Solving the divergence that may occur when executing State Machine Replication in multi-core servers
- Core Method:
  - Mixer that produce groups of requests that are unlikely to interfere
  - Fine-grained rollbacks in verification stage


dank u
ju faleminderit
Tack
Asante
谢谢
Tak
mulţumesc
kiitos
Gracias
Salamat!
Terima kasih
Aliquam
Merci
Dankie
Obrigado
ありがとう
köszönöm
grazie
Aliquam
Go raibh maith agat
děkuji
Thank you
gam