HBase Multisite Replication

Version 3

Consistency Model Background

- Strongest to weakest
  - Strict: Changes are atomic and appear to take effect instantaneously
  - Sequential: Every observer sees all changes in the same order
  - Causal: Changes that are causally related are observed in the same order by all observers
  - Eventual: When no updates occur for a period of time, eventually all updates will propagate through the system and all the replicas will be consistent
  - Weak: No guarantee that all updates will propagate and changes may appear out of order to various observers
- Strict consistency is the ideal model but it is impossible to implement in a distributed system
- Transactions for enforcing casual or stronger consistency ("synchronous replication") are difficult to scale and limit availability during network partition
- Asynchronous replication implies eventual consistency
  - Replicas are out of sync until the replication happens

Regional Consistency

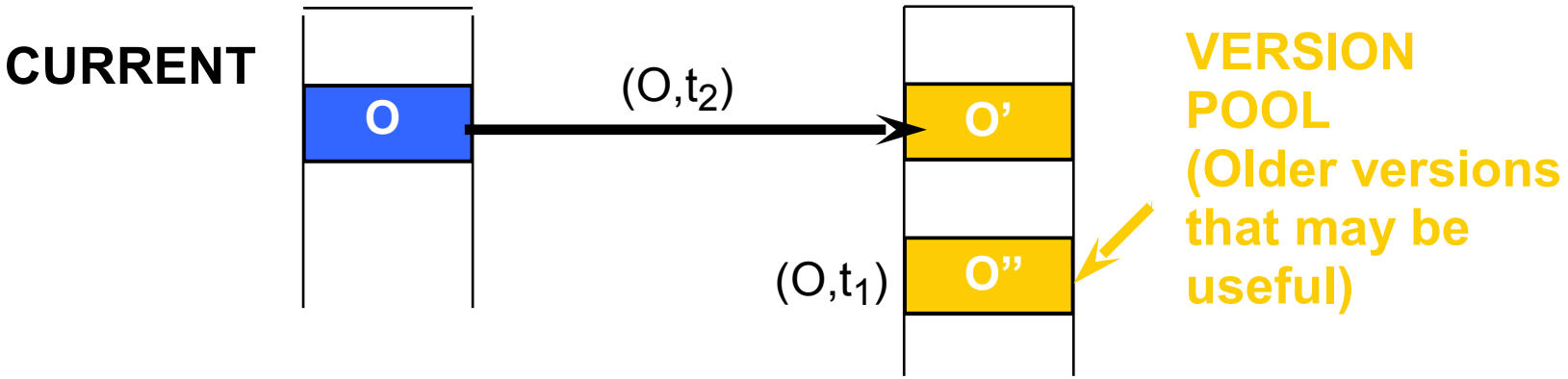
- Because eventual consistency is weaker than casual consistency, an observer of one store may observe updates out of order with respect to observers of another store

- *Stores in each cluster will be fully consistent, so try to divide the problem such that each cluster can handle it independently within some global divide-and-conquer scheme*

- *Otherwise, applications must tolerate and handle some degree of global inconsistency*

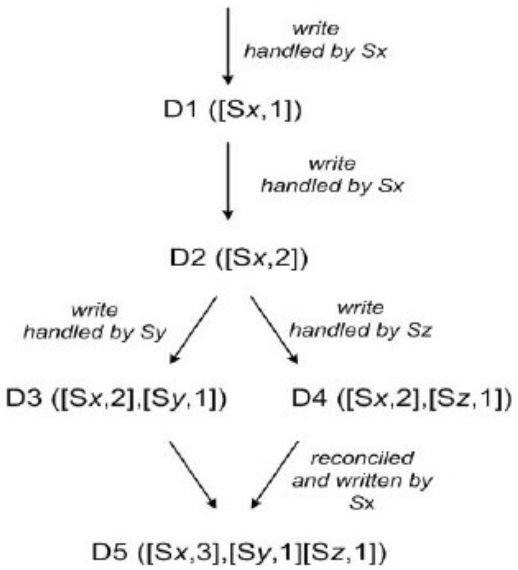
Application Strategies

- Strategies for handling eventual consistency
  - Multiversioning and timestamping
    - All edits are timestamped and the storage system supports storage and retrieval of multiple versions of a cell
    - No edit will conflict with another
    - Applications can query for the latest version according to timestamp or N versions, depending on requirements
  - HBase supports uses in its normal operation both multiversioning and timestamping, so can handle this on behalf of the application with minimal application considerations



Application Strategies

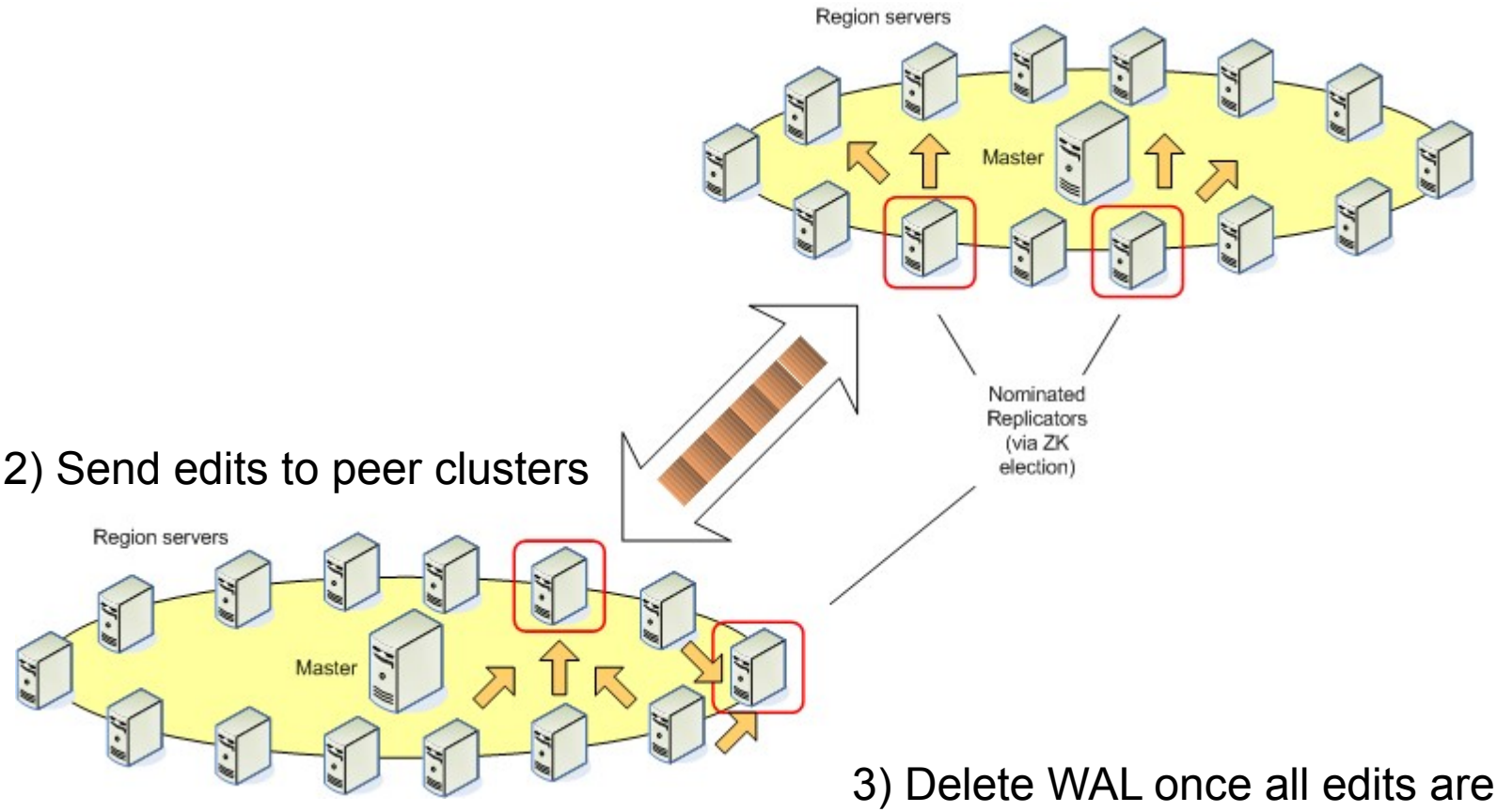
- Strategies for handling eventual consistency
  - Vector clocks (Lamport clocks)
    - Does not require synchronized clocks
    - A vector clock is effectively a list of { item, counter } pairs
    - Each node maintains a monotonic counter
    - One vector clock is associated with every version of an item
    - One can determine whether two versions of an object are on parallel branches or have a causal ordering
      - If the counters on the first object's clock are less-than-or-equal to all of the items in the second clock, then the first is an ancestor of the second and can be forgotten
      - Otherwise, the two changes are in conflict and require reconciliation
      - Amazon's Dynamo does this
      - Applications must perform reconciliation on their own



HBase Store Replication

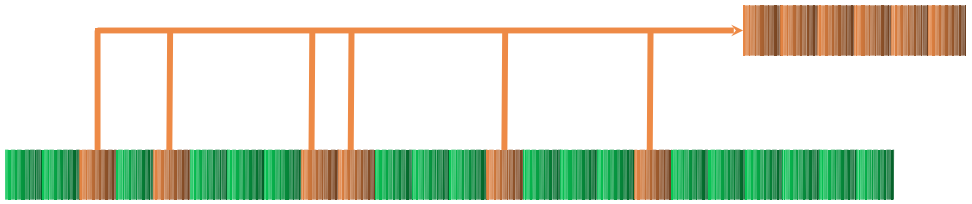
- Lazy / asynchronous
- Update everywhere
- Convergence with multiversioning instead of ACID
- Simple mesh of peers
- Use column and/or table descriptor attributes to specify the scope of data items stored there
  - Local: Do not replicate
  - Global: Replicate everywhere
- Only replicate globally scoped cells
- Run background thread on each region server that processes each HLog just after it is rolled
- Have the RegionServers nominate via ZooKeeper a subset of themselves to act as aggregator/gateways for inter-cluster replication
- Tune HLog rolling for new scenario
  - Disallow deletion of log until remotes acknowledge all replications

HBase Store Replication



3) Delete WAL once all edits are acked

1) Collect global edits from newly rolled WAL



Required Modifications

- Reserve HTD and HCD attribute for scoping ("`SCOPE`")
- Update meta table (root, meta, historian, etc.) schemas to mark them as locally scoped
- HLog entries need an attribute to track input source
  - Avoid replicating cells back again to the source peer
- Add upcall from HLog roll logic that replicator can hook
- Add internal interface for temporarily disallowing HLog deletions
- New meta table – like region historian – for replication peer entries
  - Labels
  - Peer replicator endpoint addresses
  - Last seen sequence numbers
- Implement HRS log walker and replicator thread and RPC interface for supporting the cluster replicator role
- Replication protocol

Required Modifications

- Replicator peer implementation
  - Sender side
  - Receiver side
  - Redo log for tracking edits received but not yet committed
- Implement cluster replicator role nomination and fail over (via ZK)
- Support for new shell commands
- Support online (no disable/enable required) addition and removal of column families

New Administrative Interfaces

- CREATE PEER
  - Create a new cluster peering configuration
  - HBaseAdmin

    ```
    createPeer(byte[] label, byte[] secretKey,
     InetSocketAddress[] contactAddrs)
    ```

  - Shell

    ```
    CREATE PEER <label> , <secretKey> [ , <address> ]+
    ```

- DROP PEER
  - Terminate a cluster peering arrangement and remove the configuration
  - HBaseAdmin

    ```
    dropPeer(byte[] label)
    ```

  - Shell

    ```
    DROP PEER <label>
    ```

- ENABLE PEER
  - Enable replication via a cluster peering arrangement
  - HBaseAdmin

    ```
    enablePeer(byte[] label)
    ```

  - Shell

    ```
    ENABLE PEER <label>
    ```

New Administrative Interfaces

- DISABLE PEER
  - Stop replication to/from a peer cluster
  - HBaseAdmin

    ```
    disablePeer(byte[] label)
    ```
  - Shell

    ```
    DISABLE PEER <label>
    ```
- ADD PEER ADDRESS
  - Associate a DNS name or IP address with a peer cluster
  - HBaseAdmin

    ```
    addPeerAddress(byte[] label, InetSocketAddress[] addrs)
    ```
  - Shell

    ```
    ADD PEER ADDRESS <label> <address> [ , <address> ]+
    ```
- DROP PEER ADDRESS
  - Disassociate a DNS name or IP address with a peer cluster
  - HBaseAdmin

    ```
    dropPeerAddress(byte[] label, InetSocketAddress[] addrs)
    ```
  - Shell

    ```
    DROP PEER ADDRESS <label> , <address> [ , <address> ]+
    ```

Replication Protocol

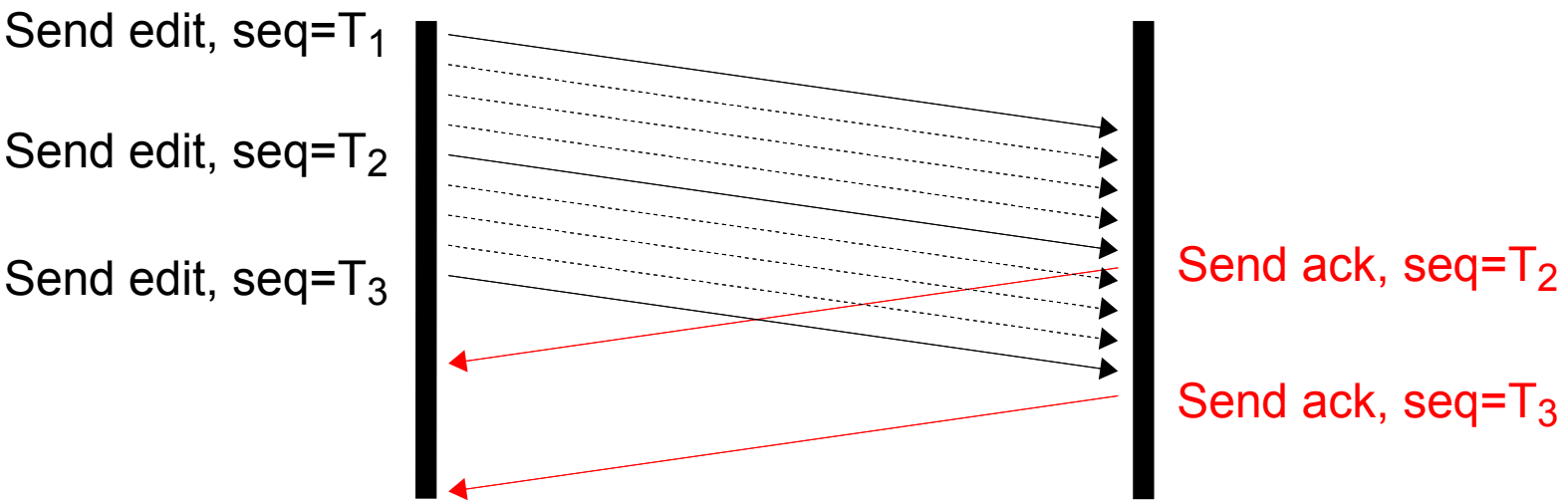
- Peer-to-peer / symmetric
- Encrypted
  - Fast block cipher
  - Shared secret keys
- Simple request-response interactions
- Optimize for bulk transfer
  - Compression
  - Cumulative acknowledgment
- Binary format
  - Language and architecture neutrality
- Multiple transport support
  - Plain TCP
  - HTTP/HTTPS
  - MQ (ActiveMQ) ?
  - Mail (JavaMail) ?

Replication Protocol

- KeyValue replication

  - Source: Send <KV, sequence, KeyValue>
  - Target: Send <KV-ACK, sequence>

  - Source must use a monotonic counter when assigning sequence numbers to cells
  - Target should send periodic cumulative acknowledgement, where the period can be configurable or can be dynamically adjusted depending on source sending rate



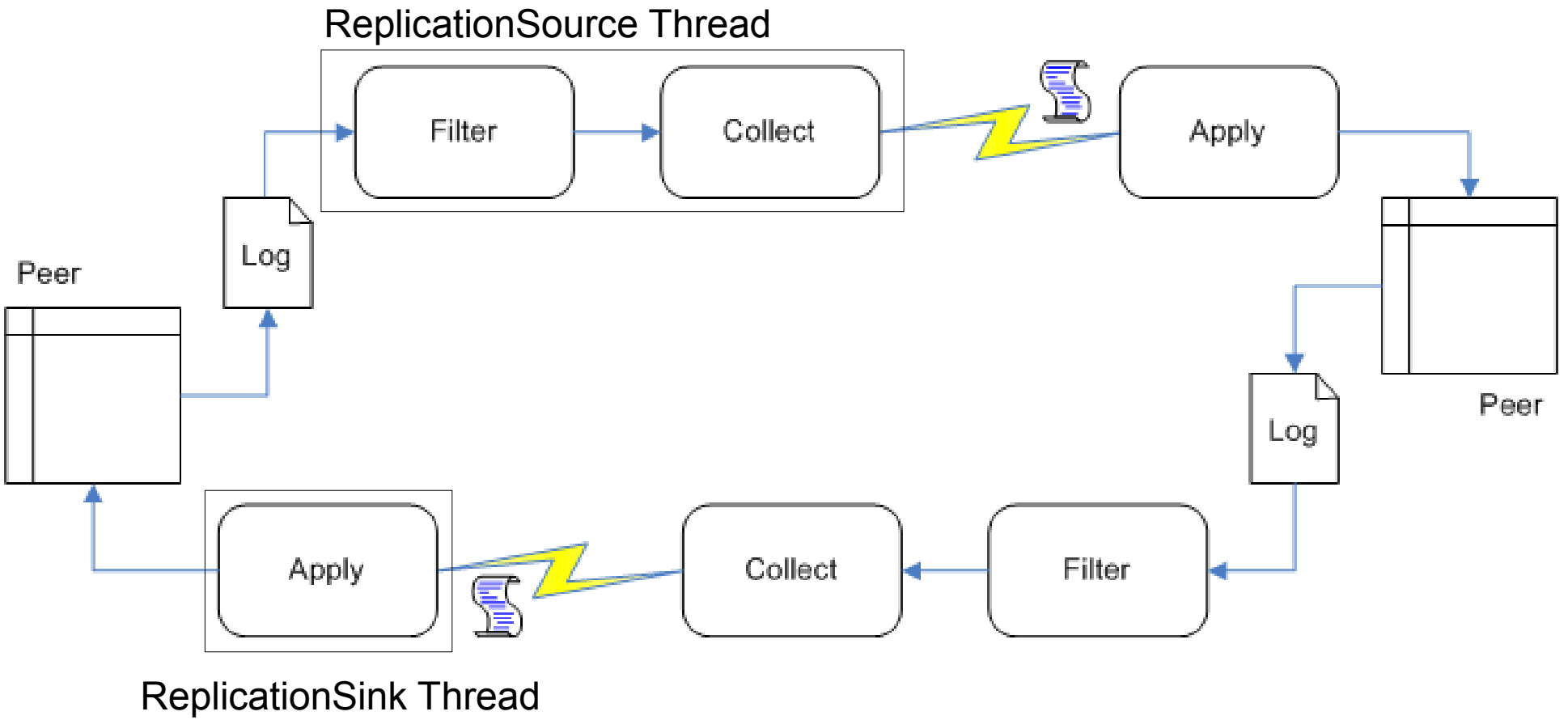
Replication Protocol

- Schema query
  - Target: Send <SCHEMA-REQ, tableName>
  - Source: Send <SCHEMA, sequence, tableName, HTD>
  - Target: Send <SCHEMA-ACK, sequence>
  - Source should initialize the value of 'sequence' to the value of System.currentTimeMillis()
  - Target should initiate a schema query whenever replication between a given peer pair is restarted, or when a cell with a key specifying a previously unseen table or column family is received
  - Target should replace local HTD with replicated HTD
- NOTE: There should be an admin setting per cluster to disable this and instead resolve by dropping cells without corresponding local schema; this setting can be stored in the peer table

Replication Protocol

- Schema update
  - Source: Send <SCHEMA, sequence, tableName, HTD>
  - Target: Send <SCHEMA-ACK, sequence>
  - Source should initialize the value of 'sequence' to the value of System.currentTimeMillis()

Replication



- One ReplicationSource thread per peer
  - Each ReplicationSource maintains a cell pointer { HLog file, offset }
  - Pointer is advanced when paired Sink acks

- One ReplicationSink per peer at the other end

Fault Tolerance and Recovery Strategies

- Replicator node failure
  - A failed replicator will shut down (or node goes offline), leading to a ZK session expiration and deletion of the corresponding ephemeral znode
  - Region servers will periodically list the children of the designated replicator root znode; if the count falls below a minimum, it will volunteer by adding a znode and starting a replicator thread

Fault Tolerance and Recovery Strategies

- Network partition
  - WALs are held until all pending cells are acked
  - Replicators will retry connections to last known peer locations until successful
  - When network connections are broken and reestablished, the replicator will restart replication at a slow pace and gradually increase the pace
    - TCP should do most of the work here

Fault Tolerance and Recovery Strategies

- Value collisions due to clients explicitly setting timestamps
  - Without any consideration to this, collisions would be resolved from the client perspective as if by an arbitrary coin toss
  - Could consider a couple of collision resolution strategies and implement them as we do Filters now
    - Users choose the best strategy for the application and associate the resolver with the corresponding column(s) in the schema
    - Collision resolution configuration can be propagated as part of schema detail
    - Resolvers operate server side: Examine conflicting user data, consult their user-supplied configuration, and choose whether to keep the existing value or replace it with the incoming one