A Brief History of Chain Replication

Christopher Meiklejohn // @cmeik
QCon 2015, November 17th, 2015

The Overview

1. **Chain Replication** for High Throughput and Availability
2. Object Storage on **CRAQ**
3. **FAWN:** A Fast Array of Wimpy Nodes
4. Chain Replication in **Theory and in Practice**
5. **HyperDex:** A Distributed, Searchable Key-Value Store
6. **ChainReaction**: a Causal+ Consistent Datastore based on Chain Replication
7. **Leveraging Sharding** in the Design of Scalable Replication Protocols

Chain Replication for High Throughput and Availability

OSDI 2004

Storage Service API

- `V <- read(objId)`
  Read the value for an object in the system

- `write(objId, V)`
  Write an object to the system

Primary-Backup
Replication

- Primary-Backup
  Primary sequences all write operations and forwards them to a non-faulty replica

- Centralized Configuration Manager
  Promotes a backup replica to a primary replica in the event of a failure

Quorum Intersection
Replication

- Quorum Intersection
  Read and write quorums used to perform requests against a replica set, ensure overlapping quorums

- Increased performance
  Increased performance when you do not perform operations against every replica in the replica set

- Centralized Configuration Manager
  Establishes replicas, replica sets and quorums

Chain Replication
Contributions

- High-throughput
  Nodes process updates in serial, responsibility of "primary" divided between the head and the tail nodes

- High-availability
  Objects are tolerant to `f` failures with only `f + 1` nodes

- Linearizability
  Total order over all read and write operations

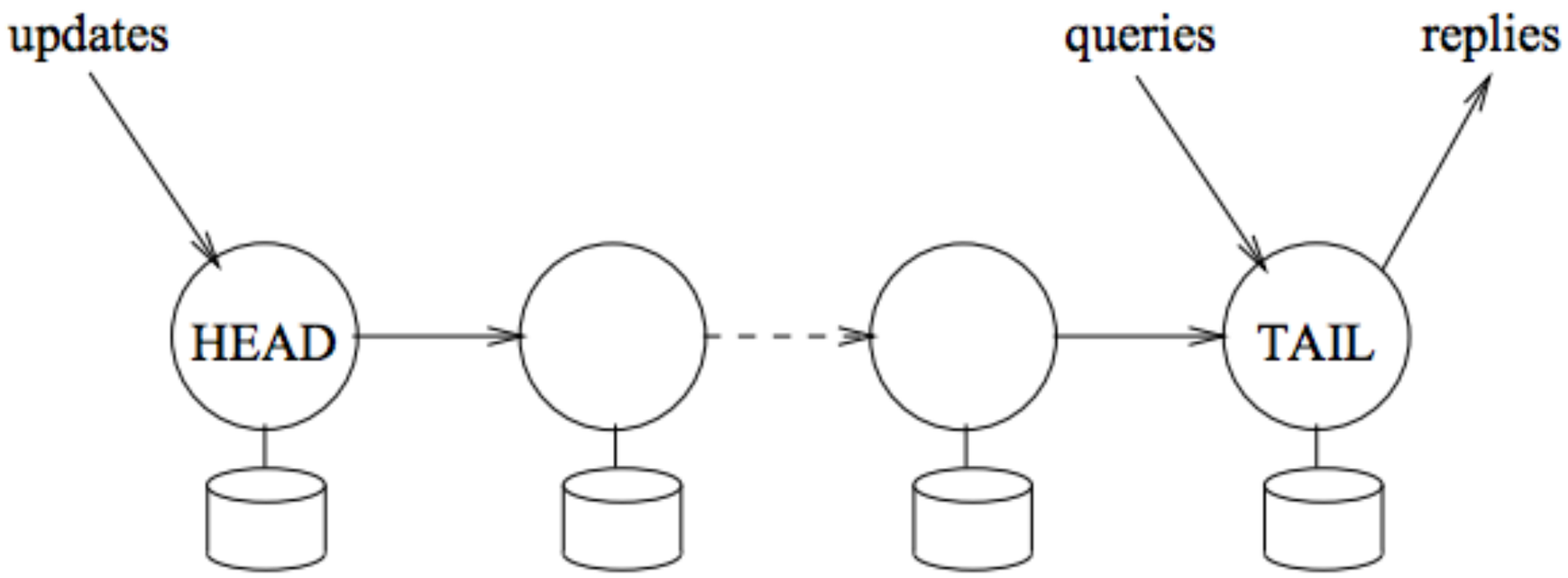

Figure 2: A chain.

Chain Replication Algorithm

- **Head applies update and ships state change**
  Head performs the write operation and send the result down the chain where it is stored in replicas history

- **Tail “acknowledges” the request**
  Tail node “acknowledges” the user and services write operations

- **“Update Propagation Invariant”**
  Reliable FIFO links for delivering messages, we can say that servers in a chain will have potentially greater histories than their successors

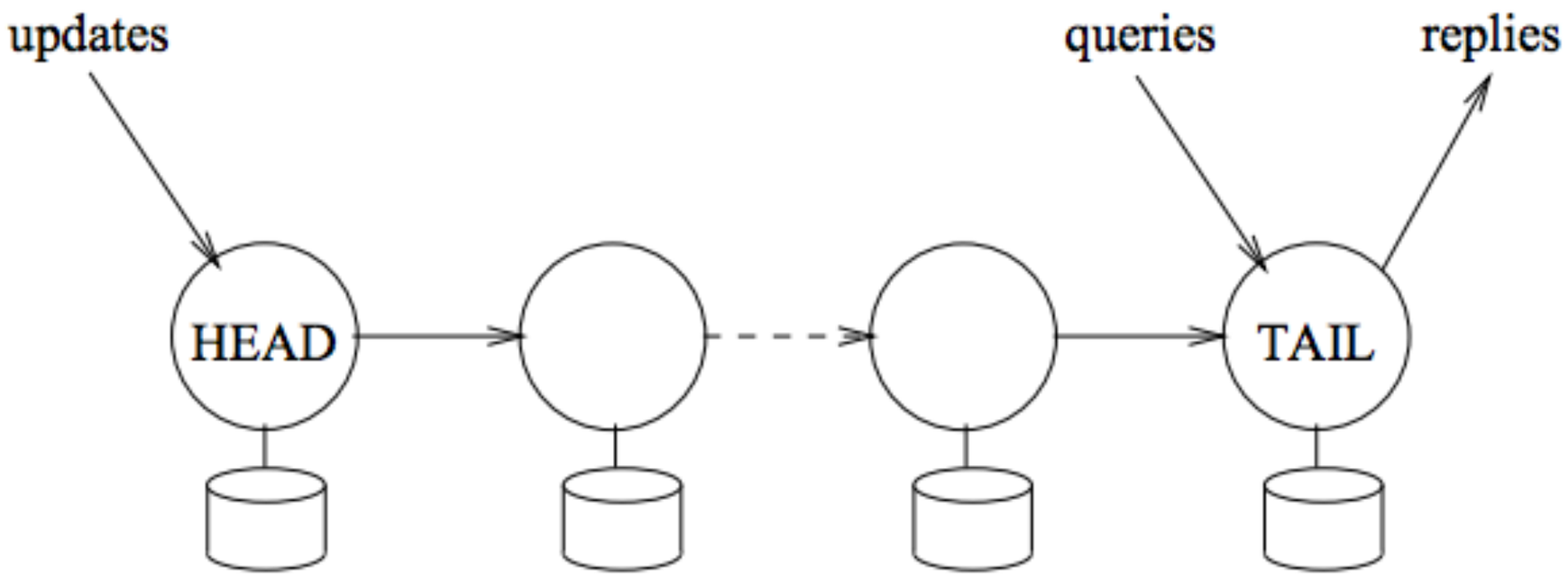

Figure 2: A chain.

Failures?

Reconfigure Chains

Chain Replication
Failure Detection

- **Centralized Configuration Manager**
  Responsible for managing the “chain” and performing failure detection
- **“Fail-stop” failure model**
  Processors fail by halting, do not perform an erroneous state transition, and can be reliably detected

Chain Replication
Reconfiguration

- Failure of the head node
  Remove H replace with successor to H

- Failure of the tail node
  Remove T replace with predecessor to T

Chain Replication
Reconfiguration

- **Failure of a “middle” node**
  Introduce acknowledgements, and track “in-flight” updates between members of a chain

- **“Inprocess Request Invariant”**
  History of a given node is the history of its successor with “in-flight” updates

Object Storage on CRAQ

CRAQ Motivation

- CRAQ
  "Chain Replication with Apportioned Queries"

- Motivation
  Read operations can only be serviced by the tail

CRAQ Contributions

- Read Operations
  Any node can service read operations for the cluster, removing hotspots

- Partitioning
  During network partitions: “eventually consistent” reads

- Multi-Datacenter Load Balancing
  Provide a mechanism for performing multi-datacenter load balancing

CRAQ Consistency Models

- **Strong Consistency**
  Per-key linearizability
- **Eventual Consistency**
  For committed writes, monotonic read consistency
- **Restricted Eventual Consistency**
  Restricted with maximal bounded inconsistency based on versioning or physical time

CRAQ Algorithm

- Replicas store multiple versions for each object
  Each object copy contains version number and a dirty/clean status

- Tail nodes mark objects “clean”
  Through acknowledgements, tail nodes mark an object “clean” and remove other versions

- Read operations only serve “clean” values
  Any replica can accept write and “query” the tail for the identifier of a “clean” version

- “Interesting Observation”
  No longer can we provide a total order over reads, only writes and reads or writes and writes.

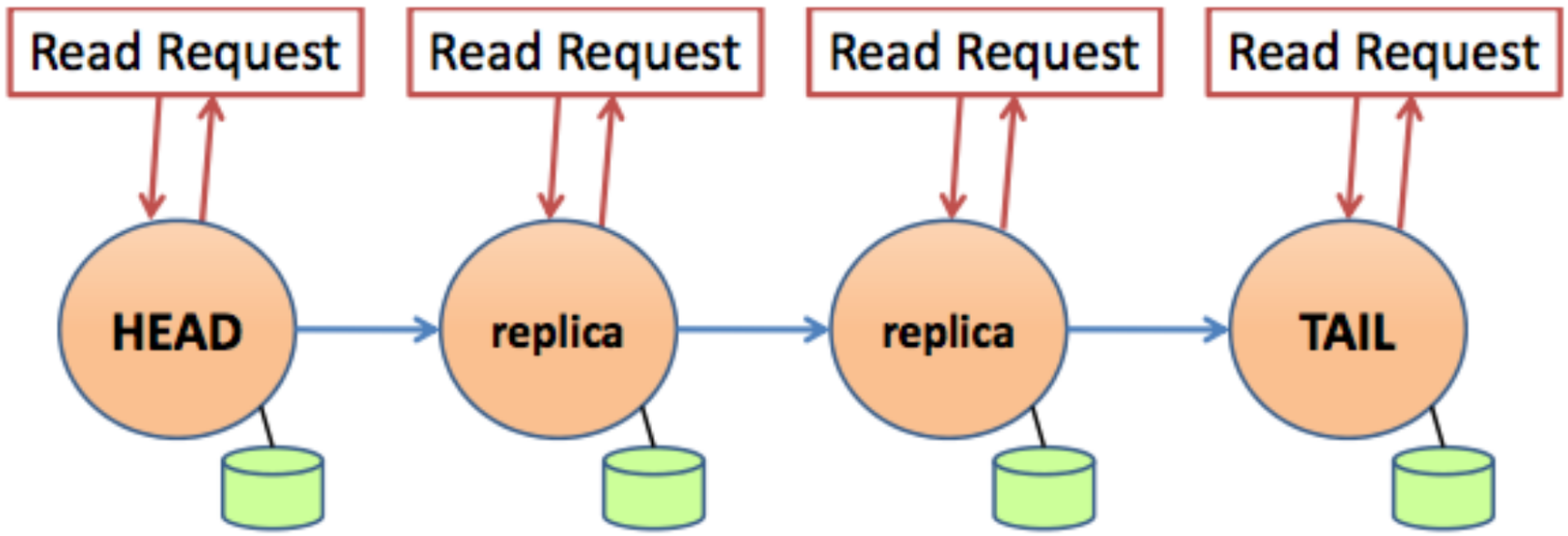

Figure 2: **Reads to clean objects in CRAQ can be completely handled by any node in the system.**

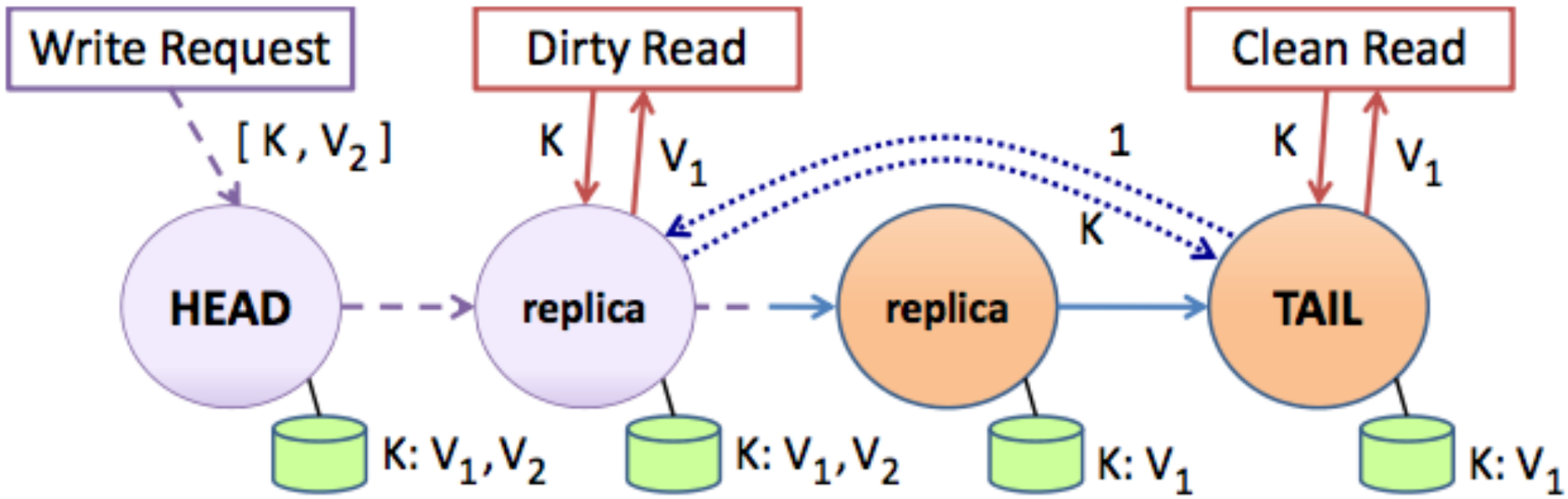

Figure 3: **Reads to dirty objects in CRAQ can be received by any node, but require small version requests (dotted blue line) to the chain tail to properly serialize operations.**

CRAQ Single-Key API

- **Prepend or append to a given object**
  Apply a transformation for a given object in the data store

- **Increment/decrement**
  Increment or decrement a value for an object in the data store

- **Test-and-set**
  Compare and swap a value in the data store

CRAQ Multi-Key API

- Single-Chain
  Single-chain atomicity for objects located in the same chain

- Multi-Chain
  Multi-Chain update use a 2PC protocol to ensure objects are committed across chains

CRAQ Chain Placement

- Multiple Chain Placement Strategies
  - “Implicit Datacenters and Global Chain Size”
    Specify number of DC’s and chain size during creation
  - “Explicit Datacenters and Global Chain Size”
    Specify datacenters and chain size per datacenter
  - “Explicit Datacenters Chain Size”
    Specify datacenters and chains size per datacenter
- “Lower Latency”
  Ability to read from local nodes reduces read latency under geo-distribution

CRAQ TCP Multicast

- **Can be used for disseminating updates**
  Chain used only for signaling messages about how to sequence update messages

- **Acknowledgements**
  Can be multicast as well, as long as we ensure a downward closed set on message identifiers

FAWN: A Fast Array of Wimpy Nodes

SOSP 2009

FAWN-KV & FAWN-DS

- “Low-power, data-intensive computing”
  Massively powerful, low-power, mostly random-access computing

- Solution: FAWN architecture
  Close the IO/CPU gap, optimize for low-power processors

  - Low-power embedded CPUs

  - Satisfy same latency, same capacity, same processing requirements

FAWN-KV

- Multi-node system named FAWN-KV
  Horizontal partitioning across FAWN-DS instances: log-structured data stores
- Similar to Riak or Chord
  Consistent hashing across the cluster with hash-space partitioning

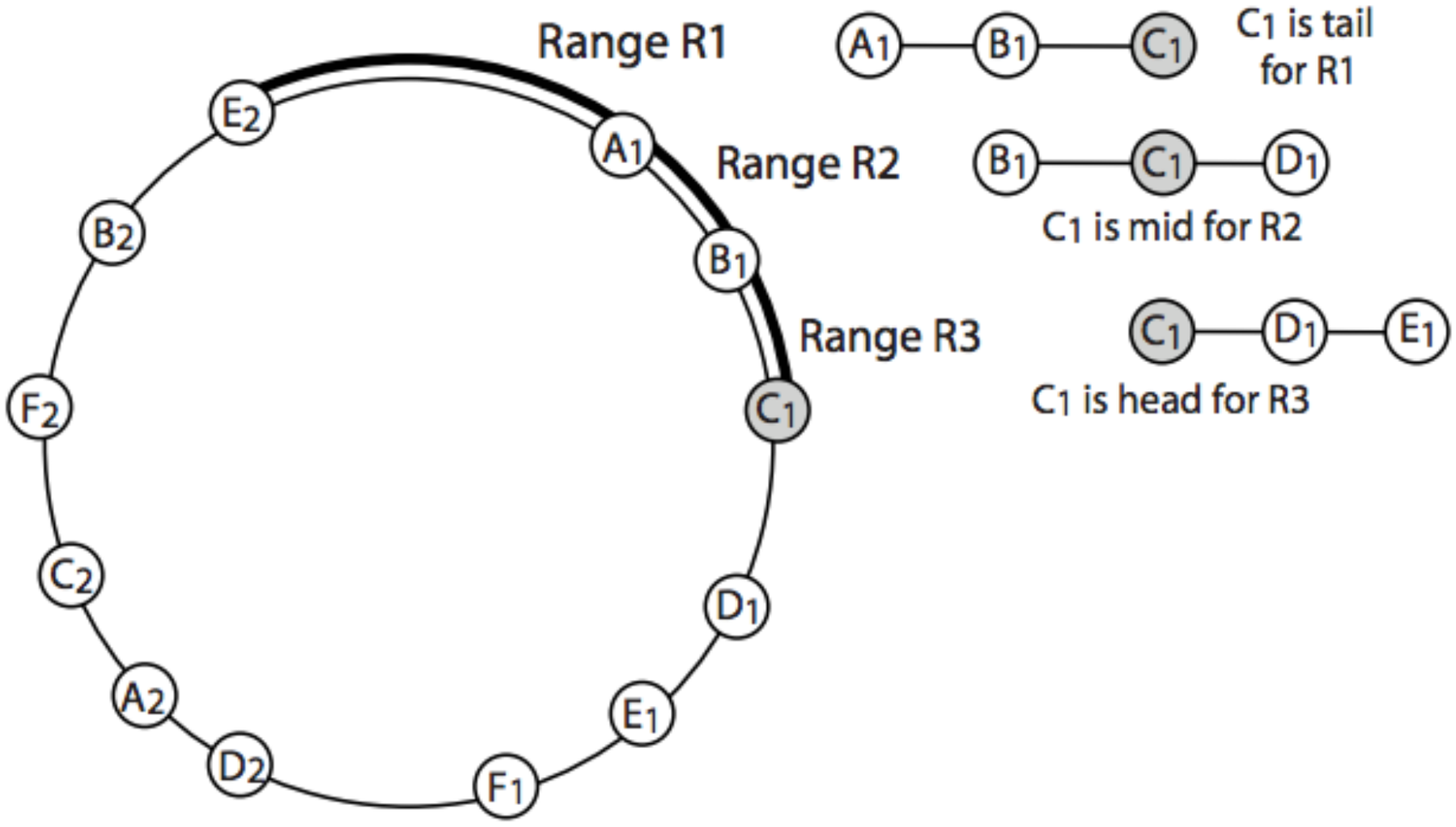

Figure 6: Overlapping Chains in the Ring – Each node in the consistent hashing ring is part of $R = 3$ chains.

FAWN-KV Optimizations

- **In-memory lookup by key**
  Store an in-memory location to a key in a log-structured data structure

- **Update operations**
  Remove reference in the log; garbage collect dangling references during compaction of the log

- **Buffer and log cache**
  Front-end nodes that proxy requests cache requests and results to those requests

FAWN-KV Operations

- Join/Leave operations
  Two phase operations: pre-copy and log flush
- Pre-copy
  Ensures that joining nodes get copy of state
- Flush
  Operations ensure that operations performed after copy snapshot are flushed to the joining node

FAWN-KV Failure Model

- Fail-Stop
  Nodes are assumed to be fail stop, and failures are detected using front-end to back-end timeouts

- Naive failure model
  Assumed and acknowledged that backends become fully partitioned: assumed backends under partitioning can not talk to each other

Chain Replication in Theory and in Practice

Hibari Overview

- Physical and Logical Bricks
  Logical bricks exist on physical and make up striped chains across physical bricks

- “Table” Abstraction
  Exposes itself as a SQL-like “table” with rows made up of keys and values, one table per key

- Consistent Hashing
  Multiple chains; hashed to determine what chain to write values to in the cluster

- “Smart Clients”
  Clients know where to route requests given metadata information

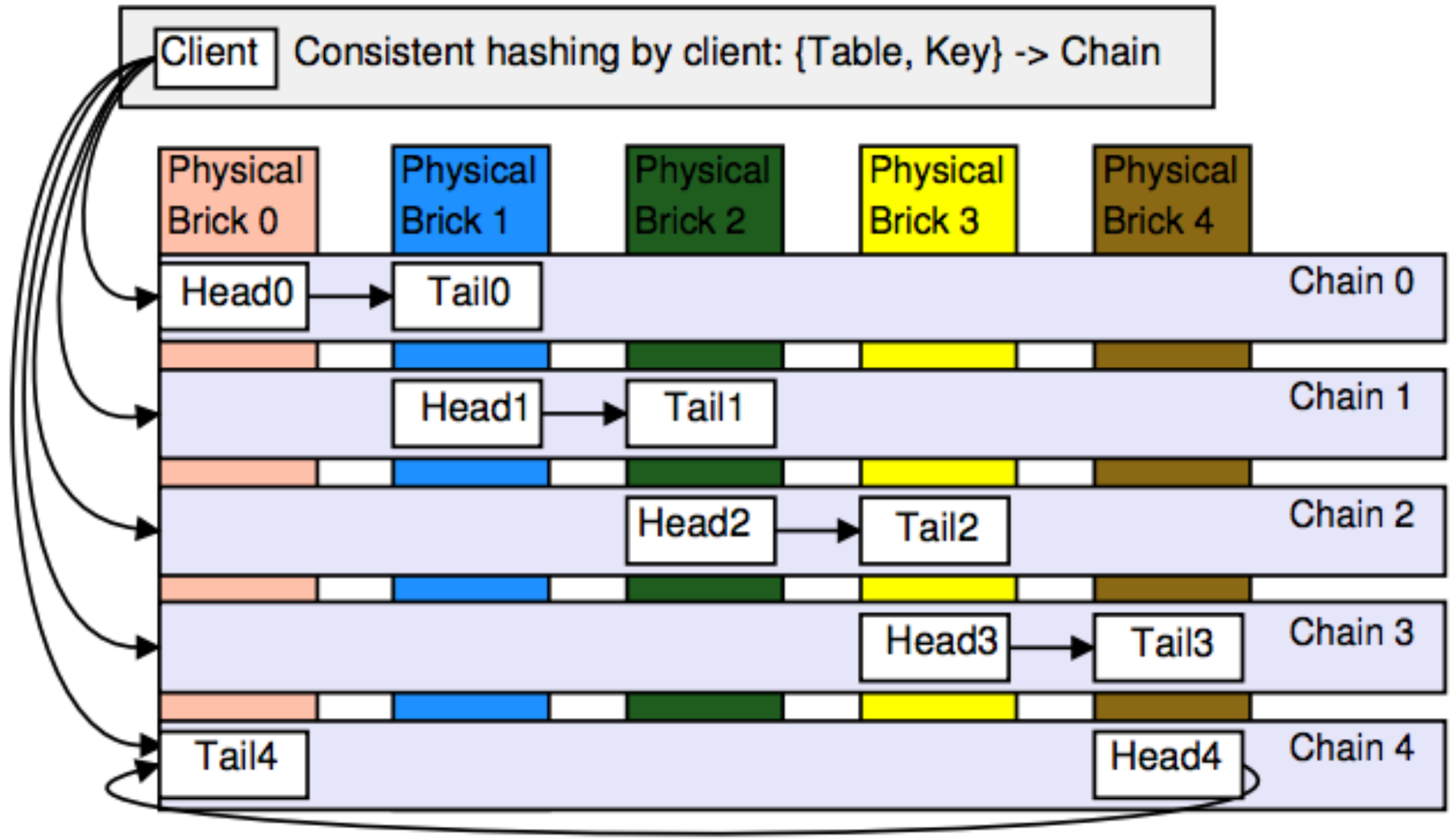

Figure 3. Hibari architecture: an alternate view of Figure 2 with each physical brick represented in a vertical column.

Hibari “Read Priming”

- **“Priming” Processes**
  In order to prevent blocking in logical bricks, processes are spawned to pre-read data from files and fill the OS page cache

- **Double Reads**
  Results in reading the same data twice, but is faster than blocking the entire process to perform a read operation

Hibari Rate Control

- Load Shedding
  Processes are tagged with a temporal time and dropped if events sit too long in the Erlang mailbox

- Routing Loops
  Monotonic hop counters are used to ensure that routing loops do not occur during key migration

Hibari Admin Server

- Single configuration agent
  Failure of this only prevents cluster reconfiguration
- Replicated state
  State is stored in the logical bricks of the cluster, but replicated using quorum-style voting operations

Hibari “Fail Stop”

- “Send and Pray”
  Erlang message passing can drop messages and only makes particular guarantees about ordering, but not delivery

- Routing Loops
  Monotonic hop counters are used to ensure that routing loops do not occur during key migration

Hibari Partition Detector

- **Monitor two physical networks**
  Application which sends heartbeat messages over two physical networks in attempt increase failure detection accuracy

- **Still problematic**
  Bugs in the Erlang runtime system, backed up distribution ports, VM pauses, etc.

Hibari “Fail Stop” Violations

- Fast chain churn
  Incorrect detection of failures result in frequent chain reconfiguration
- Zero length chains
  This can result in zero length chains if churn occurs to frequently

HyperDex: A Distributed, Searchable Key-Value Store

SIGCOMM 2011

HyperDex Motivation

- **Scalable systems with restricted APIs**
  Only mechanism for querying is by "primary key"
- **Secondary attributes and search**
  Can we provide efficient secondary indexes and search functionality in these systems?

HyperDex Contribution

- “Hyperspace Hashing”
  Uses all attributes of an object to map into multi-dimensional Euclidean space

- “Value-Dependent Chaining”
  Fault-tolerant replication protocol ensuring linearizability

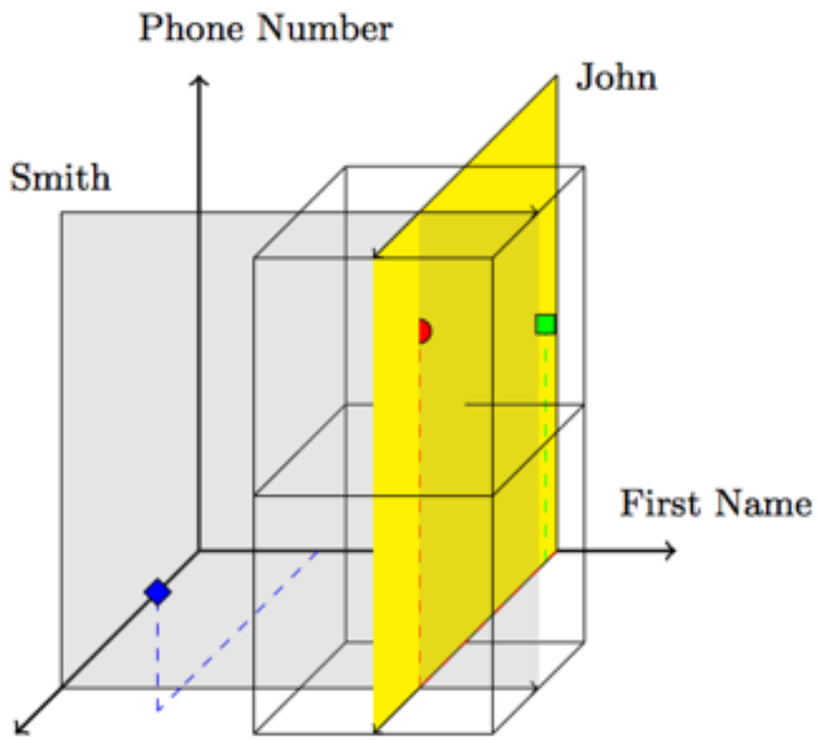

Figure 1: Simple hyperspace hashing in three dimensions. Each plane represents a query on a single attribute. The plane orthogonal to the axis for "Last Name" passes through all points for `last_name = 'Smith'`, while the other plane passes through all points for `first_name = 'John'`. Together they represent a line formed by the intersection of the two search conditions; that is, all phone numbers for people named "John Smith". The two cubes show regions of the space assigned to two different servers. The query for "John Smith" needs to contact only these servers.

HyperDex
Consistency and Replication

- “Point leader”
  Determined through hashing, used to sequence all updates for an object

- Attribute hashing
  Chain for the object is determined by hashing secondary attributes for the object

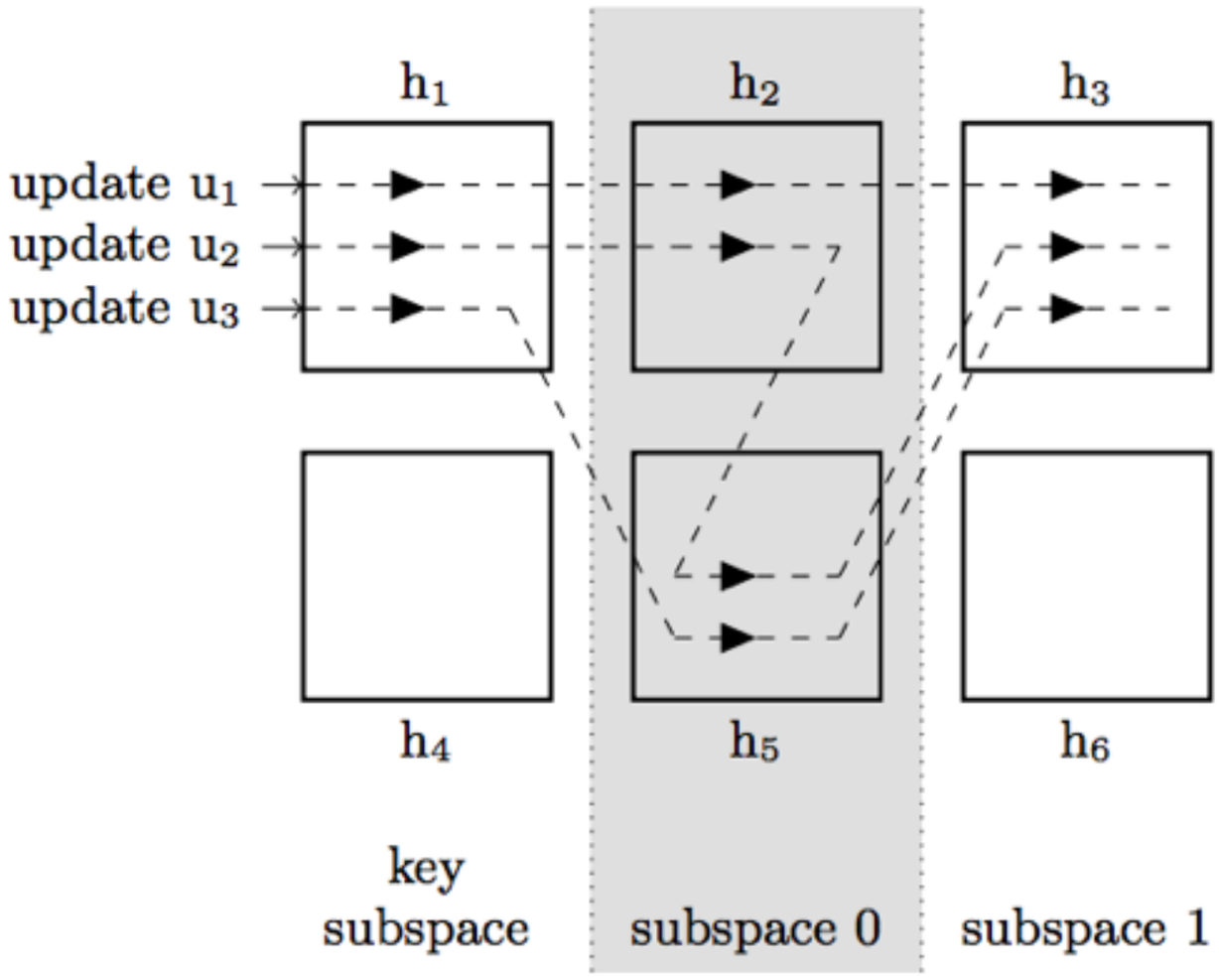

Figure 3: HyperDex's replication protocol propagates along value-dependent chains. Each update has a value-dependent chain that is determined solely by objects' current and previous values and the hyperspace mapping.

HyperDex
Consistency and Replication

- Updates “relocate” values
  On relocation, chain contains old and new locations, ensuring they preserve the ordering

- Acknowledgements purge state
  Once a write is acknowledged back through the chain, old state is purged from old locations

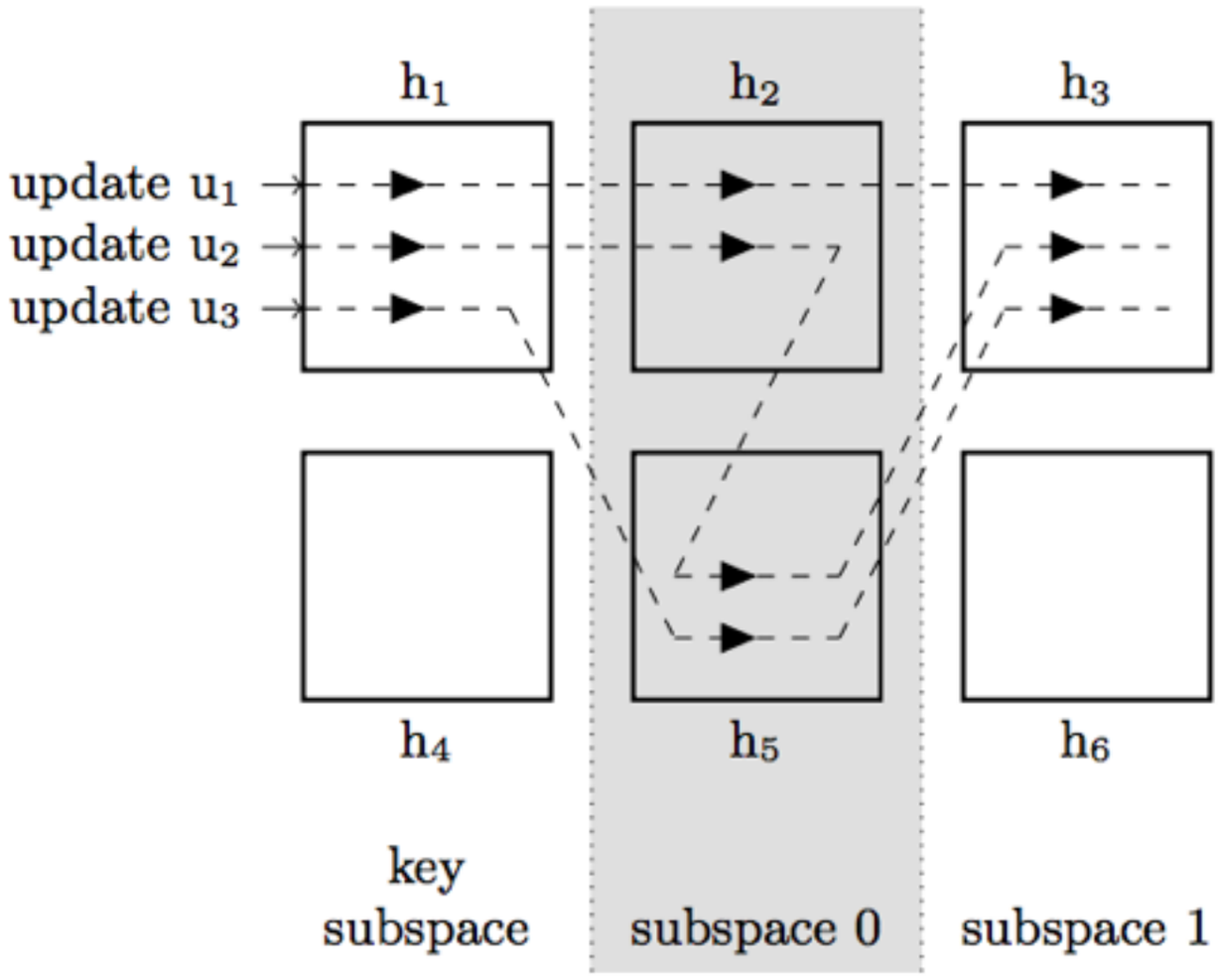

Figure 3: HyperDex's replication protocol propagates along value-dependent chains. Each update has a value-dependent chain that is determined solely by objects' current and previous values and the hyperspace mapping.

HyperDex
Consistency and Replication

- “Point leader” includes sequencing information
  To resolve out of order delivery for different length chains, sequencing information is included in the messages

- Each “node” can be a chain itself
  Fault-tolerance achieved by having each hyperspace mapping an instance of chain replication

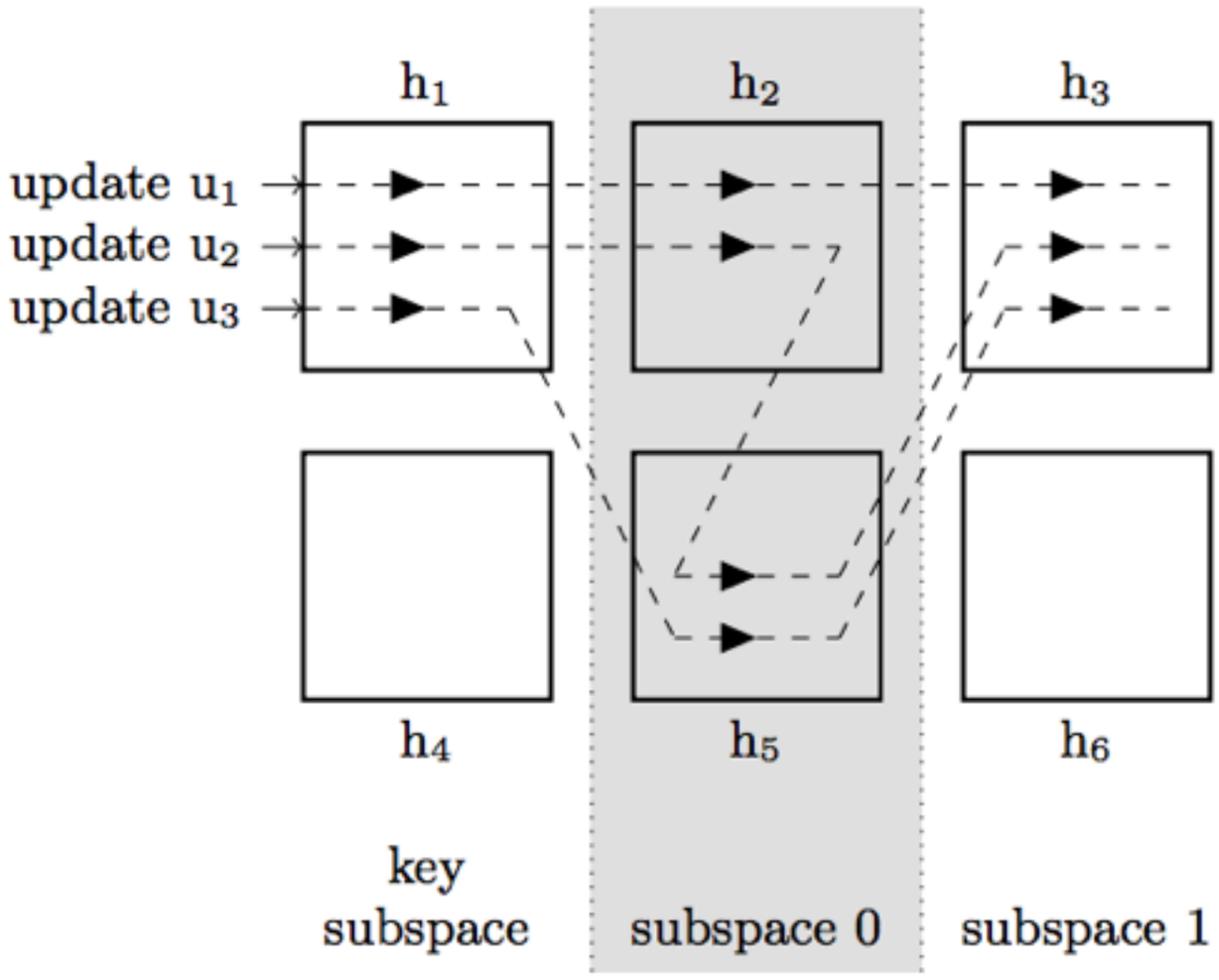

Figure 3: HyperDex's replication protocol propagates along value-dependent chains. Each update has a value-dependent chain that is determined solely by objects' current and previous values and the hyperspace mapping.

HyperDex
Consistency and Replication

- **Per-key Linearizability**
  Linearizable for all operations, all clients see the same order of events

- **Search Consistency**
  Search results are guaranteed to return all committed objects at the time of request

ChainReaction:
a Causal+ Consistent Datastore based on Chain Replication

Eurosys 2013

ChainReaction:
Motivation and Contributions

- Per-Key Linearizability
  Too expensive in the geo-replicated scenario

- Causal+ Consistency
  Causal consistency with guaranteed convergence

- Low Metadata Overhead
  Ensure metadata does not cause explosive growth

- Geo-Replication
  Define an optimal strategy for geo-replication of data

ChainReaction:
Conflict Resolution

- **“Last Writer Wins”**
  Convergent given a “synchronized” physical clock, based

- **Antidote, etc.**
  Show that CRDTs can be used in practice to make this more deterministic

ChainReaction:
Single Datacenter Operation

- Causal Reads from K Nodes
  Given UPI, assume reads from K-1 nodes observe causal consistency for keys

- Explicit Causality (not Potential)
  Explicitly transmit list of operations that are causally related to submitted update

- “Datacenter Stability”
  Update is stable within a particular datacenter and no previous update will ever be observed

ChainReaction:
Multi Datacenter Operation

- **Tracking with DC-based "version vector"**
  "Remote proxy" used to establish a DC-based version vector

- **Explicit Causality (not Potential)**
  Apply only updates where causal dependencies are satisfied within the DC based on a local version vector

- **"Global Stability"**
  Update is stable within **all datacenters** and no previous update will ever be observed

Leveraging Sharding in the Design of Scalable Replication Protocols

SOSP 2011 Poster Session
SoCC 2013

Elastic Replication:
Motivation and Contributions

- Customizable Consistency
  Decrease latency for weaker guarantees regarding consistency

- Robust Consistency
  Consistency does not require accurate failure detection

- Smooth Reconfiguration
  Reconfiguration can occur without a central configuration service

Fail-Stop: Challenges

- Primary-Backup
  False suspicion can lead to promotion of a backup while concurrent writes on the non-failed primary can be read

- Quorum Intersection
  Under reconfiguration, quorums may not intersect for all clients

Elastic Replication: Algorithm

- **Replicas contain a history of commands**
  Commands are sequenced by the head of the chain

- **Stable prefix**
  As commands are acknowledged, each replica reports the length of it's stable prefix

- **Greatest common prefix is "learned"**
  Sequencer promotes the greatest common prefix between replicas

Elastic Replication: Algorithm

- Safety
  When nodes suspect a failure in the network, nodes “wedge” where no operations can be app
    - Only updates in the history may become stable
- Liveness
  Replicas and chains are reconfigured to ensure progress
    - History is inherited from replicas and reconfigured to preserve UPI

Elastic Replication: Elastic Bands

- Horizontal partitioning
  Requests are sharded across elastic bands for scalability

- Shards configure neighboring shards
  Shards are responsible for sequencing configurations of neighboring shards

- Requires external configuration
  Even with this, band configuration must be managed by an external configuration service

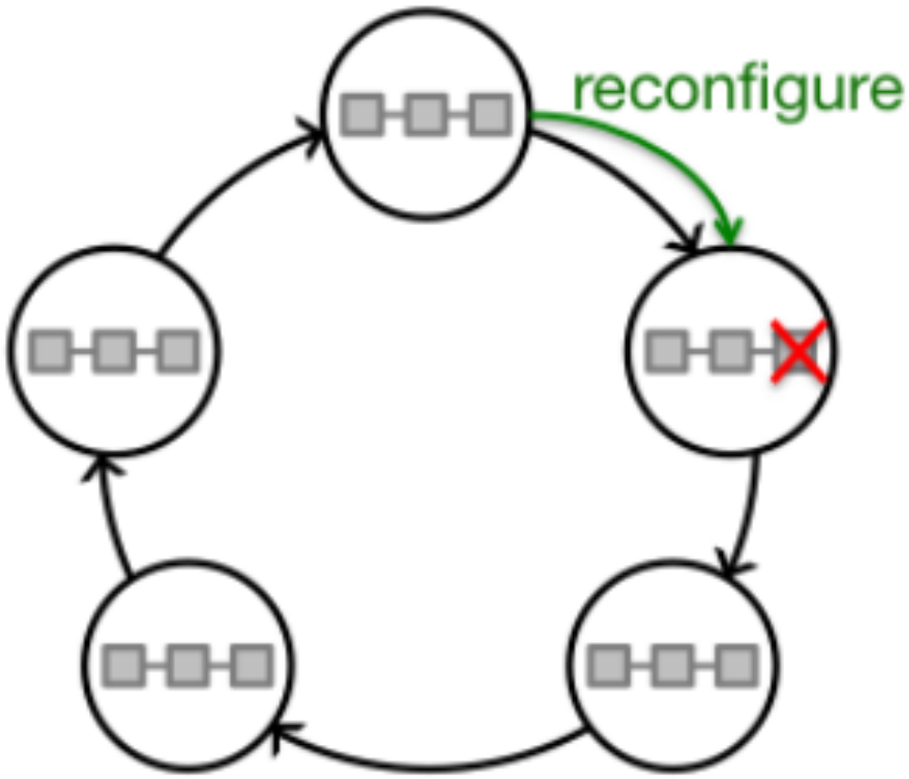

Figure 3: Example of an elastic band. Each (replicated) shard is sequenced by its predecessor on the band as indicated by the arrows. The sequencer of a shard issues new configurations when needed.

Elastic Replication: Read Operations

- **Read requests must be sent down chain**
  Read operations must be sequenced for the system to properly determine if a configuration has been wedged
- **Reads can be serviced by other nodes**
  Read out of the stabilized reads for a weaker form of consistency.

In Summary

- “Fail-Stop” Assumption
  In practice, fail-stop can be a difficult model to provide given the imperfections in VMs, networks, and programming abstractions

- Consensus
  Consensus still required for configuration, as much as we attempt to remove it from the system

- Chain Replication
  Strong technique for providing linearizability, which requires only f + 1 nodes for failure tolerance

Thanks!

Christopher Meiklejohn
@cmeik