Kai – An Open Source Implementation of Amazon's Dynamo

takemaru

Outline

- Amazon's Dynamo
  - Motivation
  - Features
  - Algorithms
- Kai
  - Build and Run
  - Internals
  - Roadmap

Dynamo: Motivation

- Largest e-commerce site
  - 75K query/sec (my estimation)
    - 500 req/sec * 150 query/req
  - O(10M) users and Many items

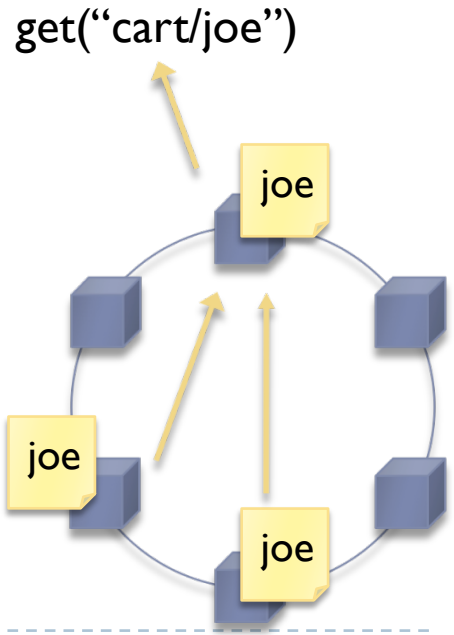
- Why not RDBMS?
  - Not easy to scaling-out or load balancing
  - Many components only need primary key access

- Databases are required just for Amazon
  - *Dynamo* for primary key accesses
  - *SimpleDB* for complex queries
  - *S3* for large files

- Dynamo is used for
  - Shopping carts, customer preferences, session management, sales rank, and product catalogs

Dynamo: Features

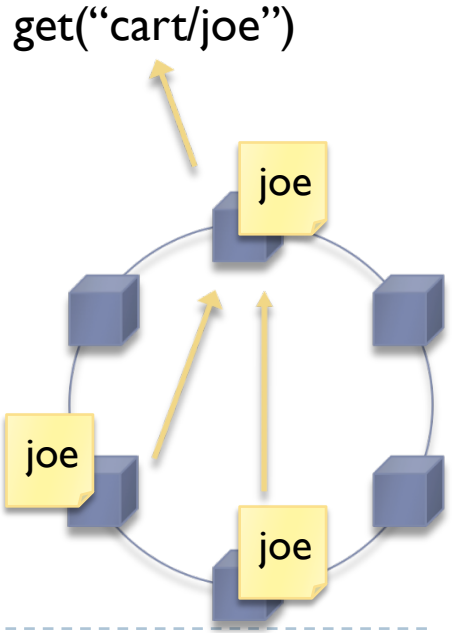
- Key, value store
  - Distributed hash table
- High scalability
  - No master, peer-to-peer
  - Large scale cluster, maybe O(1K)
- Fault tolerant
  - Even if an entire data center fails
  - Meets latency requirements in the case



Dynamo: Features, cont'd

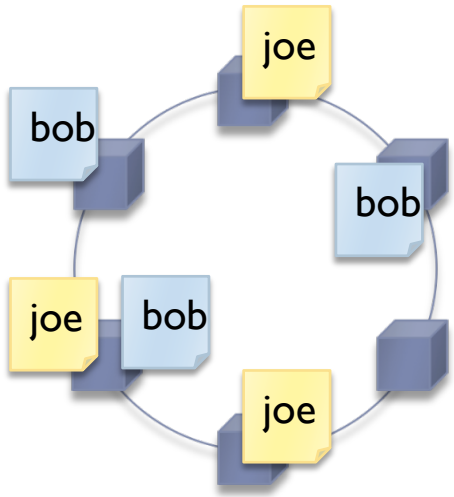
- Service Level Agreements
  - < 300ms for 99.9% queries
  - On the average, 15ms for reads and 30ms for writes
- High availability
  - No lock and always writable
- Eventually Consistent
  - Replicas are loosely synchronized
  - Inconsistencies are resolved by clients

Tradeoff between availability and consistency
- RDBMS chooses consistency
- Dynamo prefers availability



Dynamo: Overview

- Dynamo cluster (instance)
  - Consists of equivalent nodes
  - Has *N* replicas for each key



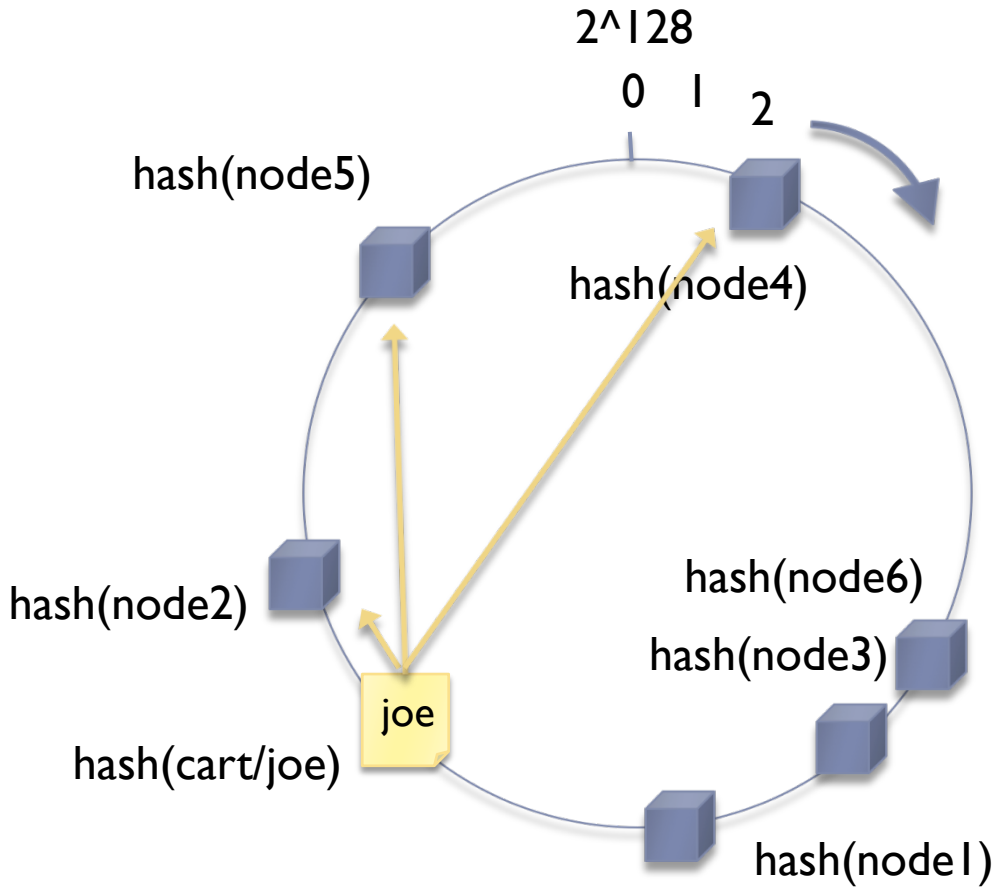
Dynamo of N = 3

- Dynamo APIs
  - get(key)
  - put(key, value, context)
    - Context is a kind of version, like *cas* of memcache
  - *delete* is not described in the paper (I guess defined)

- Requirements for clients
  - Don't need to know *ALL* nodes, unlike memcache clients
    - Requests can be sent to any node

Dynamo: Partitioning

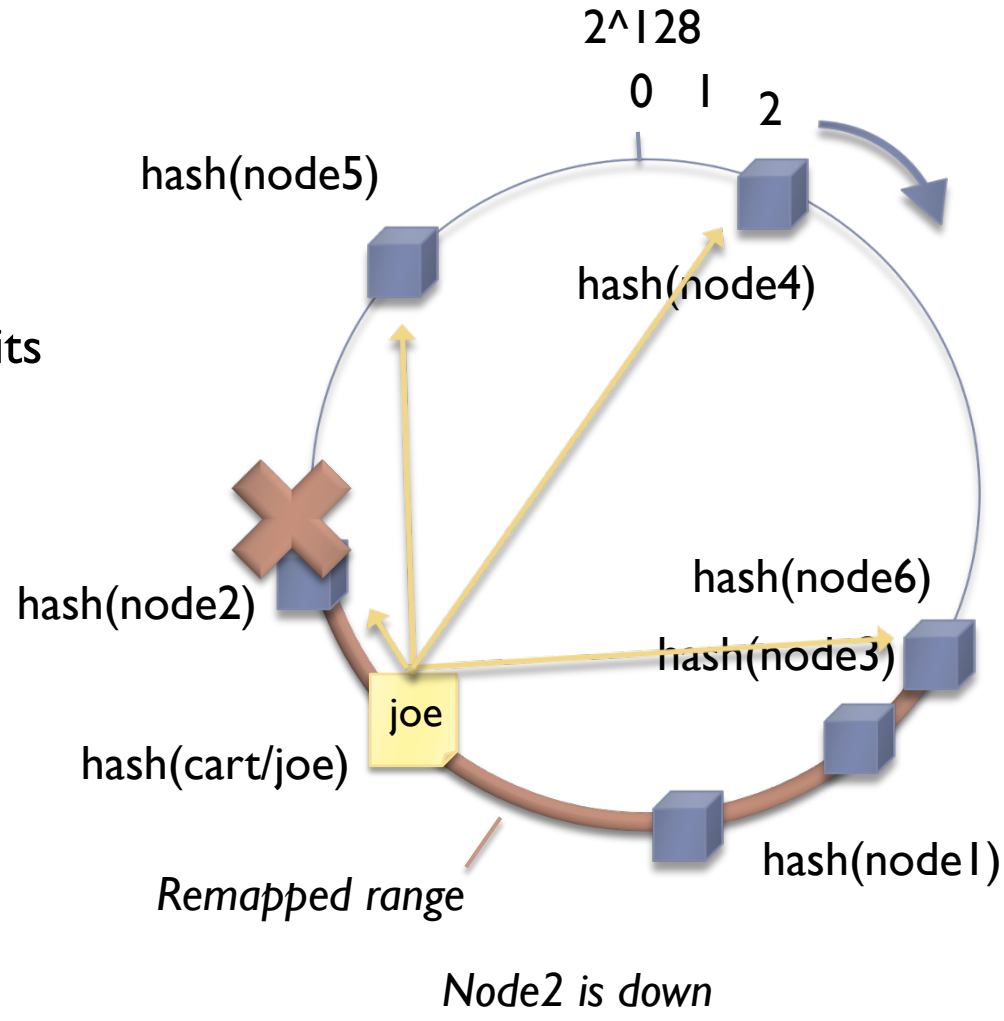
- Consistent Hashing
  - Nodes and keys are positioned at their hash values
    - MD5 (128bits)
  - Keys are stored in the following *N* nodes



Hash ring (N=3)

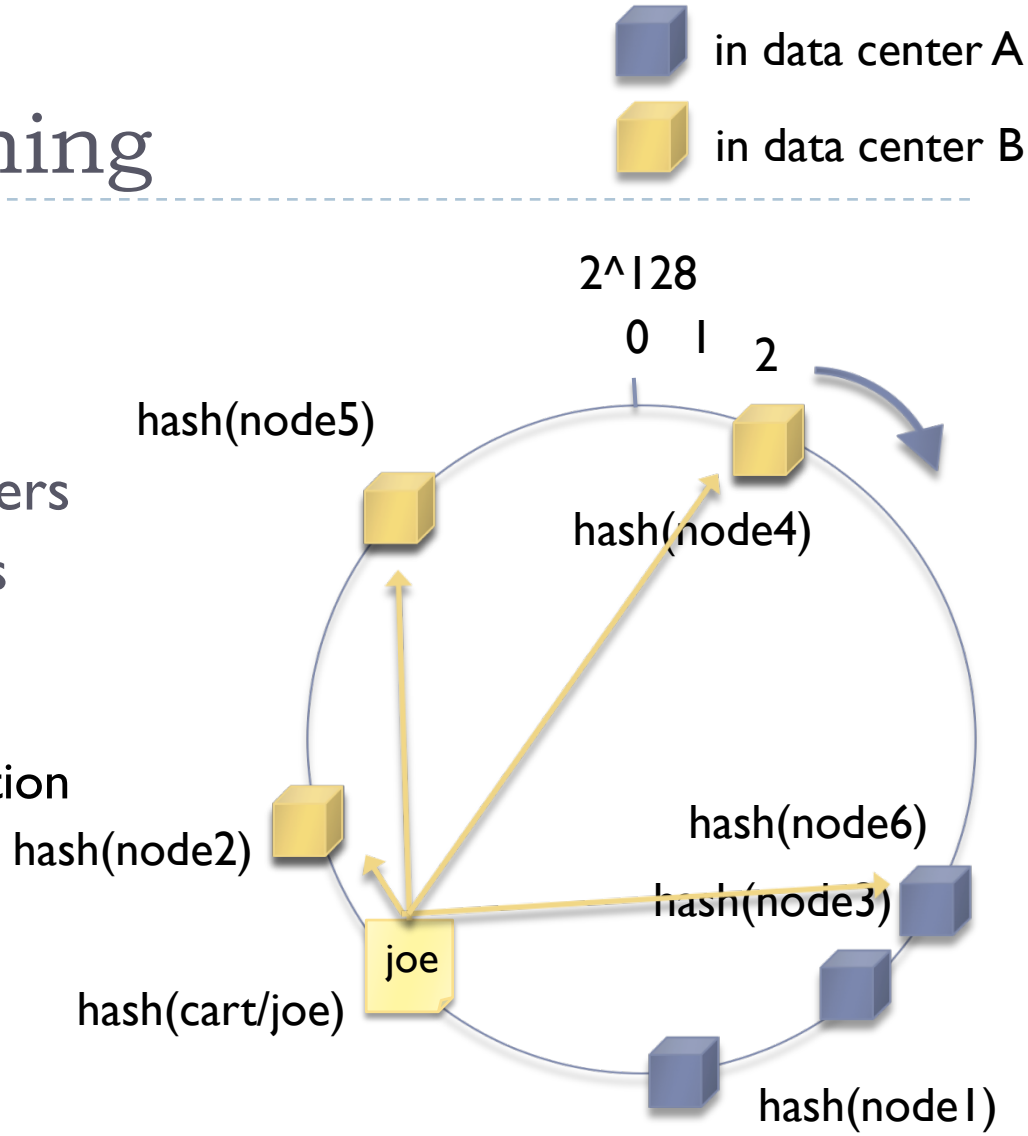
Dynamo: Partitioning, cont'd

- Advantages
  - Small # of keys are remapped, when membership is changed
    - Between down node and its *N*th predecessor



Dynamo: Partitioning

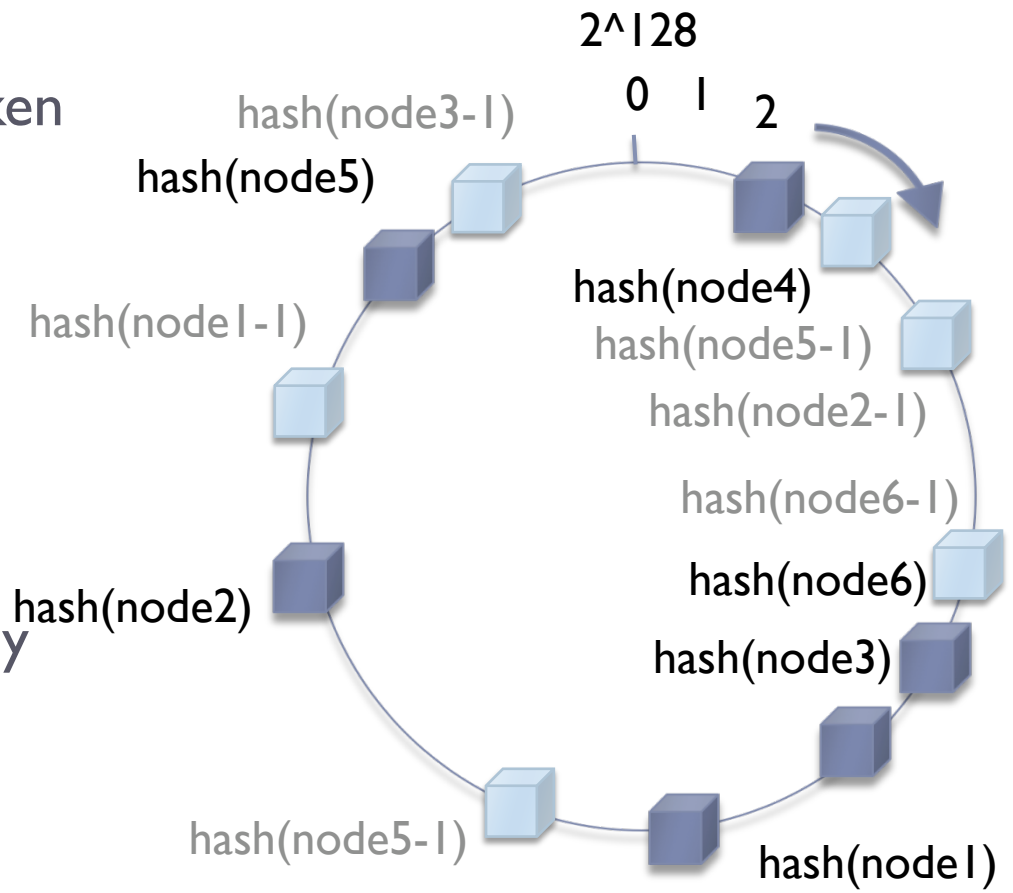
- Physical placement of replicas
  - Each key is replicated across multiple data centers
  - Arrangement scheme has not been revealed
    - Netmask is helpful, I guess
    - Impact on replica distribution is unknown
- Advantages
  - No data outage on data center failures



joe is replicated in multiple data centers

Dynamo: Partitioning, cont'd

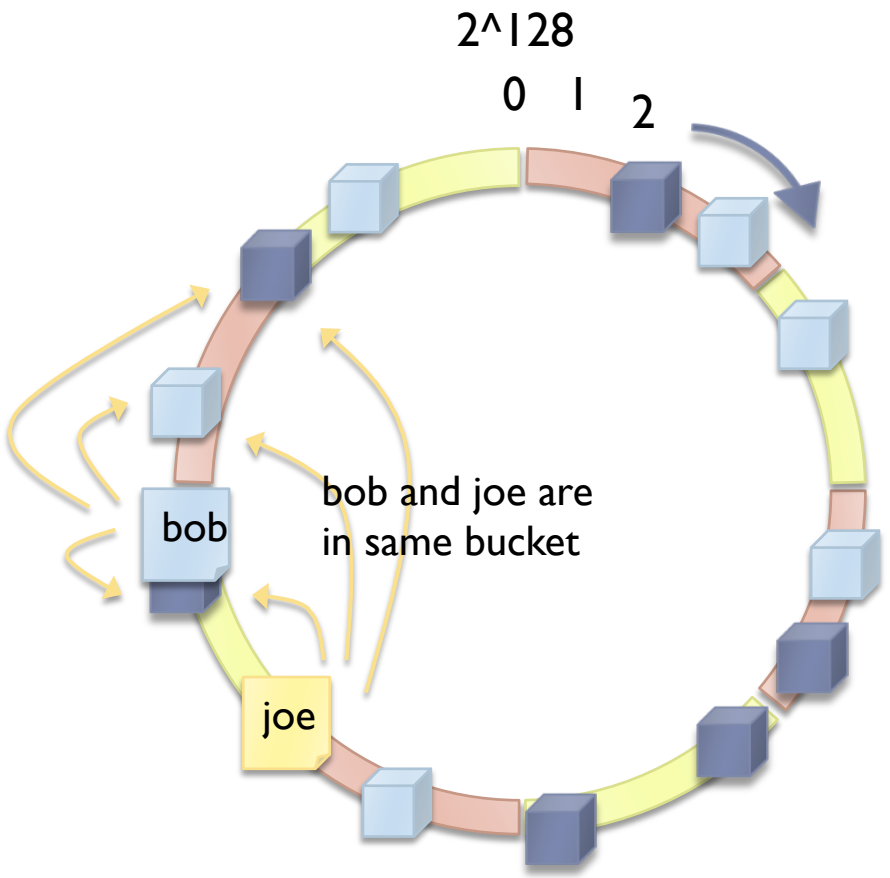
- Virtual nodes
  - Multiple positions are taken by a single physical node
    - O(100) virtual/physical
- Advantages
  - Keys are more uniformly distributed statistically
  - Remapped keys are evenly dispersed across nodes
  - # of virtual nodes can be determined based on capacity of physical node



Two virtual nodes per each physical node

Dynamo: Partitioning, cont'd

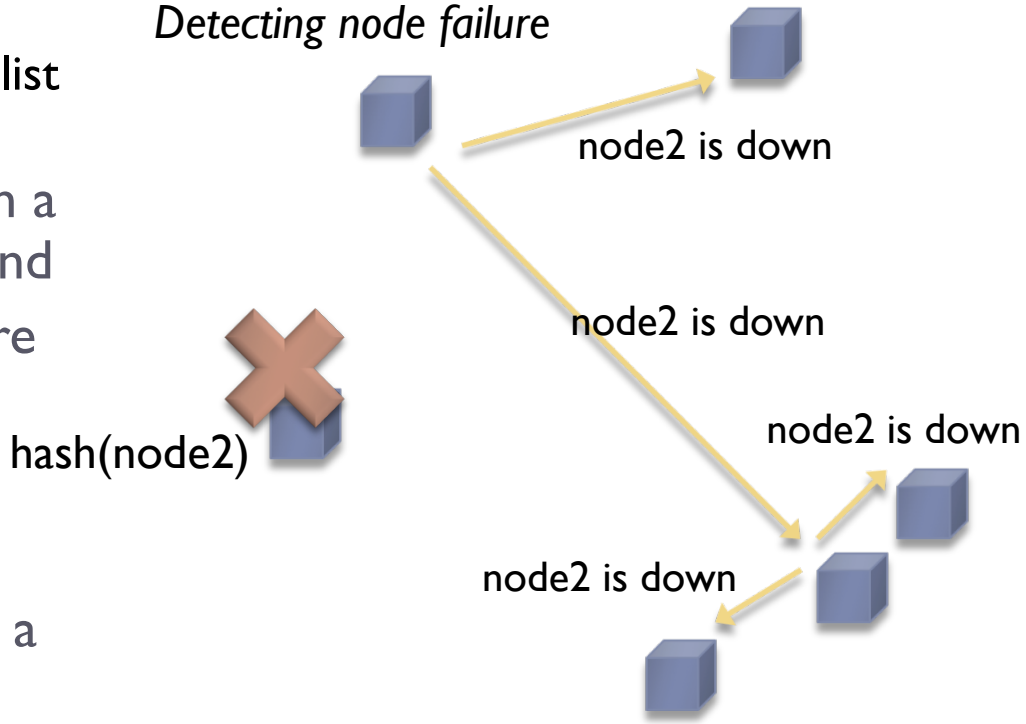
- Buckets
  - Hash ring is equally divided into buckets
    - There should be more buckets than all of virtual nodes
  - Keys in same bucket are mapped to same nodes
- Advantages
  - Keys are easily synchronized bucket by bucket
    - For Merkle trees discussed later



Divided into 8 buckets

Dynamo: Membership

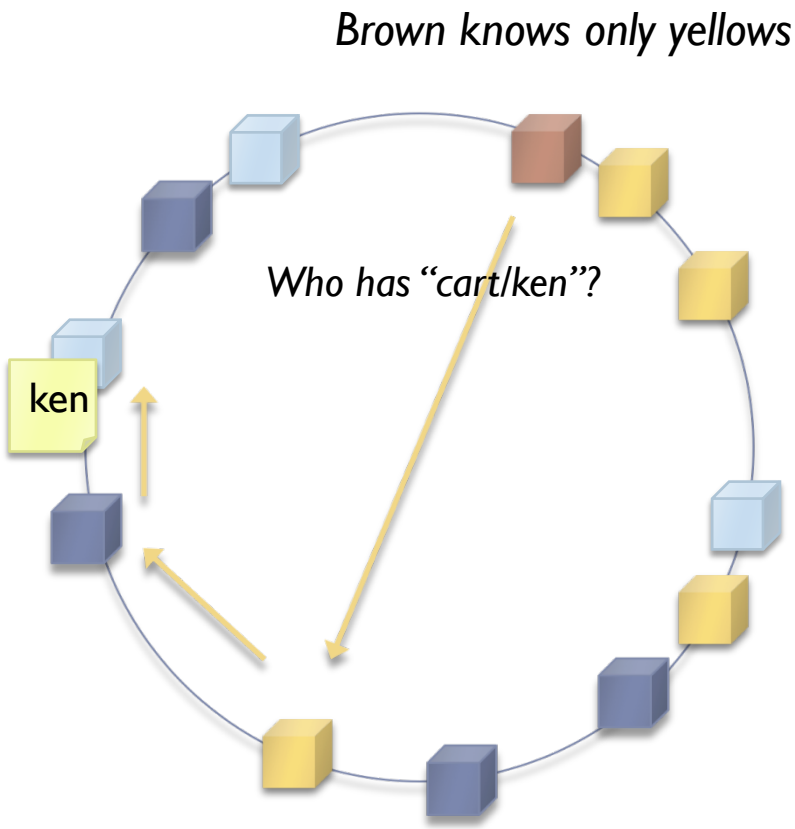
- Gossip-based protocol
  - Spreads membership like a rumor
    - Membership contains node list and change history
  - Exchanges membership with a node at random every second
  - Updates membership if more recent one received
- Advantages
  - Robust; no one can prevent a rumor from spreading
  - Exponentially rapid spread



Membership change is spread by the form of gossip

Dynamo: Membership, cont'd

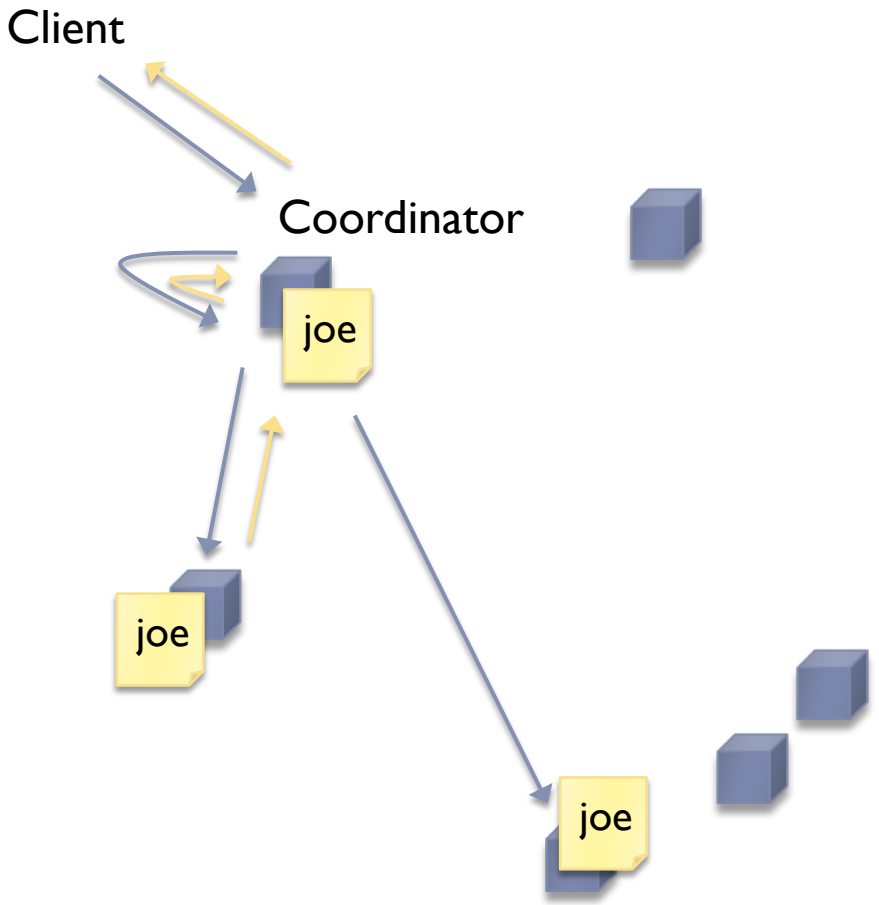
- Chord
  - For large scale cluster
    - > O(10K) virtual nodes
  - Each node needs to know only O(log *v*) nodes
    - *v* is # of virtual nodes
    - The nearer on hash ring, the more to know
  - Messages are routed hop by hop
    - Hop count is < O(log *v*)
  - For details, see original paper



Routing in Chord

Dynamo: *get*/*put* Operations

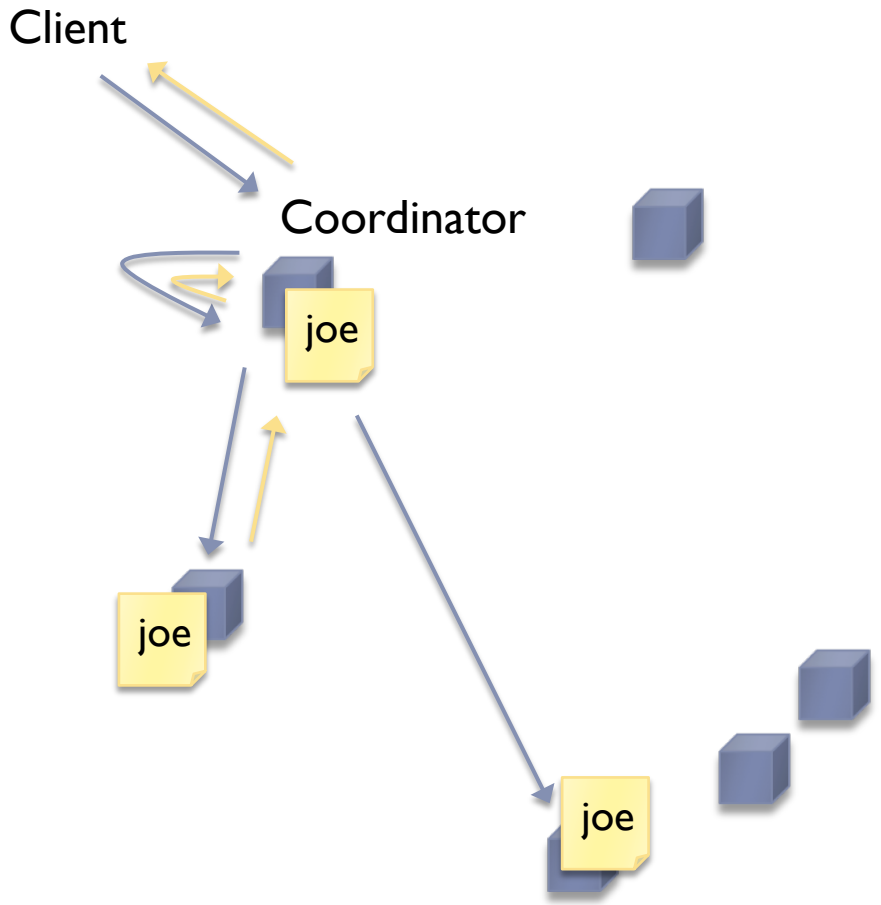
- Client
  1. Sends a request any of Dynamo node
  - The request is forwarded to coordinator
    - Coordinator: one of nodes associated with the key
- Coordinator
  1. Chooses *N* nodes by using consistent hashing
  2. Forwards a request to *N* nodes
  3. Waits responses from *R* or *W* nodes, or timeouts
  4. Checks replica versions if *get*
  5. Sends a response to client



get/put operations for N,R,W = 3,2,2

Dynamo: *get/put* Operations, Cont'd

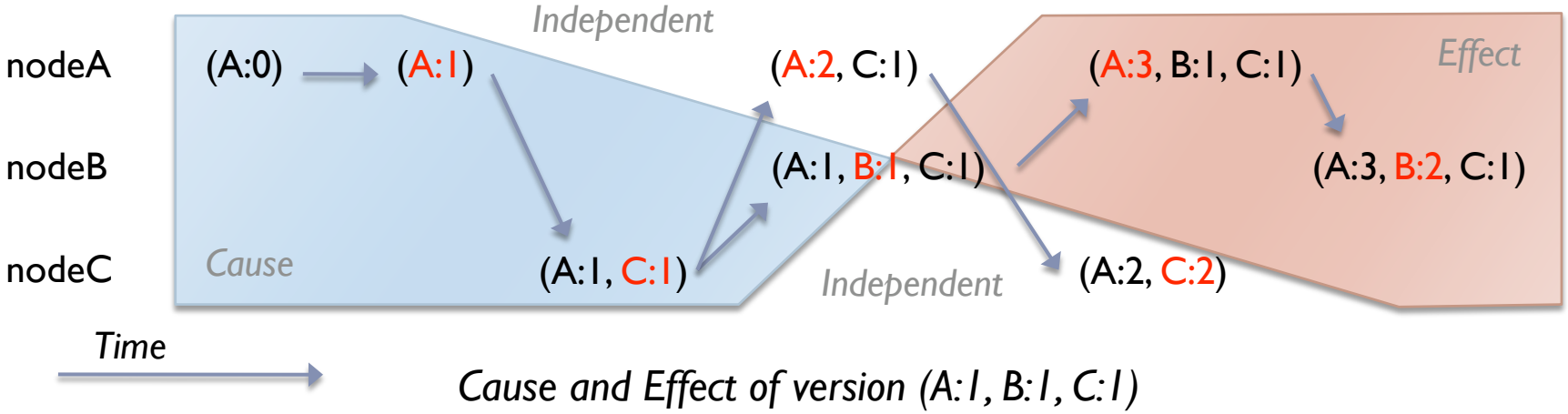
- Quorum
  - Parameters
    - *N*: # of replicas
    - *R*: min # of successful reads
    - *W*: min # of successful writes
  - Conditions
    - *R+W > N*
      - Key can be read from *at least* one node which has successfully written it
    - *R < N*
      - Provides better latency
    - *N,R,W* = 3,2,2 in common



get/put operations for N,R,W = 3,2,2

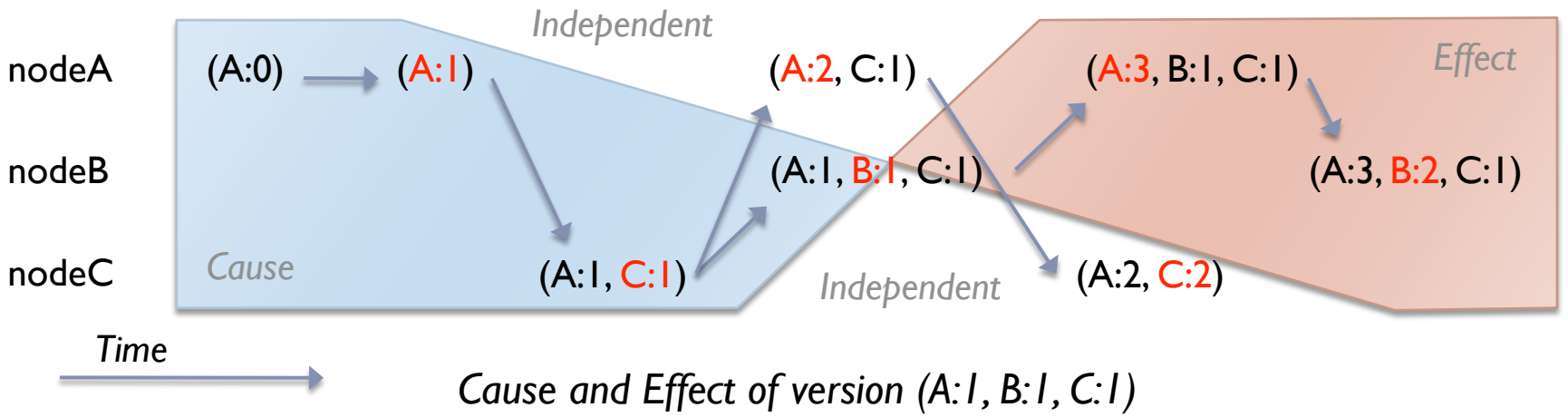
Dynamo: Versioning

- Vector Clocks
  - Ordering of versions in distributed manner
    - No master, no clock synchronization
    - Parallel branches can be detected
  - List of clocks (counters) associated with each node
    - Each clock is managed by coordinator
    - (A:2, C:1) means that updated when A's clock was 2 and C's was 1



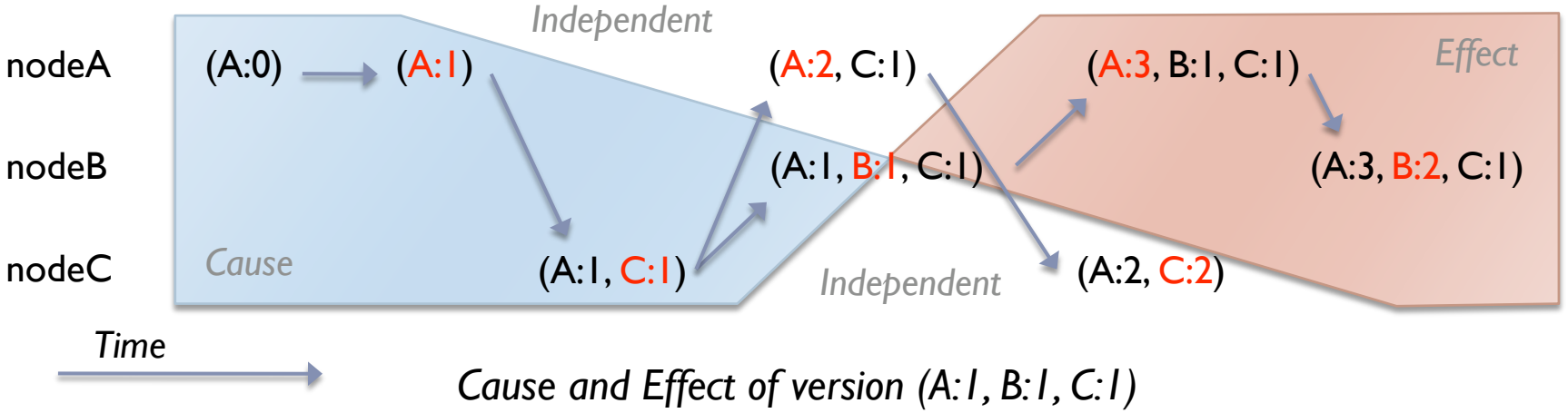
Dynamo: Versioning, cont'd

- Vector Clocks
  - When a coordinator updates a key, …
    - Increments its own clock
    - Replaces clock in vector of the key



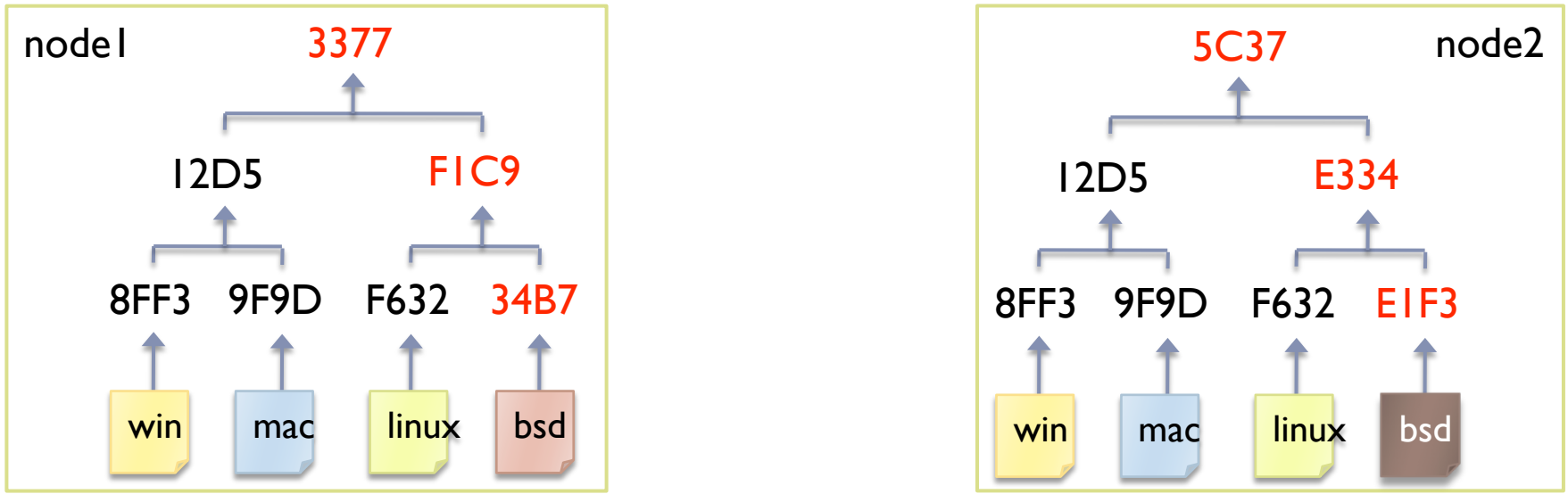
Dynamo: Versioning, cont'd

- Vector Clocks
  - How to determine ordering of versions?
    - If all clocks are equal to or greater than those of the other
      - The one is more recently updated
        - e.g. (A:1, B:1, C:1) < (A:3, B:1, C:1)
    - Otherwise
      - They are parallel branches and cannot be ordered
        - e.g. (A:1, B:1, C:1) ? (A:2, C:1)



Dynamo: Synchronization

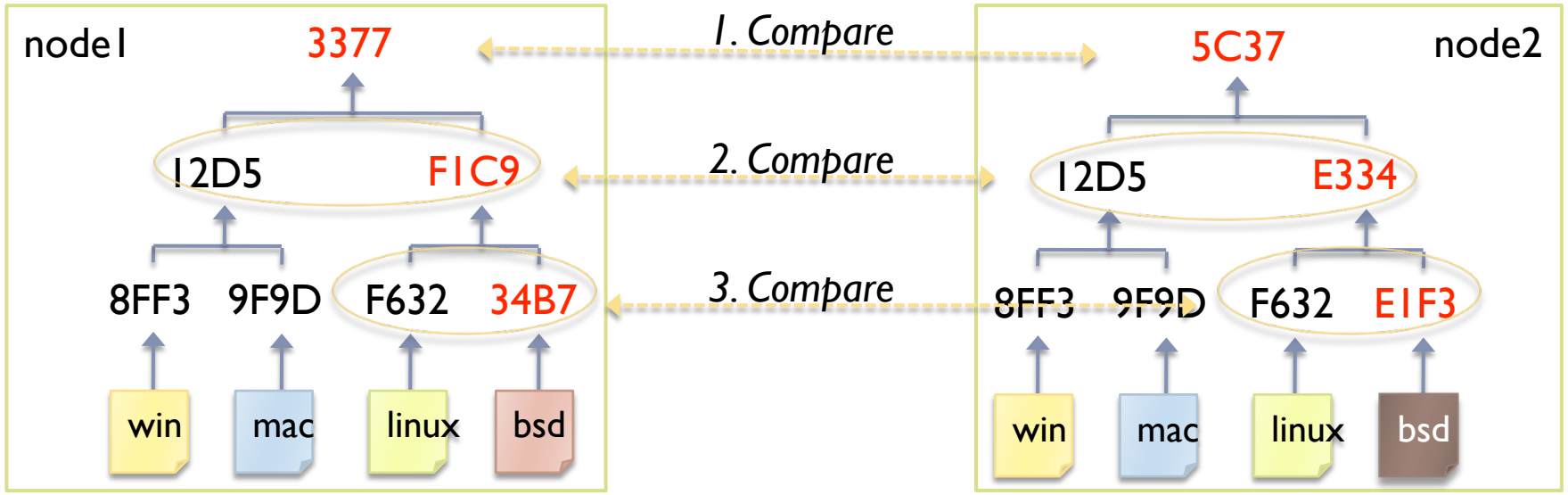
- Sync replicas with Merkle tree
  - Hierarchical checksums (Merkle tree) are kept to be calculated
    - Trees are constructed for each bucket
  - Synchronization is executed periodically or when membership changes



Comparison of hierarchical checksums in Merkle trees

Dynamo: Synchronization, cont'd

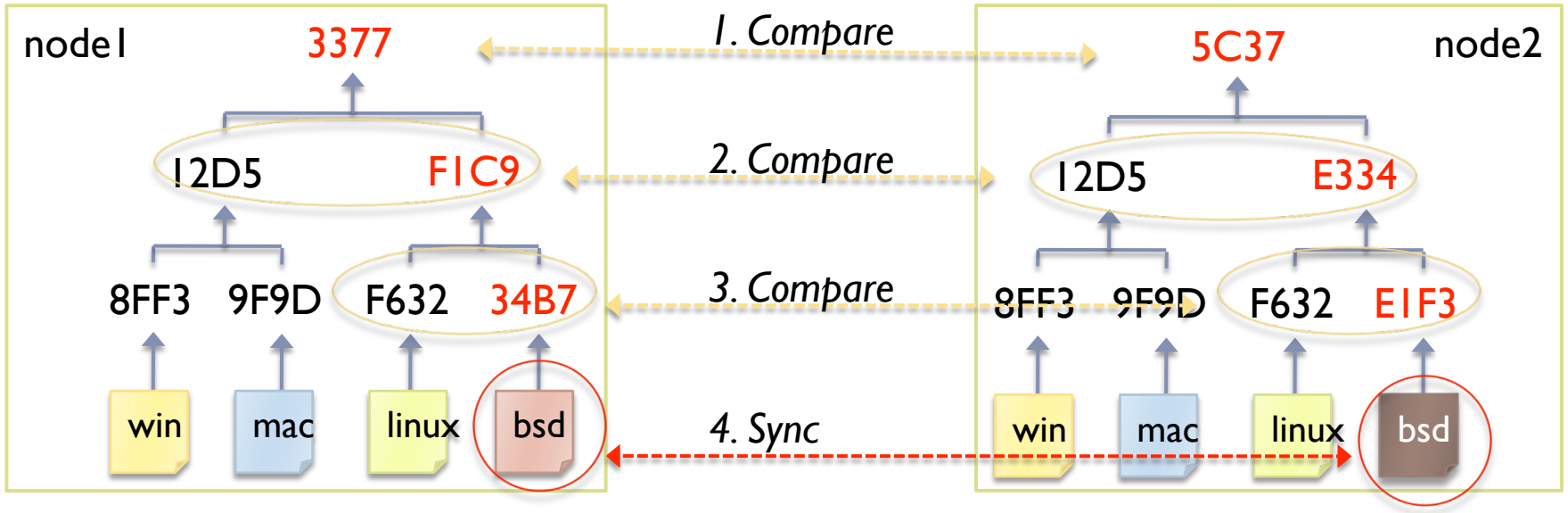
- Sync replicas with Merkle tree
  - Compares Merkle tree with other nodes
    - From root to leaf, until checksum corresponds with each other



Comparison of hierarchical checksums in Merkle trees

Dynamo: Synchronization, cont'd

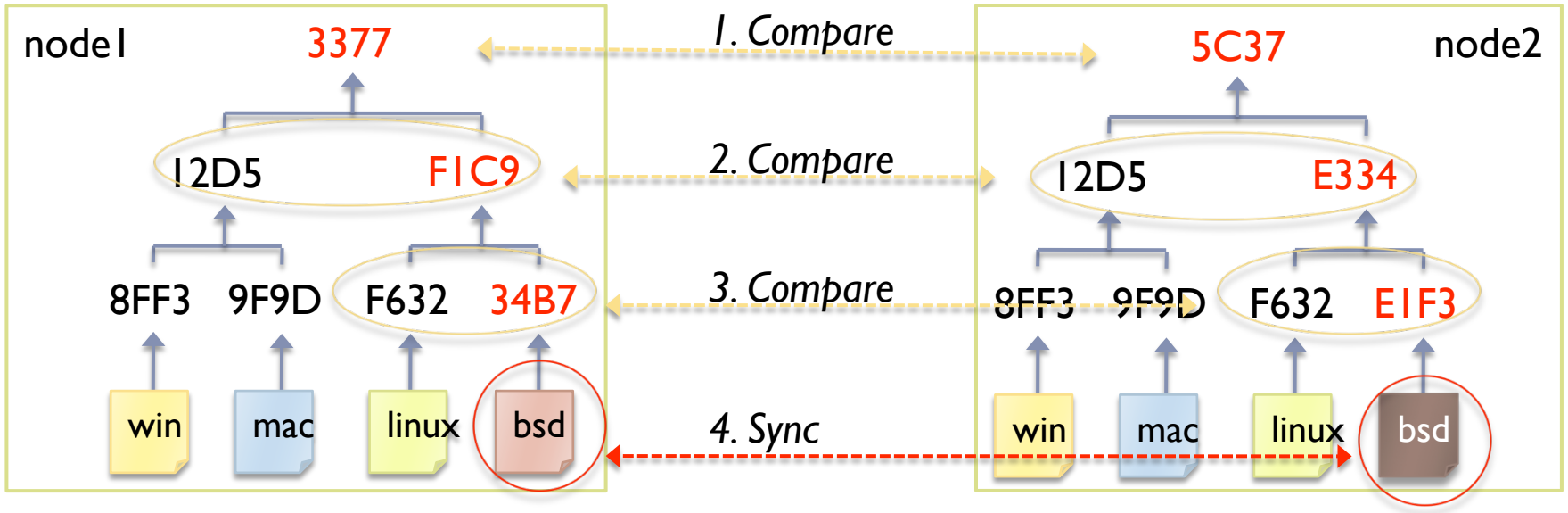
- Sync replicas with Merkle tree
  - Synchronize out-of-date replicas
    - In background as low priority task
    - If versions cannot be ordered, no sync will be done



Comparison of hierarchical checksums in Merkle trees

Dynamo: Synchronization, cont'd

- Advantages of Merkle tree
  - Comparisons can be reduced, if most of replicas are synchronized
    - Root checksums are equal, and no more comparison is required



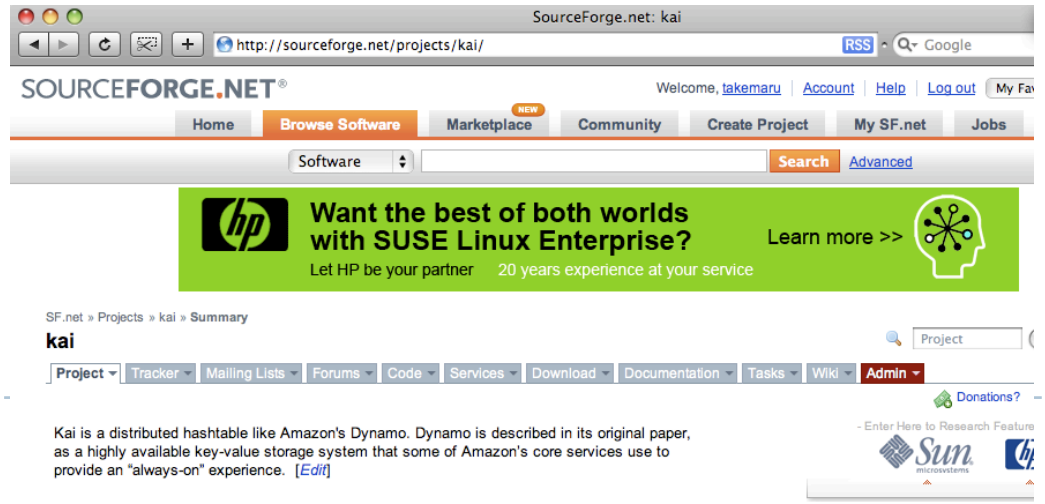
Comparison of hierarchical checksums in Merkle trees

Dynamo: Implementation

- Implementation
  - Written in Java
  - Closed source ☹
- APIs
  - Over HTTP
- Storage
  - BDB or MySQL
- Security
  - No requirements

Kai: Overview

- Kai
  - Open source implementation of Amazon's Dynamo
    - Named after my origin
    - *OpenDynamo* had been taken by a project not related to Amazon's Dynamo ☹
  - Written in Erlang
  - memcache API
  - Found at http://sourceforge.net/projects/kai/



Kai: Building Kai

- Requirements
  - Erlang OTP (>= R12B)
  - make
- Build

```
% svn co http://kai.svn.sourceforge.net/svnroot/kai/trunk kai
% cd kai/
% make
% make test
```

  - Edits RUN_TEST in *Makefile* according to your enbironment before *make test*, if not MacOSX
    - *./configure* will be coming

Kai: Configuration

- *kai.config*
  - All parameters are optional since having default values

Parameter	Description	Default value
logfile	Name of log file	Standard output
hostname	Hostname, which should be specified if the computer has multiple network interfaces	Auto detection
port	Port # of internal API	11011
memcache_port	Port # of memcache API	11211
n, r, w	N, R, W for quorum	3, 2, 2
number_of_buckets	# of buckets, which must correspond with other nodes	1024
number_of_virtual_nodes	# of virtual nodes	128

Kai: Running Kai

- Run as a stand alone server

```
% erl -pa src -config kai -kai n 1 -kai r 1 -kai w 1

1> application:load(kai).
2> application:start(kai).
```

  - Arguments

Arguments	Description
-pa src	Adds directory *src* to load path
-config kai	Loads configuration file *kai.config*
-kai n1 -kai r1 -kai w1	Overwrites configuration as N, R, W = 1, 1, 1

  - Access to 127.0.0.1:11211 by your memcache client

Kai: Running Kai, cont'd

- Run as a cluster system
  - Start three nodes on different port #'s in a single computer

Terminal 1

```
% erl -pa src -config kai -kai port 11011 -kai memcache_port 11211

1> application:load(kai).
2> application:start(kai).
```

Terminal 2

```
% erl -pa src -config kai -kai port 11012 -kai memcache_port 11212

1> application:load(kai).
2> application:start(kai).
```

Terminal 3

```
% erl -pa src -config kai -kai port 11013 -kai memcache_port 11213

1> application:load(kai).
2> application:start(kai).
```

Kai: Running Kai, cont'd

- Run as a cluster system
  - Connect all nodes by informing of their neighbors

```
3> kai_api:check_node({{127,0,0,1}, 11011}, {{127,0,0,1}, 11012}).
4> kai_api:check_node({{127,0,0,1}, 11012}, {{127,0,0,1}, 11013}).
5> kai_api:check_node({{127,0,0,1}, 11013}, {{127,0,0,1}, 11011}).
```

  - Access to 127.0.0.1:11211-11213 by your memcache client

  - Adding a new node to existed cluster

```
% Link a new node to the existed cluster
% (kai_api:check_node/2 establishes one-directional link)
1> kai_api:check_node(NewNode, NodeInCluster).

% Wait till synchronizing buckets…

% Link a node in the cluster to the new node
2> kai_api:check_node(NodeInCluster, NewNode).
```

Kai: Internals

Function	Module	Comments
Partitioning	*kai_hash*	• Not considering physical placement
Membership	*kai_network*	• Not including Chord • Will be renamed to *kai_membership*
Coordinator	*kai_memcache*	• Will be re-implemented in *kai_coordinator*
Versioning		• Not implemented yet
Synchronization	*kai_sync*	• Not including Merkle tree
Storage	*kai_store*	• Using *ets*
Internal API	*kai_api*	
API	*kai_memcache*	• *get*, *set*, and *delete*
Logging	*kai_log*	
Configuration	*kai_config*	
Supervisor	*kai_sup*	

Kai: *kai_hash*

- Base module
  - *gen_server*
- Current status
  - Consistent hashing
    - 32bit hash space
    - Virtual nodes
    - Buckets
- Future work
  - Physical placement
  - Parallel access will be allowed for read operation
    - To avoid waiting long while re-hashing
    - No use *gen_server:call*

Synopsis

```
kai_hash:start_link(),

# Re-hashes when membership changes
{replaced_buckets, ListOfReplacedBuckets} =
    kai_hash:update_nodes(ListOfNodesToAdd, ListOfNodesToRemove),

# Finds nodes associated with Key to get/put
{nodes, ListOfNodes} = kai_hash:find_nodes(Key).
```

Kai: *kai_network*, will be renamed to *kai_membership*

- Base module
  - *gen_fsm*
- Current status
  - Built-in distributed facilities, such as EPMD, are not used, since they don't seem to be scalable
  - Gossip-based protocol
- Future work
  - Chord or Kademlia
    - Kademlia is used by BitTorrent

Synopsis

```
kai_network:start_link(),

# Checks whether Node is alive, and updates consistent hashing if needed
kai_network:check_node(Node),

# In background, kai_network checks a node at random every second
```

Kai: *kai_coordinator*, now implemented in *kai_memcache*

- Base module
  - *gen_server* (in plan)
- Current status
  - Implemented in *kai_memcache*
  - Quorum
- Future work
  - Separated from *kai_memcache*
  - Requests from clients will be routed to coordinators
    - Currently, a node receiving requests behaves like a coordinator

Synopsis (in plan)

```
kai_coordinator:start_link(),

% Sends a get request to N nodes, and returns received data to kai_memcache
Data = kai_coordinator:get(Key),

% Sends a put request to N nodes, where Data is a variable of data record
kai_coordinator:put(Data).
```

Kai: *kai_version*

- Base module
  - *gen_server* (in plan)
- Current Status
  - Not implemented yet
- Future work
  - Vector Clocks

Synopsis (in plan)

```
kai_version:start_link(),

% Updates clock of LocalNode in VectorClocks
kai_version:update(VectorClocks, LocalNode),

% Generates ordering of vector clocks
{order, Order} = kai_version:order(VectorClocks1, VectorClocks2).
```

Kai: *kai_sync*

- Base module
  - *gen_fsm*
- Current status
  - Sync data if not exists
    - Versions are not compared
- Future work
  - Bulk transport
    - Downloads whole bucket when membership changes
  - Parallel download
    - Synchronizes a bucket with multiple nodes
  - Merkle tree

Synopsis

```
kai_sync:start_link(),

# Compares key list in Bucket with other nodes,
# and retrieves those which don't exist
kai_sync:update_bucket(Bucket),

# In background, kai_sync synchronizes a bucket at random every second
```

Kai: *kai_store*

- Base module
  - *gen_server*
- Current status
  - ets, which is Erlang built-in memory storage, is used
- Future work
  - Persistent storage without capacity limit
    - dets, mnesia, or MySQL
    - Unfortunately, tables in dets and mnesia < 4GB
  - Delaying deletion

Synopsis

```
kai_store:start_link(),

# Retrieves Data associated with Key
Data = kai_store:get(Key),

% Stores Data, which is a variable of data record
kai_store:put(Data).
```

Kai: *kai_api*

- Base module
  - *gen_tcp*
- Current status
  - Internal API
    - RPC for *kai_hash*, *kai_store*, and *kai_network*
- Future work
  - Process pool
    - Restricts max # of processes to receive API calls
  - Connection pool
    - Re-uses TCP connections

Synopsis

```
kai_api:start_link(),

# Retrieves node list from Node
{node_list, ListOfNodes} = kai_api:node_list(Node),

# Retrieves Data associated with Key from Node
Data = kai_api:get(Node, Key).
```

Kai: *kai_memcache*

- Base module
  - *gen_tcp*
- Current status
  - *get*, *set*, and *delete* of memcache API are supported
    - *exptime* of *set* must be zero, since Kai is persistent storage
    - *get* returns multiple values if multiple versions are found
- Future work
  - *cas* and *stats*
  - Process pool
    - Restricts max # of processes to receive API calls

Synopsis in Ruby

```
require 'memcache'

cache = MemCache.new '127.0.0.1:11211'

# Stores 'value' with 'key'
cache['key'] = 'value'

# Retrieves data associated with 'key'
p cache['key']
```

Kai: Testing

- Execution
  - Done by *make test*
- Implementation
  - *common_test*
    - Built-in testing framework
  - Test servers
    - Written from scratch currently
    - Can *test_server* be used?

Kai: Miscellaneous

- Node ID

```
# Nodes are identified by socket addresses of internal API
{Addr, Port} = {{192,168,1,1}, 11011}.
```

- Data structure

```
# This data structure includes value as well as metadata
-record(data,     {key, bucket, last_modified, checksum, flags, value}).

# This includes only metadata, such as key, bucket #, and MD5 checksum
-record(metadata, {key, bucket, last_modified, checksum}).
```

- Return values

```
# Returns 'undefined' if data associated with Key is not found
undefined = kai_store:get(Key).

# Returns Reason with 'error' tag when something wrong happens
{error, Reason} = function(Args).
```

Kai: Roadmap

1. Basic implementation
   - Current version

2. Almost Dynamo

Module	Task
kai_hash	Parallel access will be allowed for read operation
kai_coordinator	Requests from clients will be routed to coordinators
kai_version	Vector clocks
kai_sync	Bulk and parallel transport
kai_store	Persistent storage
kai_api	Process pool
kai_memcache	Process pool and cas

Kai: Roadmap, cont'd

3. Dynamo

Module	Task
kai_hash	Physical Placement
kai_membership	Chord or Kademlia
kai_sync	Merkel tree
kai_store	Delaying deletion
kai_api	Connection pool
kai_memcache	stats

- On development environments
  - *configure, test_server*

Conclusion

- Join us if interested in!

http://sourceforge.net/projects/kai/