Dynamo: Amazon’s Highly Available Key-value Store

Tallat M. Shafaat – tallat(@)kth.se
KTH – Royal Institute of Technology

Dynamo

- A distributed storage system
- Reliability and fault tolerance at massive scale
- Availability providing an ”always-on” experience
- Cost-effectiveness
- Performance

Context

- A distributed storage system for Amazon's e-commerce platform
  - Scale
    - Shopping cart served tens of millions requests; over 3 million checkouts in a single day
  - Highly available
    - Unavailability == $$$
  - Simple
    - Key-values / No complex queries
  - Managed system
    - Manually add/remove nodes
    - No security requirements
    - No byzantine nodes
  - Guarantee service level agreements

CAP Theorem

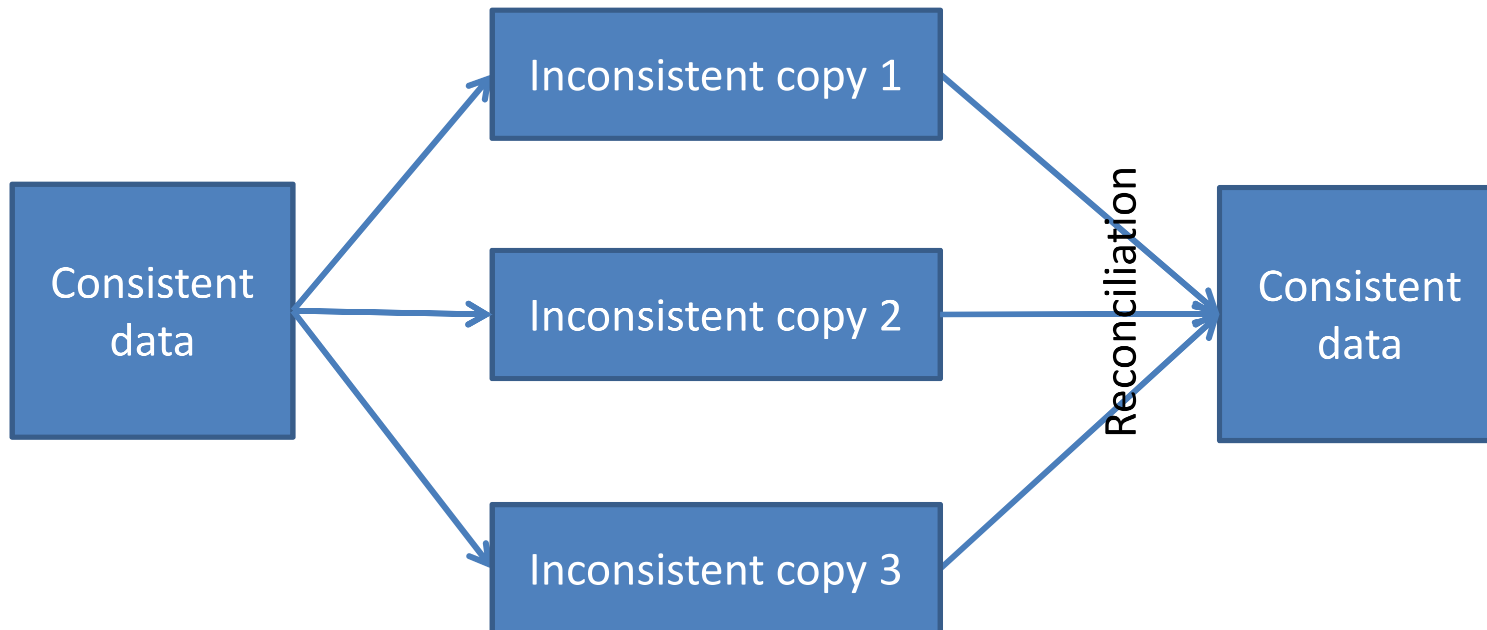
- Only two possible at the same time
  - Consistency
  - Availability
  - Partition tolerance
- Dynamo's target applications:
  - Availability and Parition-tolerance
  - Eventual consistency

Clients view on Consistency

- Strong consistency.
  - Single storage image. Informally, after an update completes any subsequent access will return the updated value.
- Weak consistency.
  - The system does not guarantee that subsequent accesses will return the updated value.
  - Inconsistency window.
- Eventual consistency.
  - Form of weak consistency
  - If no new updates are made to the object, eventually all accesses will return the last updated value.

Eventual consistency

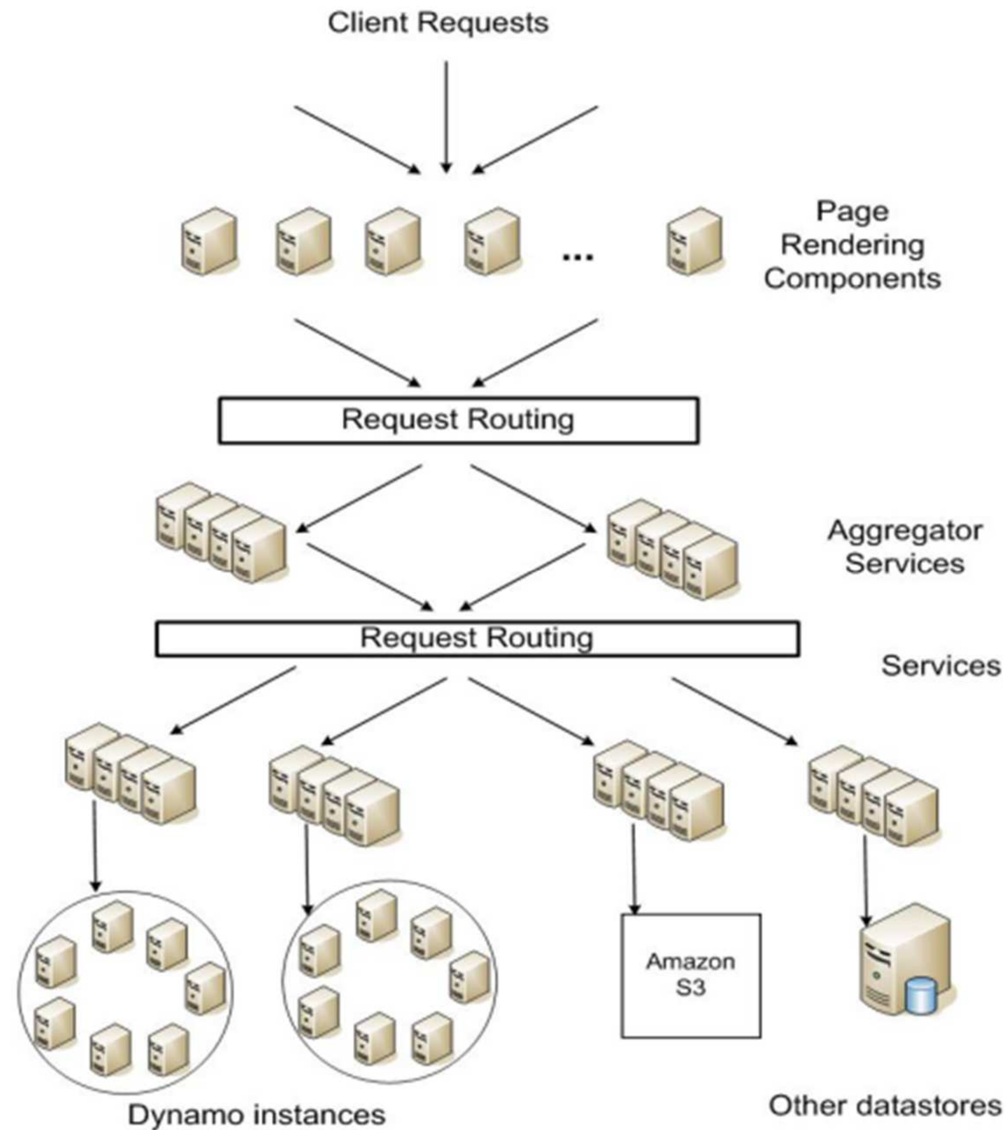
- Causal consistency
- Read-your-writes consistency, etc.
- Dynamo uses object versioning



Assumptions/Requirements

- Query model
  - Simple read/write operations on small data items
- ACID properties
  - Weaker consistency model
  - No isolation, only single key updates
- Efficiency
  - Tradeoff between performance, cost efficiency, availability and durability guarantees

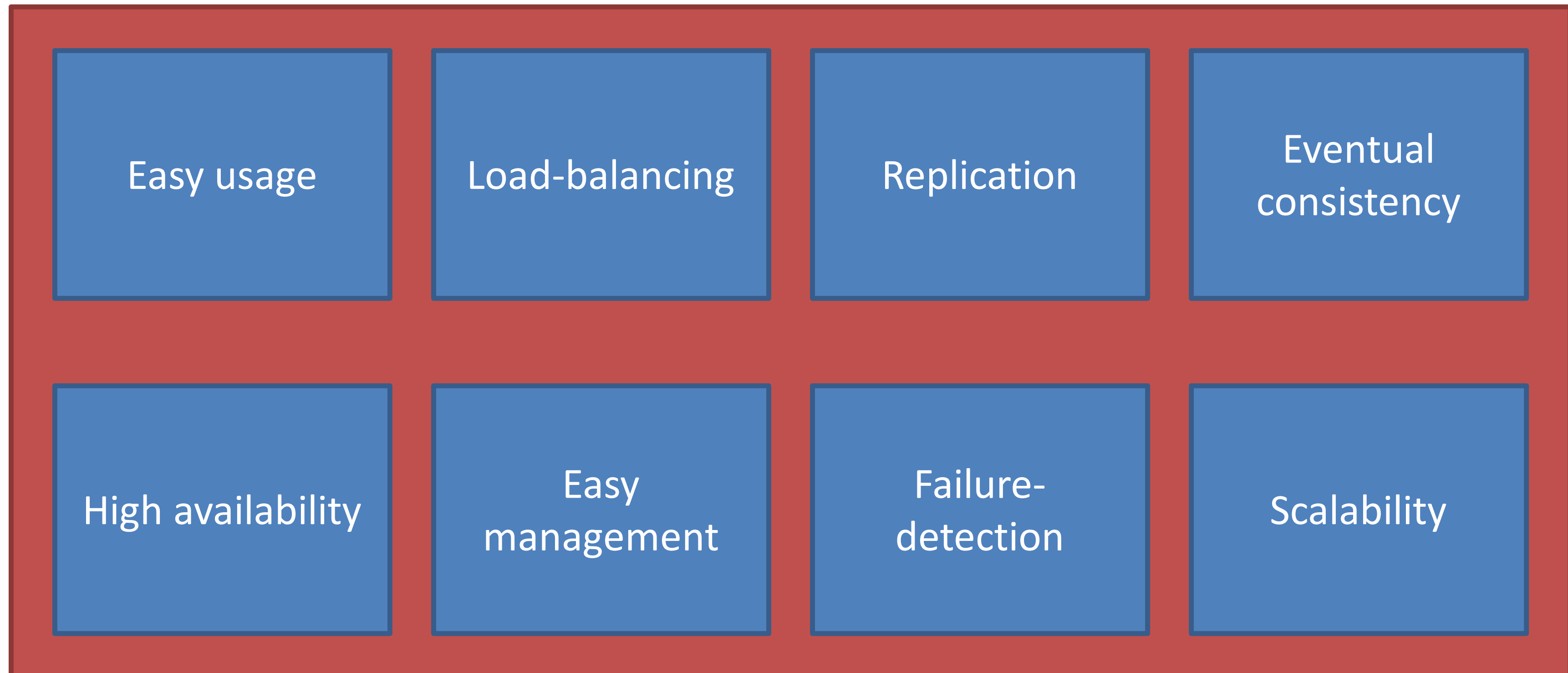
Architecture



Design considerations

- Sacrifice strong consistency for availability
- Conflict resolution
  - When
    - During read instead of write
  - Who
    - Client vs. Data store
- Incremental scalability
  - Scale by adding one machine at a time
- Symmetry
  - All nodes have the same role
- Decentralization
  - No central point
- Heterogeneity
  - Different machine configurations (CPU, memory, etc.)

The big picture

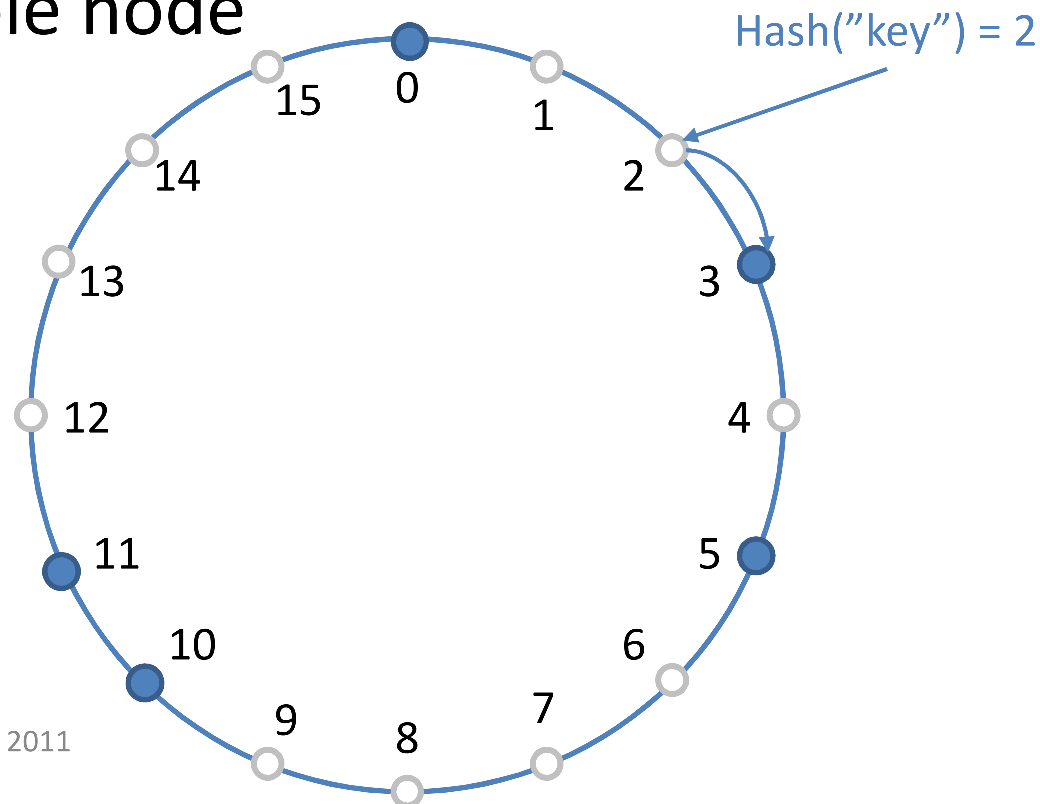


Easy usage: Interface

- get(key)
  - return single object or list of objects with conflicting versions and context
- put(key, context, object)
  - store object and context under key
- Context encodes system meta-data, e.g. version number

Data partitioning

- Use consistent hashing
- Nodes are assigned uniform random identifiers
- To store key-value items, hash key and put on responsible node



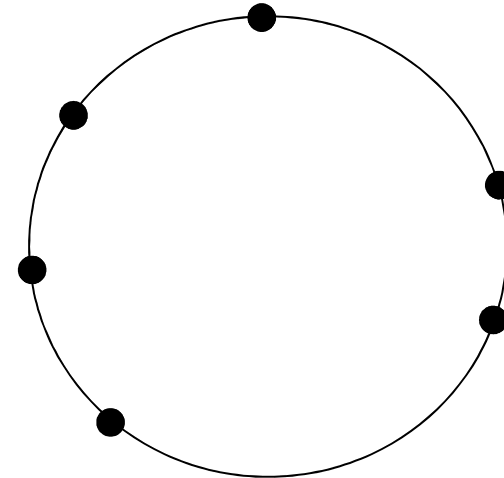
Load balancing

- Consistent hashing may lead to load imbalance

- Load
  - Storage bits
  - Popularity of the item
  - Processing required to serve the item
  - …

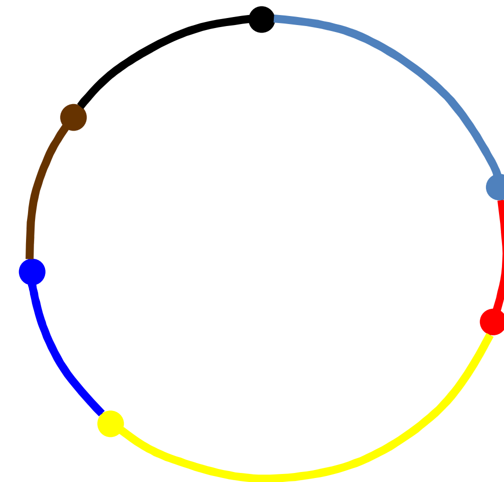
Load imbalance (1/4)

- Node identifiers may not be balanced



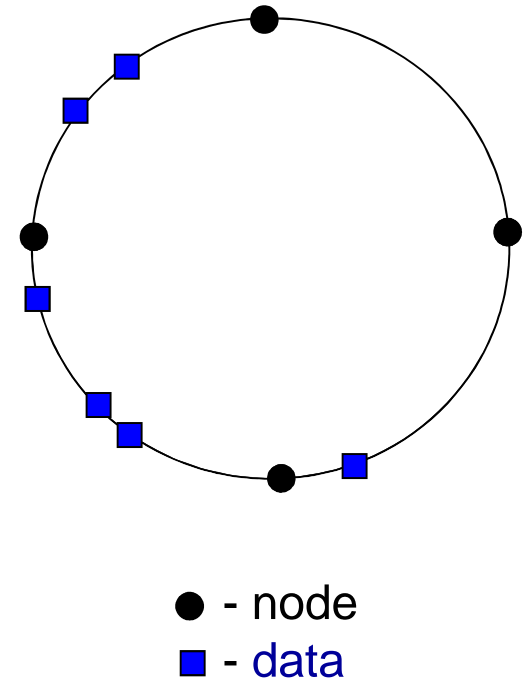
Load imbalance (1/4)

- Node identifiers may not be balanced



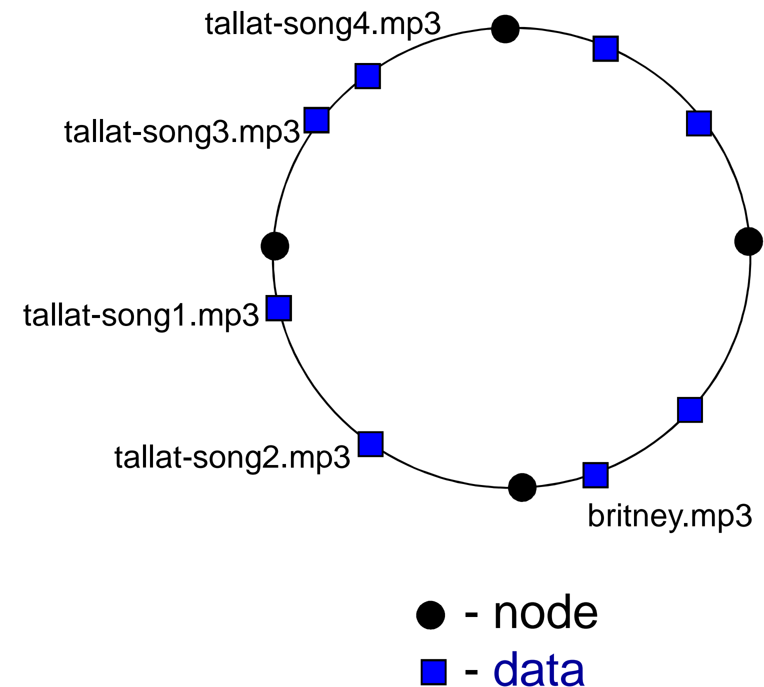
Load imbalance (2/4)

- Node identifiers may not be balanced
- Data identifiers may not be balanced



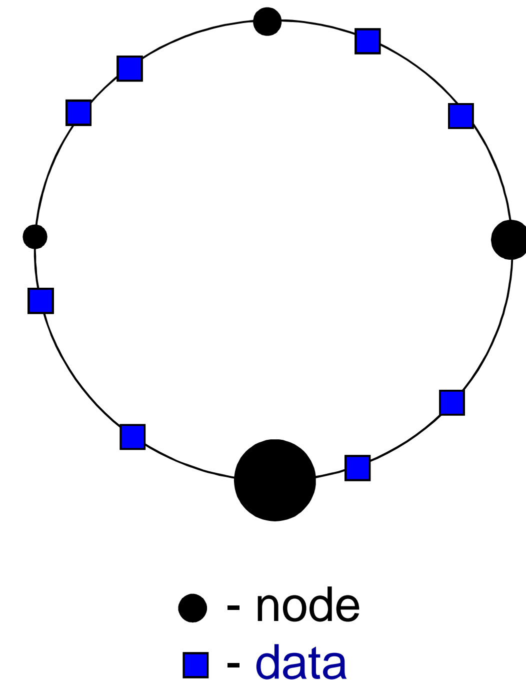
Load imbalance (3/4)

- Node identifiers may not be balanced
- Data identifiers may not be balanced
- Hot spots



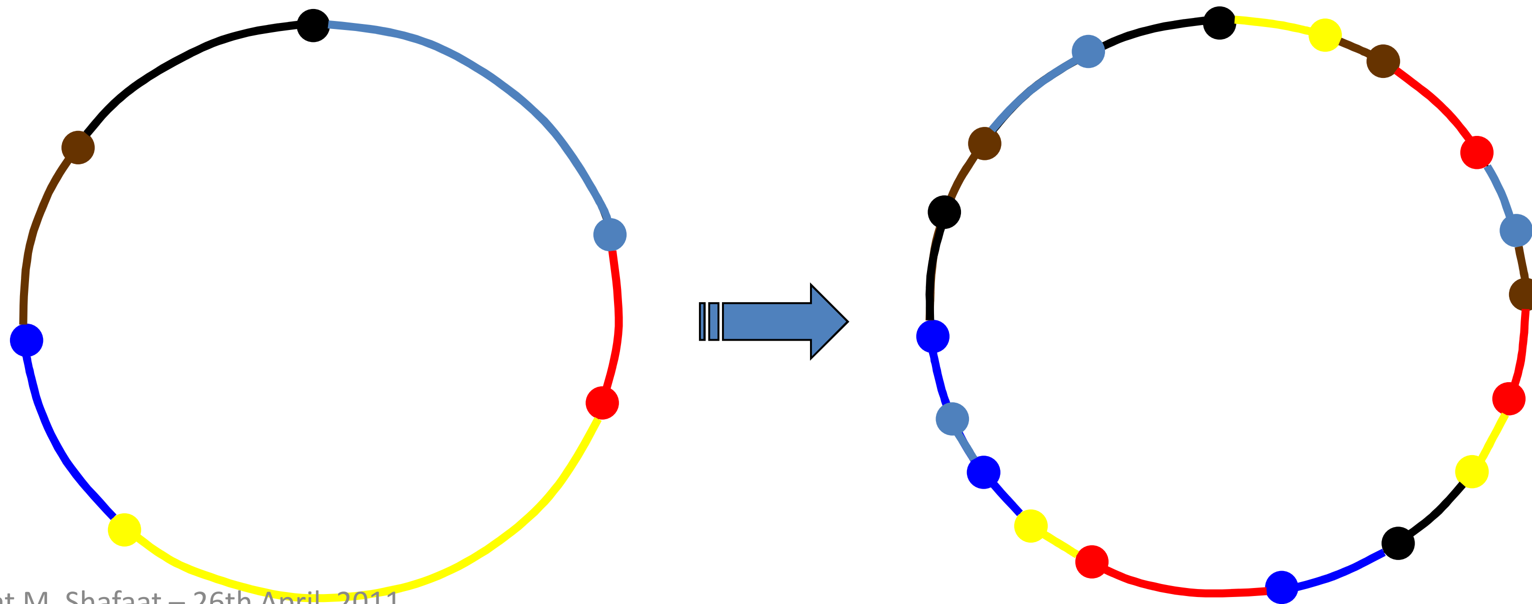
Load imbalance (4/4)

- Node identifiers may not be balanced
- Data identifiers may not be balanced
- Hot spots
- Heterogeneous nodes



Load balancing via Virtual Servers

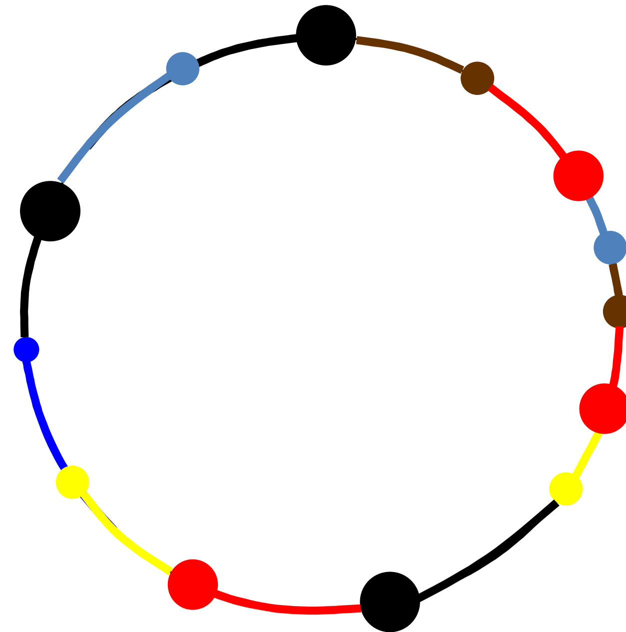
- Each physical node picks multiple random identifiers
  - Each identifier represents a virtual server
  - Each node runs multiple virtual servers
- Each node responsible for noncontiguous regions



Virtual Servers

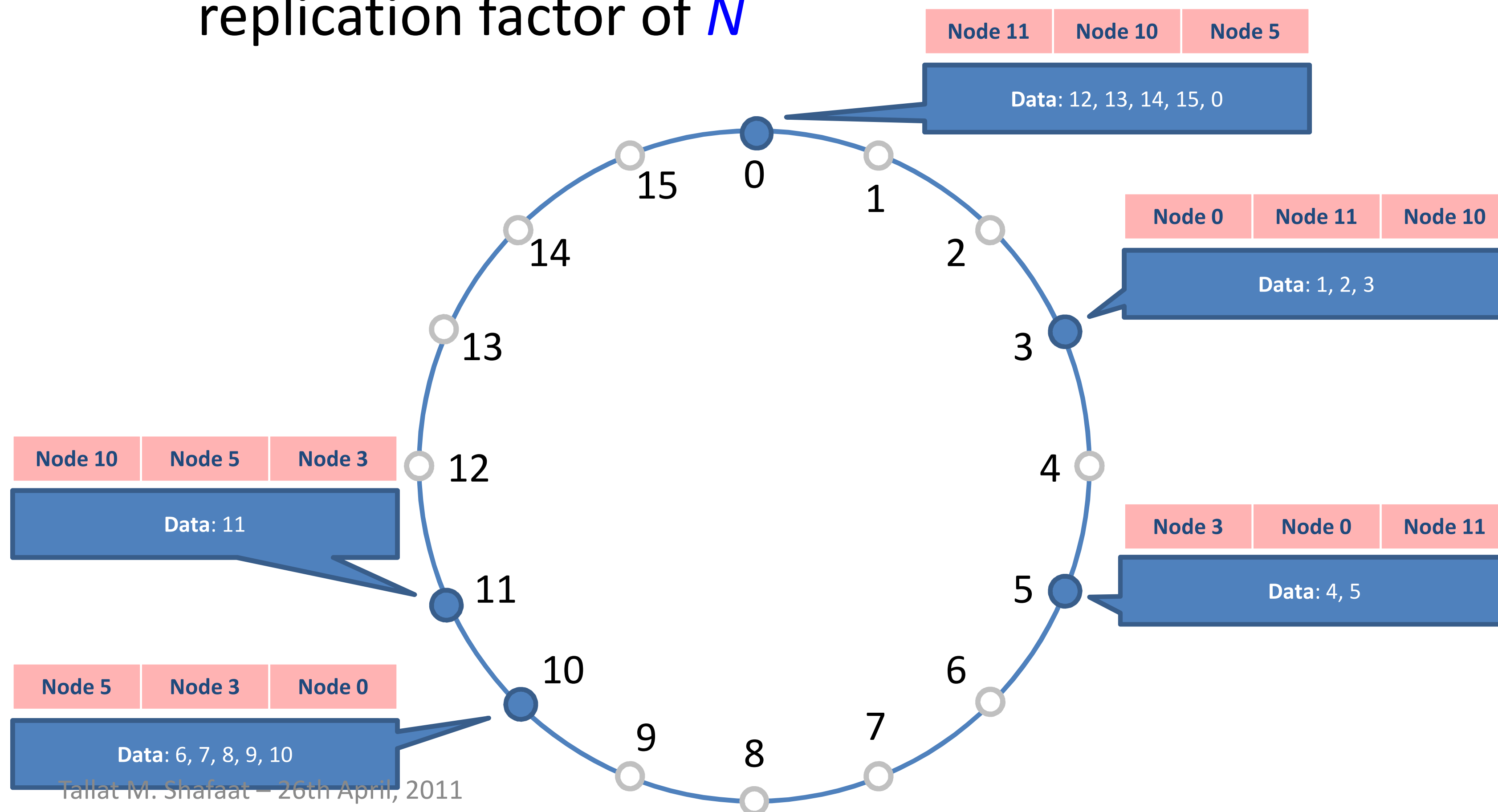
- How many virtual servers?
  - For homogeneous, all nodes run **log N** VSs
  - For heterogeneous, nodes run **clogN** VSs, where ‘c’ is
    - small for weak nodes
    - large for powerful nodes

- Move virtual servers from heavily loaded physical nodes to lightly loaded physical nodes

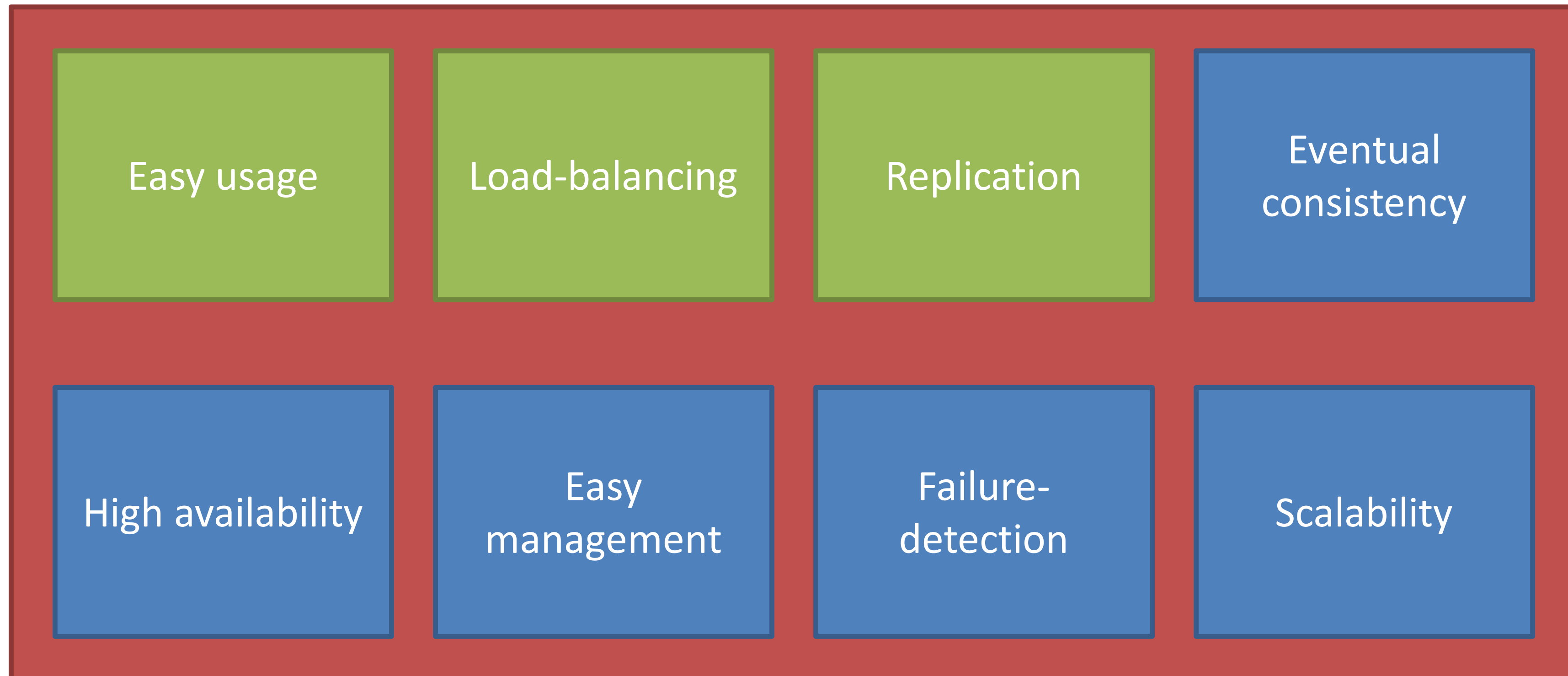


Replication

- Successor list replication
  - Replicate the data of your *N* closest neighbors for a replication factor of *N*



The big picture

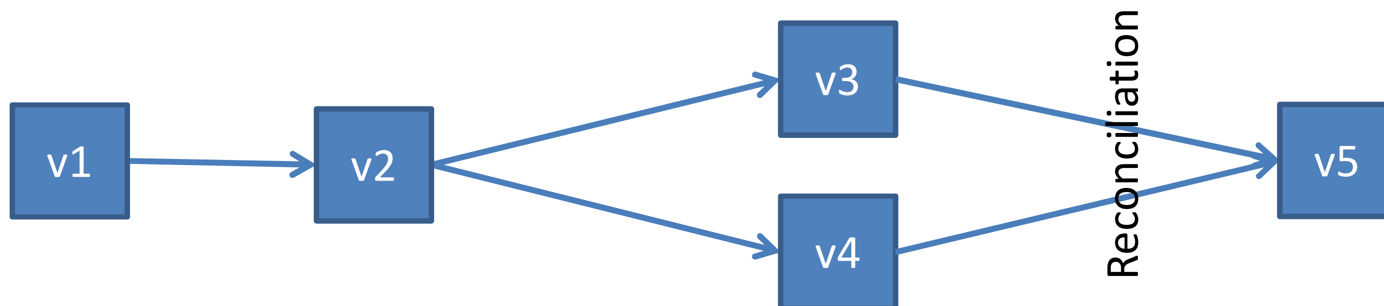


Data versioning (1/3)

- Updates are propagated asynchronously
  - Replicas eventually become consistent
- Each update/modification of an item results in a new and immutable version of the data
  - Multiple versions of an object may exist
- New versions can subsume older versions
  - Syntactic reconciliation
  - Semantic reconciliation

Data versioning (2/3)

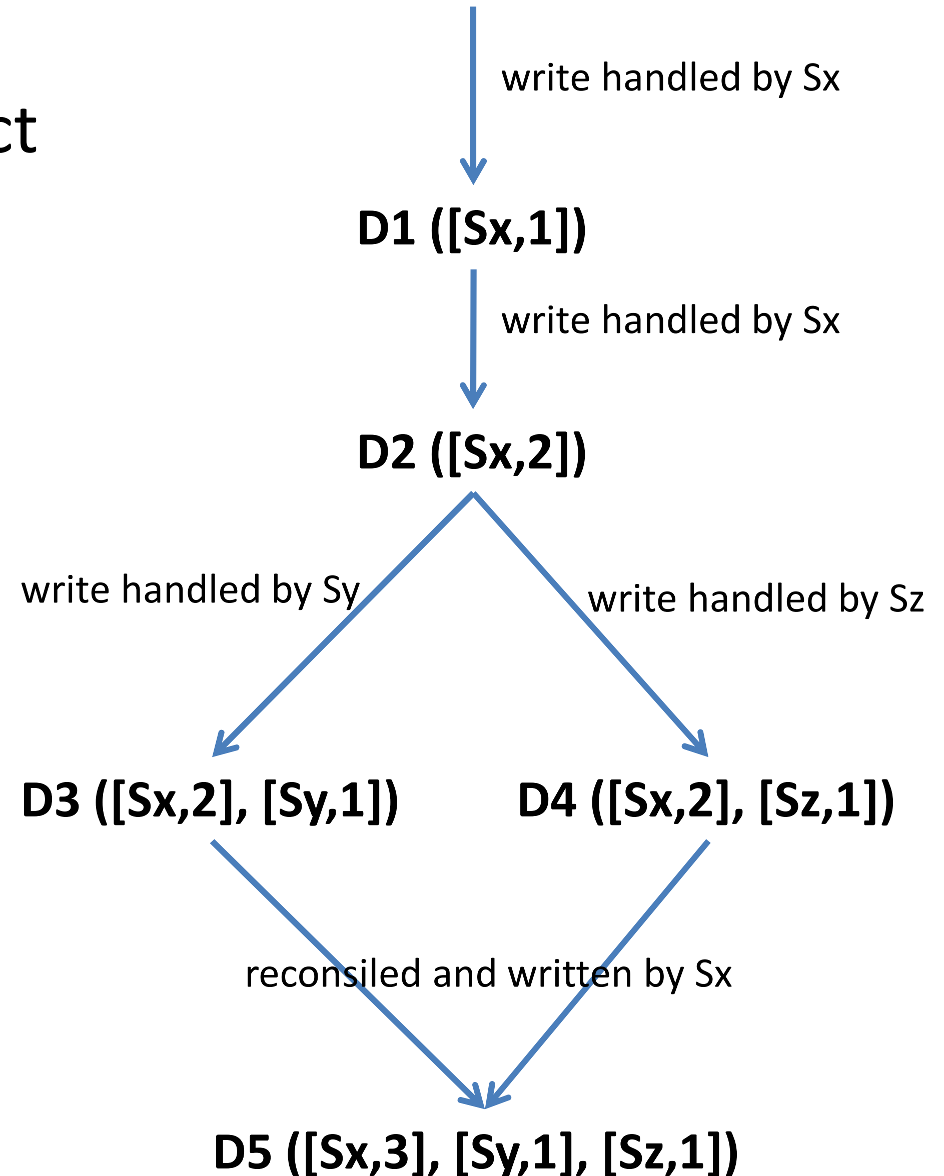
- Version branching can happen due to node failures, network failures/partitions, etc.
  - Target applications are aware that multiple versions can exist
- Use vector clocks for capturing causality
  - If causal: older version can be forgotten
  - If concurrent: conflict exists, requiring reconciliation



- A put requires a context, i.e. which version to update

Data versioning (3/3)

- Client C1 writes new object
  - say via Sx
- C1 updates the object
  - say via Sx
- C1 updates the object
  - say via Sy
- C2 reads D2 and updates the object
  - Say via Sz
- Reconciliation

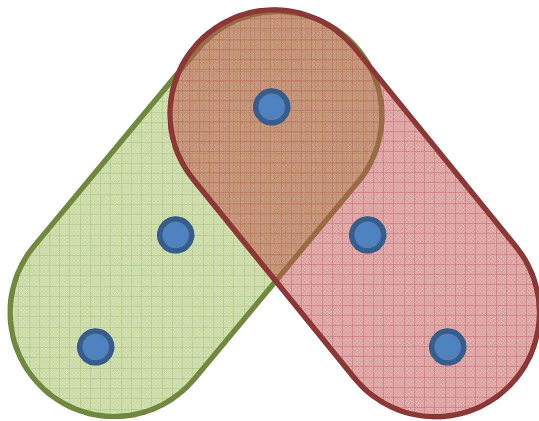


Execution of operations

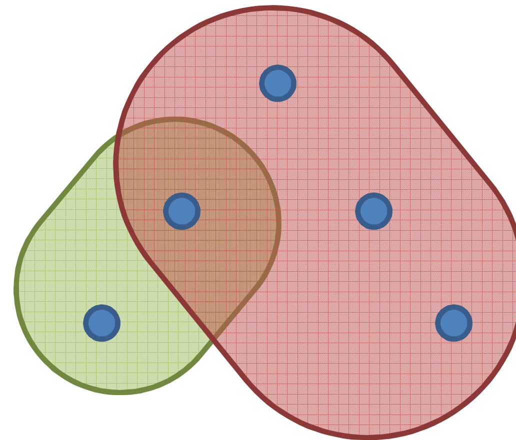
- put and get operations
- Client can send the request
  - to the node responsible for the data
    - Save on latency, code on client
  - to a generic load balancer
    - Extra hop

Quorum systems

- R / W : minimum number of nodes that must participate in a successul read / write
- R + W > N (overlap)



R=3, W=3, N=5



R=4, W=2, N=5

put (key, value, context)

- Coordinator generates new vector clock and writes the new version locally
- Send to N nodes
- Wait for response from W-1 nodes

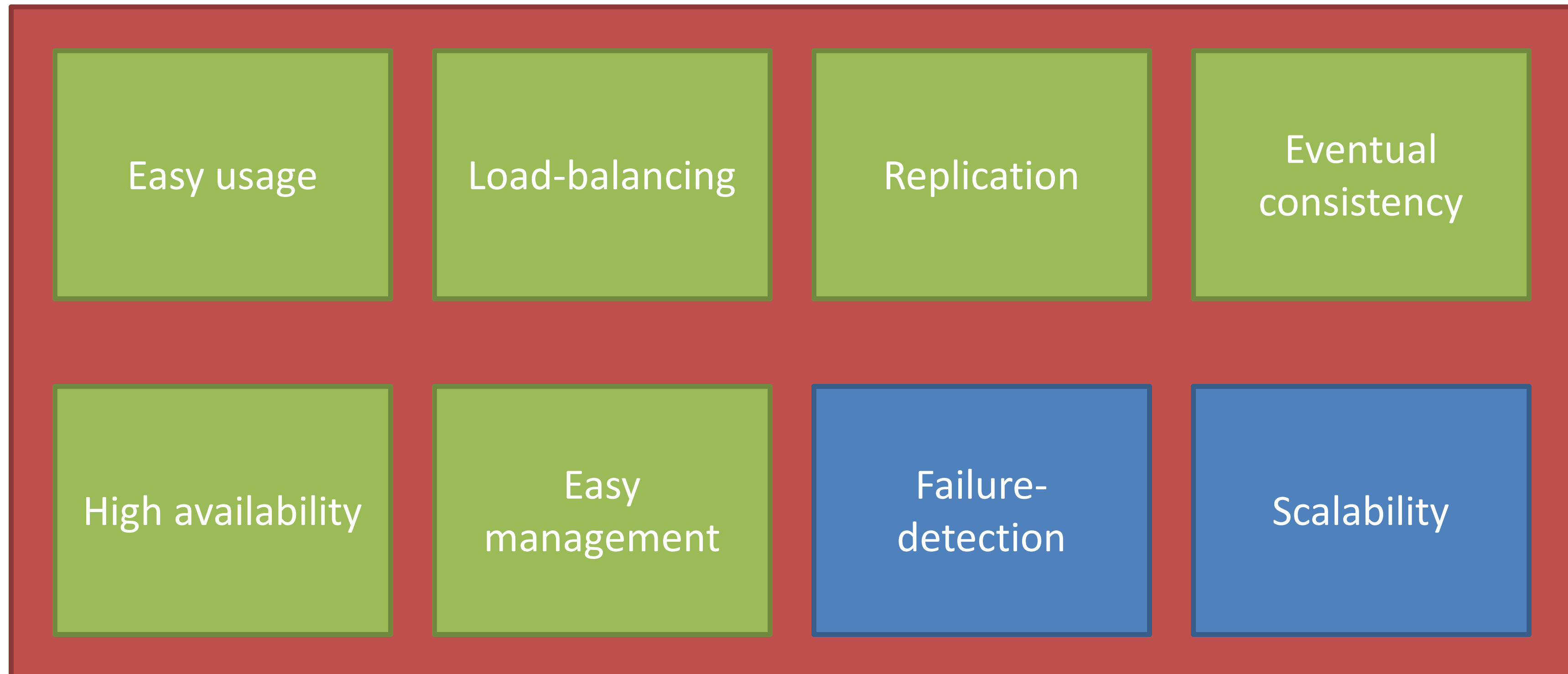
- Using W=1
  - High availability for writes
  - Low durability

(value, context) $\leftarrow$ get (key)

- Coordinator requests existing versions from N
- Wait for response from R nodes
- If multiple versions, return all versions that are causally unrelated
- Divergent versions are then reconciled
- Reconciled version written back

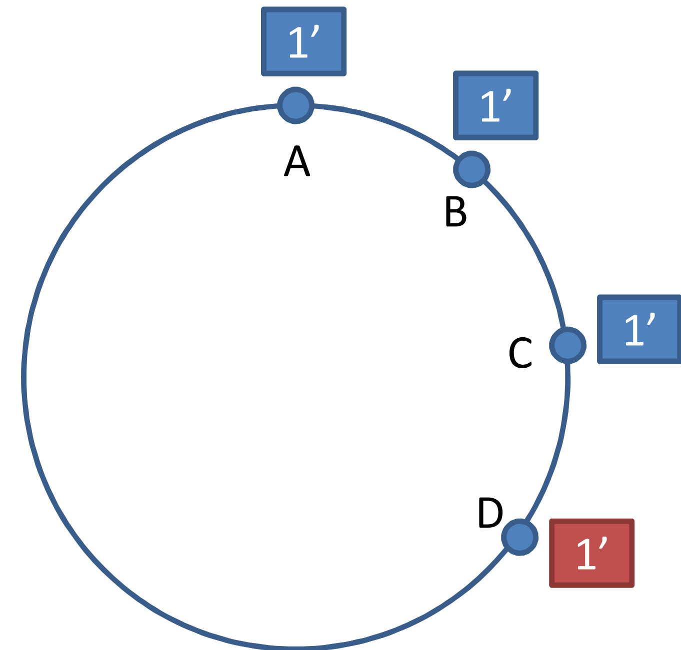
- Using R=1
  - High performance read engine

The big picture



Handling transient failures

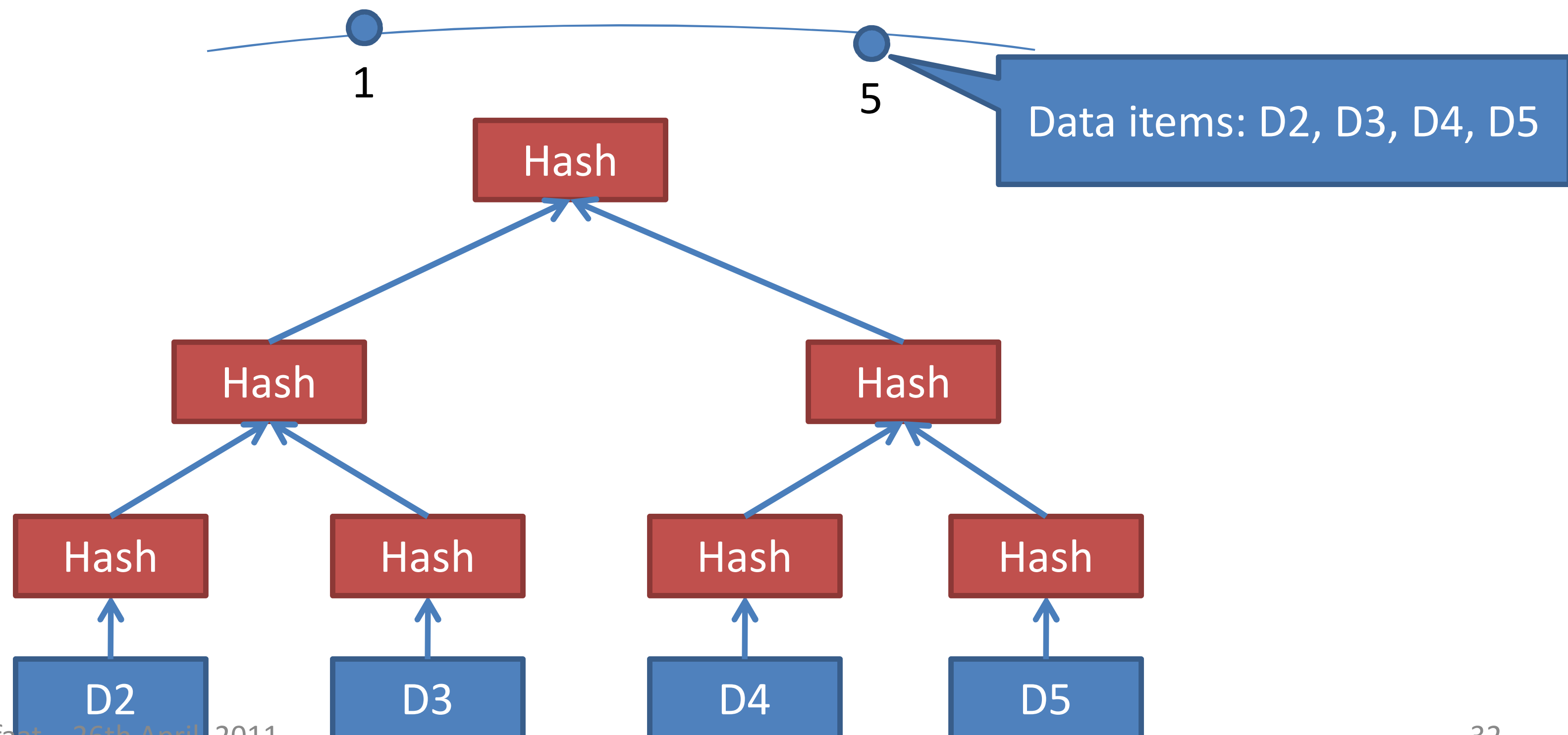
- A managed system
- Which N nodes to update?
- Say A is unreachable
- ’put’ will use D
- Later, D detects A is alive
  - send the replica to A
  - remove the replica



- Tolerate failure of a data center
  - Each object replicated across multiple data centers

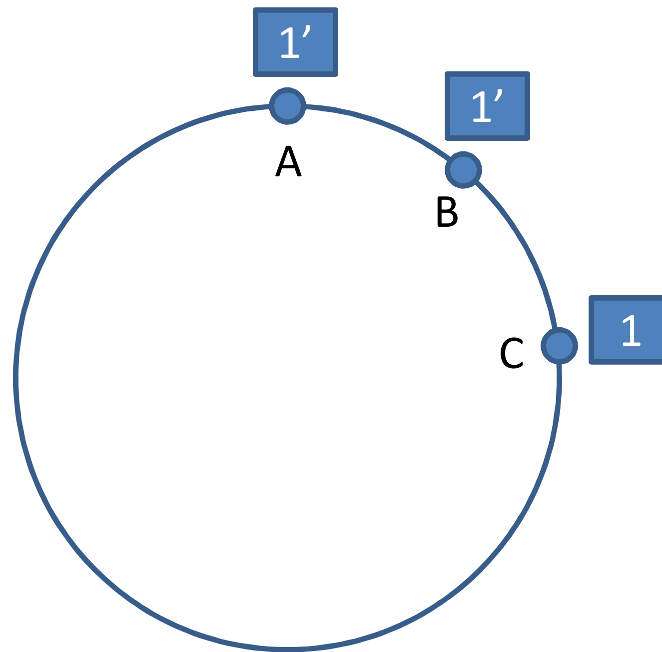
Handling permanent failures (1/2)

- Anti-entropy for replica synchronization
- Use Merkle trees for fast inconsistency detection and minimum transfer of data



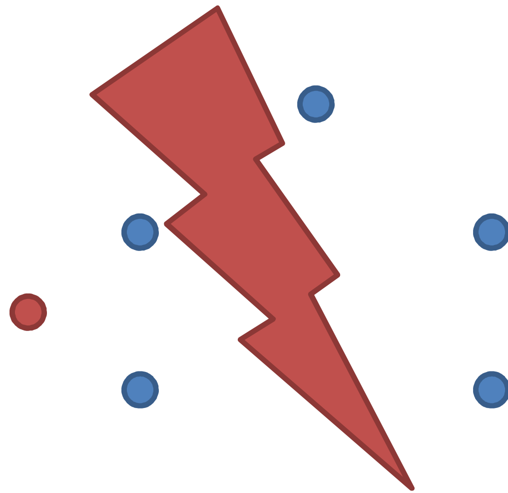
Handling permanent failures (2/2)

- Nodes maintain Merkle tree of each key range
- Exchange root of Merkle tree to check if the key ranges are up-to-date



Quorums under failures

- Due to partitions, quorums might not exist
- Create transient replicas
- Reconcile after partition heals



R=3, W=3, N=5

Membership

- A managed system
  - Administrator explicitly adds and removes nodes
- Receiving node stores changes with time stamp
- Gossiping to propagate membership changes
  - Eventually consistent view
  - O(1) hop overlay
    - log(n) hops, e.g. n=1024, 10 hops, 50ms/hop, 500ms

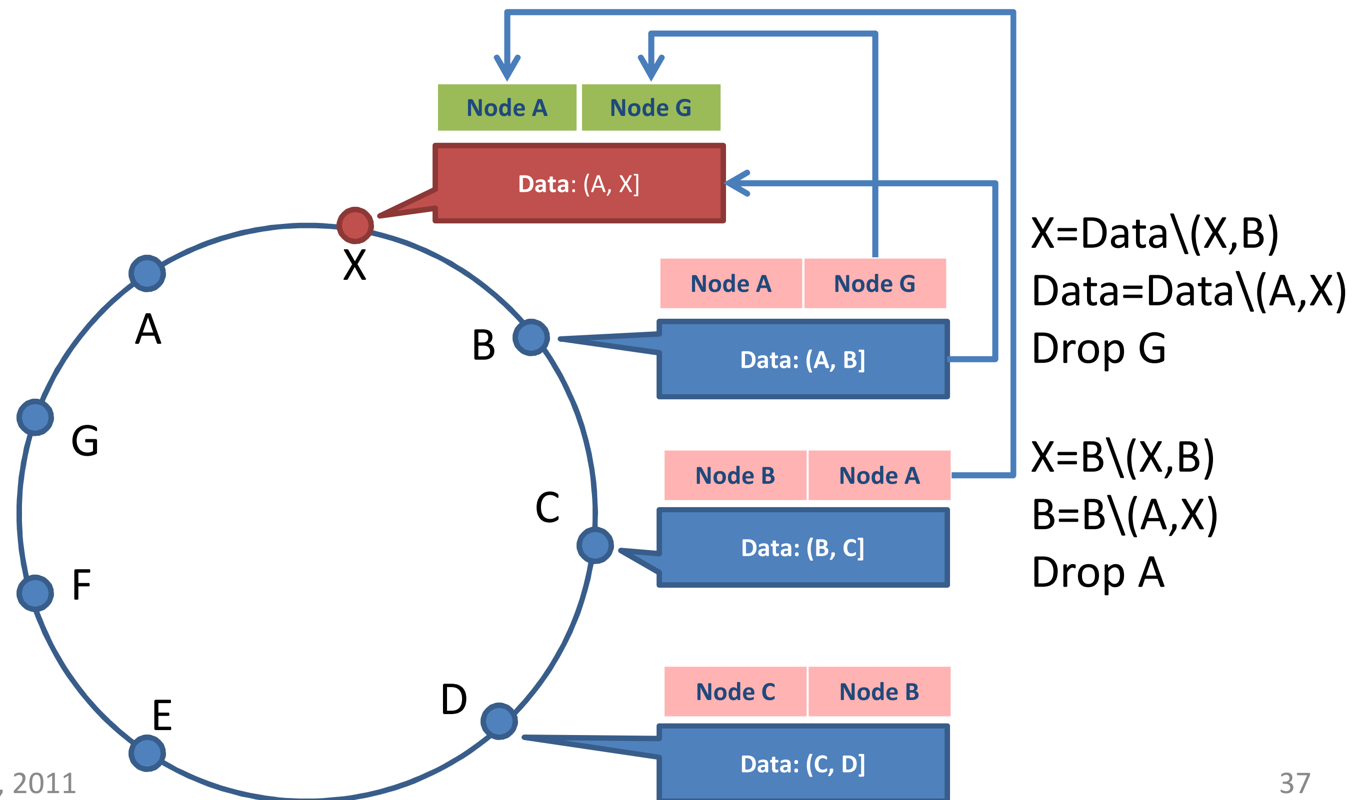
Failure detection

- Passive failure detection
  - Use pings only for detection from failed to alive
  - A detects B as failed if it doesnt respond to a message
  - A periodically checks if B is alive again

- In the absense of client requests, A doesn’t need to know if B is alive
  - Permanent node additions and removals are explicit

Adding nodes

- A new node X added to system
- X is assigned key ranges w.r.t. its virtual servers
- For each key range, it transfers the data items



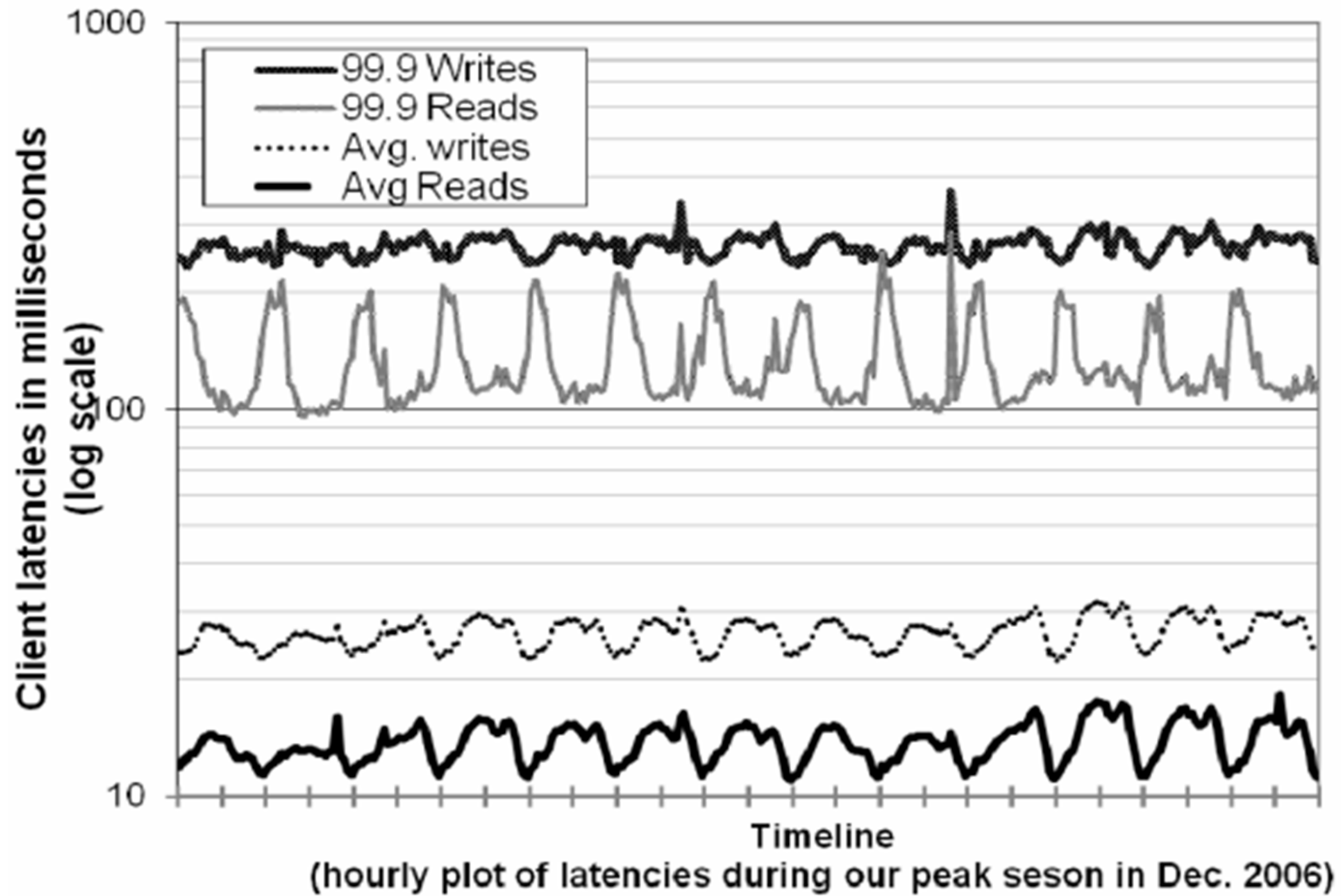
Removing nodes

- Reallocation of keys is a reverse process of adding nodes

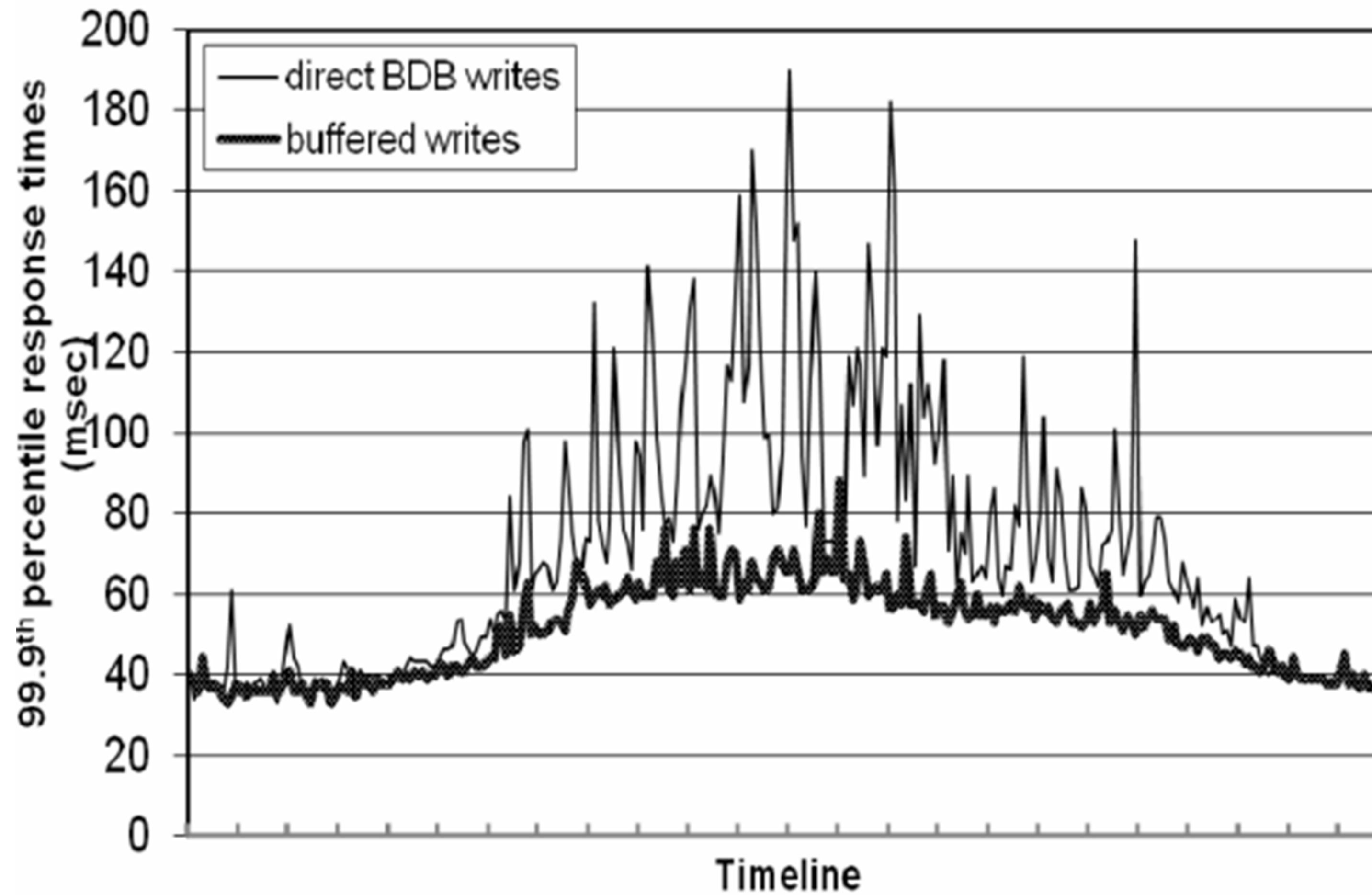
Implementation details

- Local persistence
  - BDS, MySQL, etc.
- Request coordination
  - Read operation
    - Create context
    - Syntactic reconciliation
    - Read repair
  - Write operation
    - Read-your-writes

Evaluation

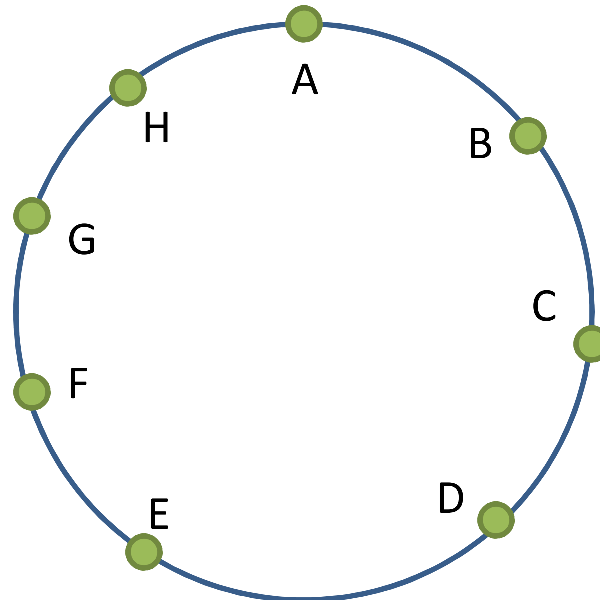


Evaluation



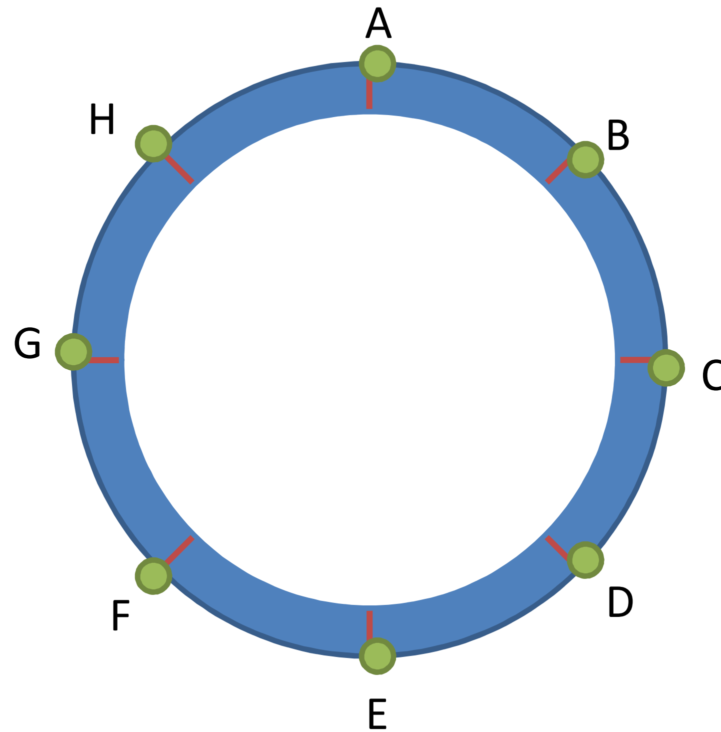
Partitioning and placement (1/2)

- Data ranges are not fixed
  - More time spend to locate items
  - More data storage needed for indexing
- Inefficient bootstrapping
- Difficult to archive the whole data



Partitioning and placement (2/2)

- Divide data space into equally sized ranges
- Assign ranges to nodes



Versions of an item

- Reason
  - Node failues, data center failures, network partitions
  - Large number of concurrent writes to an item
- Occurence
  - 99.94 % one version
  - 0.00057 % two versions
  - 0.00047 % three versions
  - 0.00009 % four versions
- Evaluation: versioning due to concurrent writes

Client vs Server coordination

- Read requests coordinated by any Dynamo node
- Write requests coordinated by a node replicating the data item

- Request coordination can be moved to client
  - Use libraries
  - Reduces latency by saving one hop
  - Client library updates view of membership periodically

End notes

- ”... decentralized techniques can be combined to provide a single highly-available system.”

Readings

- **Dynamo: Amazon's highly available key-value store,** Giuseppe DeCandia et. al., SOSP 2007.
- **Bigtable: A Distributed Storage System for Structured Data,** Fay Chang et. al., OSDI 2006.
- **Casandra** - http://cassandra.apache.org/
- **Eventual consistency -** http://www.allthingsdistributed.com/ 2008/12/eventually_consistent.html
- Key values stores, No SQL, CouchDB, Redis, Voldemort, MongoDB, Hbase

Master thesis

- Build a large-scale elastic data store
  - Consistent and Parition-tolerant
  - Highly available
- Using concepts such as atomic registers, and replicated state machines, and P2P techniques
- Implementation in Java (Kompics)
- Contact if interested