Background

- Distributed Key/Value stores provide a simple **`put/get`** interface
- Great properties: scalability, availability, reliability
- Increasingly popular both within data centers

Dynamo

Cassandra

Voldemort

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia etc.

Presented by:
Tony Huang

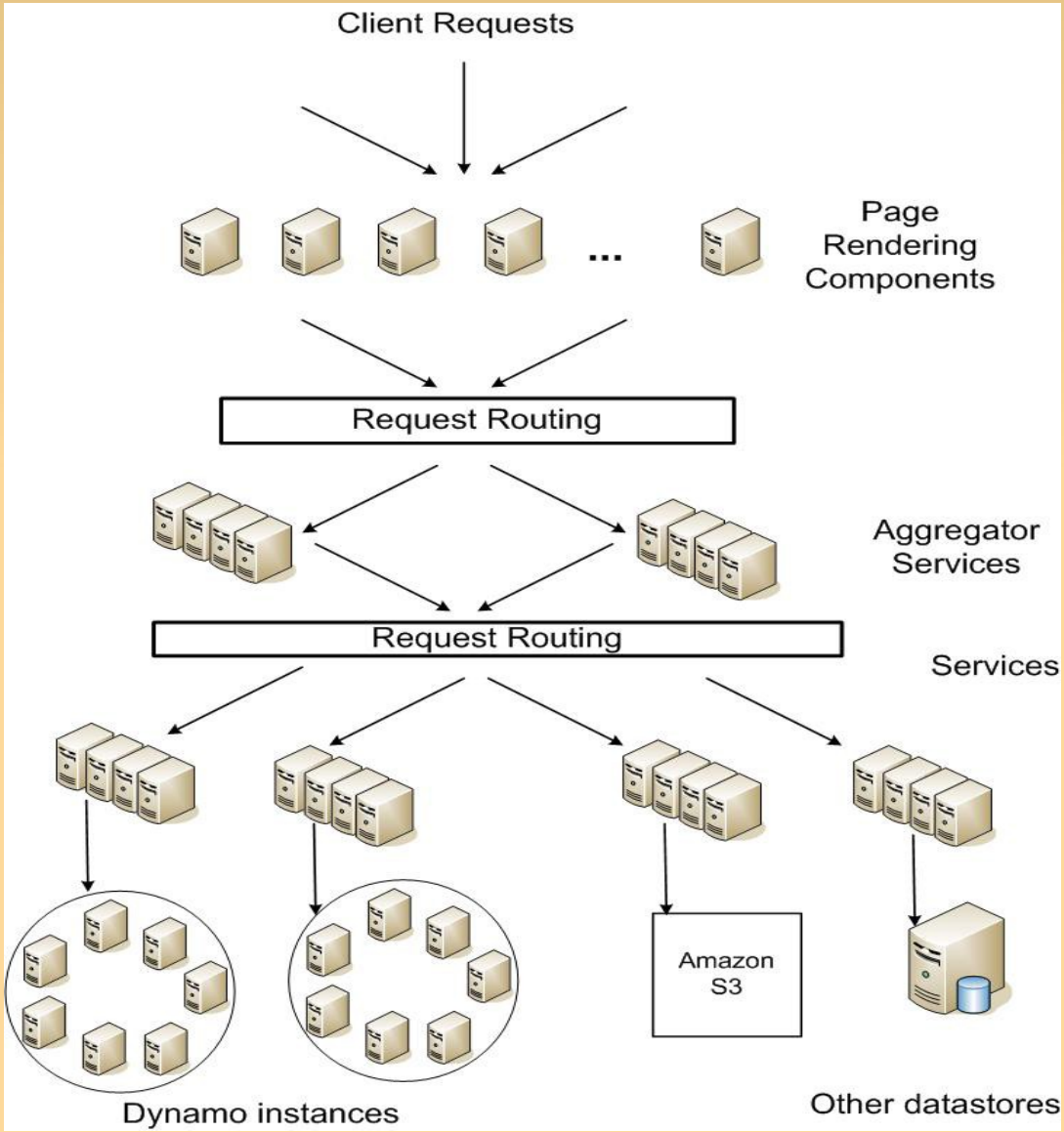
Motivation

- Highly scalable and reliable.
- Tight control over the trade-offs between availability, consistency, cost-effectiveness and performance.
- Flexible enough to let designer to make trade-offs.
- Simple primary-key access to data store.
  - Best seller list, shopping carts, customer preference, session management, sale rank, etc.

Assumptions and Design Consideration

- Query Model
  - Simple read and write operations to a data item that is uniquely identified by a key.
  - Small objects, ~1MB.
- ACID (Atomicity, Consistency, Isolation, Durability)
  - Trade consistency for availability.
  - Does not provide any isolation guarantees.
- Efficiency
  - Stringent SLA requirement.
- Assumed non-hostile environment.
  - No authentication or authorization.
- Conflict resolution is executed during read instead of write.
  - Always writable.
  - Performed either by data store or application

Amazon's Platform Architecture

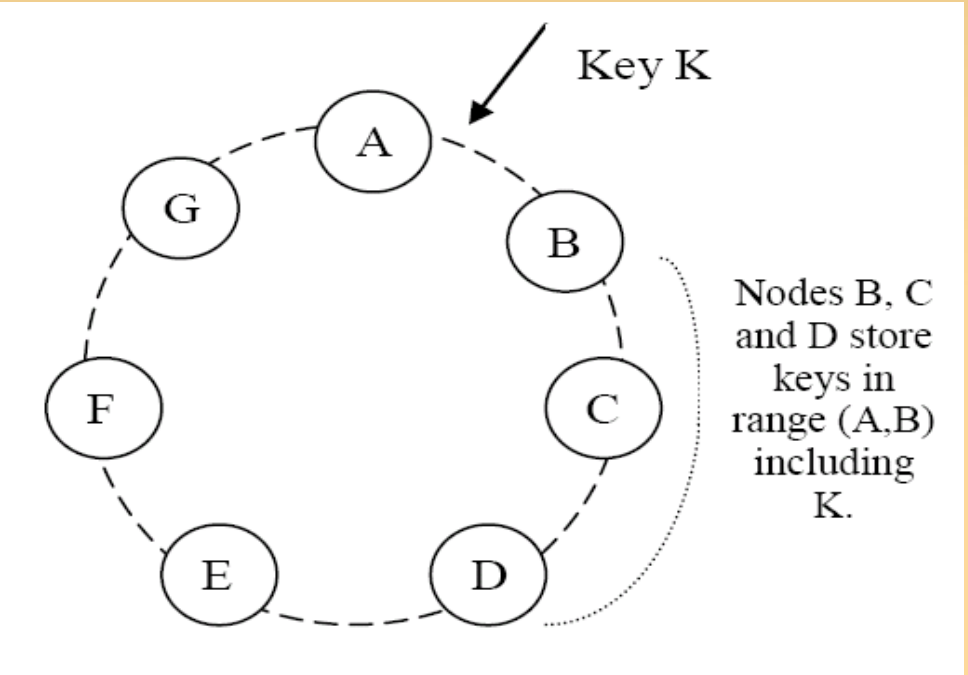


Techniques

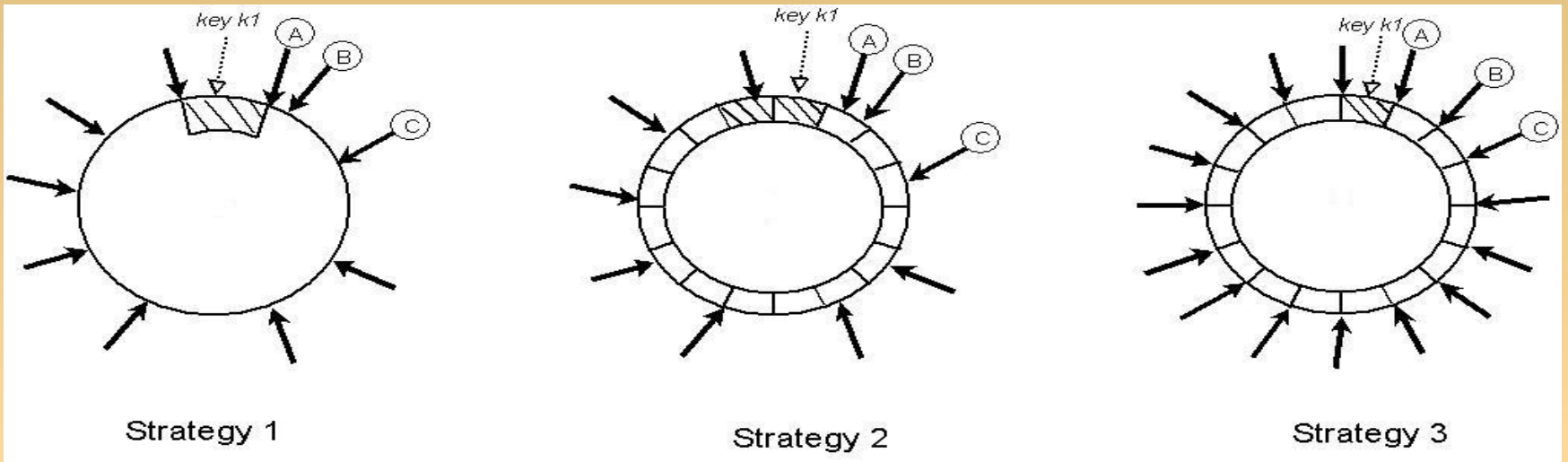
Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

Partitioning

- Consistent hashing: the output range of a hash function is treated as a fixed circular space or “ring”.
- ”Virtual Nodes”: Each node can be responsible for more than one virtual node.
  - Node fails: load evenly dispersed across the rest.
  - Node joins: its virtual nodes accept a roughly equivalent amount of load from the rest.
  - Heterogeneity.

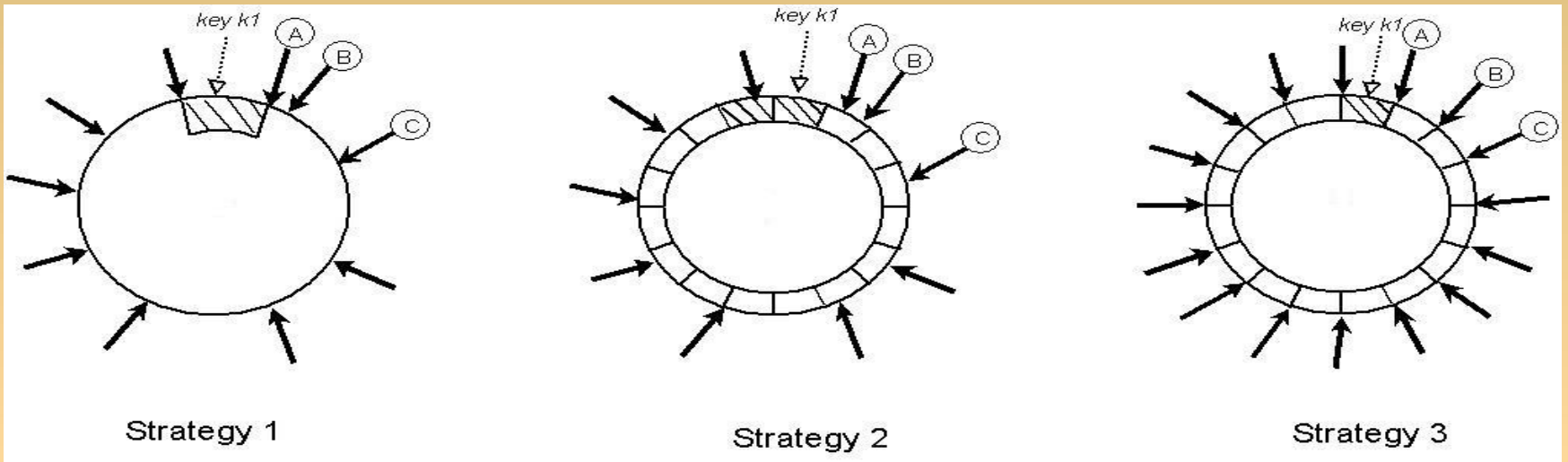


Load Distribution



- Strategy 1: T random tokens per node and and partition by token value.
  - Ranges vary in size and frequently change.
  - Long bootstrapping.
  - Difficult to take a snapshot.

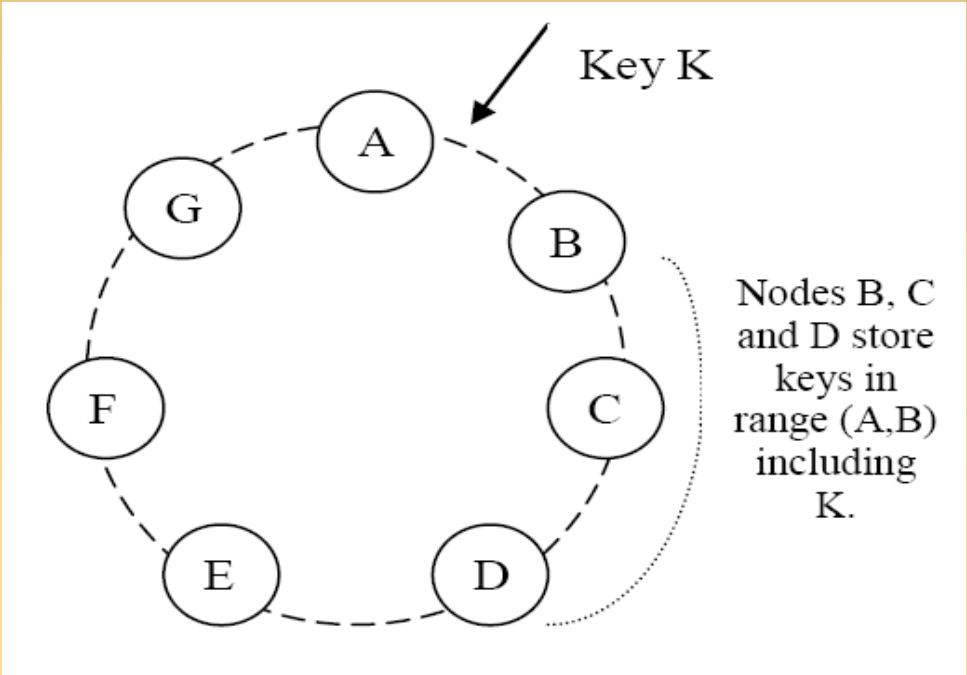
Load Distribution



- Strategy 2: T random tokens per node, partition by token value.
  - Turn out to be the worst, why?
- Strategy 3: Q/S tokens per node, equal-sized partitions.
  - Best load balancing configuration.
  - Drawback: Changing node membership requires coordination.

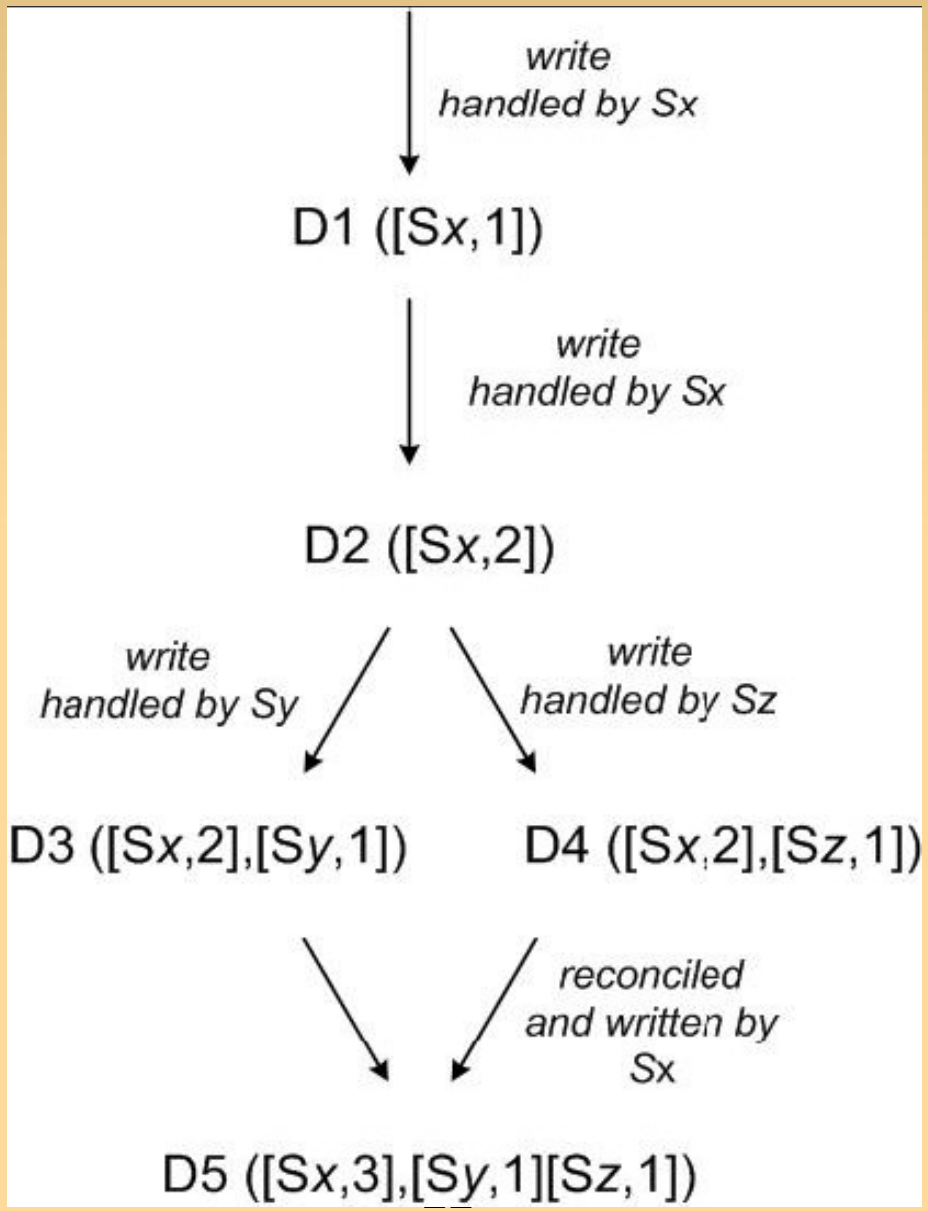
Replication

- Each data item is replicated at N hosts.
- "*preference list*": The list of nodes that is responsible for storing a particular key.
  - Improvement: The preference list contains only distinct physical nodes.



Data Versioning

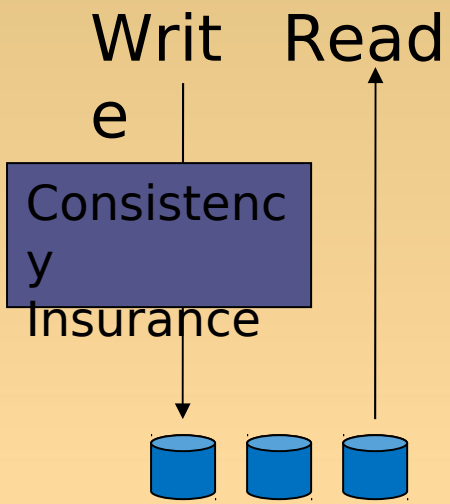
- A vector clock is a list of (node, counter) pairs.
- Every version of every object is associated with one vector clock.
- Client perform reconciliation when system can not.



Quorum for Consistency

- R: min num of nodes in a successful read.
- W: min num of nodes in a successful write.
- N: Num of machines in System.
- Different combination of R and W results in systems for different purpose.

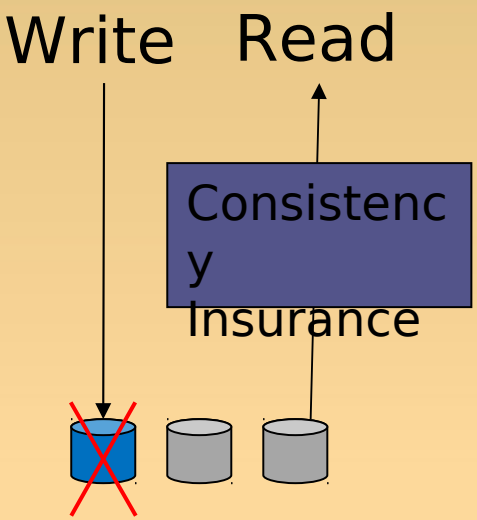
Quorum for Consistency



- Read Engine

Write: 3

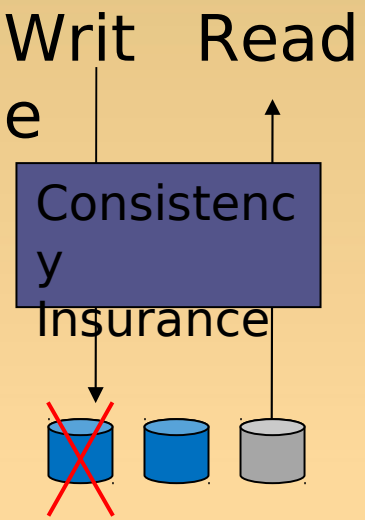
Read: 1



- Always writable, but high risk on inconsistency.

Write: 1

Read: ?



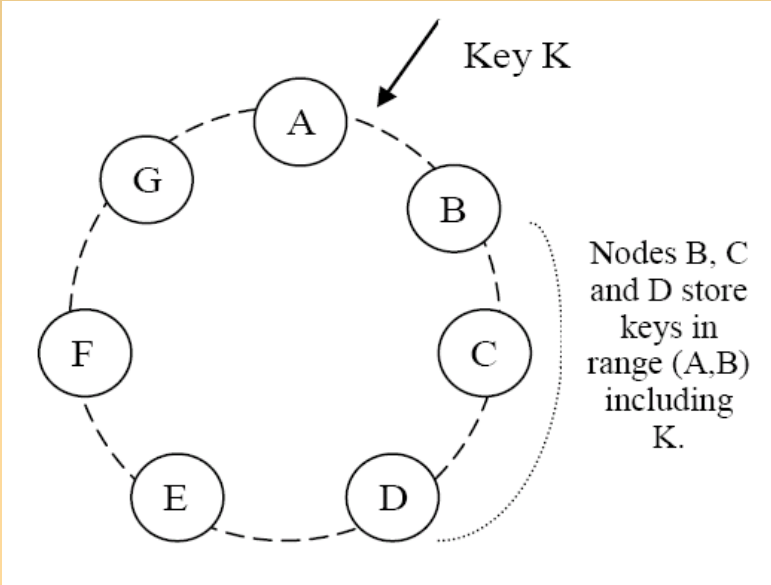
- Normally

Write: 2

Read: 2

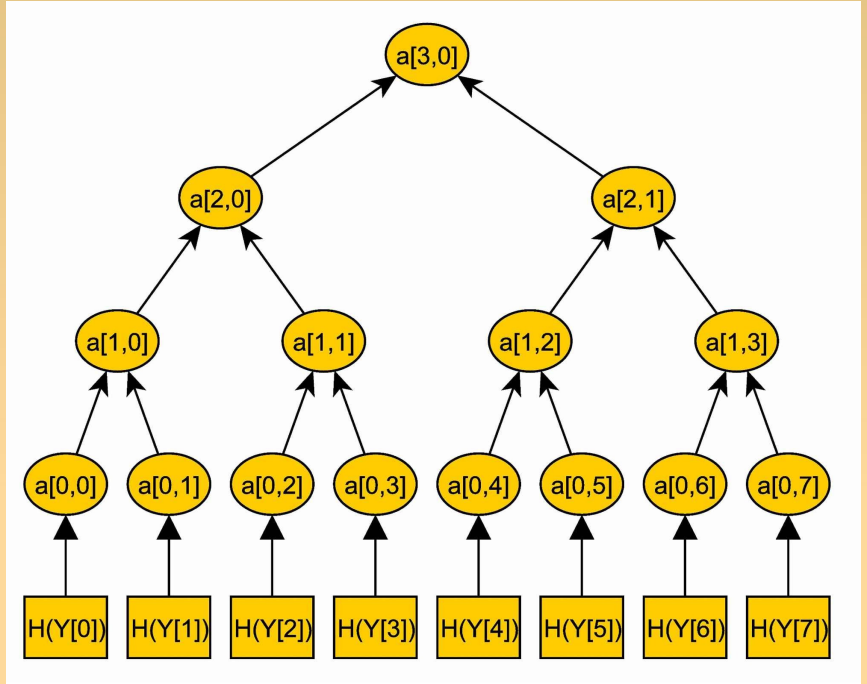
Hinted Handoff

- Assume N = 3. When A is temporarily down or unreachable during a write, send replica to D.
- D is hinted that the replica is belong to A and it will deliver to A when A is recovered.
  - What if A never recovered?
  - What if D fails before A recovers?



Replica Synchronization

- Merkle trees:
  - Hash tree.
  - Leaves are hashes of individual keys.
  - Parent nodes are hashes of their children.
- Reduce amount of data required while checking for consistency.



Membership and Failure Detection

- Manually signal membership change.
- Gossip-based protocol propagates membership changes.
- Some Dynamo nodes as seed nodes for external discovery.
  - Potential single point of failure?
- Local detection of neighbor failure
  - Gossip style protocol to propagate failure information.

Discussion

- What applications are suitable Dynamo (shopping cart, what else?)
- What applications are NOT suitable for Dynamo.
- How can you adapt Dynamo to store large data?
- How can you make Dynamo secure?

Comet: An Active Distributed Key-Value Store

Roxana Geambasu, Amit Levy, Yoshi Kohno,
Arvind Krishnamurthy, and Hank Levy

Presented by Shen Li

Outline

- Background
- Motivation
- Design
- Application

Background

- Distributed Key/Value stores provide a simple **`put/get`** interface
- Great properties: scalability, availability, reliability
- Widely used in P2P systems and is becoming increasingly popular in data centers

Dynamo

Cassandra

Voldemort

Background

- Many applications may share the same key/value storage system.

Outline

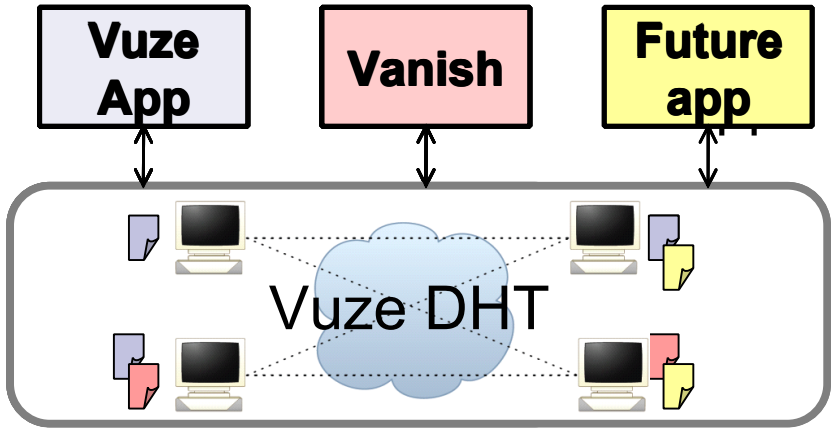
- Background
- Motivation
- Design
- Application

Motivation

- Increasingly, key/value stores are shared by many apps
  - Avoids per-app storage system deployment
- Applications have different (even conflicting) needs:
  - Availability, security, performance, functionality
- But today's key/value stores are one-size-fits-all

Motivating Example

- Vanish is a self-destructing data system above Vuze
- Vuze problems for Vanish:
  - Fixed 8-hour data timeout
  - Overly aggressive replication, which hurts security
- Changes were simple, but deploying them was difficult:
  - Need Vuze engineer
  - Long deployment cycle
  - Hard to evaluate before deployment



Solution

- Build Extensible Key/Value Stores
- Allow apps to customize store's functions
  - Different data lifetimes
  - Different numbers of replicas
  - Different replication intervals
- Allow apps to define new functions
  - Tracking popularity: data item counts the number of reads
  - Access logging: data item logs readers' IPs
  - Adapting to context: data item returns different values to different requestors

Solution

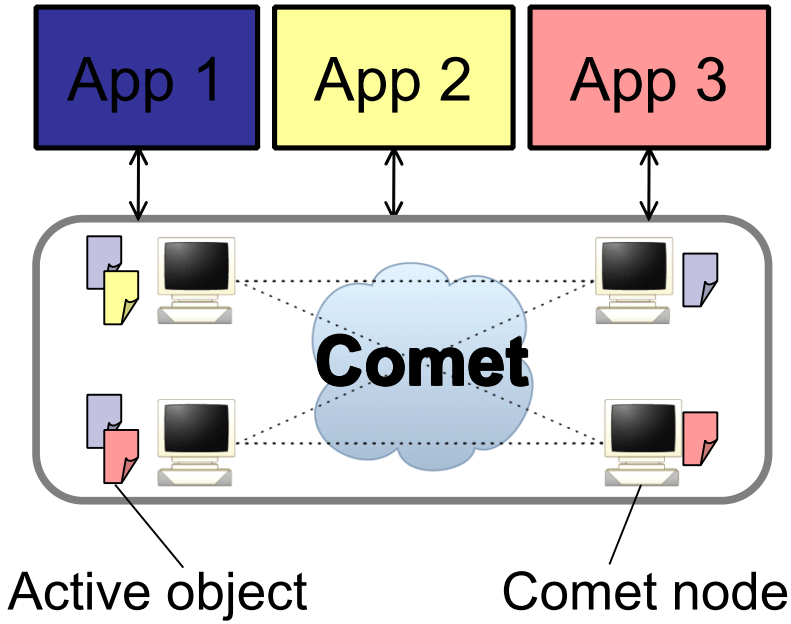
- It should also be simple!
  - Allow apps to inject tiny code fragments (10s of lines of code)
  - Adding even a tiny amount of programmability into key/value stores can be extremely powerful

Outline

- Background
- Motivation
- Design
- Application

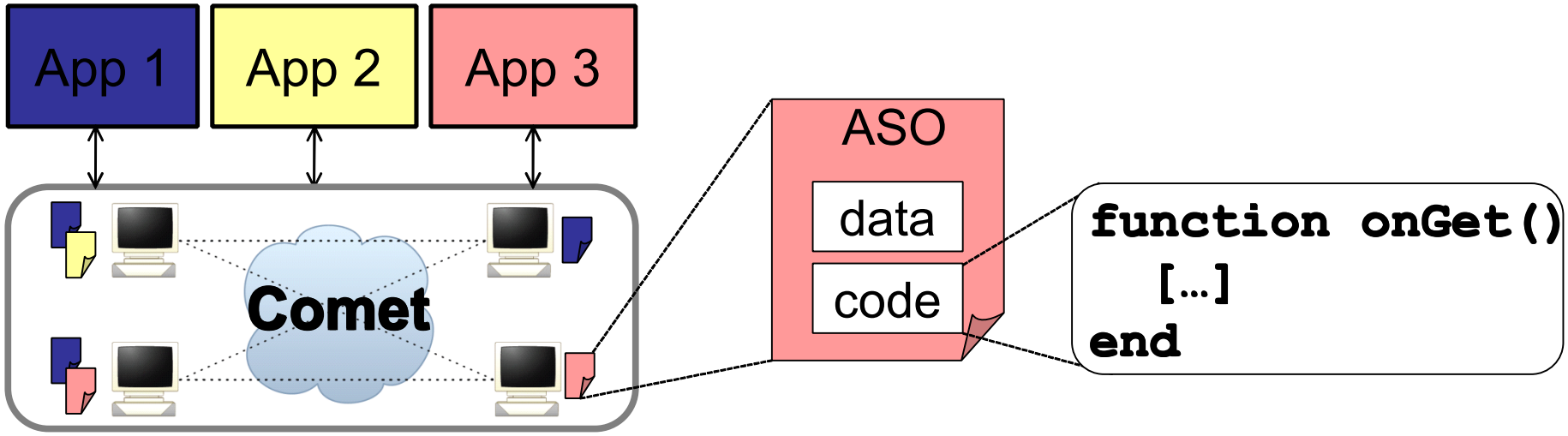
Design

- DHT that supports application-specific customizations
- Applications store active objects instead of passive values
  - Active objects contain small code snippets that control their behavior in the DHT



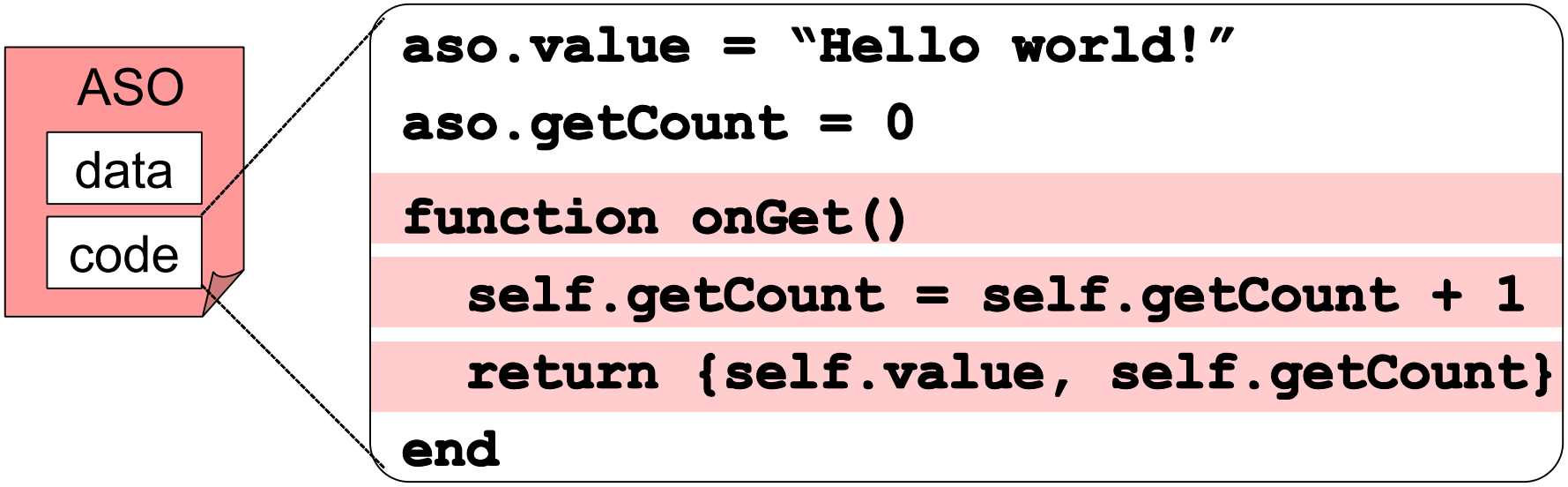
Active Storage Objects

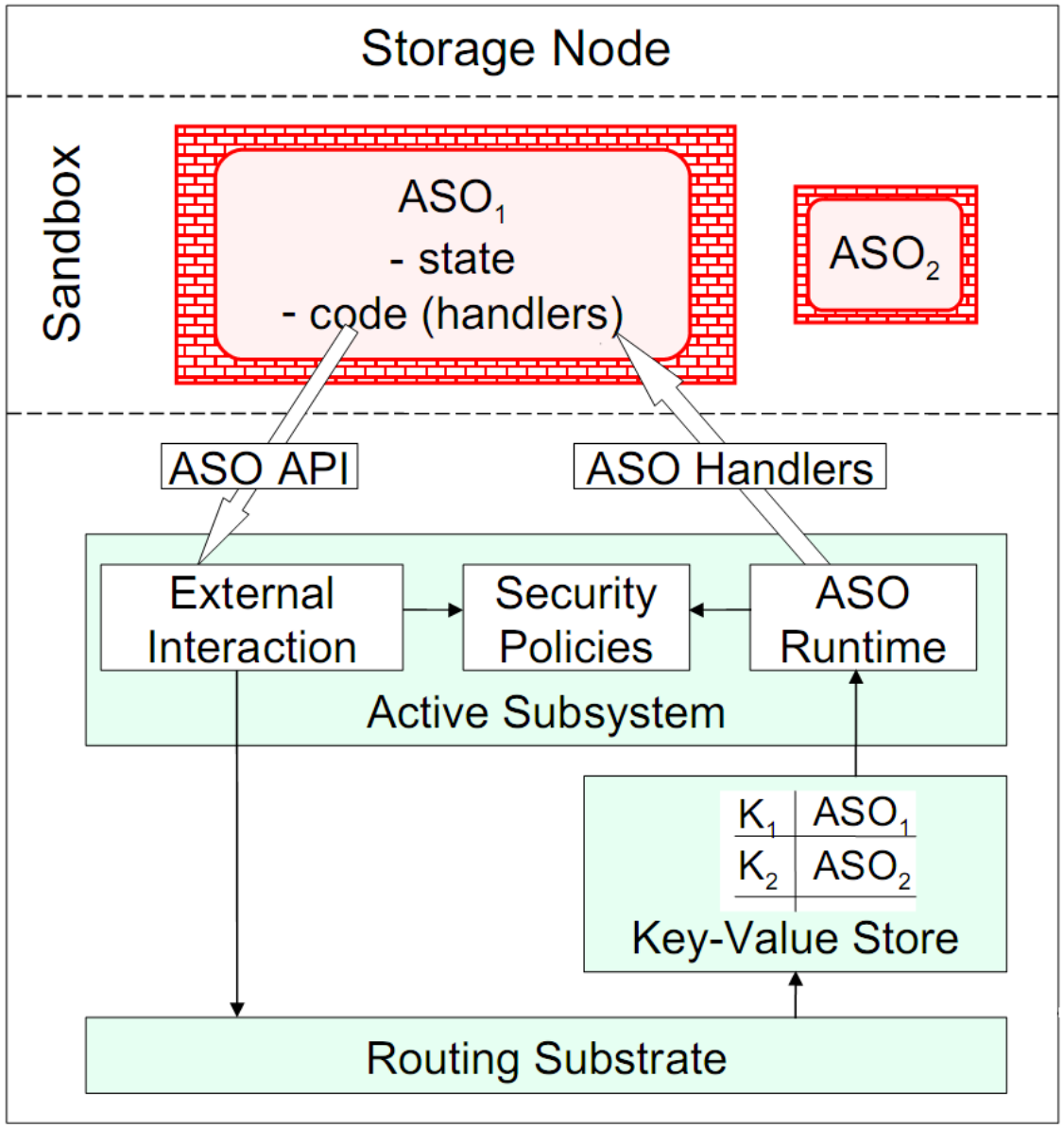
- The ASO consists of data and code
  - The data is the value
  - The code is a set of handlers and user defined functions



ASO Example

- Each replica keeps track of number of gets on an object.




Storage Node
Sandbox
ASO1
- state
- code (handlers)
ASO2
ASO API
ASO Handlers
External Interaction
Security Policies
ASO Runtime
Active Subsystem
K1 ASO1
K2 ASO2
Key-Value Store
Routing Substrate

ASO Extension API

Intercept accesses	Periodic Tasks	Host Interaction	DHT Interaction
onPut(*caller*)	**onTimer**()	**getSystemTime**()	**get**(*key, nodes*)
onGet(*caller*)		**getNodeIP**()	**put**(*key*, *data*, *nodes*)
onUpdate(*caller*)		**getNodeID**()	**lookup**(*key*)
		getASOKey()	
		deleteSelf()	

- Both local and remote recourses are restricted

Local Restriction

- Runtime library
  - Only math packet, string manipulation, and table manipulation.
- CPU
  - 100K bytecode instructions per handler invocation
- memory
  - 100KB per ASO

Remote Restriction

- ASO can only interact specific nodes
  - neighbors responsible for its replication
  - remote node, once per previous interaction
- ASO can only communication with specific ASOs
  - ASOs under the same key
- Message generating rate is limited

Outline

- Background
- Motivation
- Design
- Application

Application

- Three example
  - Application-specific DHT customization
  - Proximity-based distributed tracker
  - Self-monitoring DHT

Application-Specific DHT

- Example: customize the replication scheme

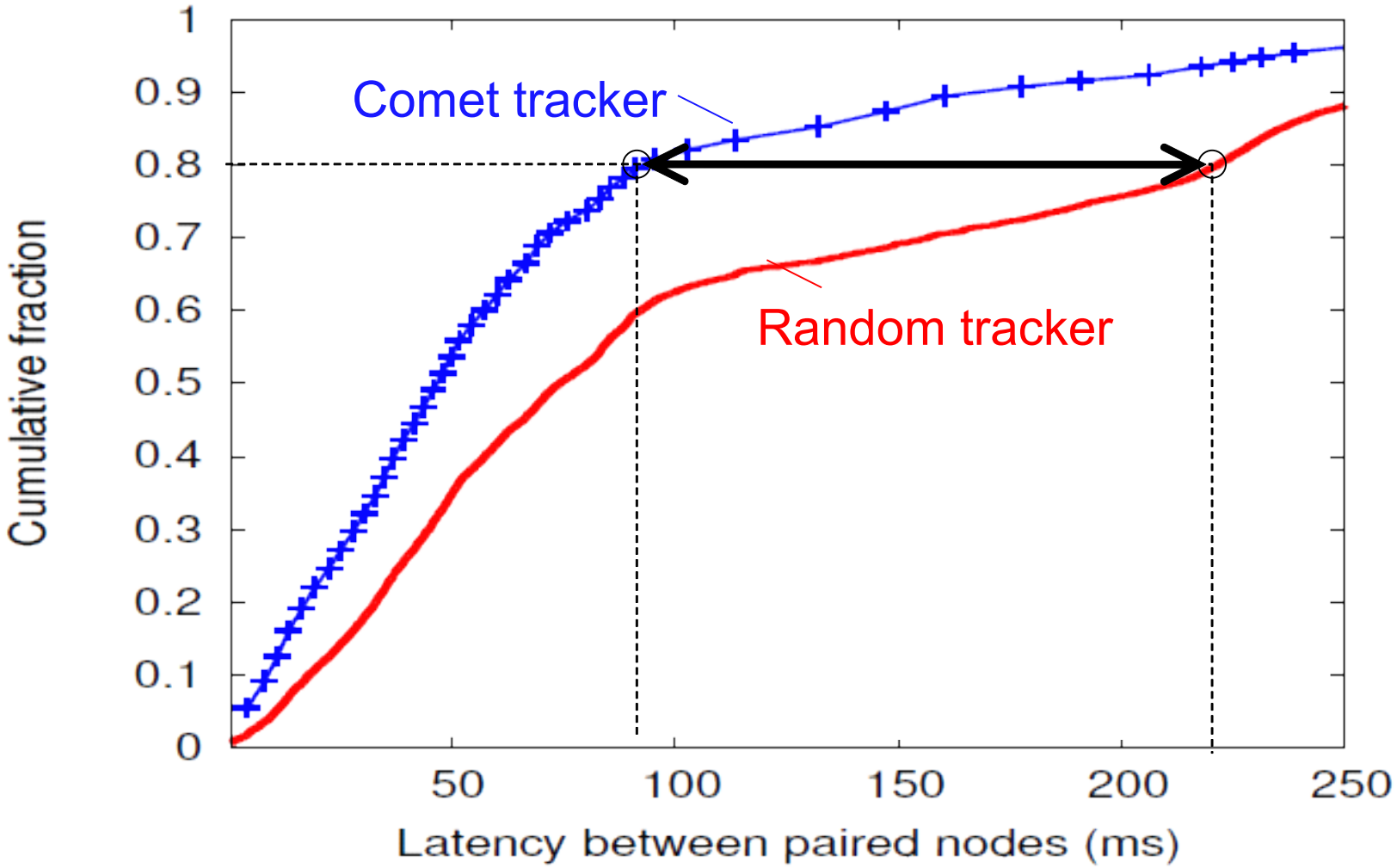
```
function aso:selectReplicas(neighbors)
  [...]
end

function aso:onTimer()
  neighbors = comet.lookup()
  replicas = self.selectReplicas(neighbors)
  comet.put(self, replicas)
end
```

Distributed Tracker

- Traditional distributed trackers return a randomized subset of the nodes
- Comet: a proximity-based distributed tracker
  - Peers **`put`** their IPs and Vivaldi coordinates at **`torrentID`**
  - On **`get`**, the ASO computes and returns the set of closest peers to the requestor

distributed tracker



Self-Monitoring DHT

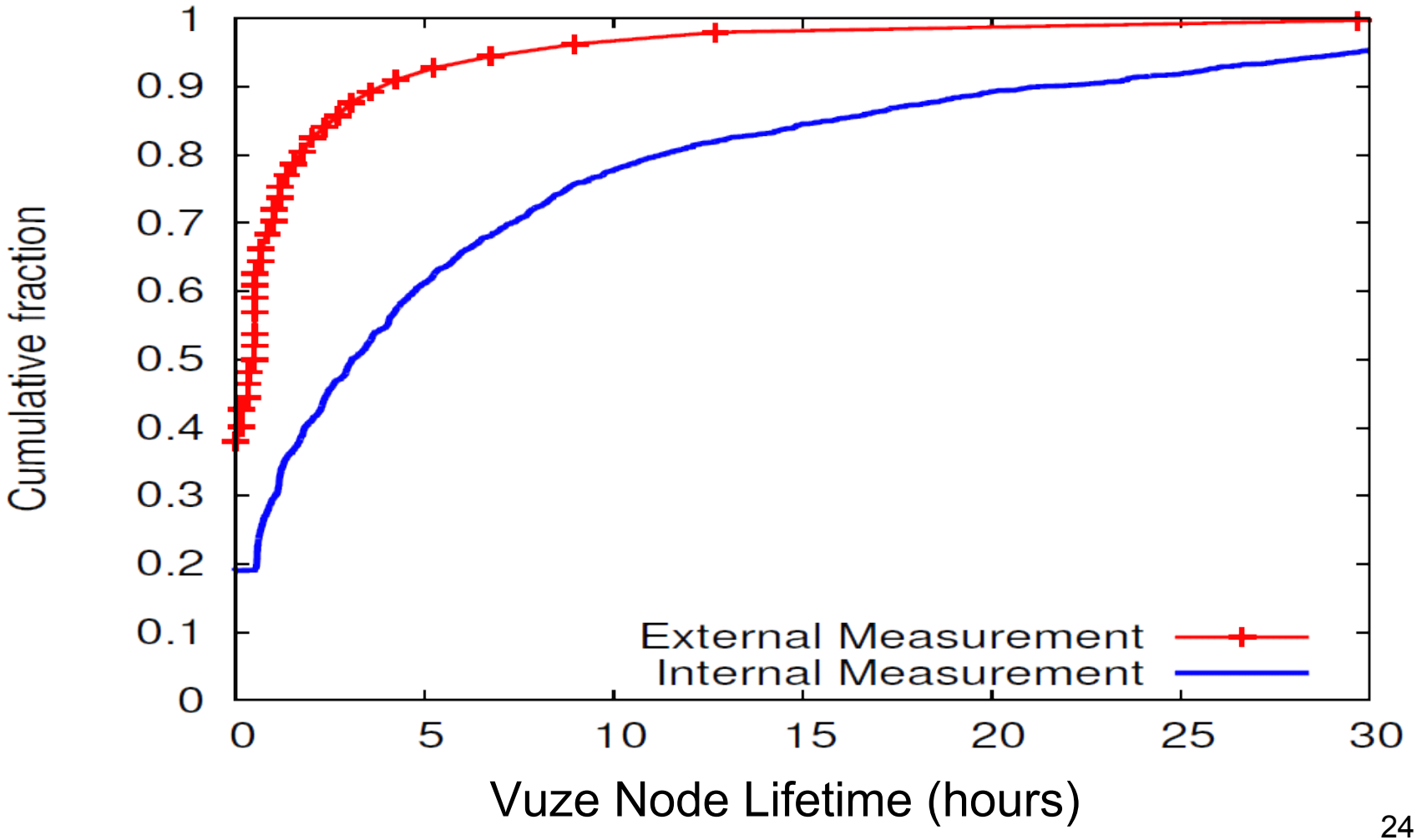
- Example: monitor a remote node's neighbors
  - **`Put`** a monitoring ASO that "pings" its neighbors periodically

```
aso.neighbors = {}

function aso:onTimer()
  neighbors = comet.lookup()
  self.neighbors[comet.systemTime()] = neighbors
end
```

- Useful for internal measurements of DHTs
  - Provides additional visibility over external measurement (e.g., NAT/firewall traversal)

Self-Monitoring DHT



Discusion

1. Is Comet safe enough? Can you come up with an idea to bring it down?
2. Do you agree with the point that Comet trades too much performance for security? Why?
3. If you are service provider of one DHT, would you like to embed Comet into your network? Why?
4. Can you come up with some practical applications that can benefits from Comet?