DISTRIBUTED SYSTEMS OVERVIEW

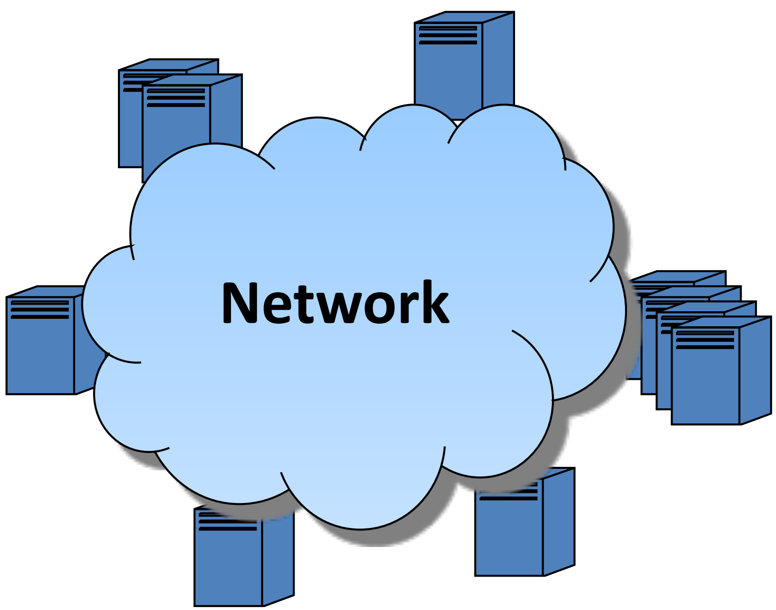
WHAT IS A DISTRIBUTED SYSTEM?

Introductory lecture overview

- Distributed systems definitions
- Distributed systems characteristics
- Distributed systems by example
- Massively scalable key-value stores
- Google's BigTable sketch
- Amazon/Facebook's Dynamo/Cassandra sketch

Working definition

A distributed system is a system that is comprised of several **physically disjoint compute resources** interconnected by a **network**.

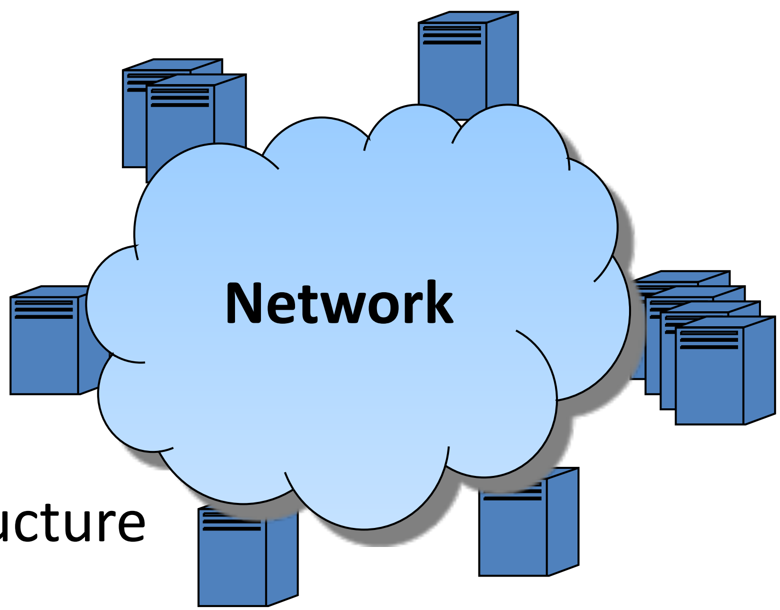


Working definition

A distributed system is a system that is comprised of several **physically disjoint compute resources** interconnected by a **network**.

MapReduce
(Hadoop)

Google infrastructure
(BigTable)



Peer-to-peer
(Bitcoin, BitTorrent)

World Wide Web
(Akamai CDN)

Other definitions & views

- A distributed system is one in which **hardware** or **software components** located at **networked computers communicate** and **coordinate** their actions only by **passing messages**.
  - *By Coulouris et al.*

- A distributed system is a **collection of independent computers** that appears to its users as **a single coherent system**.
  - *By Tanenbaum & van Steen.*

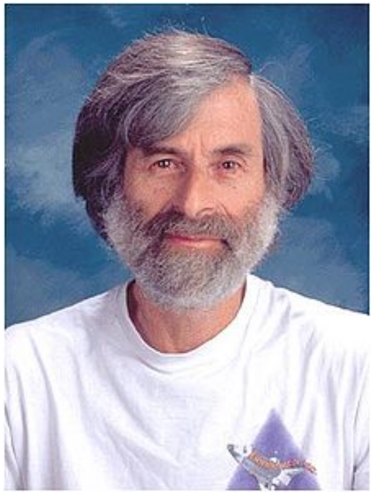
“Introduction to distributed systems design”

- A distributed system is an **application** that executes a collection of **protocols** to **coordinate the actions** of multiple **processes** on a **network,** such that all components **cooperate** together to perform a **single** or small set of **related tasks**.
  - By Google Code University

Leslie Lamport's anecdotal remark

- *"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable*. " (While at DEC SRC, 1985-2001)

- "Father" of distributed systems
  - Turing Award 2013
  - Inventor of LaTeX, ☺



Leslie Lamport

Our focus and perspective in this course

Setting expectations

- Mostly practical perspective
- Some foundations and fundamentals
- Understand best practices
- See our working definition
- Keep an open mind
  - Don’t expect rigorously formalized definitions
  - Don’t expect binary precisions
- Our view on distributed systems is about **managing trade-offs** and how to **navigate the systems design space**

Terminology

The nomenclature is not uniform in the community

- Strive to use the term ***node*** to refer to a physically separable computing node in our systems
- Other often synonymously used terms
  - Process, client (?), server, machine, container, …
- Strive to use the term ***message*** to refer to the unit of communication among nodes
- Other often synonymously used terms
  - Packet(s), communication, data, RPC, …
- It is not just us, it's the literature and who you talk to

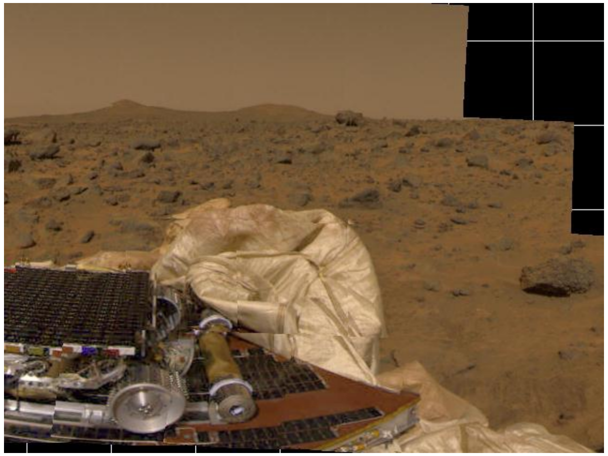
Why Build a Distributed System?

- Centralized system is simpler in all regards
  - Local memory, storage
  - Failure model
  - Maintenance
  - Data security

Why Build a Distributed System?

But…

Mars Pathfinder, July 1997



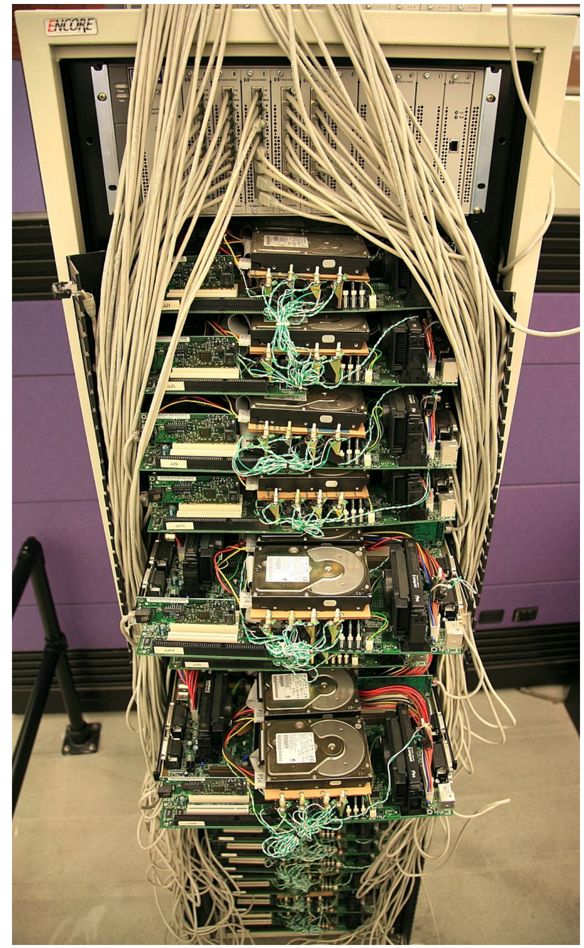
- **Vertical scaling** costs more than **horizontal scaling**
- Availability and redundancy
- Single point of failure

- Many resources are inherently distributed
- Many resources used in a shared fashion

First Google Computer, Cluster

(http://www.computerhistory.org/collections/catalog/102662167)

[Wikipedia]



Related Disciplines …

Networking

- Layers, protocols, TCP/IP
- Latency
- Communication

Databases

- Data management
- Transactions
- Consistency

Distributed Systems

Security

- Threats, defenses
- *Privacy, encryption*

Parallel computing

- Concurrency
- Massively parallel, HPC
- NUMA, UMA

Tentative Course Outline

Subject to change, stay tuned

- Time in distributed systems
- Coordination and agreement
- Consensus with Paxos
- Replication
- Consistency and transactions
- Consistent hashing, CAP theorem, web caching
- Distributed file systems (GFS)
- MapReduce, Spark
- Peer-to-peer systems, distributed hash tables (DHTs)
- Blockchains
- Auxiliary topics: Publish/Subscribe, clouds

Self-study questions

- Find more formal definitions of distributed systems and contrast them to our points of view.
- Compare today’s pricing of a vertically scaled machine to a horizontally scaled one with equal resources.
- Find other terms used for node and message by going through online popular press articles on systems.
- What are other related disciplines you have come across in your studies, if any?
- Is the client-server paradigm a distributed computing paradigm, argue for or against?

The Eight Fallacies of Distributed Computing

Peter Deutsch

Essentially everyone, when they first build a distributed application, makes the following eight assumptions. All prove to be false in the long run and all cause *big* trouble and *painful* learning experiences.

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

For more details, read the article by Arnon Rotem-Gal-Oz

https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing

DISTRIBUTED SYSTEMS OVERVIEW

SELECTED CHARACTERISTICS

Characteristics of distributed systems

- Reliable
- Fault-tolerant
- Highly available
- Recoverable
- Consistent
- Scalable
- Predictable performance
- Secure
- Heterogeneous
- Open

Also known as the

ilities

(non-functional requirements)

*Many of them still pose **significant challenges** in theory and in practice!*

Reliability

- Probability of a system to **perform** its **required functions** under **stated conditions** for a **specified period of time**.
- To run continuously (correctly) without failure

- Expressed as
  Mean Time Between Failure (MTBF), failure rate

Availability and high-availability

- Proportion of time a system is in a **functioning state**, i.e., can be used, **(1 – unavailable).**
- **Ratio** of time usable over entire time
  - Informally, uptime / (uptime + downtime)
  - System that **can be used 100 hrs** out of **168 hrs** has **availability of 100/168**
- Specified as decimal or percentage
  - Five nines is 0.99999 or 99.999% available

Nines - Class of 9

# Nines	Avail. (%)	Downtime per			
		year	month	week	day
1 x 9	90	36.5 d	3 d	16.8 h	2.4 h
2 x 9	99	3.65 d	7.2 h	1.68 h	14.4 mins
4 x 9	99.99	52.56 min	4.32 min	60.48 s	8.64 s
5 x 9	99.999	5.256 min	25.92 s	6.048 s	864 ms
6 x 9	99.9999	31.536 s	2.592 s	604.8 ms	86.4 ms
9 x 9	99.9999999	31.536 ms	2.592 ms	604.8 µs	86.4 µs

Nines - Class of 9

- Frequently used for telecommunication systems
- More a marketing term
- Does not capture impact or cost of downtime

"According to Google, its Gmail service was ***available*** *99.984 percent of the time in 2010 …"* by P. Lilly

Availability ≠ Reliability

- System going down 1 ms every 1 hr has an availability of more than 99.9999%
  – Highly available, but also highly unreliable

- A system that never crashes, but is taken down for two weeks per year
  – Highly reliable, but only about 96% available

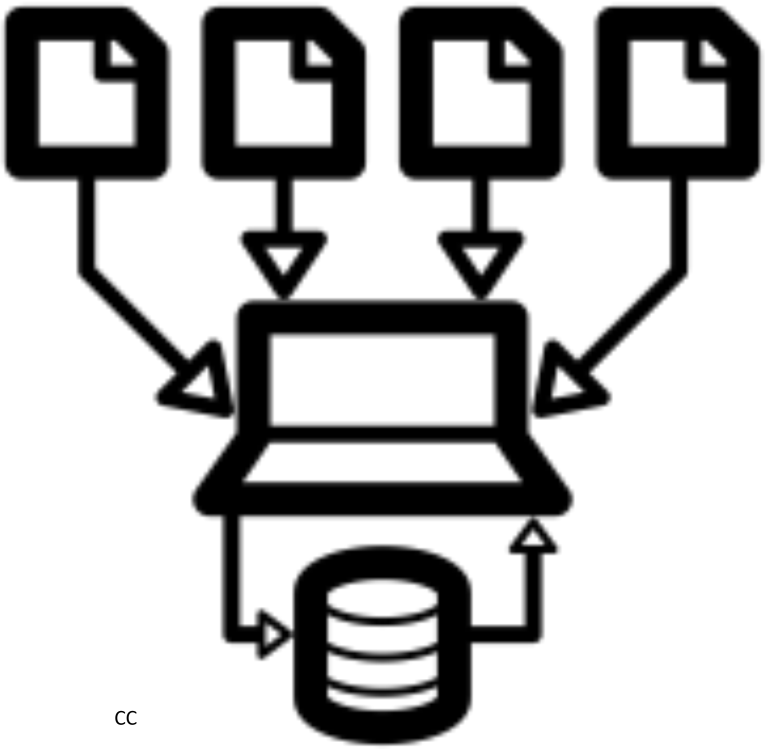
Distributed Systems Design Fallacies

- **Assumptions** (novice) designers of distributed systems often make **that turn out to be false**
- Originated in 1994 by Peter Deutsch, Sun Fellow, Sun Microsystems

- **The eight fallacies**
  - The network is reliable.
  - Latency is zero.
  - Bandwidth is infinite.
  - The network is secure.
  - Topology doesn't change.
  - There is one administrator.
  - Transport cost is zero.
  - The network is homogeneous

Self-study questions

- Look at popular cloud providers and seek to identify the Class of 9 they offer their customers – who promises the most availability and at what cost?



DISTRIBUTED SYSTEMS BY EXAMPLE

MASSIVELY SCALABLE KEY-VALUE STORES

Key-value stores

- *What is a key-value-store?*
- *Why are key-value stores needed*?
- Key-value-store client interface
- Key-value stores in practice
- Common features & non-features
- Apache HBase
- Apache Cassandra

What mechanisms make them work?

What are key-value stores?

- Container for key-value pairs (databases)
- Distributed, multi-component, systems
- NoSQL semantics (non-relational)
- KV-stores offer **simpler query semantics** in exchange for **increased scalability**, **speed**, **availability**, and **flexibility**
- Data model not new

DBMS (SQL)

Students Table

Student	ID*
John Smith	084
Jane Bloggs	100
John Smith	182
Mark Antony	219

Activities Table

ID*	Activity*	Cost
084	Swimming	$17
084	Tennis	$36
100	Squash	$40
100	Swimming	$17
182	Tennis	$36
219	Golf	$47
219	Swimming	$15
219	Squash	$40

- Relational data schema
- Data types
- Foreign keys
- Full SQL support

Key-value store

Key	Value
John Smith	{Activity:Name= Swimming}
Jane Bloggs	{Activity:Cost=57}
Mark Anthony	{ID=219}

- No data schema
- Raw byte access
- No relations
- Single-row operations

Why are key-value stores needed?

- Today's internet applications
  - Huge amounts of stored data (1 PB = 10^{15} bytes)
  - Huge number of Internet users (e.g., 3.4 billion)
  - Frequent updates
  - Fast retrieval of information
  - Rapidly changing data definitions
- Ever more users, ever more data

Why are key-value stores needed?

- Horizontal scalability
  - User growth, traffic patterns change
  - Adapt to number of requests, data size
- Performance
  - High speed for single-record read and write operations
- Flexibility
  - Adapt to changing data definitions

Why are key-value stores needed?

- Reliability
  - Thousands of components at play
  - Uses commodity hardware: failure is the norm
  - Provide failure recovery
- Availability and geo-distribution
  - Users are worldwide
  - Guarantee fast access

Key-value store client interface

- Main operations
  - Write/update **put**(key, value)
  - Read **get**(key)
  - Delete **delete**(key)

- Usually no aggregation, no table joins, no transactions!

Hbase: Key-value store client interface

```
Configuration conf = HBaseConfiguration.create();
conf.set("hbase.zookeeper.quorum", "192.168.127.129");

HTable table = new HTable(conf, „MyBaseTable");

Put put = new Put(Bytes.toBytes("key1"));
put.add(Bytes.toBytes("colfam1"), Bytes.toBytes(„value"), Bytes.toBytes(200));
table.put(put);

Get get = new Get(Bytes.toBytes("key1"));
Result result = table.get(get);
byte[] val = result.getValue(Bytes.toBytes("colfam1"), Bytes.toBytes(„value"));
System.out.println("Value: " + Bytes.toInt(val));
```

Hbase: Key-value store client interface

Initialization Using ZooKeeper

```
Configuration conf = HBaseConfiguration.create();
conf.set("hbase.zookeeper.quorum", "192.168.127.129");

HTable table = new HTable(conf, „MyBaseTable");

Put put = new Put(Bytes.toBytes("key1"));
put.add(Bytes.toBytes("colfam1"), Bytes.toBytes(„value"), Bytes.toBytes(200));
table.put(put);

Get get = new Get(Bytes.toBytes("key1"));
Result result = table.get(get);
byte[] val = result.getValue(Bytes.toBytes("colfam1"), Bytes.toBytes(„value"));
System.out.println("Value: " + Bytes.toInt(val));
```

Hbase: Key-value store client interface

```
Configuration conf = HBaseConfiguration.create();
conf.set("hbase.zookeeper.quorum", "192.168.127.129");

HTable table = new HTable(conf, „MyBaseTable");

Put put = new Put(Bytes.toBytes("key1"
put.add(Bytes.toBytes("colfam1"), Byt  toBytes(„value"), Bytes.toBytes(200));
table.put(put);

Get get = new Get(Bytes.toBytes("key1"));
Result result = table.get(get);
byte[] val = result.getValue(Bytes.toBytes("colfam1"), Bytes.toBytes(„value"));
System.out.println("Value: " + Bytes.toInt(val));
```

Initialization
Using
ZooKeeper

Column Family:
"Schema"

Hbase: Key-value store client interface

```
Configuration conf = HBaseConfiguration.create();
conf.set("hbase.zookeeper.quorum", "192.168.127.129");

HTable table = new HTable(conf, „MyBaseTable");

Put put = new Put(Bytes.toBytes("key1"
put.add(Bytes.toBytes("colfam1"), Byt    toBytes(„value"), Bytes.toBytes(200));
table.put(put);

Get get = new Get(Bytes.toBytes(“key1"));
Result result = table.get(get);
byte[] val = result.getValue(Bytes.toBytes("colfam1"), B
System.out.println("Value: " + Bytes.toInt(val));
```

Initialization
Using
ZooKeeper

Column Family:
“Schema”

Column:
Defined at run-time
(“wide column”
stores)

Key-value store in practice

- BigTable
- Apache HBase
- Apache Cassandra
- Redis
- Amazon Dynamo
- Yahoo! PNUTS

Common elements of key-value stores

- Failure detection, failure recovery *(cf. Coordination Lecture)*
- Replication *(cf. Replication Lecture)*
  - Store and manage multiple copies of data
- Memory store, write ahead log (WAL)
  - Keep data in memory for fast access
  - Keep a commit log as ground truth
- Versioning (*cf. Time Lecture*)
  - Store different versions of data
  - Timestamping

Self-study questions

If you are not familiar with SQL, simply find an online tutorial

- How would you select, project and join tables with the key-value store API vs. via SQL?
- What are the main differences in realizing the above operations between either models?
- Elicit the main differences between traditional key-value stores (e.g., Berkeley DB) and the massive scale key-value stores we introduced.
- Can SQL be layered on top of a key-value store API, argue for or against?

DISTRIBUTED SYSTEMS BY EXAMPLE

BIGTABLE / HBASE

BigTable

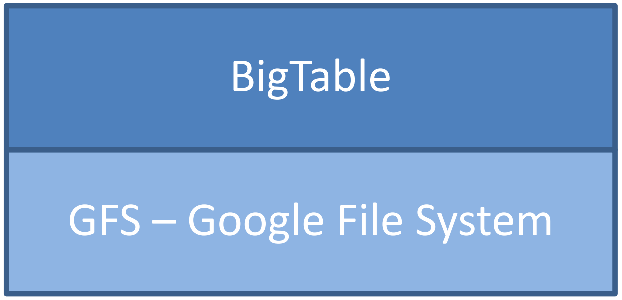
- Engineered at Google, 2004
- Designed for petabyte scale
- Internal use for web indexing, personalized search, Google Earth, Google Analytics, Google Finance
- Based on Google File System (GFS), *cf. GFS et al. Lecture*

BigTable

- Engineered at Google, 2004
- Designed for petabyte scale
- Internal use for web indexing, personalized search, Google Earth, Google Analytics, Google Finance
- Based on Google File System (GFS), *cf. GFS et al. Lecture*

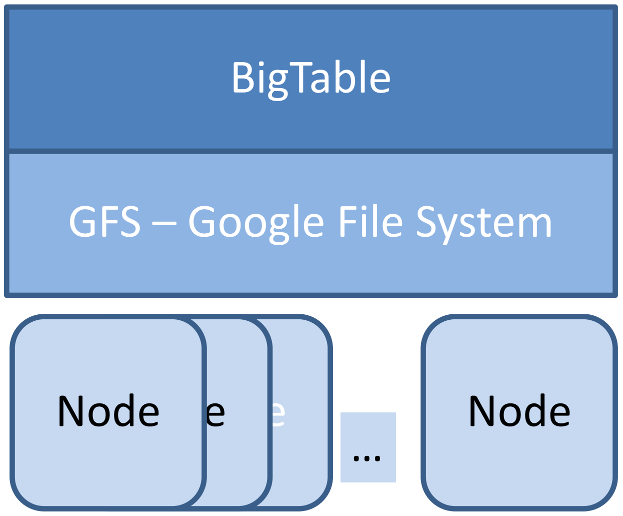
BigTable

- Engineered at Google, 2004
- Designed for petabyte scale
- Internal use for web indexing, personalized search, Google Earth, Google Analytics, Google Finance
- Based on Google File System (GFS), *cf. GFS et al. Lecture*



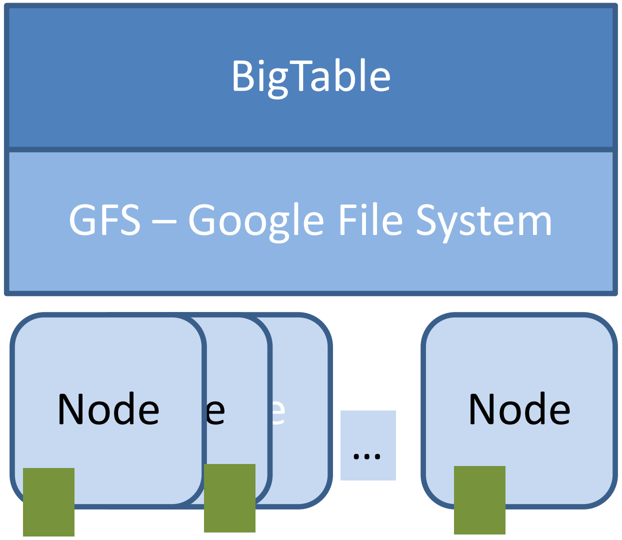
BigTable

- Engineered at Google, 2004
- Designed for petabyte scale
- Internal use for web indexing, personalized search, Google Earth, Google Analytics, Google Finance
- Based on Google File System (GFS), *cf. GFS et al. Lecture*



BigTable

- Engineered at Google, 2004
- Designed for petabyte scale
- Internal use for web indexing, personalized search, Google Earth, Google Analytics, Google Finance
- Based on Google File System (GFS), *cf. GFS et al. Lecture*

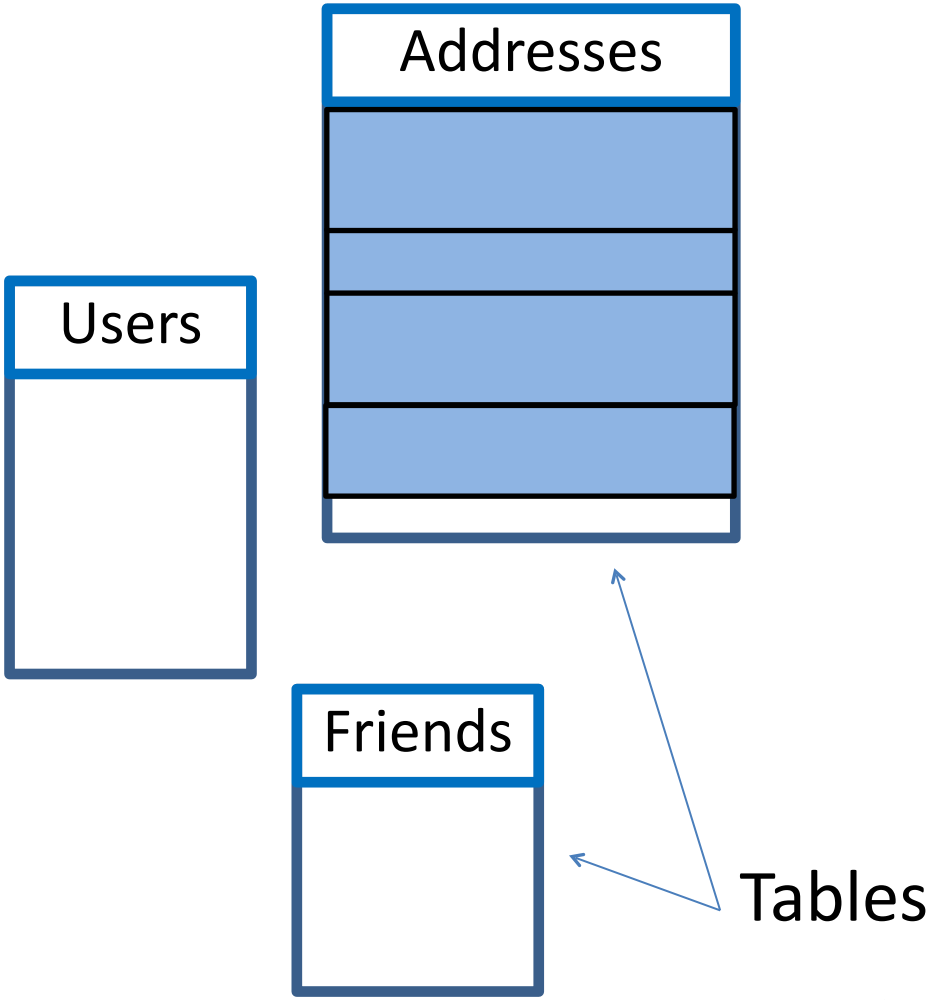


BigTable: Tables & Tablets

(Logical Organization)

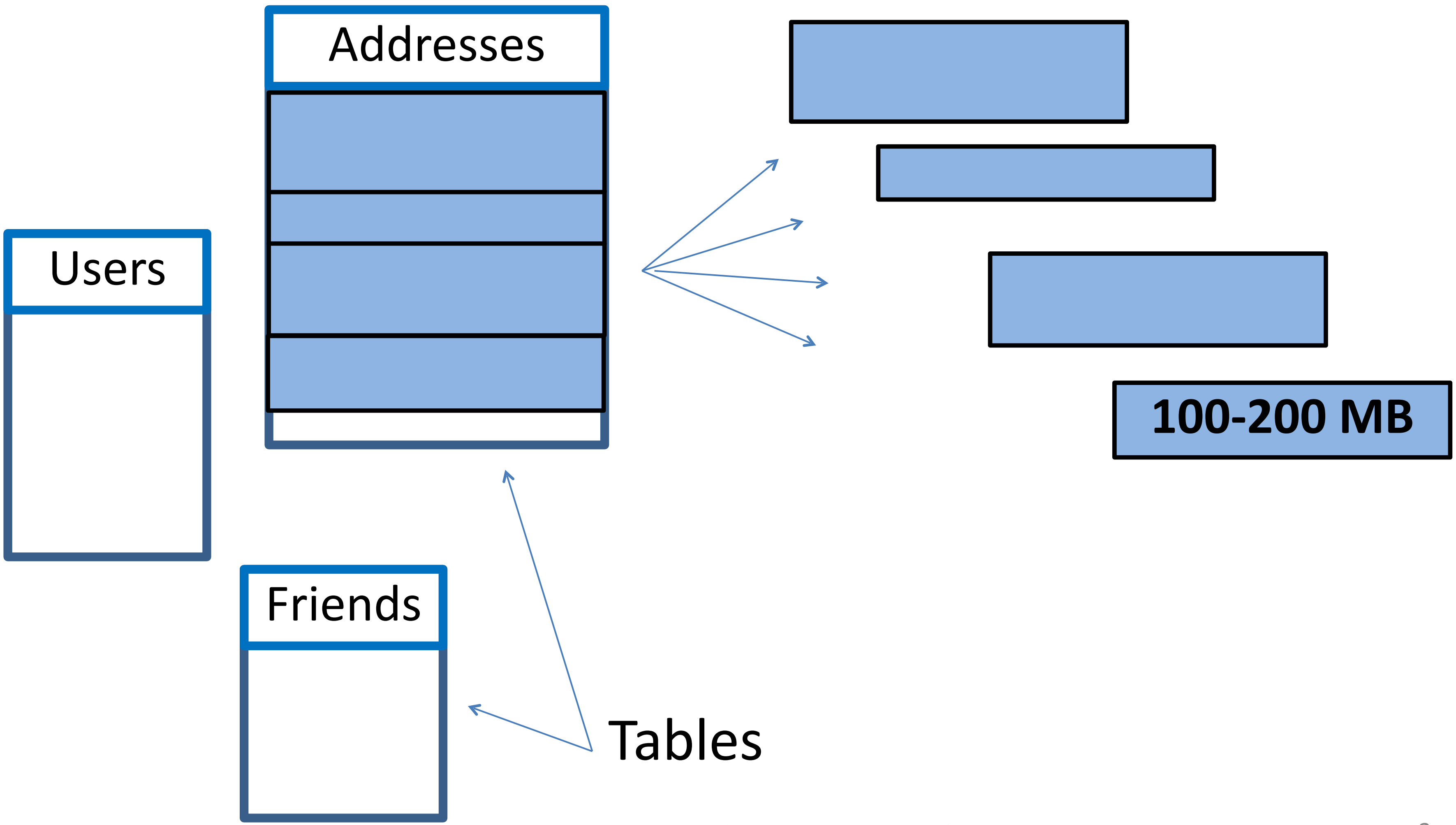
BigTable: Tables & Tablets

(Logical Organization)



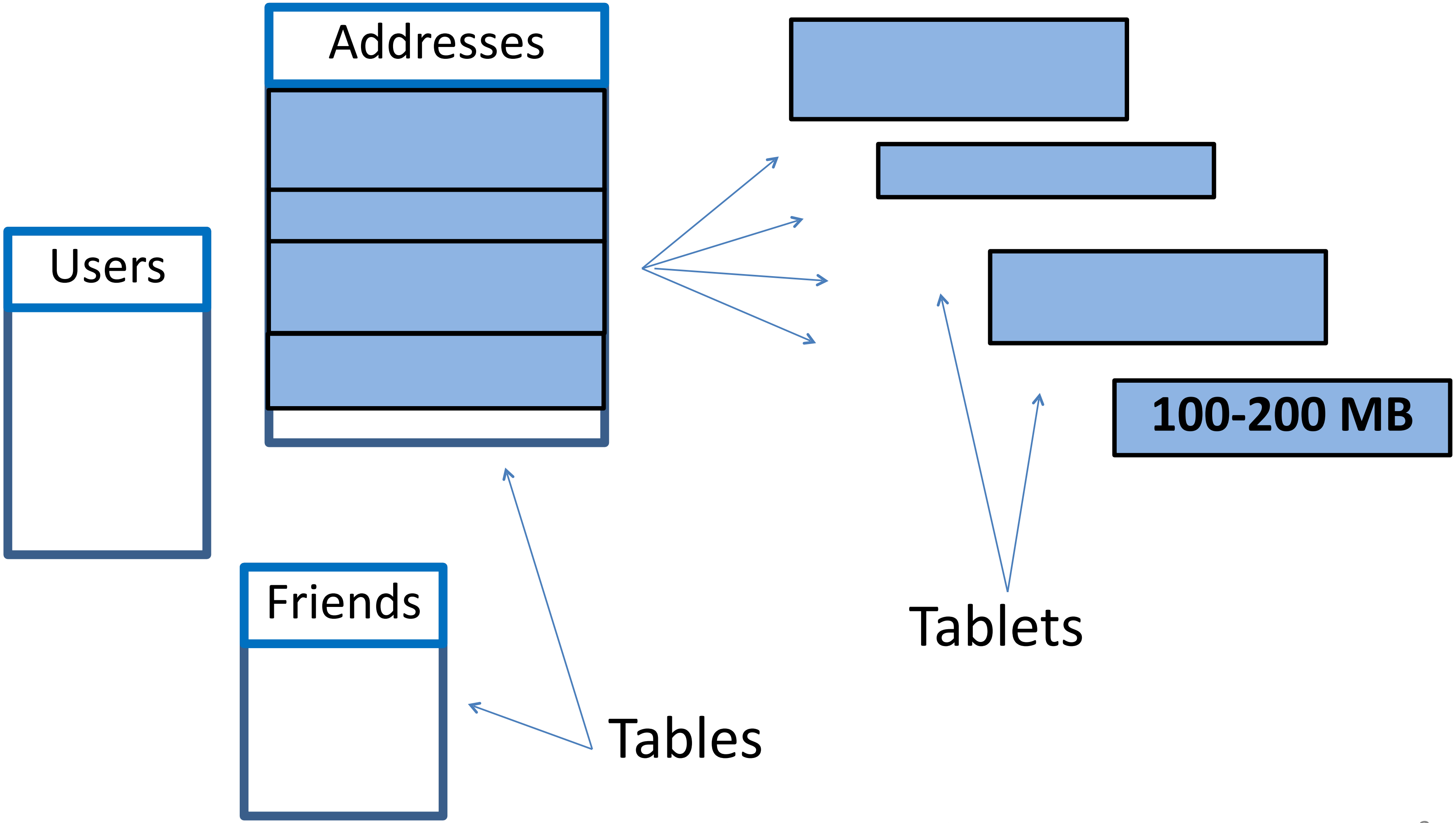
BigTable: Tables & Tablets

(Logical Organization)



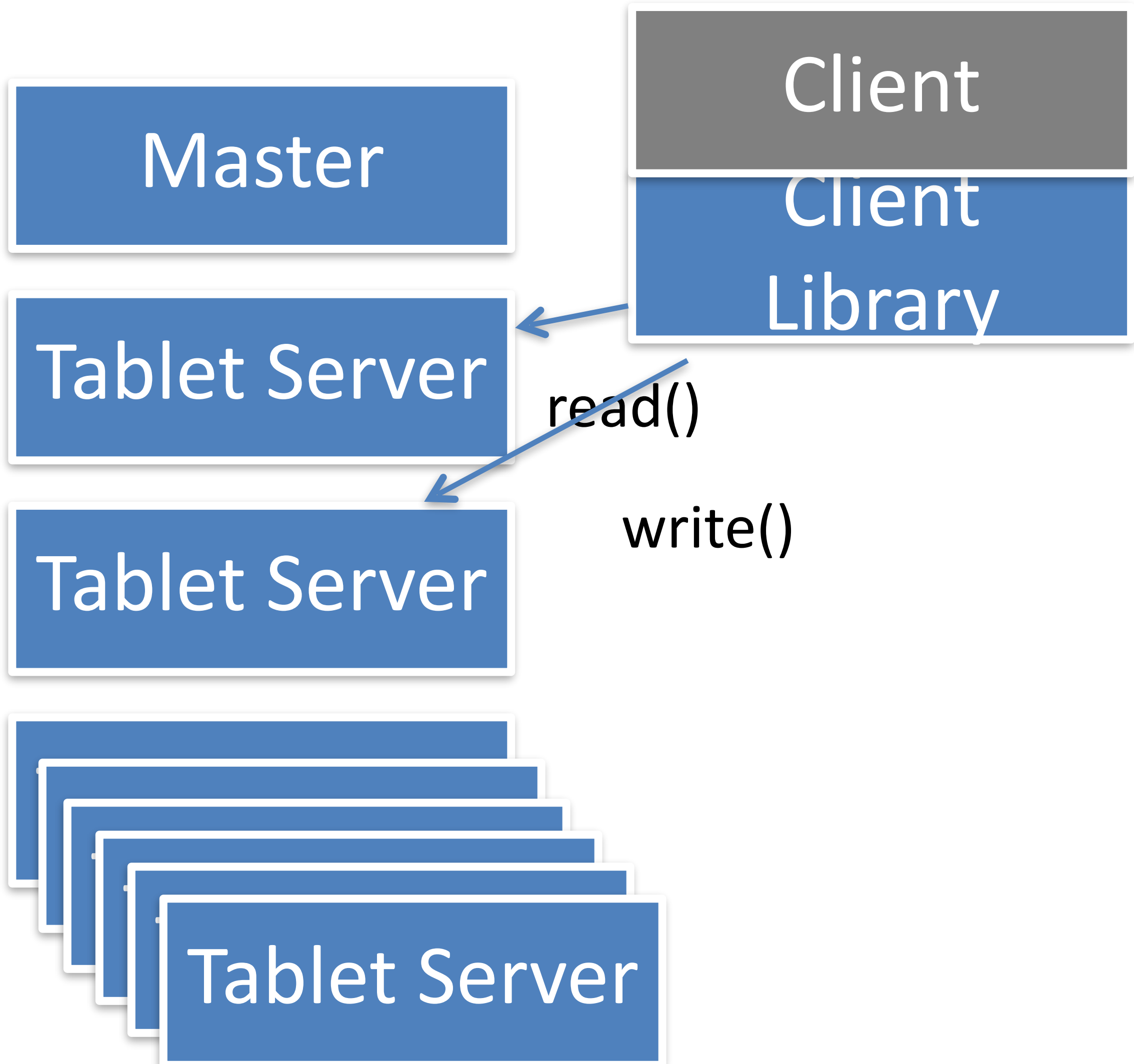
BigTable: Tables & Tablets

(Logical Organization)



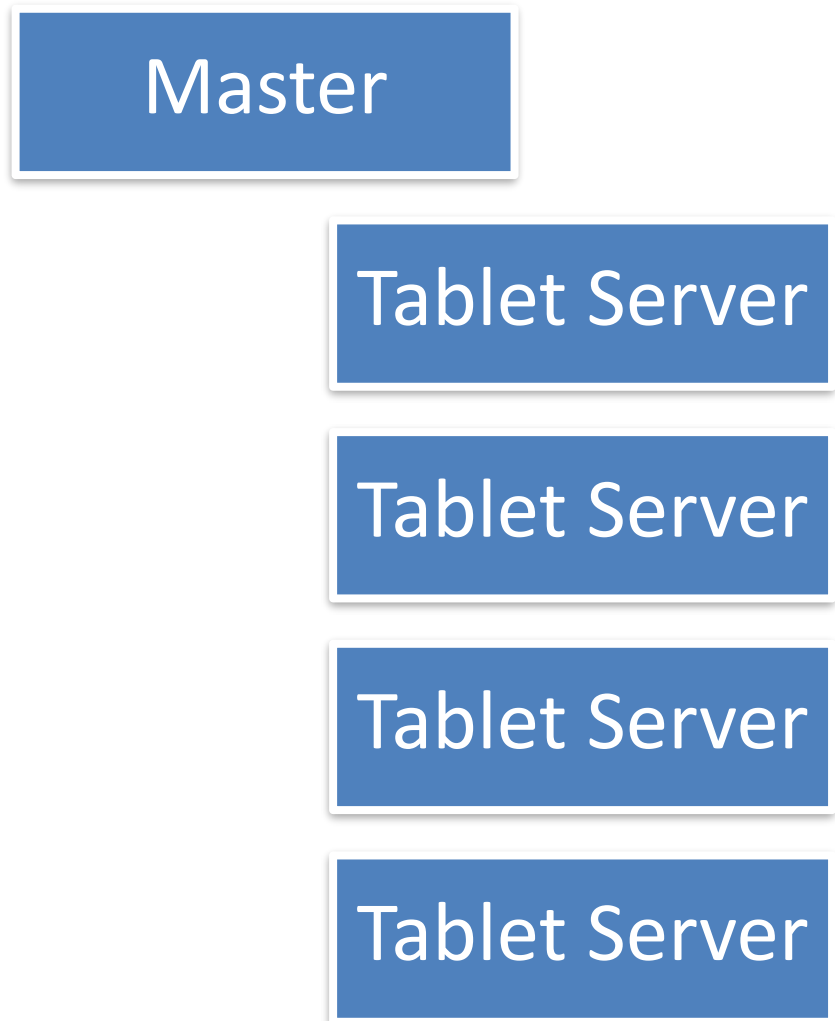
BigTable Components

- Client library
- Master
  - Metadata operations
  - Load balancing
- Tablet server
  - Data operations



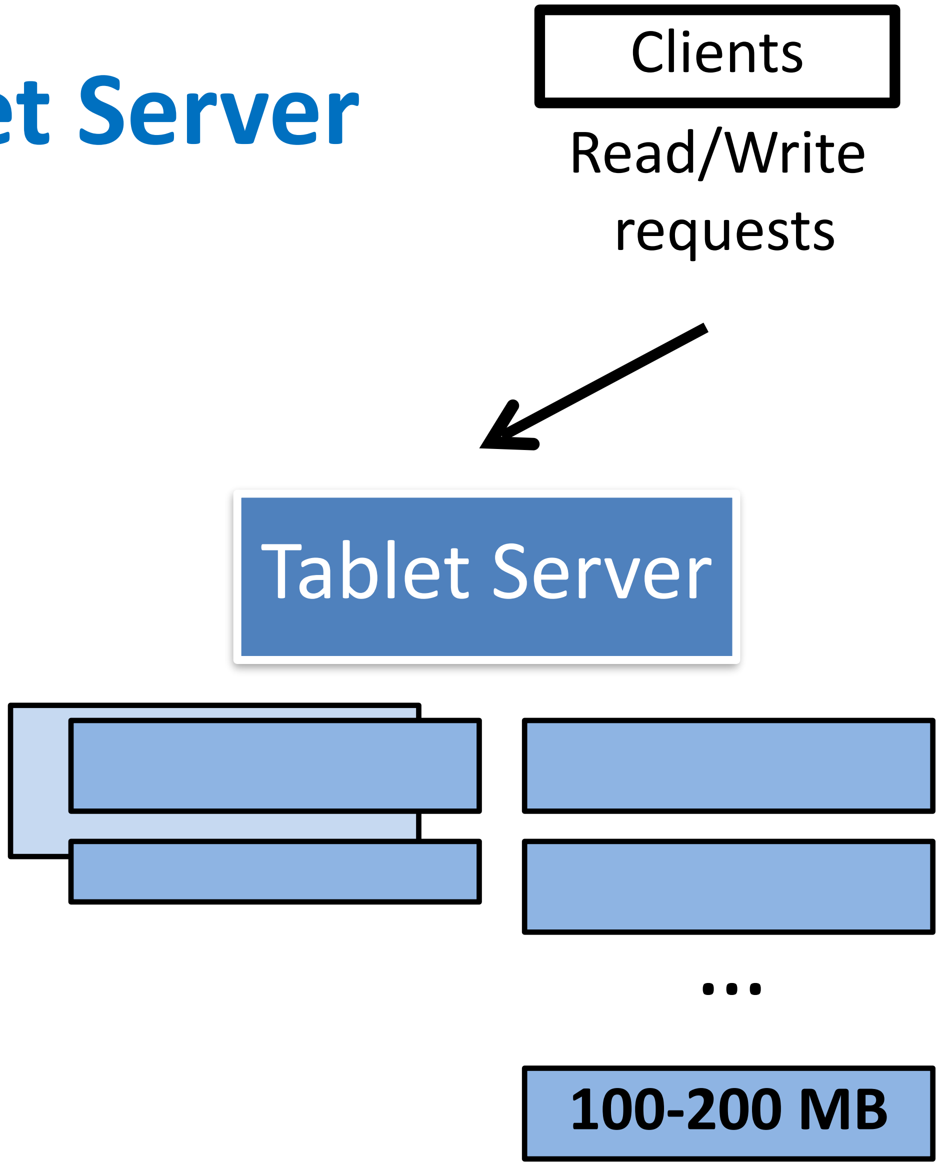
Master

- Assigns tablets to tablet servers
- Balance tablet server load
- Detects addition and expiration of tablet servers



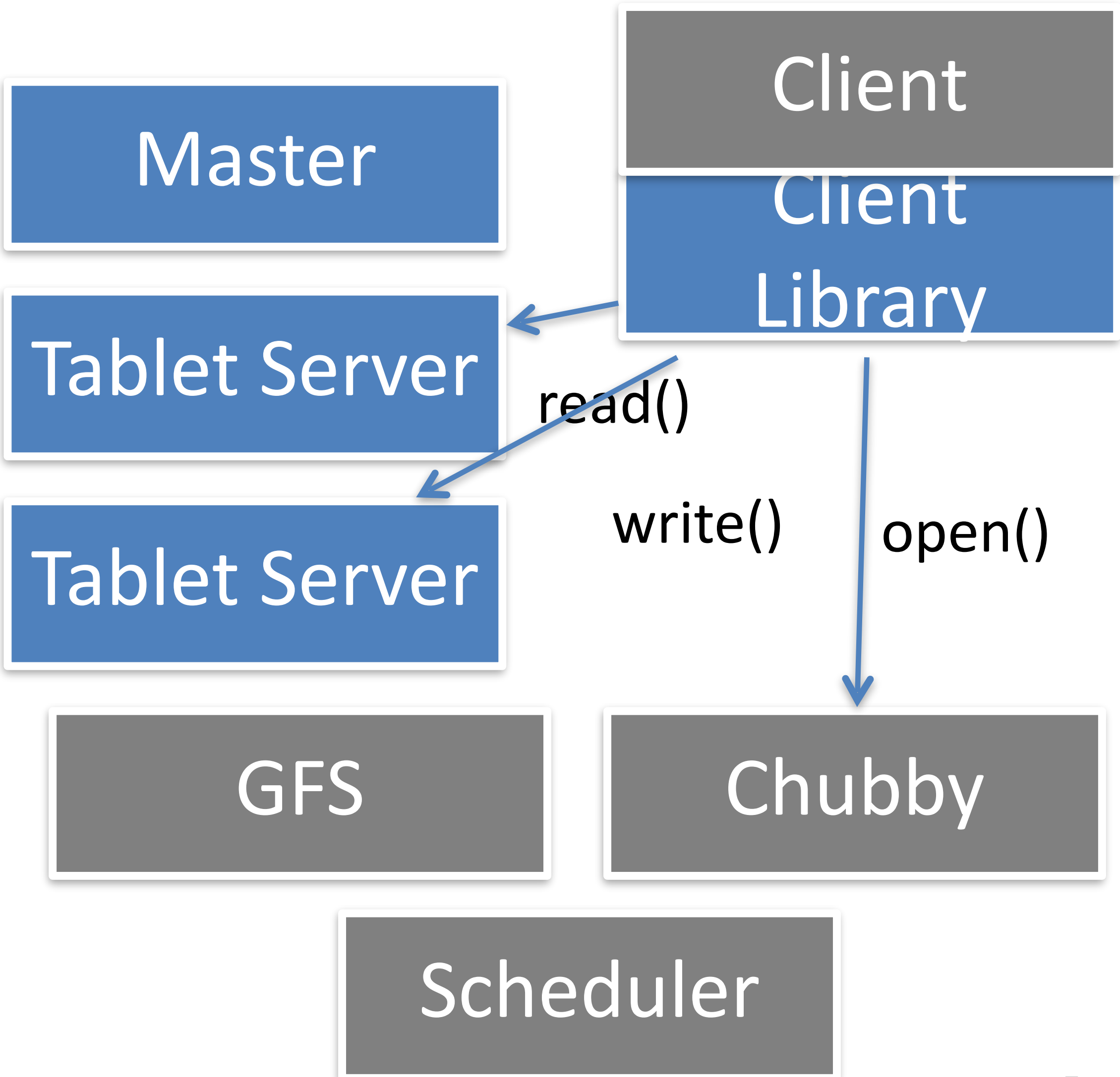
Tablet Server

- Manages a set of tablets (up to a thousand)
- Handles read and write requests for the tablets it manages
- Splits tablets that have grown too large – merges as well



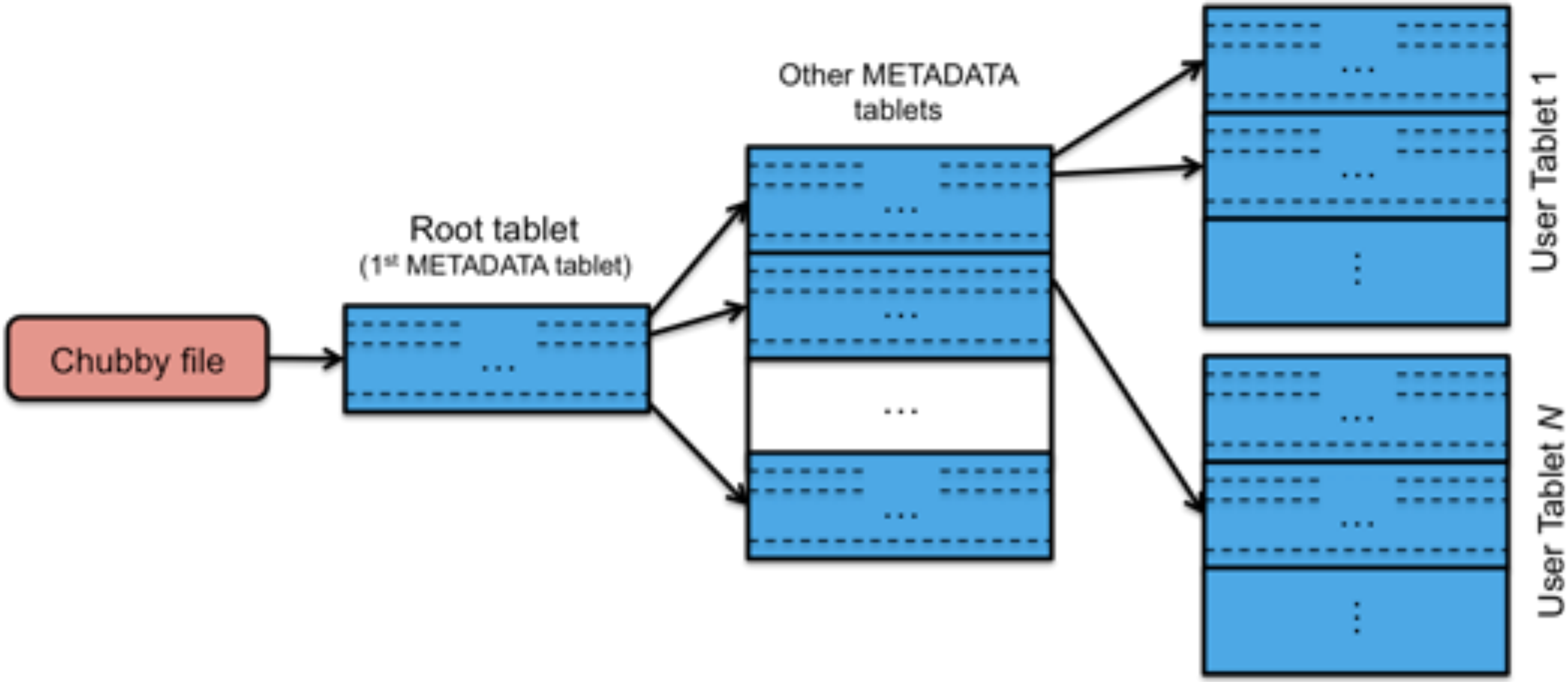
BigTable Building Blocks

- Chubby (*cf. Paxos Lecture*)
  - Lock service
  - Metadata storage
- GFS (*cf. GFS et al. Lecture*)
  - Data, log storage
  - Replication
  - Uses Sorted Strings Table files (SSTables)
- Scheduler
  - Monitoring
  - Failover



Tablet location hierarchy

Determine location of tablet– heavy caching



3 Levels: 2^17 (METADATA tablets) * 2^17 (user tablets) = 2^34 tablets

Apache HBase

- Open-source re-implementation of BigTable concepts
- E.g., Facebook Messenger uses HBase

- Different names for similar components
  - GFS → HDFS
  - Chubby → ZooKeeper
  - BigTable → HBase
  - MapReduce → Hadoop

HBase architecture overview I

- Client library
  - Issues put, get, delete operations
- ZooKeeper (Chubby)
  - Distributed lock service for HBase components
  - Based on ZAB – ZooKeeper Atomic Broadcast (Paxos)
- HRegion (tablet)
  - Tables are split into multiple key-regions

"*Because coordinating distributed systems is a zoo*."

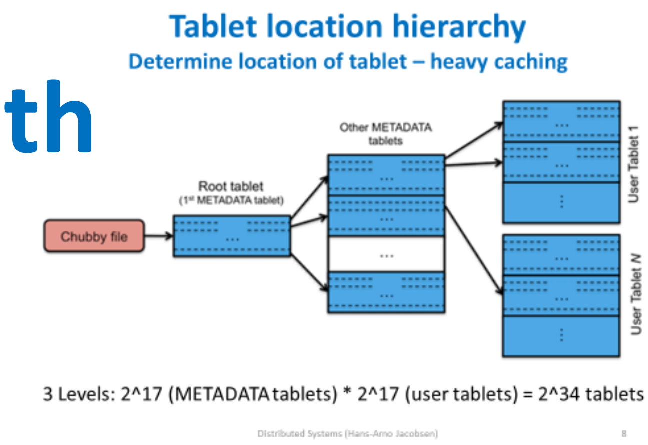
HBase architecture overview II

- HRegionServer (tablet server)
  - Processes operations for data / key-regions (tablets)
  - Can host multiple key-regions (tablets)
  - Answers client requests
- HMaster (master)
  - Coordinates components
    - Startup, shutdown, failure of region servers (tablet servers)
    - Opens, closes, assigns, moves regions (tablets)
  - Not on read or write path

HBase architecture overview III

- Write Ahead Log (WAL)
  - For failure recovery, persist a log of operations, before anything
- MemStore
  - Keep data in main memory, periodically sync to disk
  - Retain hot data in main memory
- HDFS (GFS)
  - Underlying distributed file system
  - Table data is stored as HFile format (SSTable in GFS)
  - Replicates data over multiple data nodes

HBase read-path



Client wants to read key *k1* from table *t1*

Client

ZooKeeper

Region Server 1

-ROOT-: Addresses of meta tablets

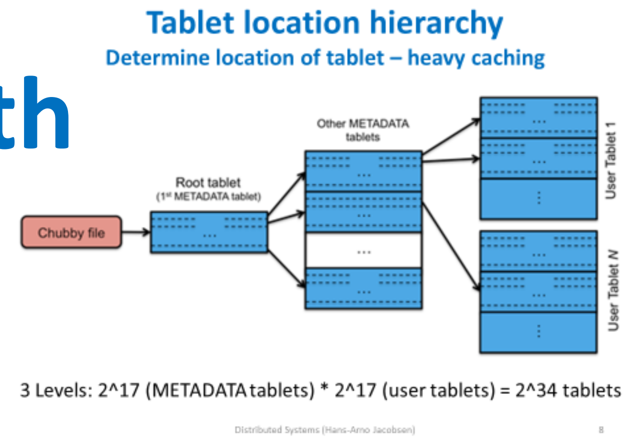
Region Server 2

.META. 1: Addresses of key ranges

Region Server 3

Region 1 for Table t1: Values for key range

HBase read-path



Client wants to read key *k1* from table *t1*

Client

1 Requests -ROOT-

ZooKeeper

Region Server 1

-ROOT-: Addresses of meta tablets

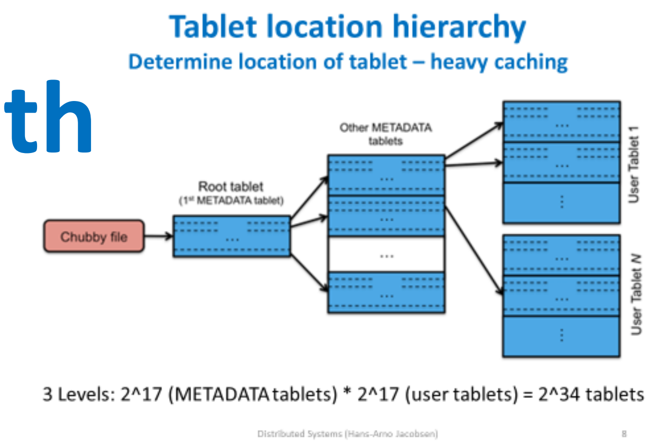
Region Server 2

.META. 1: Addresses of key ranges

Region Server 3

Region 1 for Table t1: Values for key range

HBase read-path



Client wants to read key *k1* from table *t1*

Client

Cache Addresses

1 Requests -ROOT-

ZooKeeper

Region Server 1

-ROOT-: Addresses of meta tablets

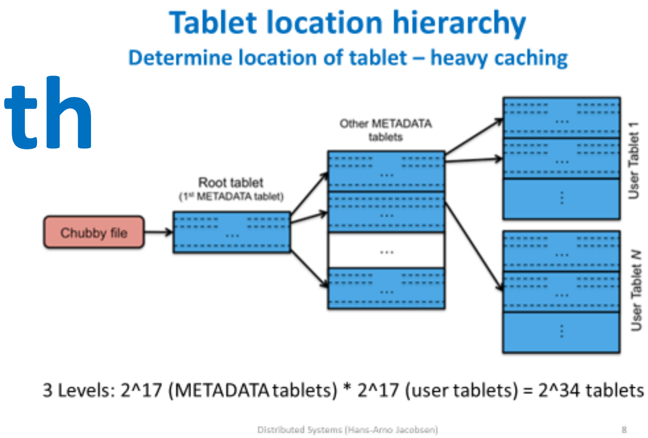
Region Server 2

.META. 1: Addresses of key ranges

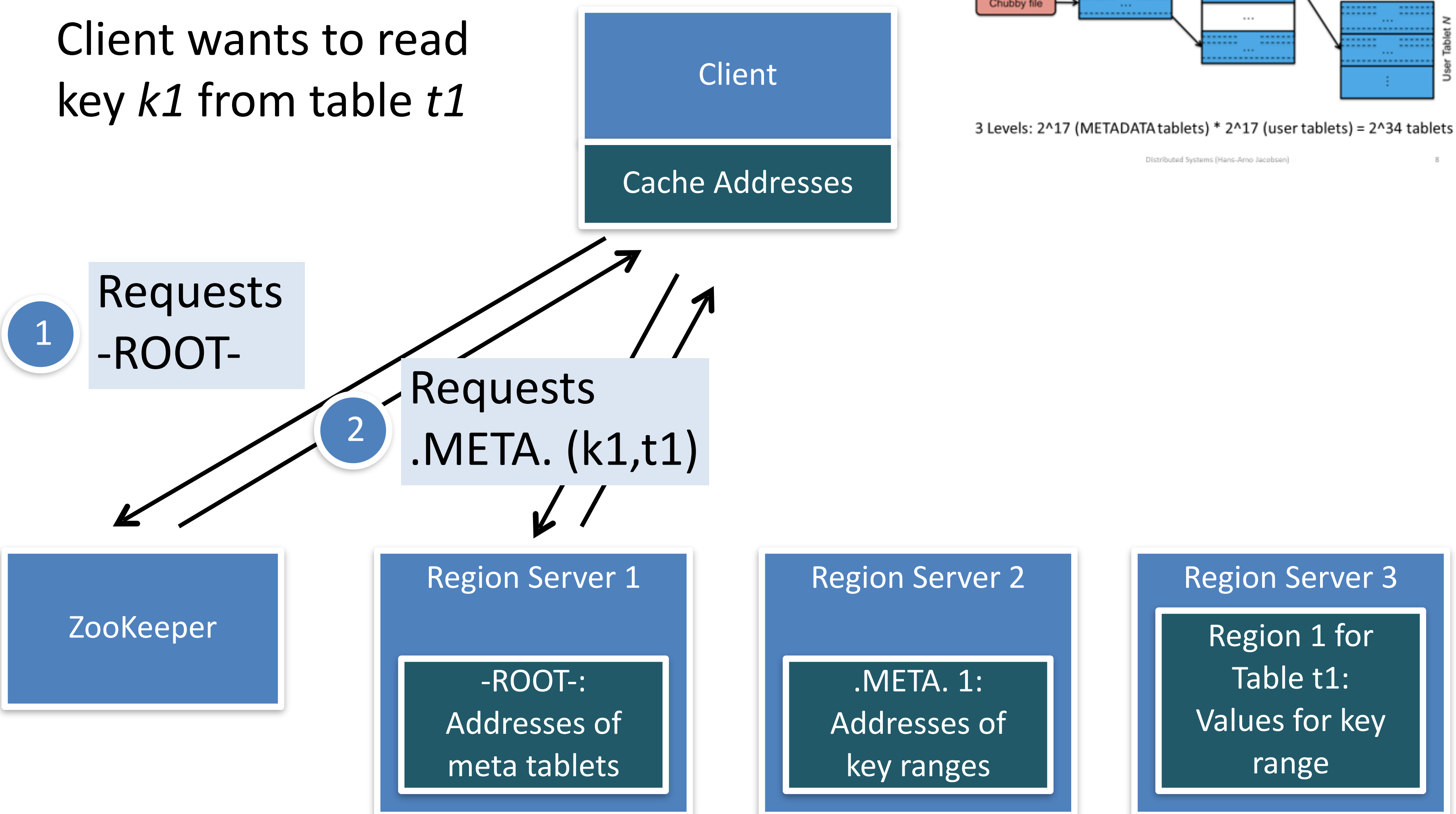
Region Server 3

Region 1 for Table t1: Values for key range

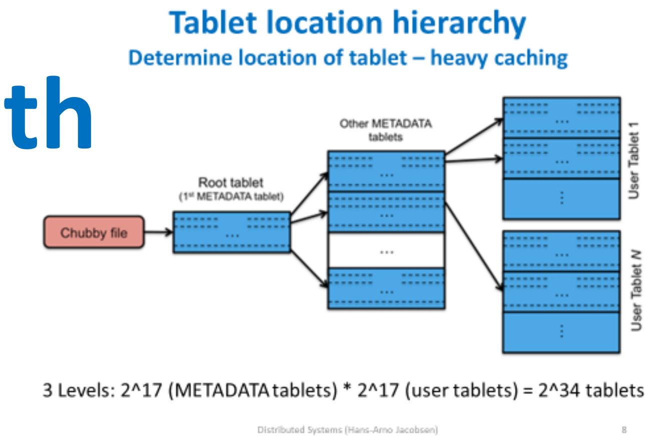
HBase read-path



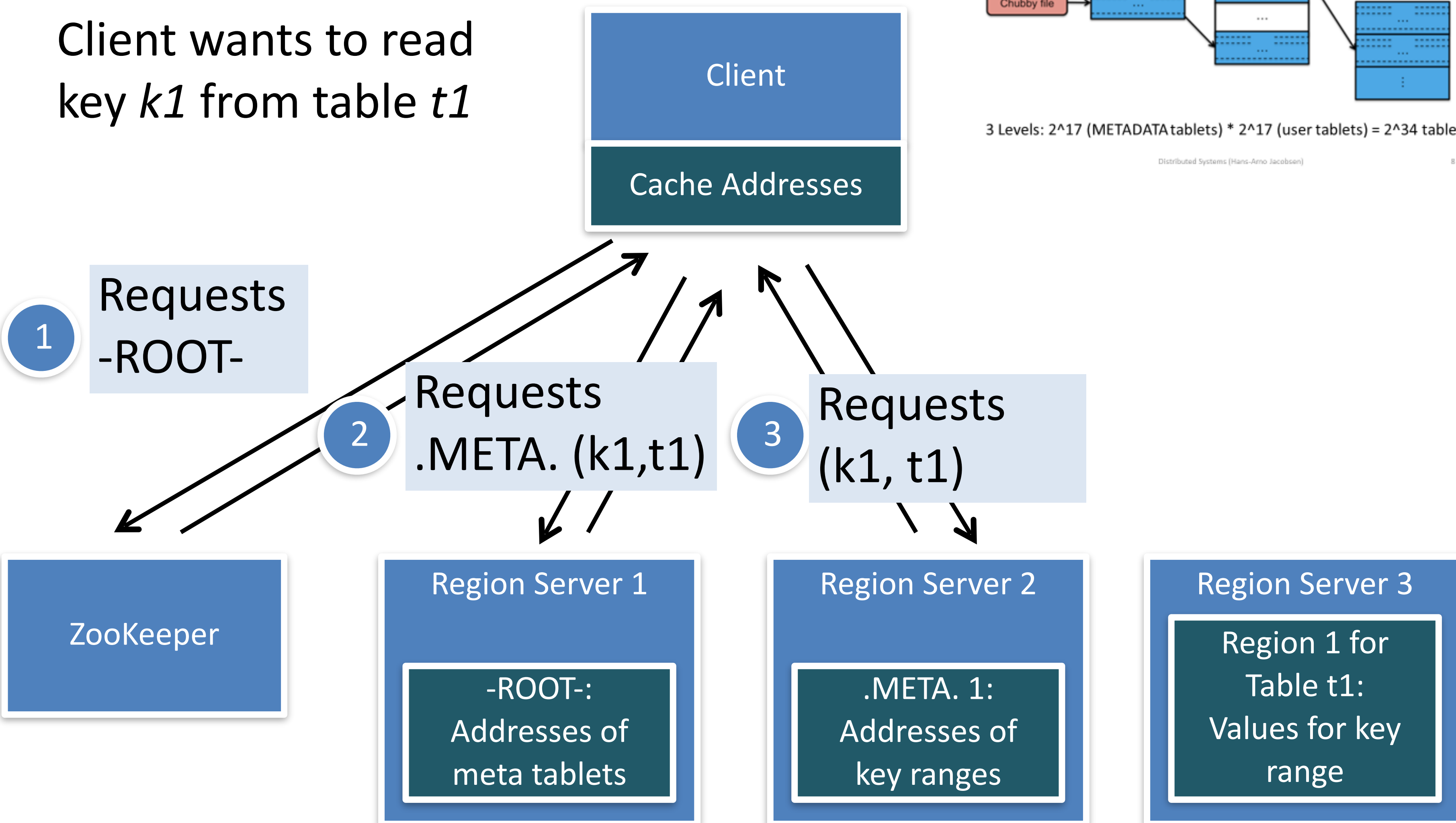
Client wants to read key *k1* from table *t1*



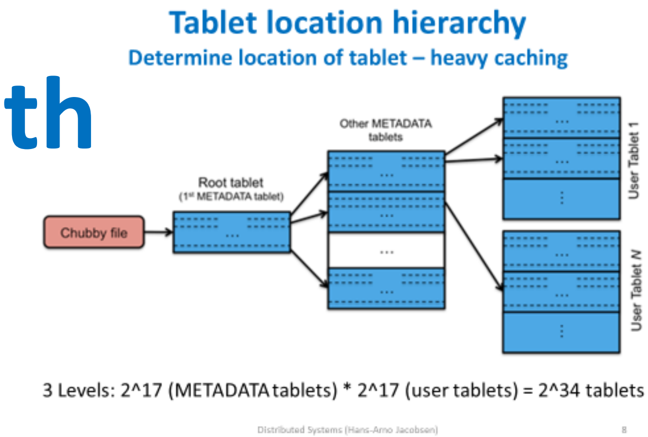
HBase read-path



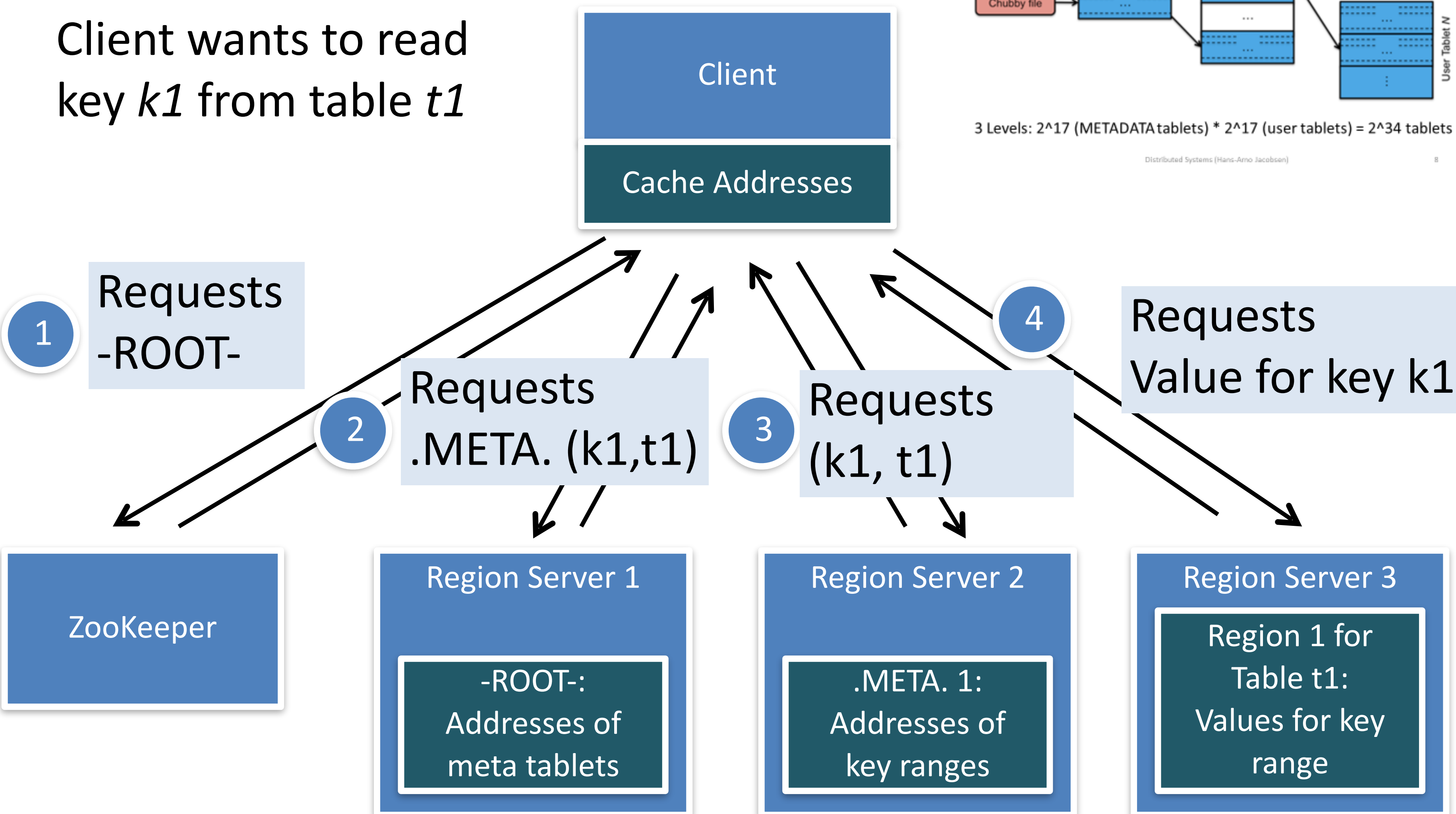
Client wants to read key *k1* from table *t1*



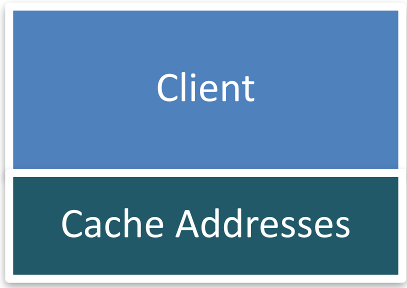
HBase read-path



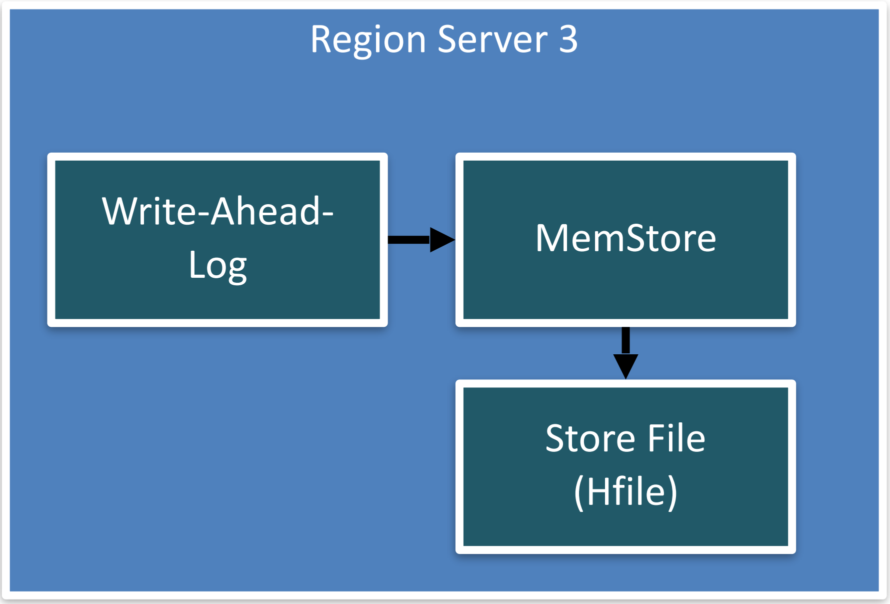
Client wants to read key *k1* from table *t1*



HBase write-path

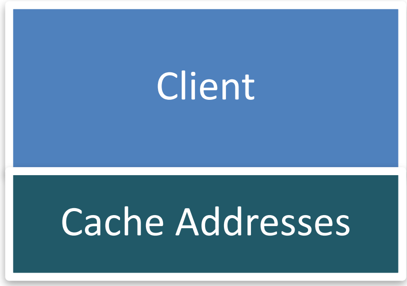


Client wants to store a value under key *k1* in table *t1*

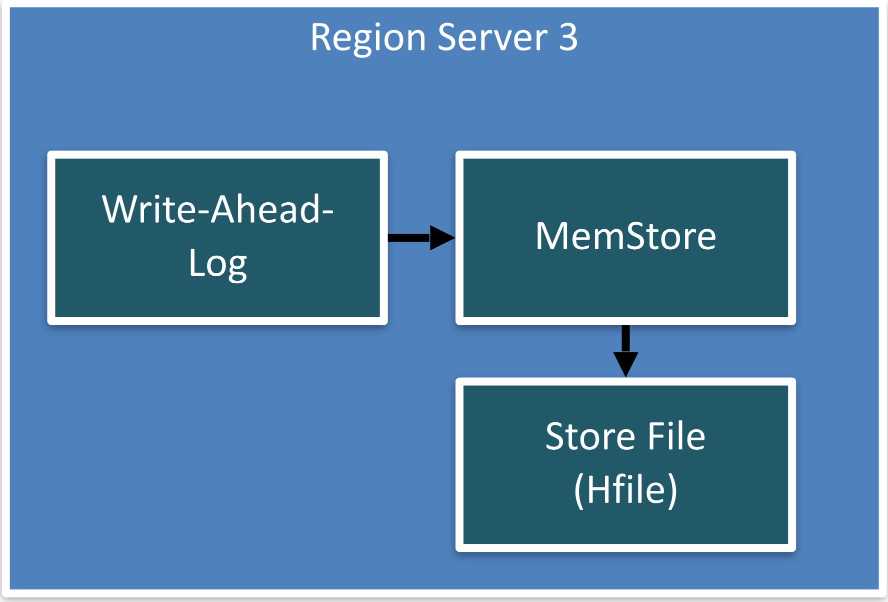


HBase write-path

1 Perform the same four steps as on read-path. If address of region server (RS) is cached go directly to RS



Client wants to store a value under key *k1* in table *t1*

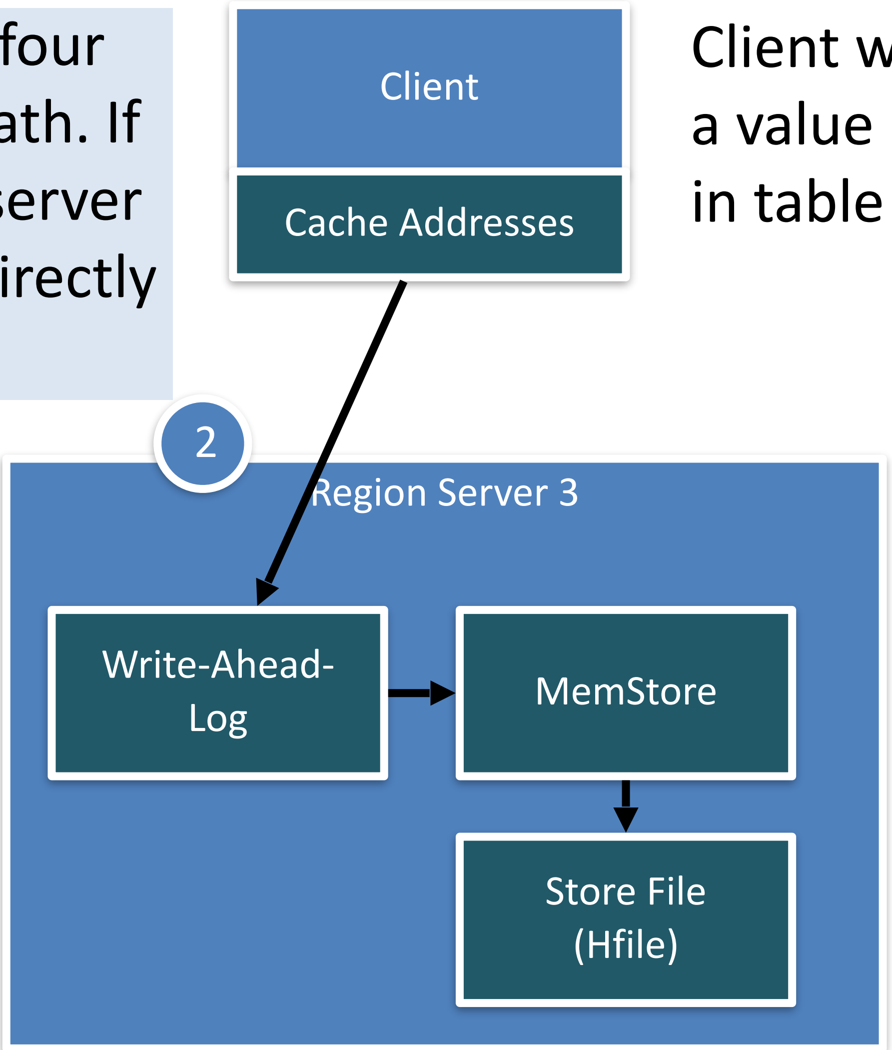


HBase write-path

1 Perform the same four steps as on read-path. If address of region server (RS) is cached go directly to RS

Client wants to store a value under key *k1* in table *t1*

2 Send put(k1, value) to RS



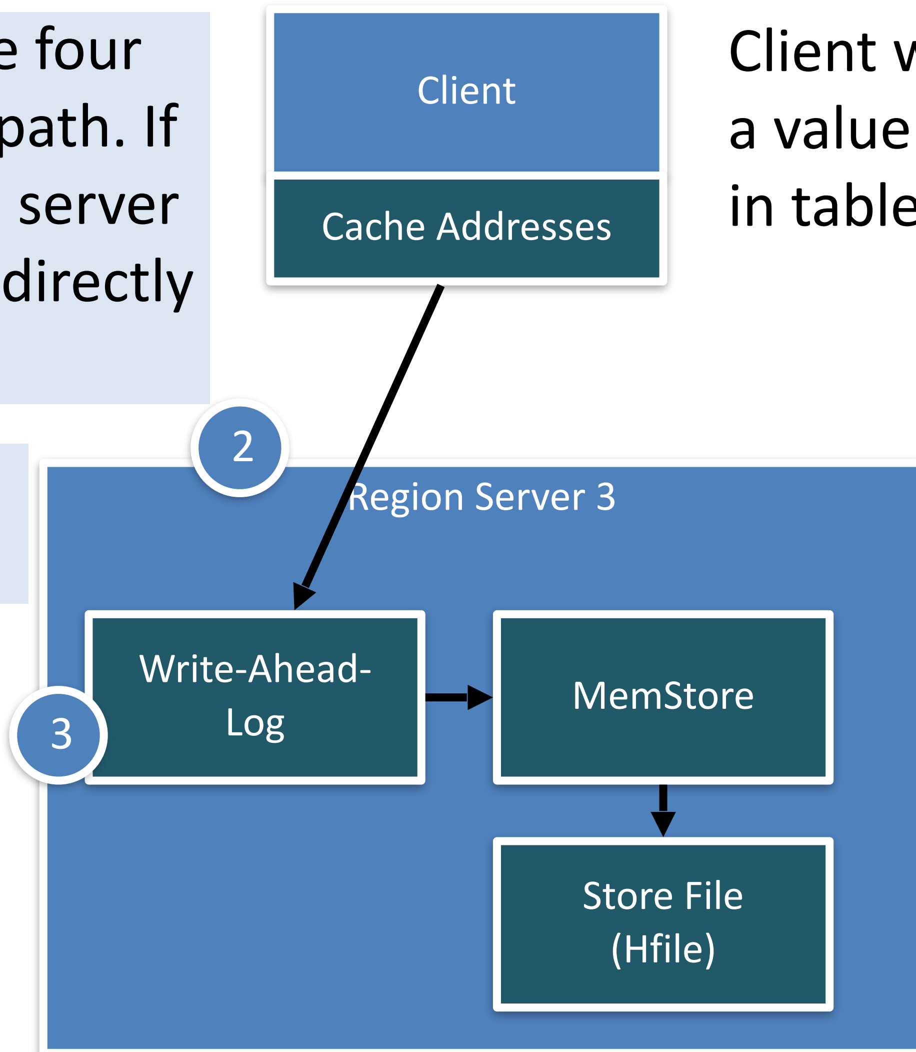
HBase write-path

1 Perform the same four steps as on read-path. If address of region server (RS) is cached go directly to RS

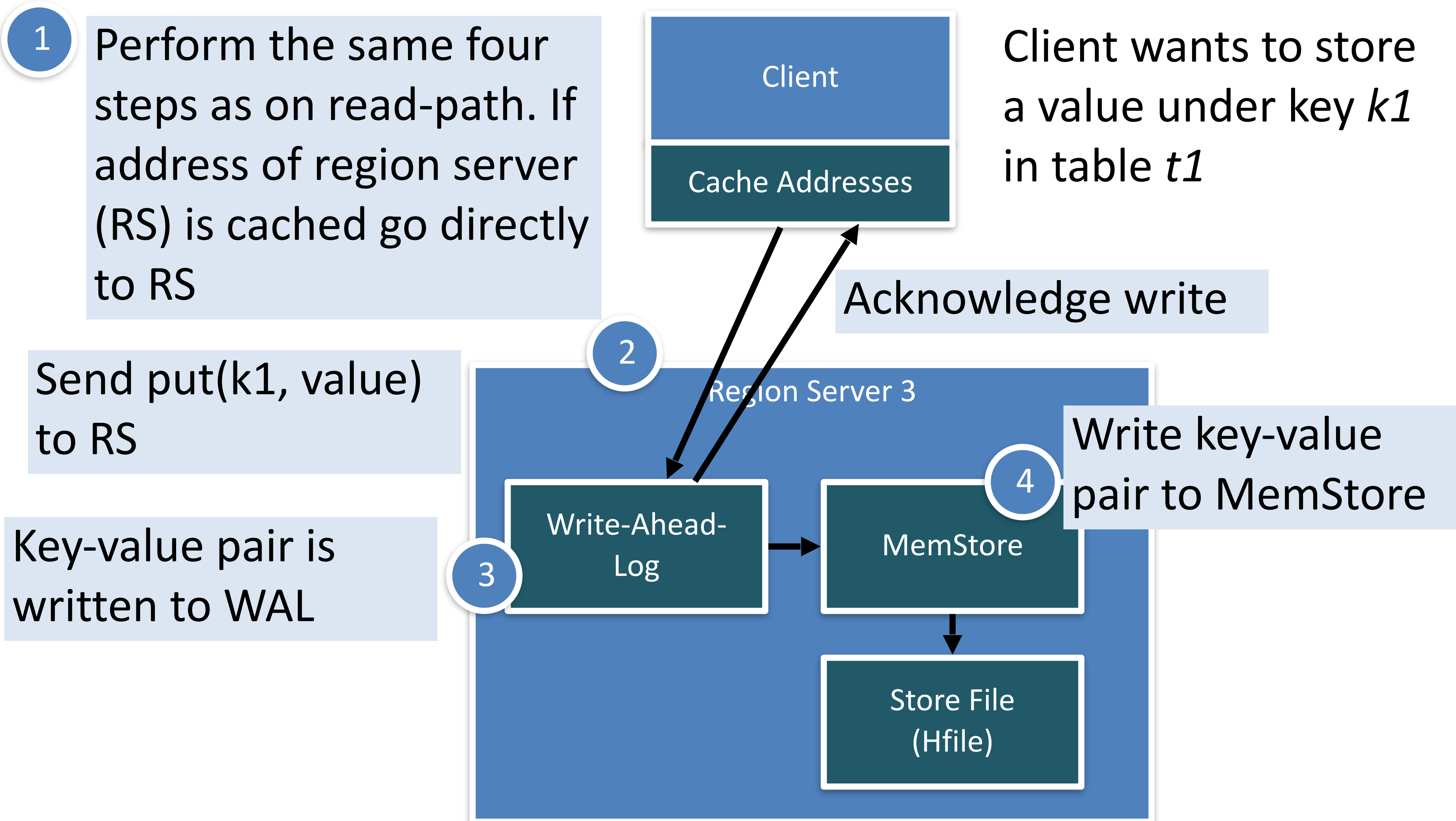
2 Send put(k1, value) to RS

3 Key-value pair is written to WAL

Client wants to store a value under key *k1* in table *t1*



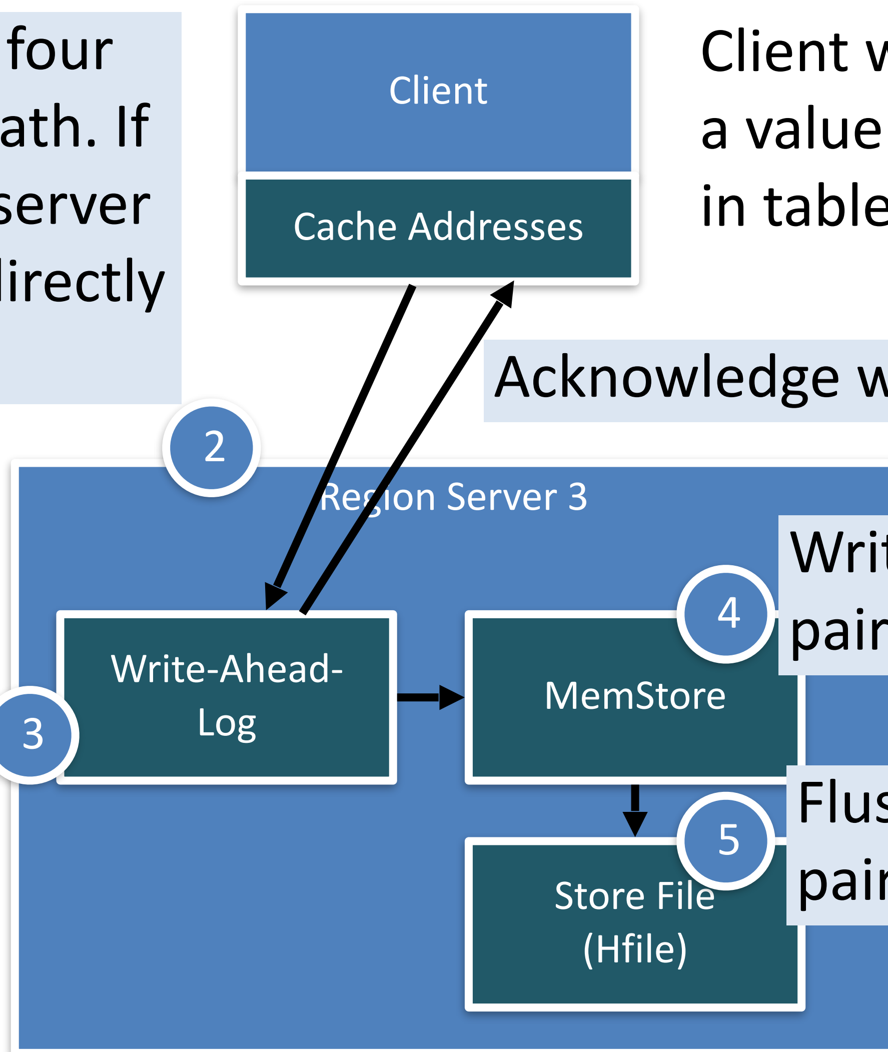
HBase write-path



HBase write-path

1 Perform the same four steps as on read-path. If address of region server (RS) is cached go directly to RS

Client wants to store a value under key *k1* in table *t1*



2 Send put(k1, value) to RS

3 Key-value pair is written to WAL

4 Write key-value pair to MemStore

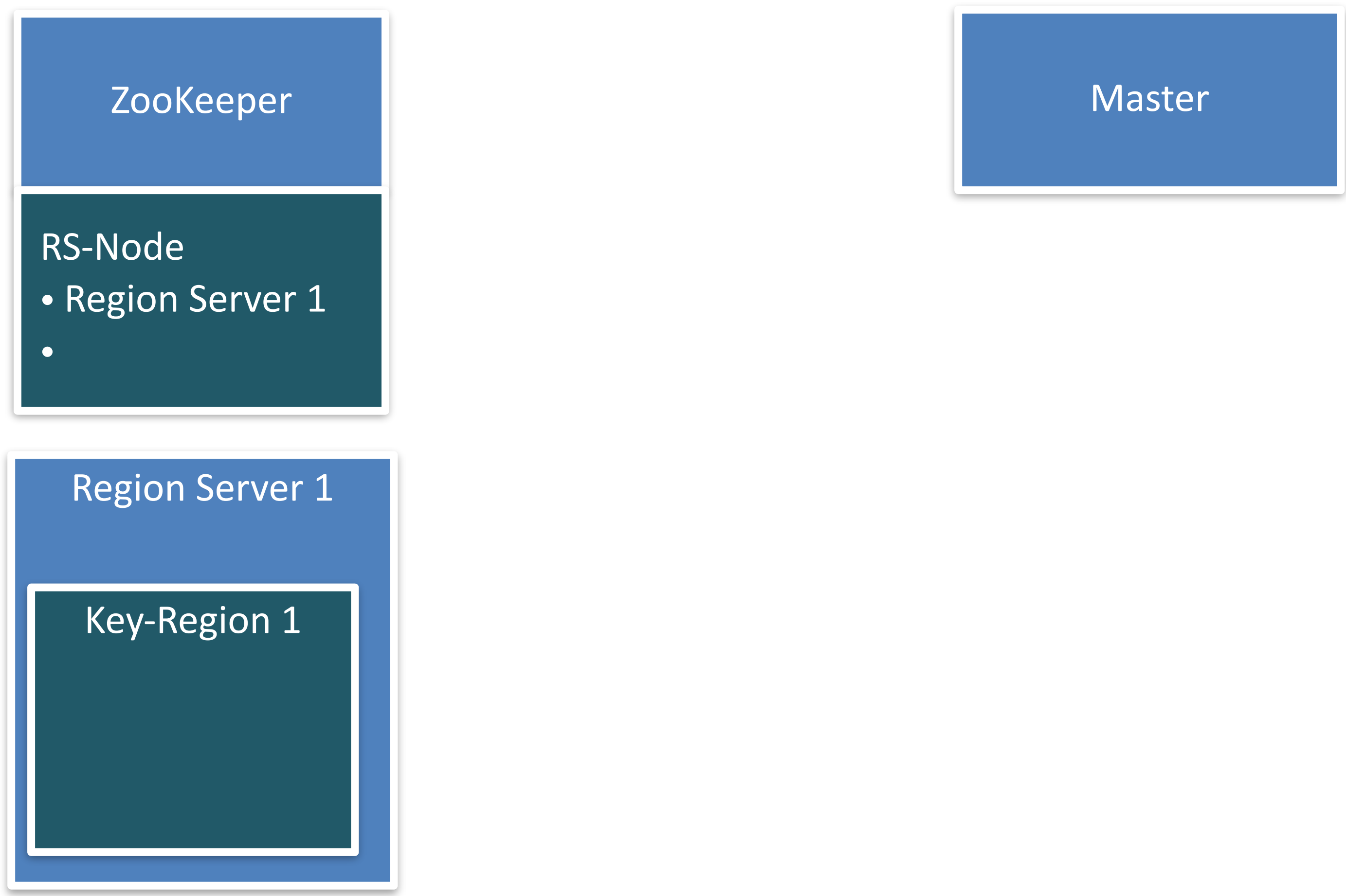
5 Flush key-value pair to HDFS

Acknowledge write

HBase: Scalability and fault tolerance

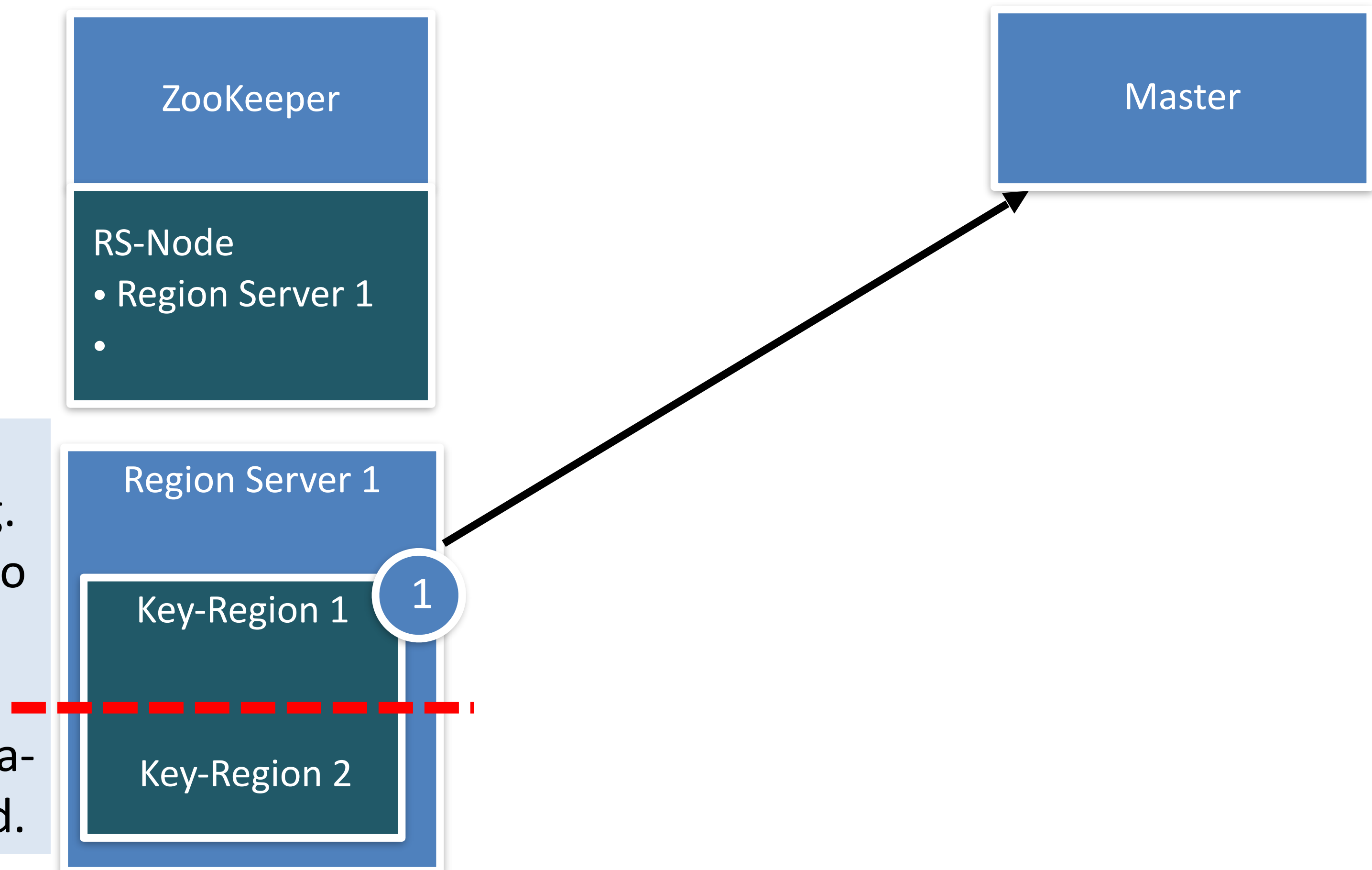
- Components can be added on-the-fly
- Add multiple backup master servers
  - Avoid single point of failure
  - In case of crash, backup master takes over
  - Leader election using ZooKeeper
- Add multiple region servers
  - **Horizontal** scalability
  - Master takes care of **load balancing**

HBase scalability

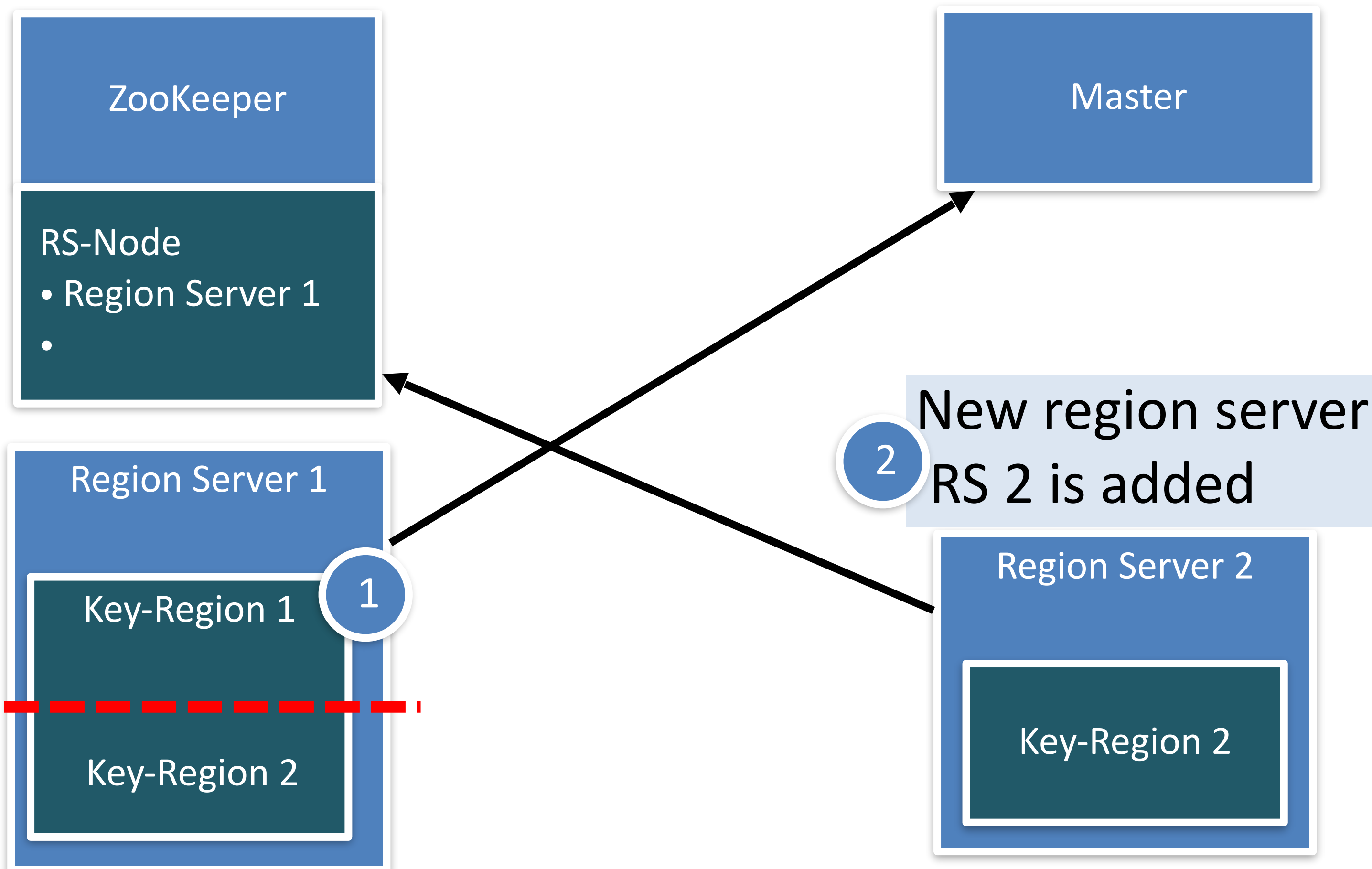


HBase scalability

- Table of RS 1 grows too big.
- RS splits it into 2 regions.
- Master is notified, Meta-table updated.

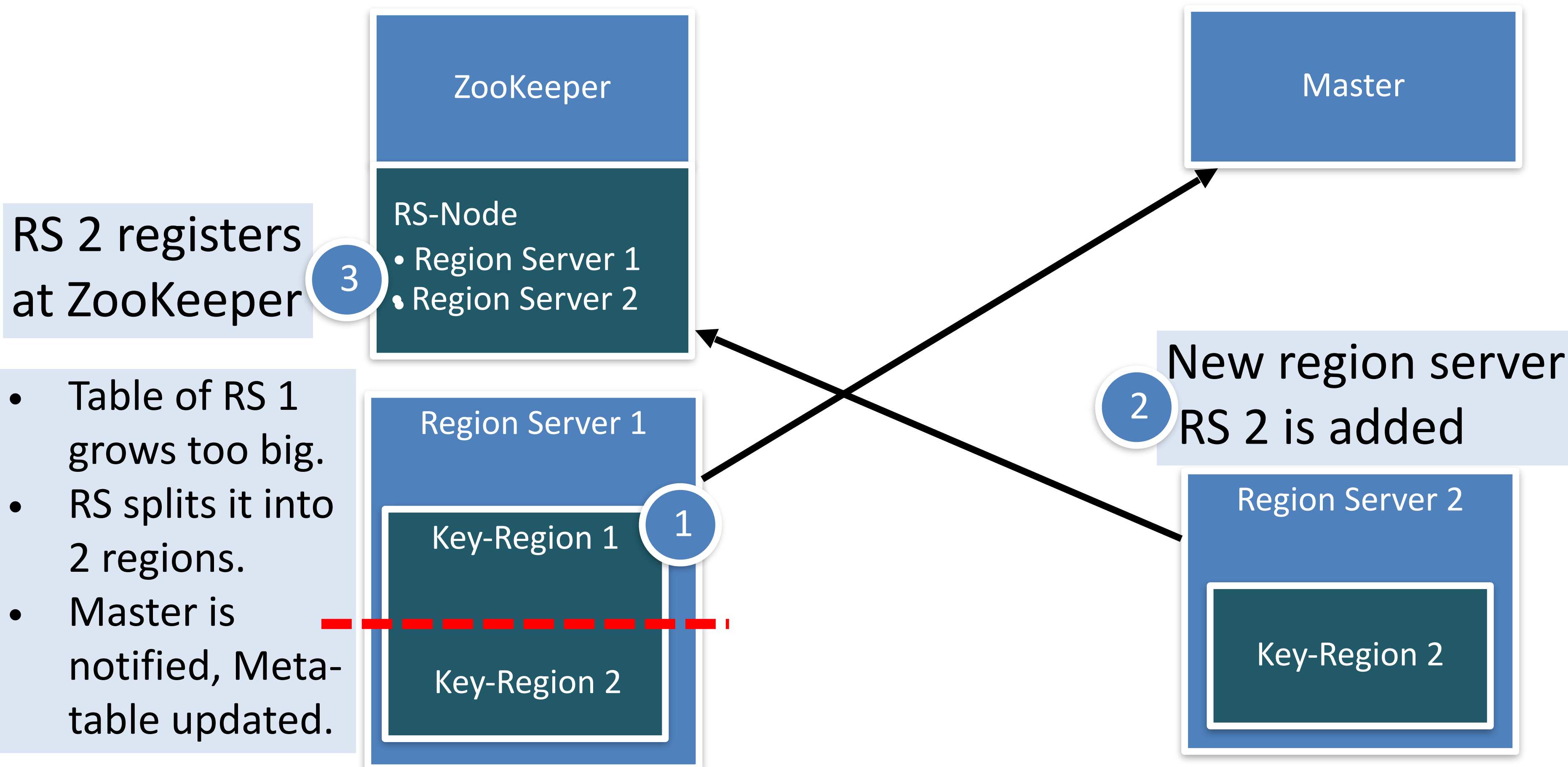


HBase scalability



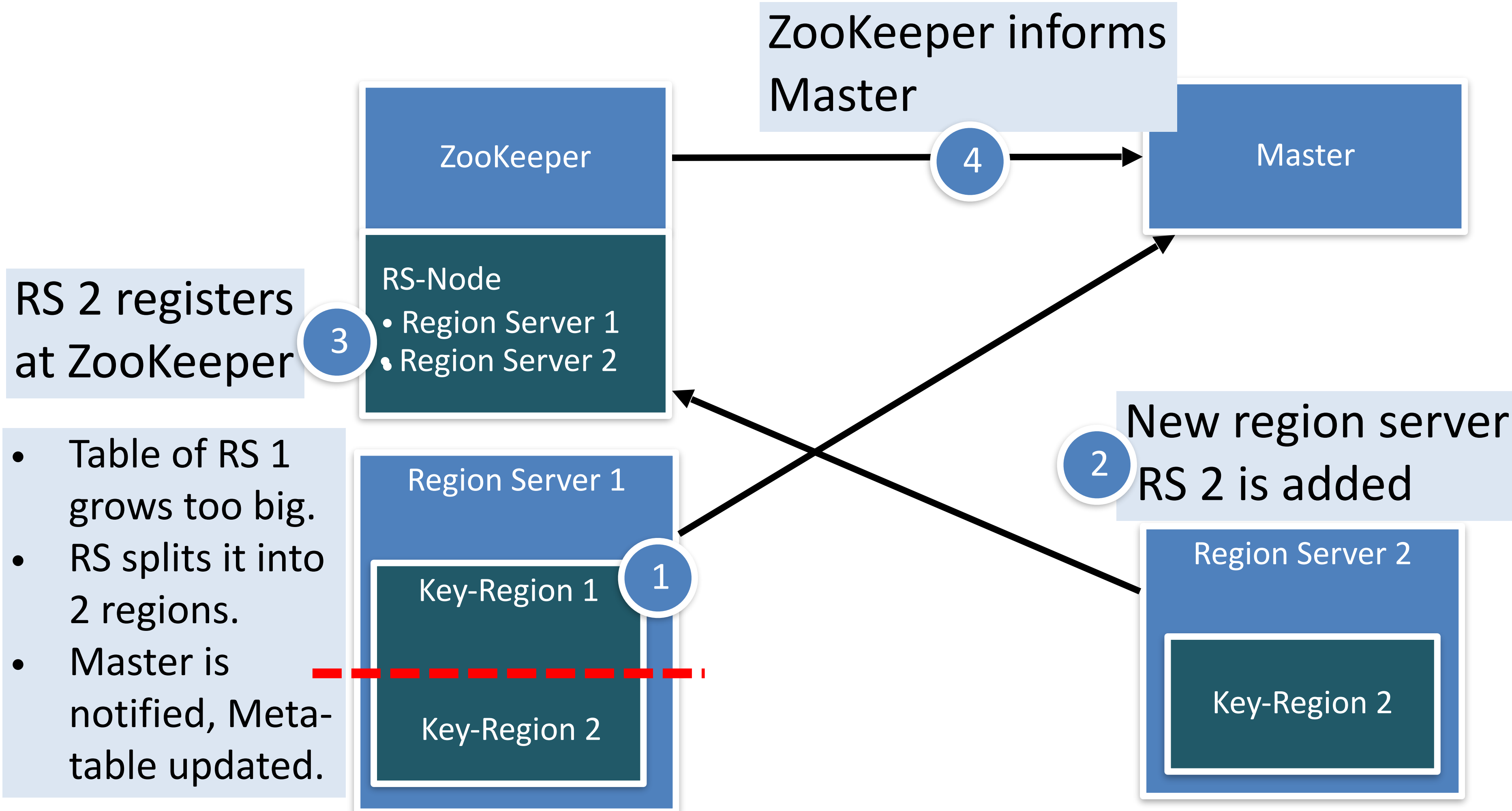
- Table of RS 1 grows too big.
- RS splits it into 2 regions.
- Master is notified, Meta-table updated.

HBase scalability



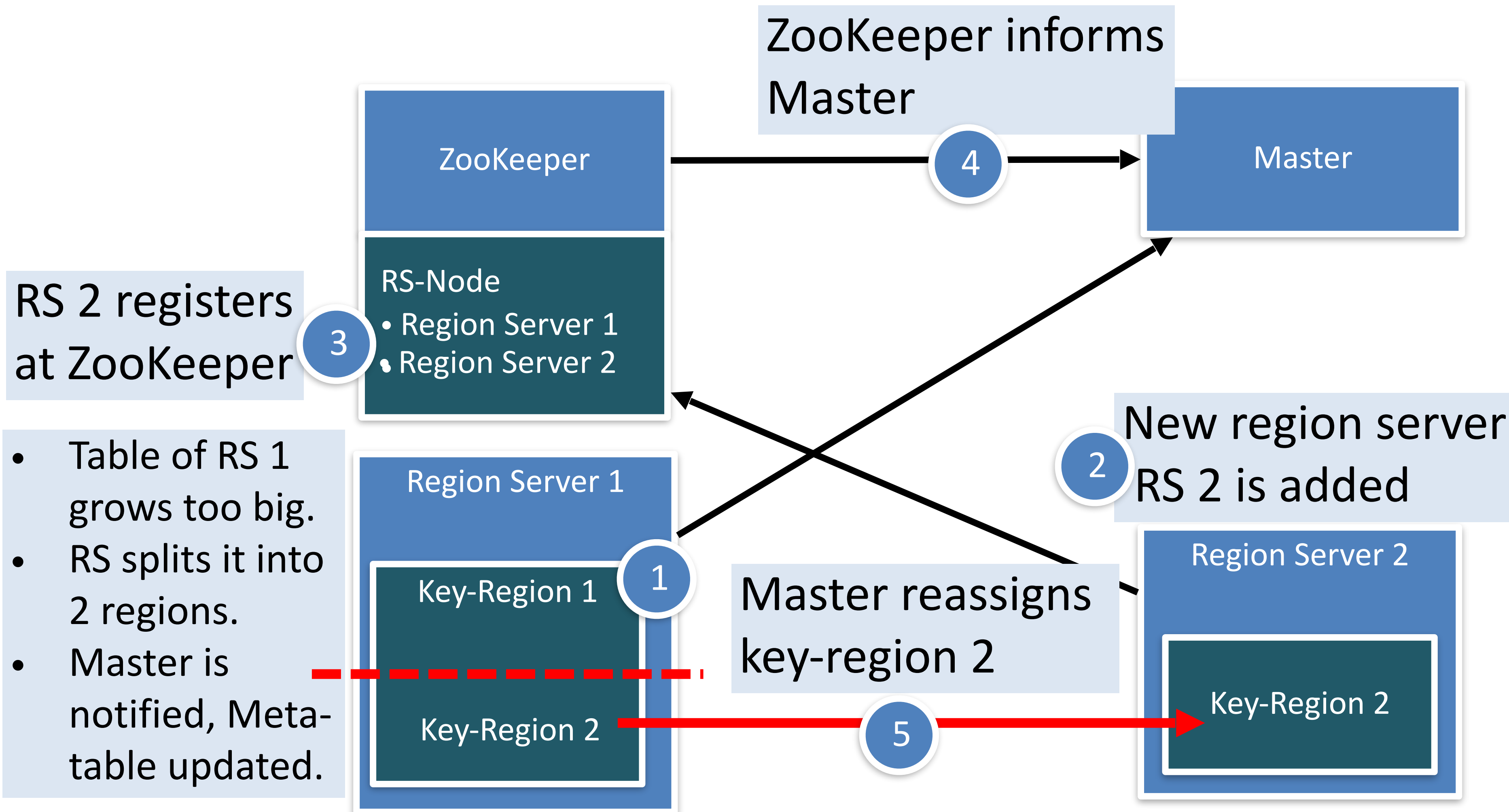
- Table of RS 1 grows too big.
- RS splits it into 2 regions.
- Master is notified, Meta-table updated.

HBase scalability

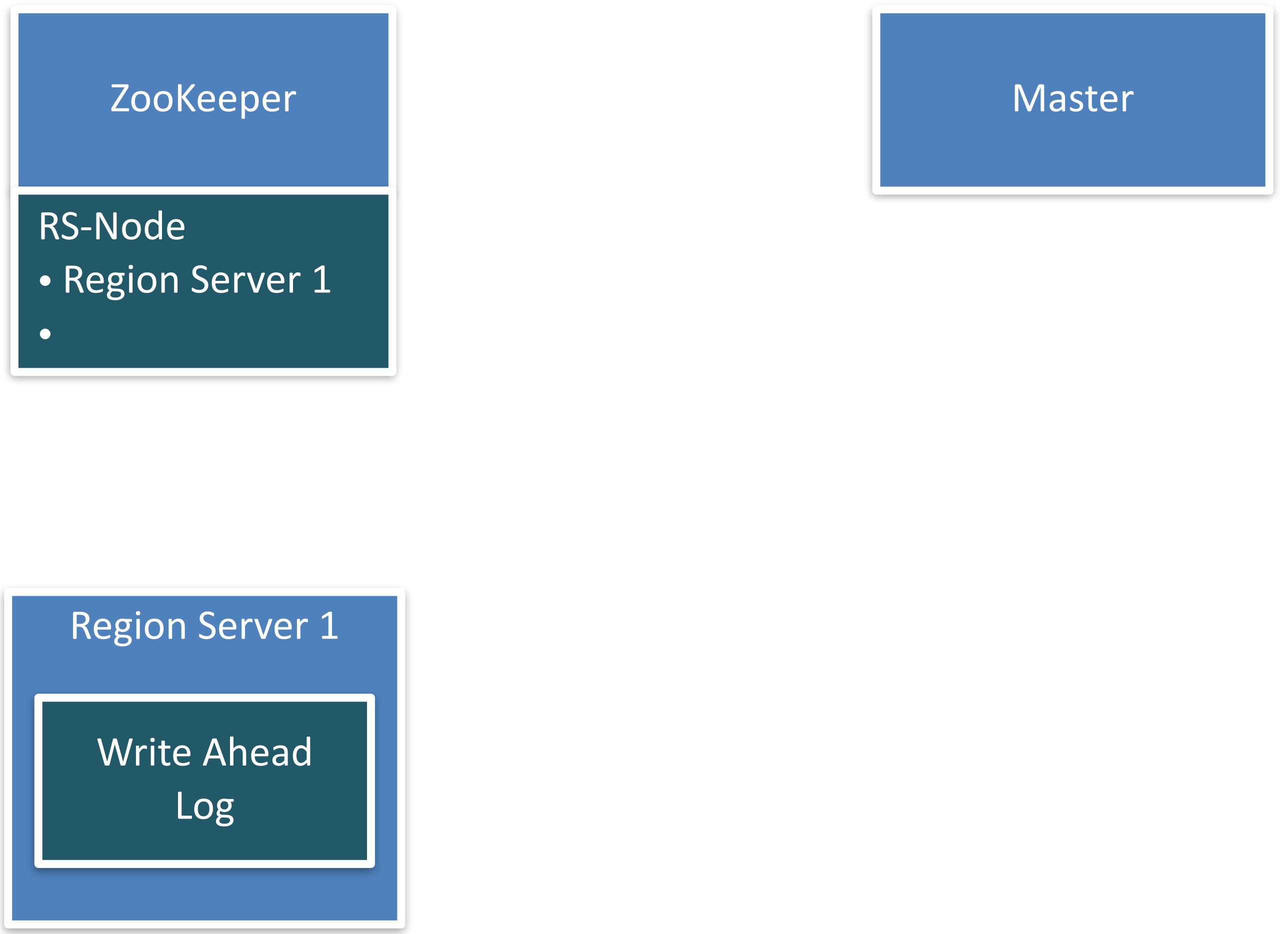


- Table of RS 1 grows too big.
- RS splits it into 2 regions.
- Master is notified, Meta-table updated.

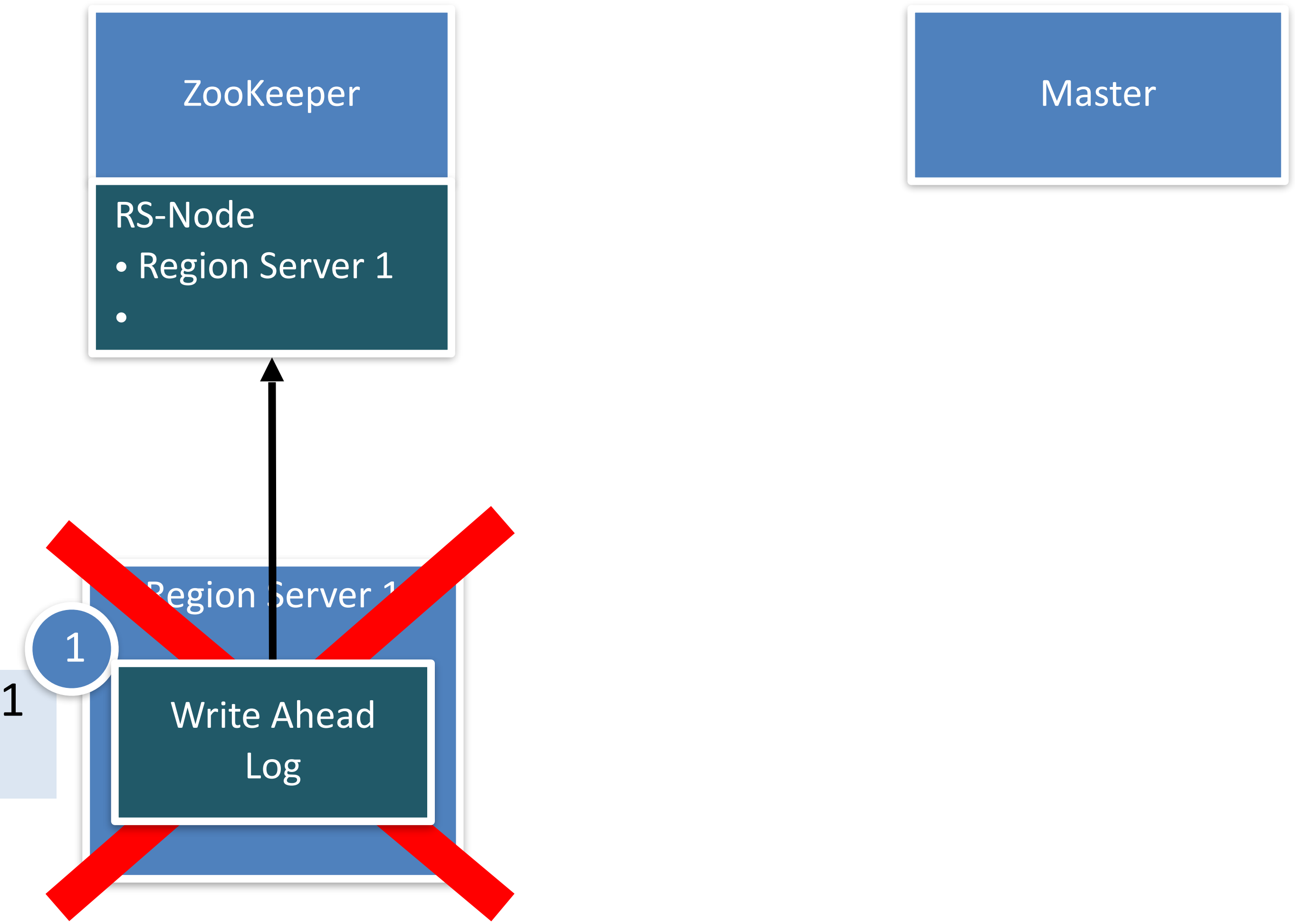
HBase scalability



HBase storage unit failure

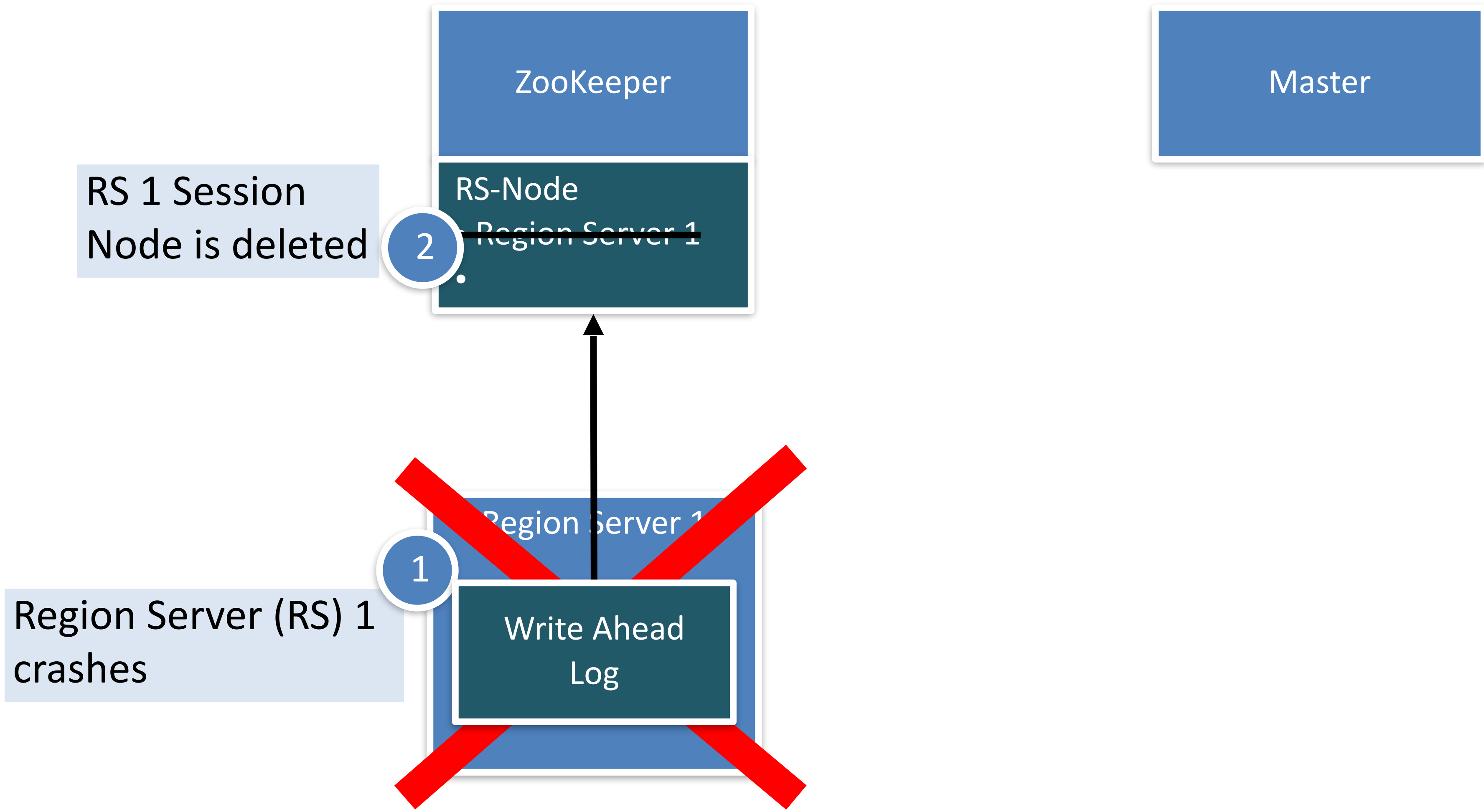


HBase storage unit failure

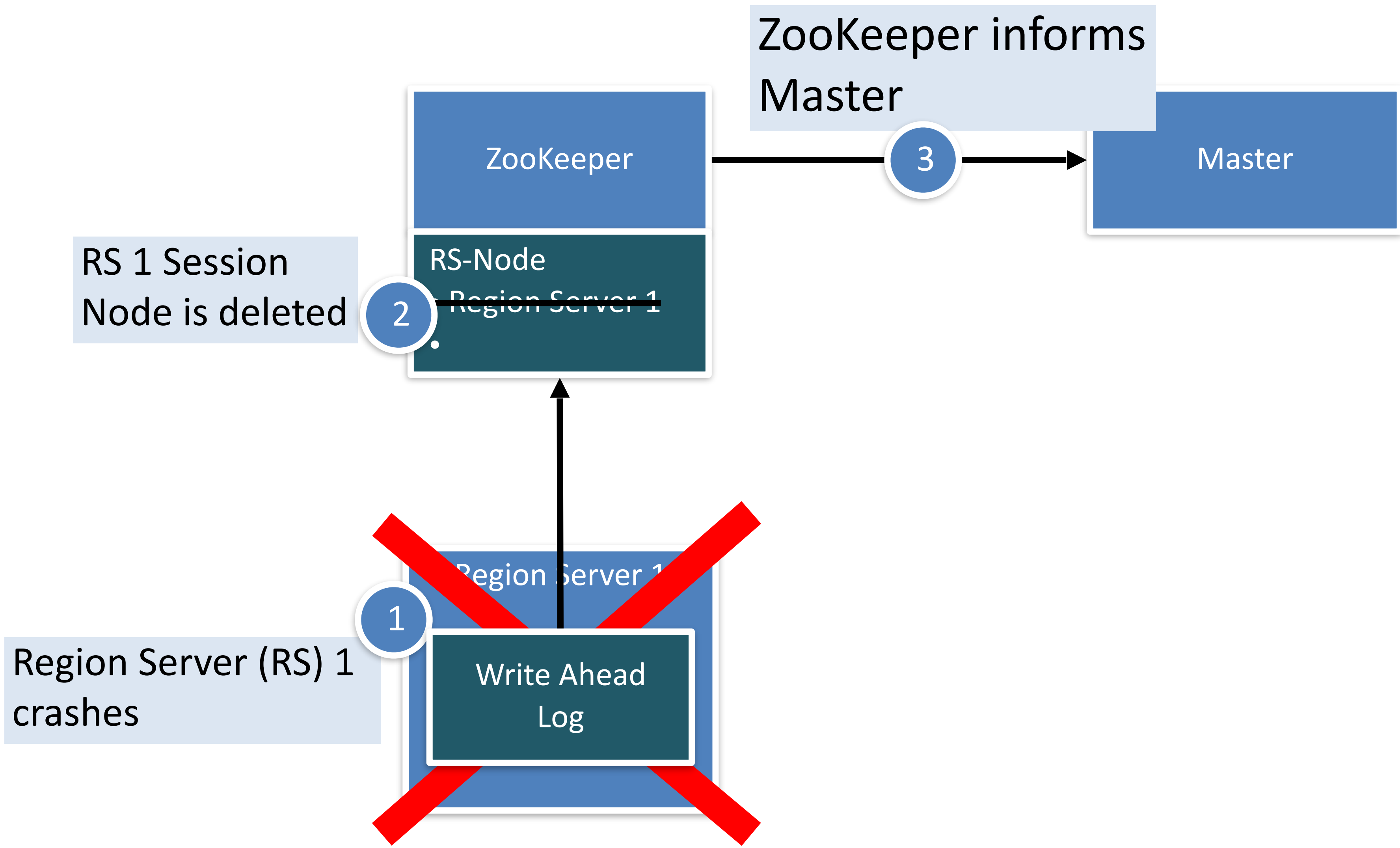


Region Server (RS) 1 crashes

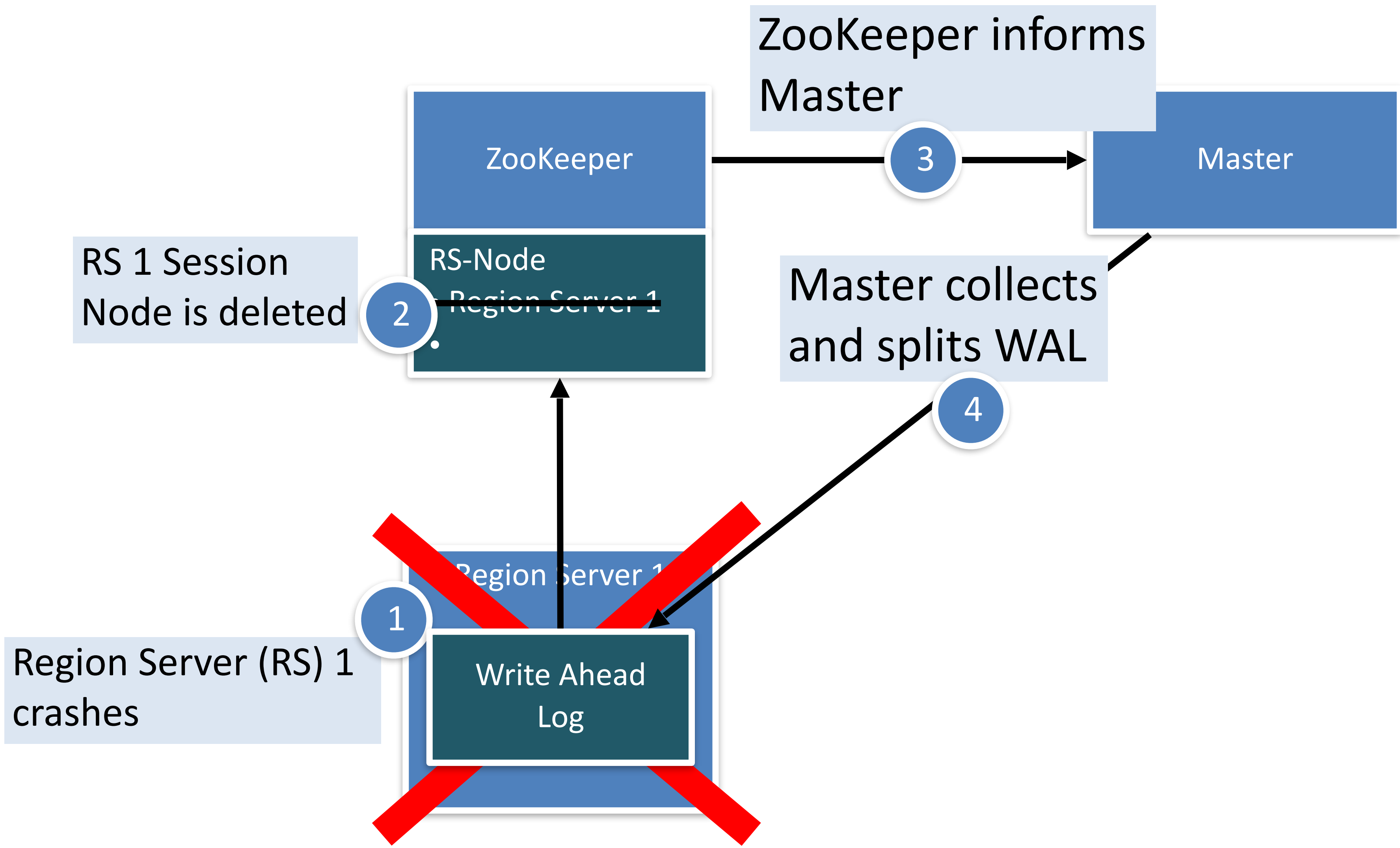
HBase storage unit failure



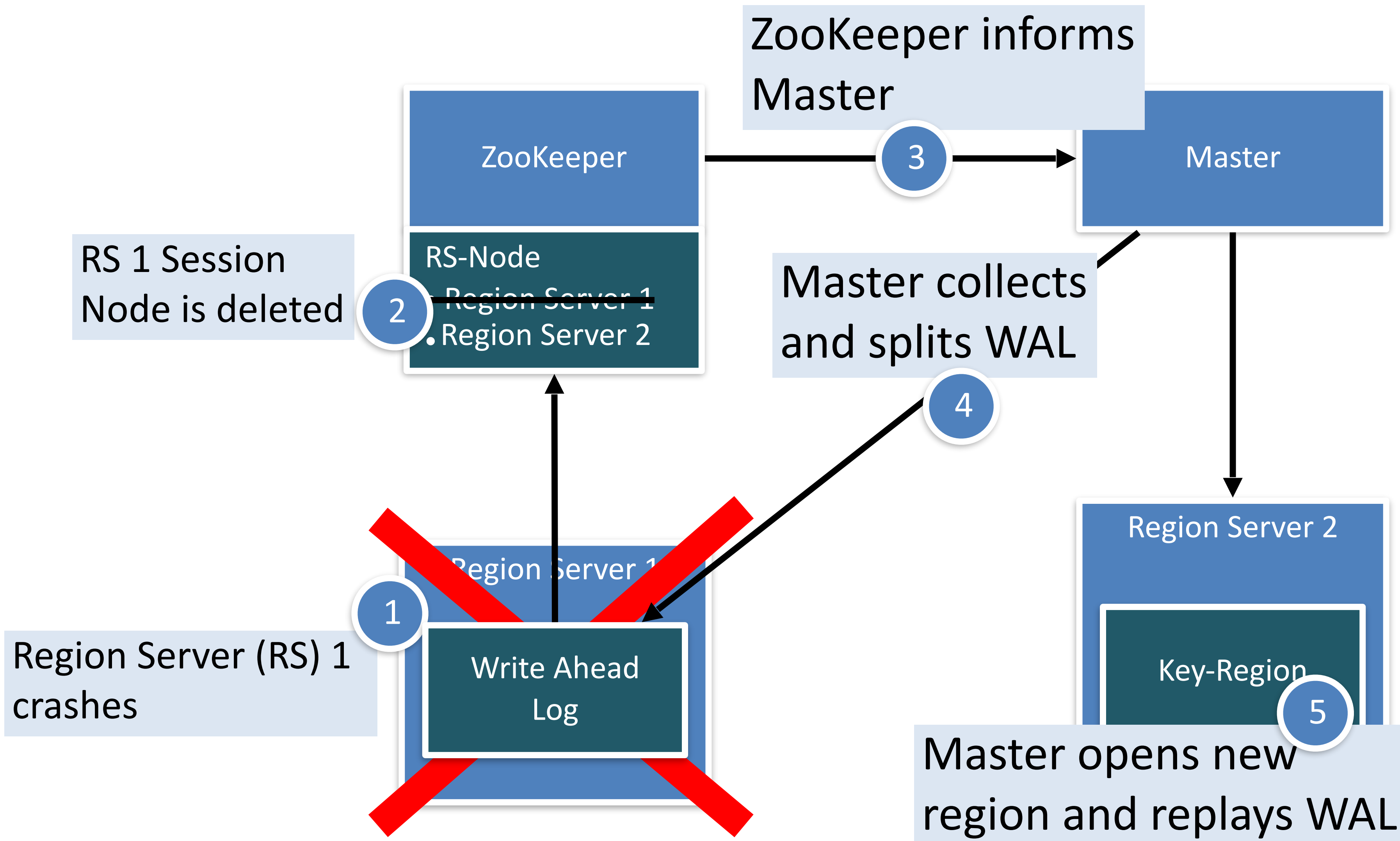
HBase storage unit failure



HBase storage unit failure



HBase storage unit failure



Summary on BigTable and HBase I

- Partitioning of data for **horizontal scalability**
  - Tables → Regions (Tablets)
  - **Load-balanced** amongst Region Servers (TabletServer)
  - Write-Ahead-Log for **failure recovery**
  - **Decouple write** from actual I/O of value to disk
  - Use **MemStore** et al. to accommodate fast write
- Centralized management
  - (H)Master – single point of failure
  - **Backup *masters*** for failover, **leader election** needed
  - Not involved in read/write path (not a bottleneck)

Summary on Bigtable and HBase II

- **Coordination**
  - ZooKeeper (Chubby) lock service
  - **Leader election**, server status, region directory, …
  - Sessions (leases) for timeout (**failure detection**)
  - Mechanisms for **high availability** and reliability
  - **Paxos,** atomic broadcast to replicate coordination state
  - Cache meta-data replies to avoid frequent communication

- **Distributed file system**
  - HDFS (GFS)
  - Store data as Hfiles (SSTables)
  - **Data is replicated** for availability

Summary Big Picture

BigTable vs. MapReduce

BigTable

- Layered on top of GFS
- Data storage and access
- BigTable: read/write web data

MapReduce

- Layered on top of GFS
- Batch analytics
- MapReduce: offline batch processing (*cf. MapReduce Lecture*)

Google File System: common persistent storage layer (*cf. GFS et al. Lecture*)

Self-study questions

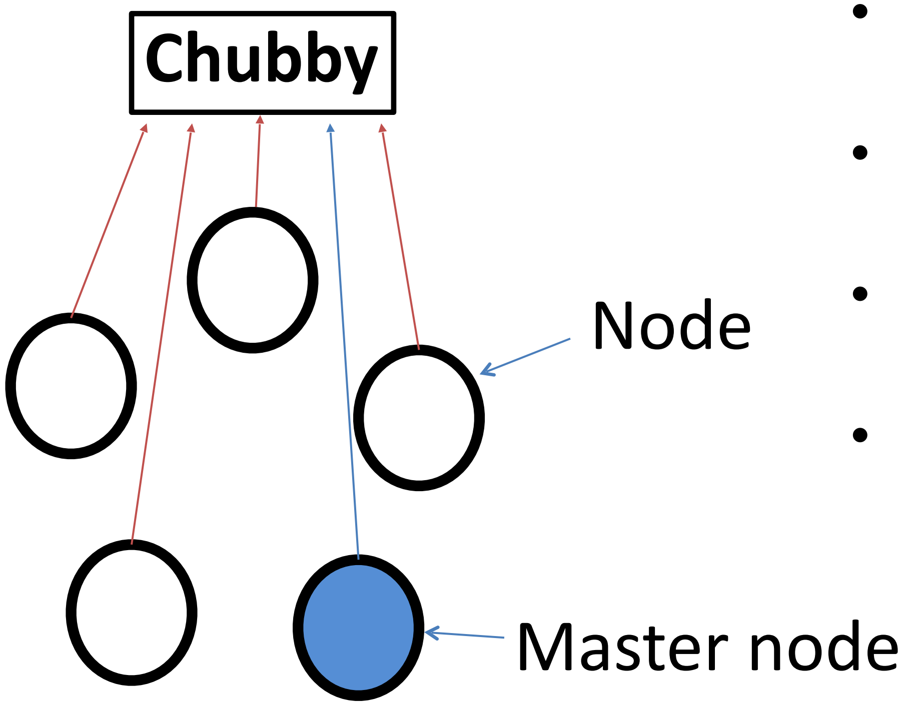
- Would the BigTable architecture make sense without relying on a distributed file system layer for storage, argue for or against?
- Contrast the life of a read request vs. write request issued from a client to BigTable, what processing stages can you identify?
- How many bytes could BitTable/Hbase address, assuming tables are of set sizes (e.g., 1MB, 100MB, 200MB etc.)
- Why are read and write requests not channelled through the Master, argue for or against?
- What happens if a client request from a client sent to a Tablet Server is not serviced due to Tablet Server crash?
- Why is the Master a single-point of failure?

DISTRIBUTED SYSTEMS BY EXAMPLE

CHUBBY (ZOOKEEPER)

Chubby Lock Service

Highly-available, persistent, distributed lock, coordination service



Sample use in BigTable

- Ensure at most one active BigTable master at any time
- Store bootstrap location of data (root tablet)
- Discover tablet servers (manage their lifetime)
- Store schema information

Chubby Lock Service

High
coor

Cf. Coordination and Agreement Lecture

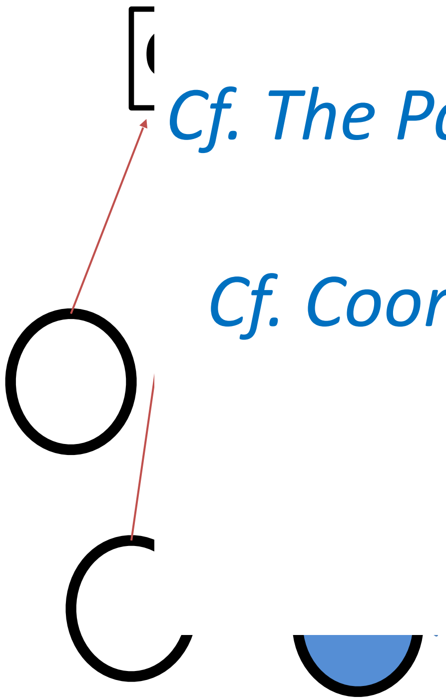
e BigTable

Cf. The Paxos Consensus Algorithm Lecture

of data

Cf. Coordination with Zookeeper Lecture

anage



Master node

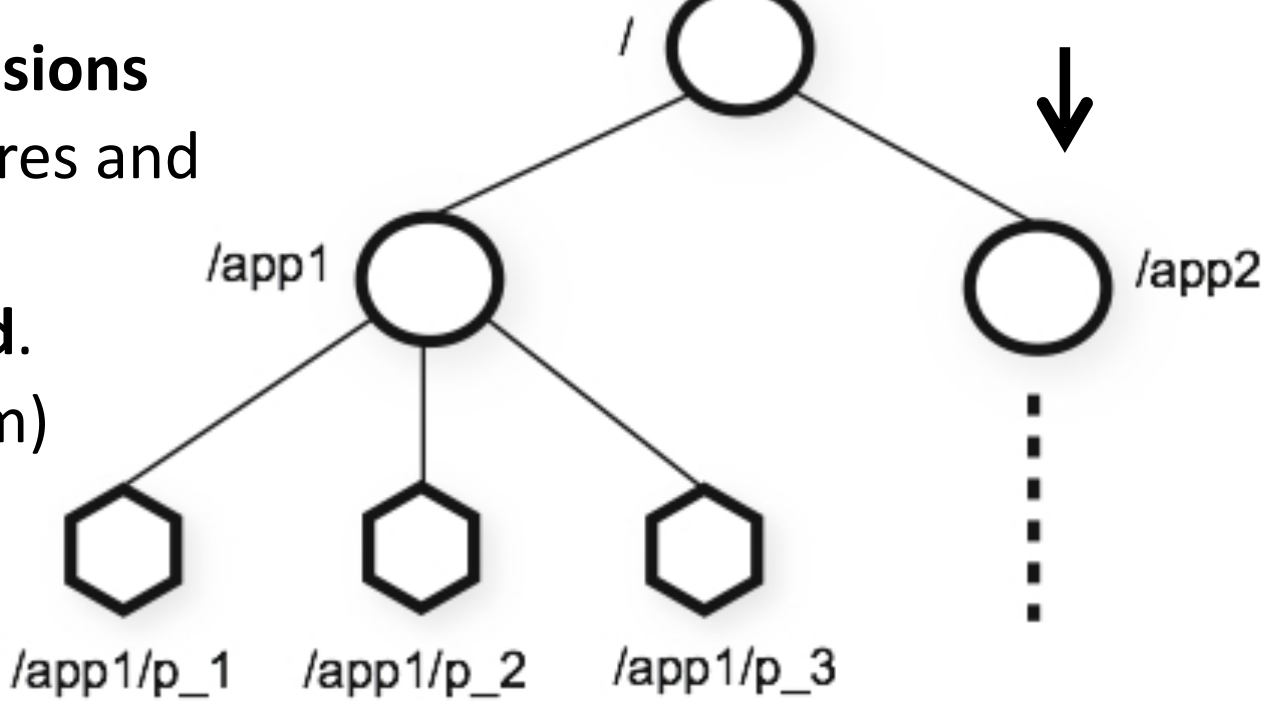
Lock Service Operational Model

- Knows about and manages directories, **files**
- Directories, files can serve as **locks**
- Reads, writes are **atomic**

Chubby Clients

Read/Write requests

Clients maintain **sessions**
If session lease expires and
can't be renewed,
locks are **released**.
(timeout mechanism)



Lock Service Availability

- Comprised of **five** active **replicas**
  - **Consistently replicate writes (*cf. Replication Lecture*)**
- One replica is designated as master
  - Need to **elect** master (**leader**) (*cf. Coordination Lecture*)
  - Chubby master is different from BigTable master!
- Service is up when:
  - Majority of replicas are running and
    - A **quorum** of replicas is established
    - Can communicate with one another

Core Mechanisms

- Ensure one active BigTable master at any time
  - **Leader election** in distributed systems
  - *Cf. Coordination and Agreement Lecture*
- Keep replicas consistent in face of failures
  - **Paxos algorithm** based on replicated state machines (RSM)
  - Atomic broadcast
  - *Cf. Paxos Lecture, Replication Lecture*

Chubby Example: Leader election

- Electing a leader node: supported by acquiring an exclusive lock on a file (**clients represent partaking nodes**)
- Clients concurrently **open a file** and attempt to acquire the file lock in write mode
- One client **succeeds** (i.e., becomes the **leader**) and writes its name to the file
- Other clients **fail** (i.e., become **replicas**) and discover the name of the leader by reading the file

Chubby Example: Leader election

```
Open("/ls/cell1/somedir/file1",
      "write mode")

if (successful) {  // leader
      setContents(primary_identity)
} else {           // replica
      Open("/ls/cell1/somedir/file1",
            "read mode",
            "file-modification event")
      On modification notification
            primary = getContentsAndStat()
}
```

obtain file handle

write to file

subscribe to modification event

read from file

Self-study questions

- Why do we need another database (Chubby) in addition to BigTable?
- How could the sketched Chubby API be used for locking and for mutual exclusion?

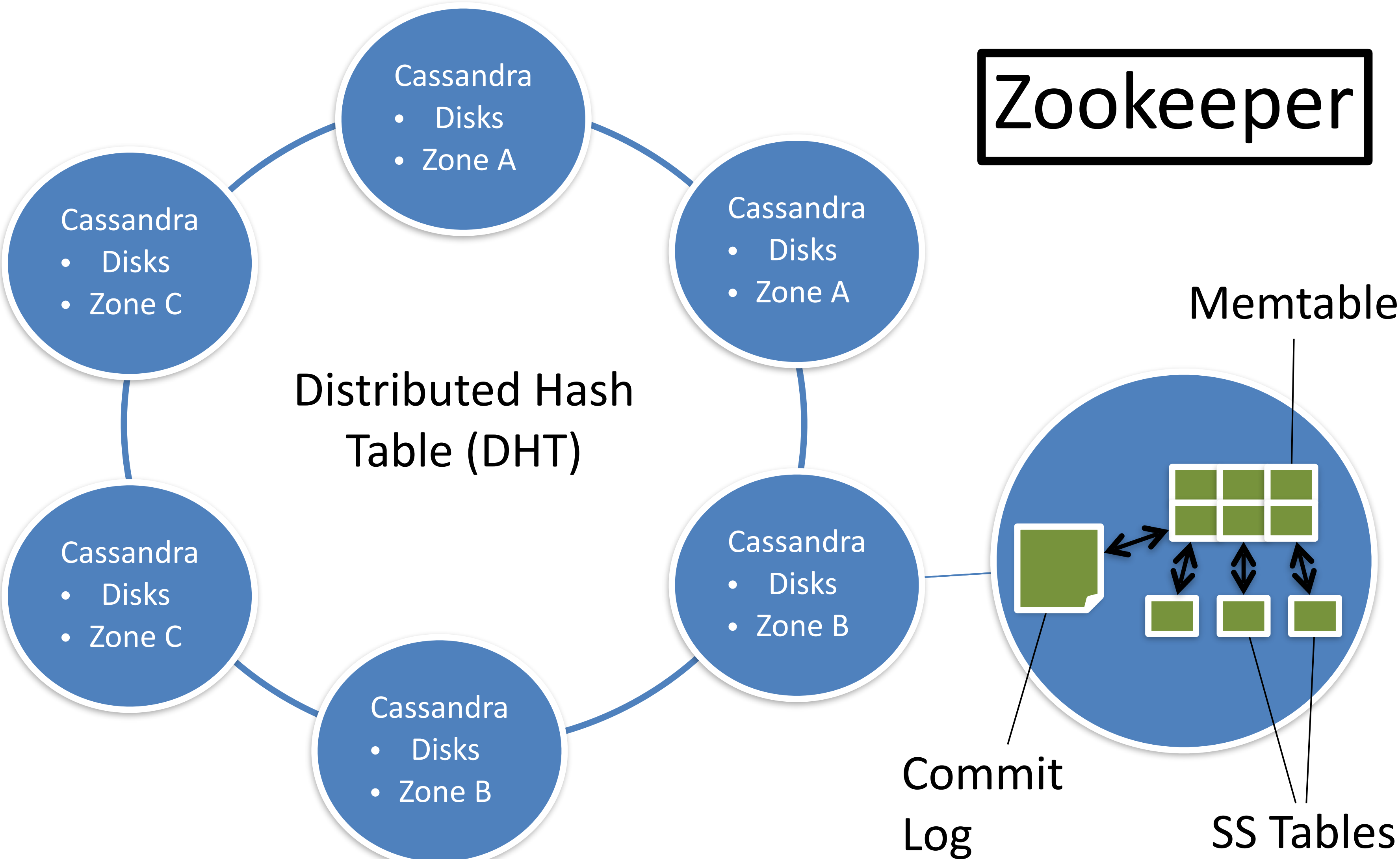
DISTRIBUTED SYSTEMS BY EXAMPLE

DYNAMO / CASSANDRA

Cassandra

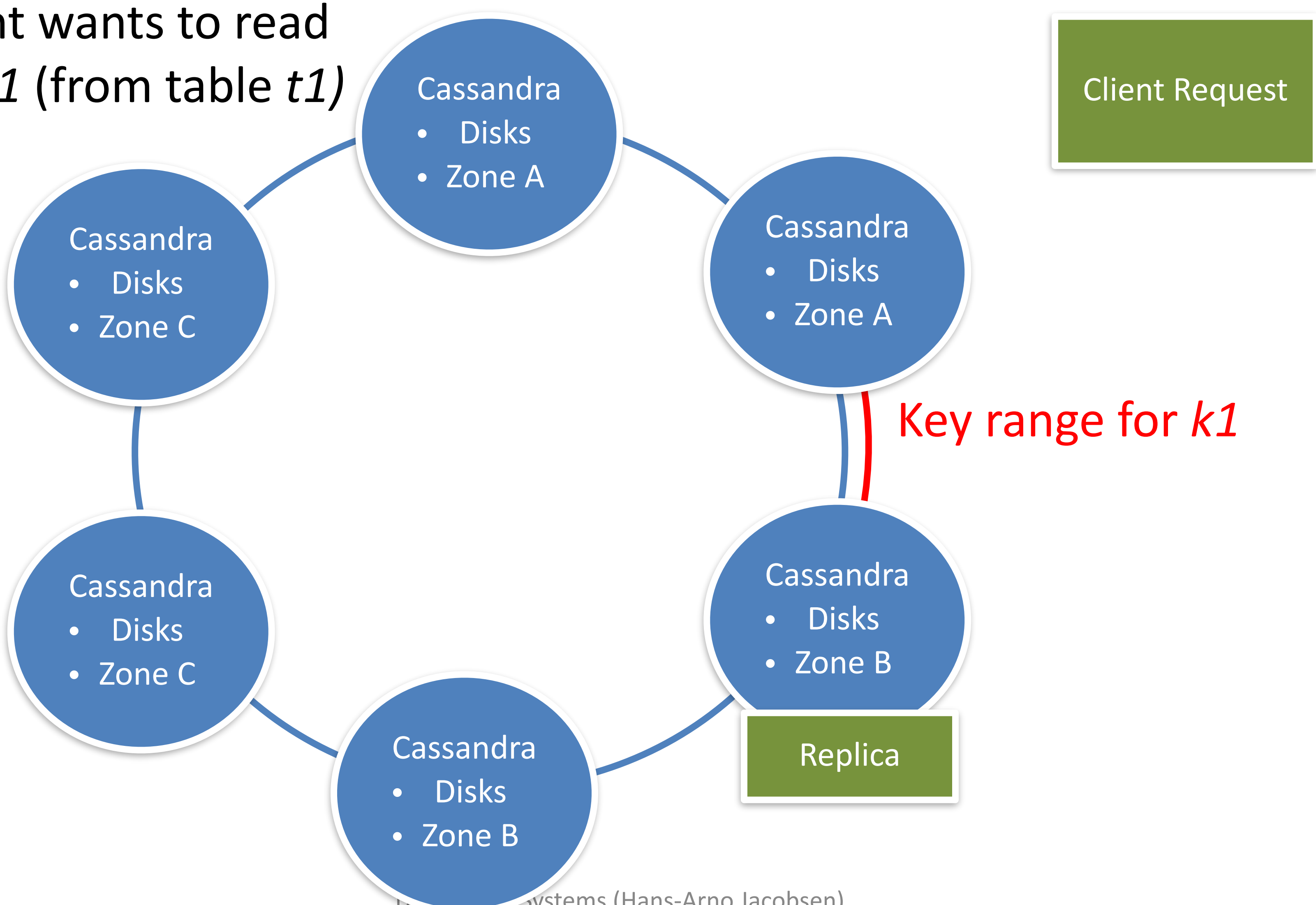
- Developed by Facebook
- Based on Amazon Dynamo (but open-source)
- Structured storage nodes (**no GFS** used)
- **Decentralized** architecture (no master assignment)
- **Consistent hashing** for load balancing
- Eventual consistency
- **Gossiping** to exchange information

Cassandra architecture overview



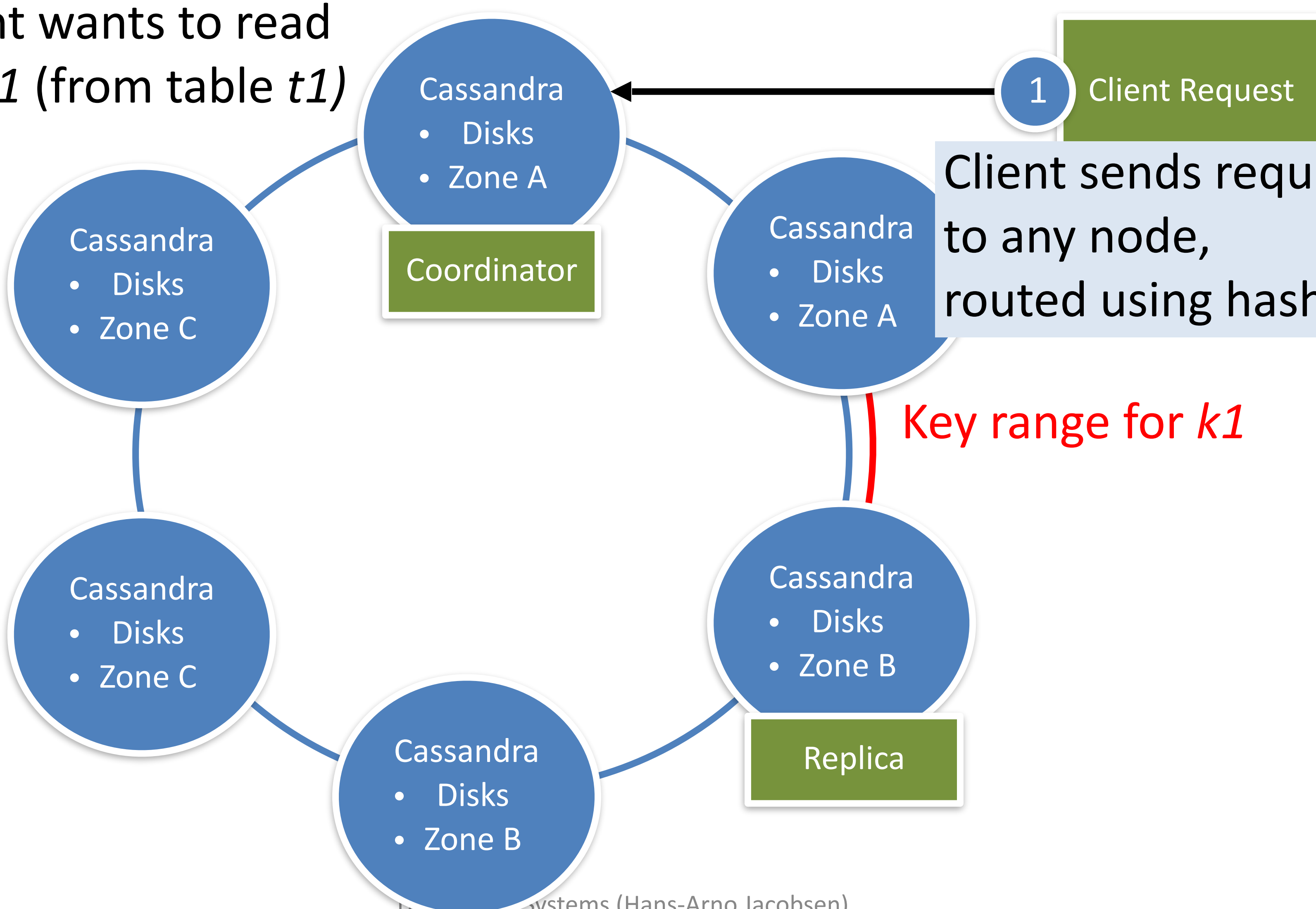
Cassandra global read-path

Client wants to read key *k1* (from table *t1*)



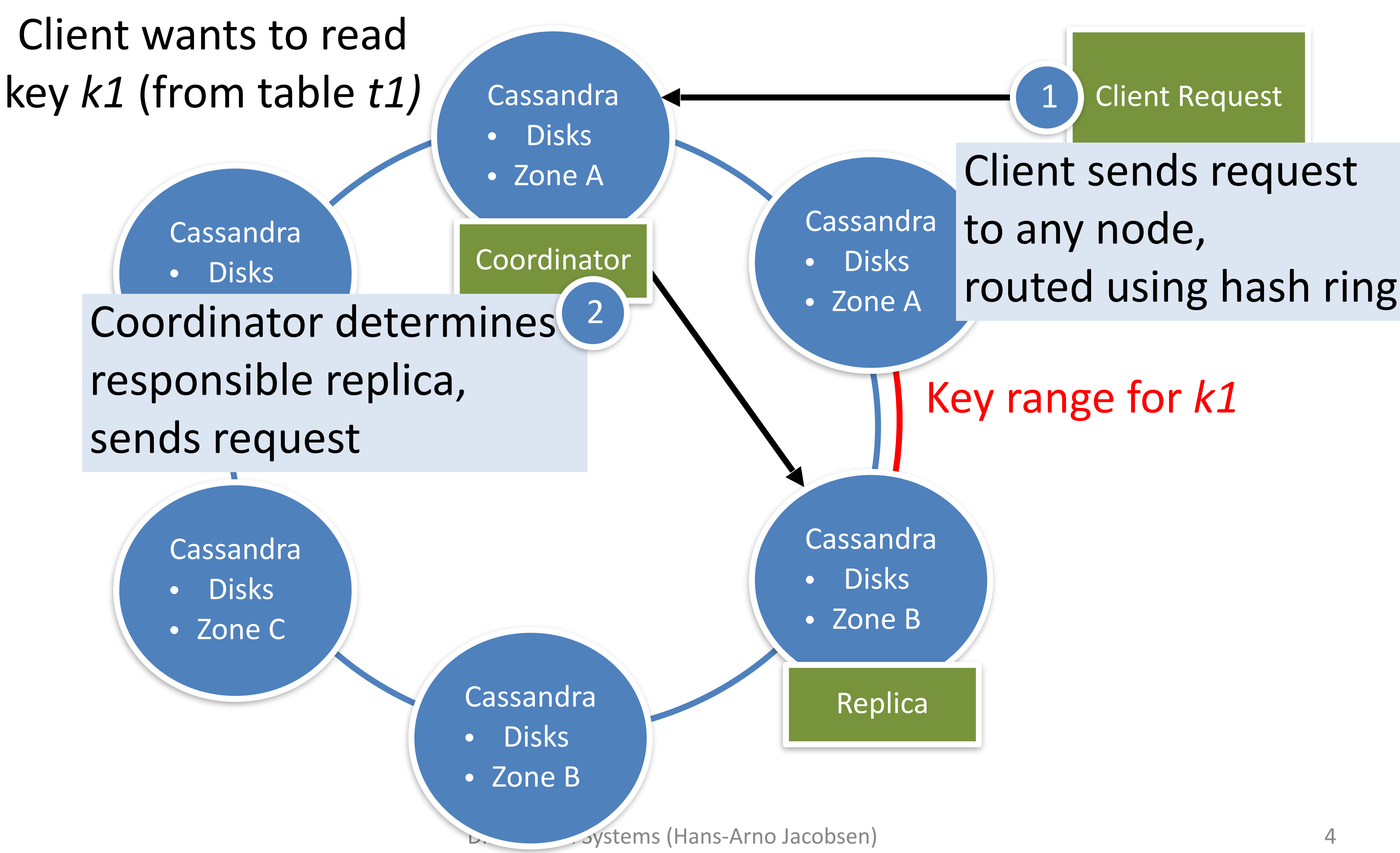
Cassandra global read-path

Client wants to read key *k1* (from table *t1*)

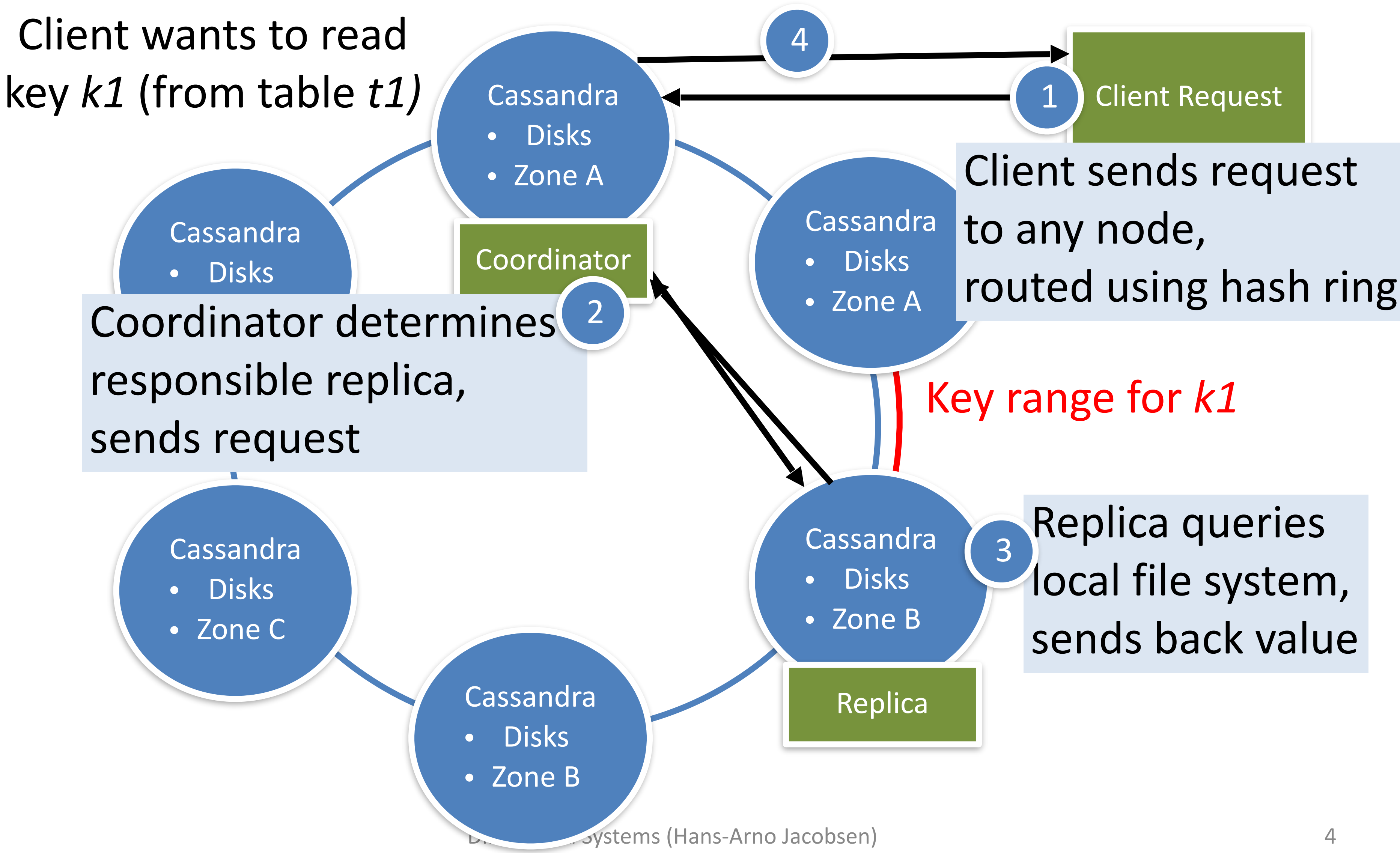


Client sends request to any node, routed using hash ring

Cassandra global read-path

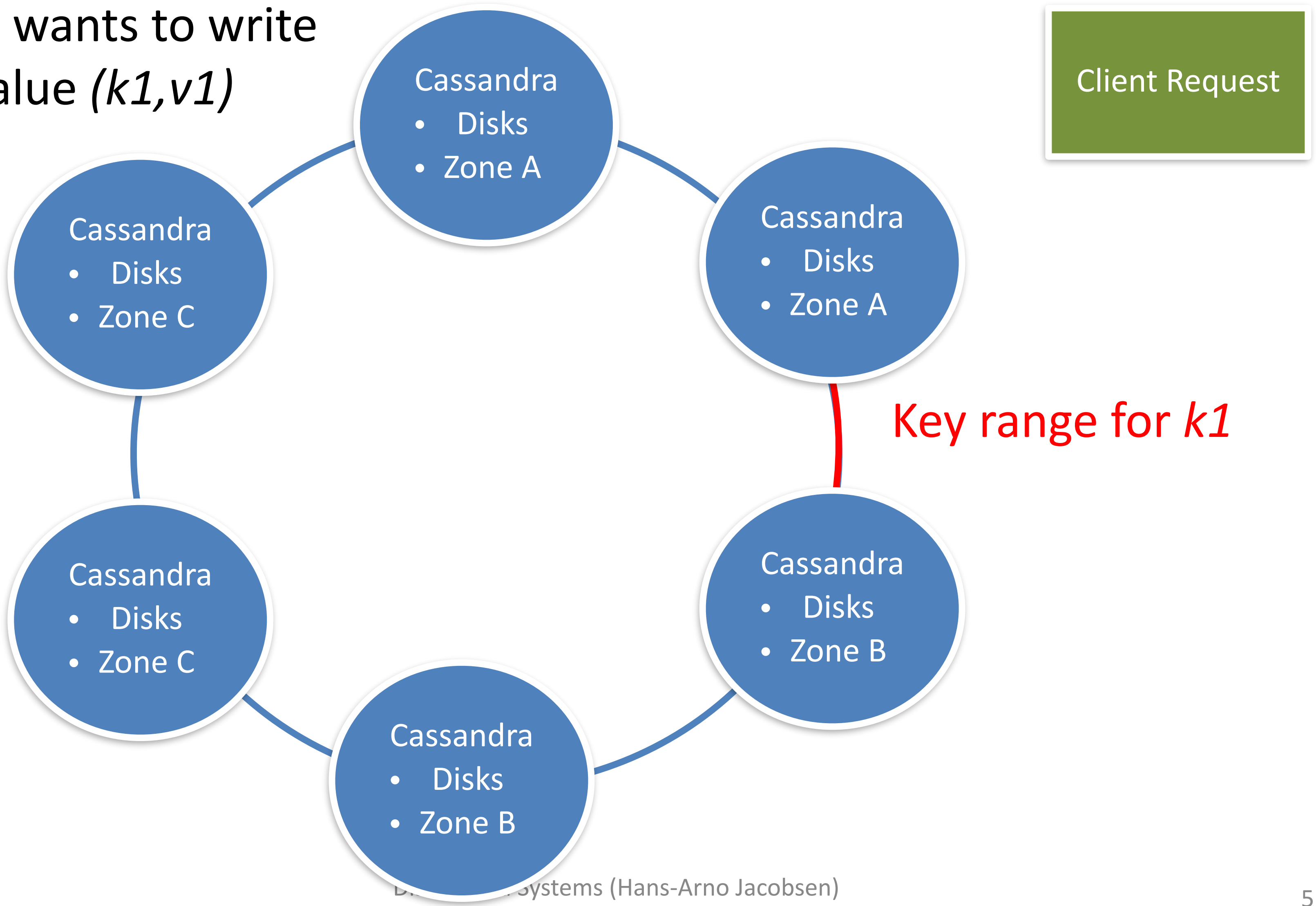


Cassandra global read-path



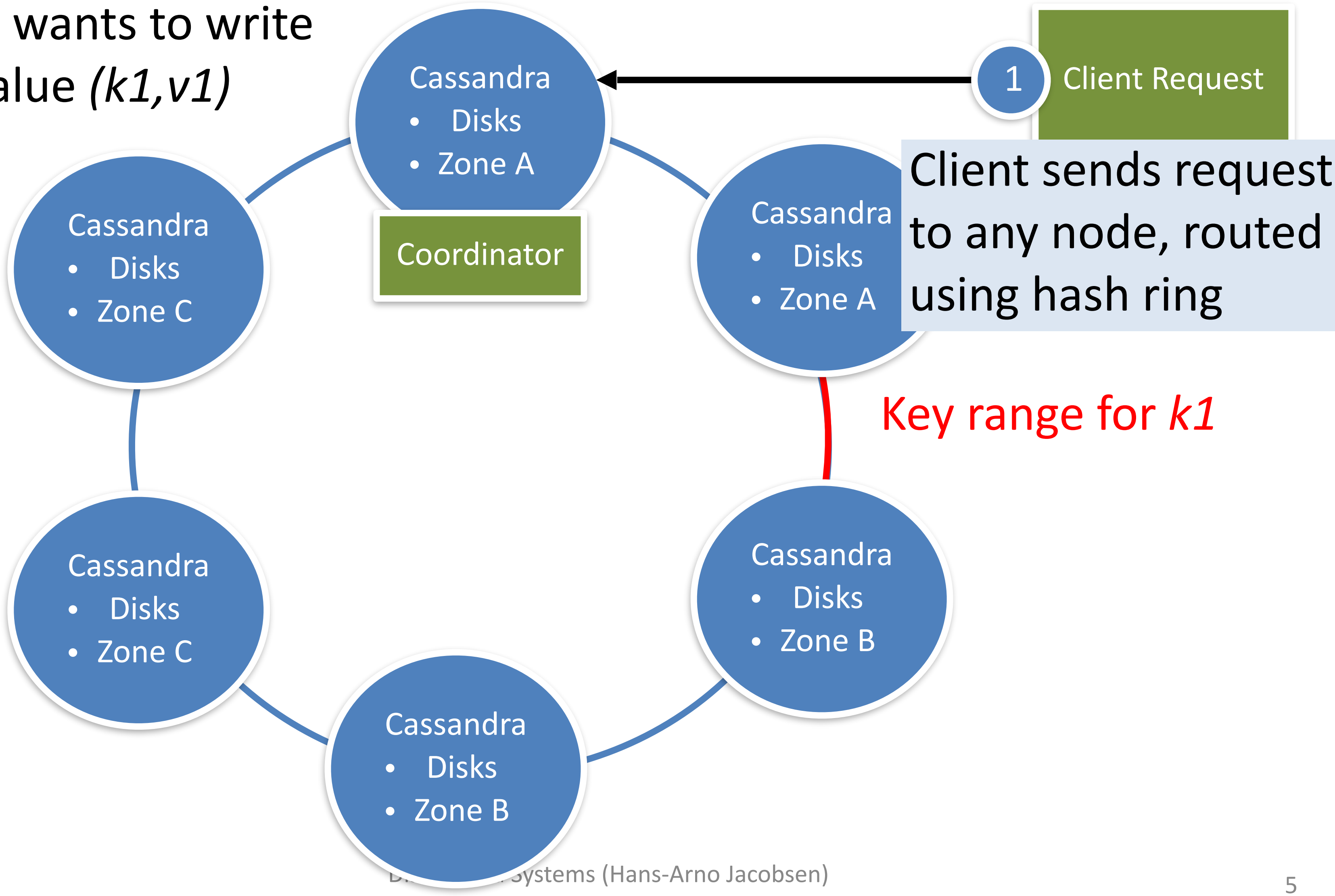
Cassandra global write-path

Client wants to write key-value *(k1,v1)*



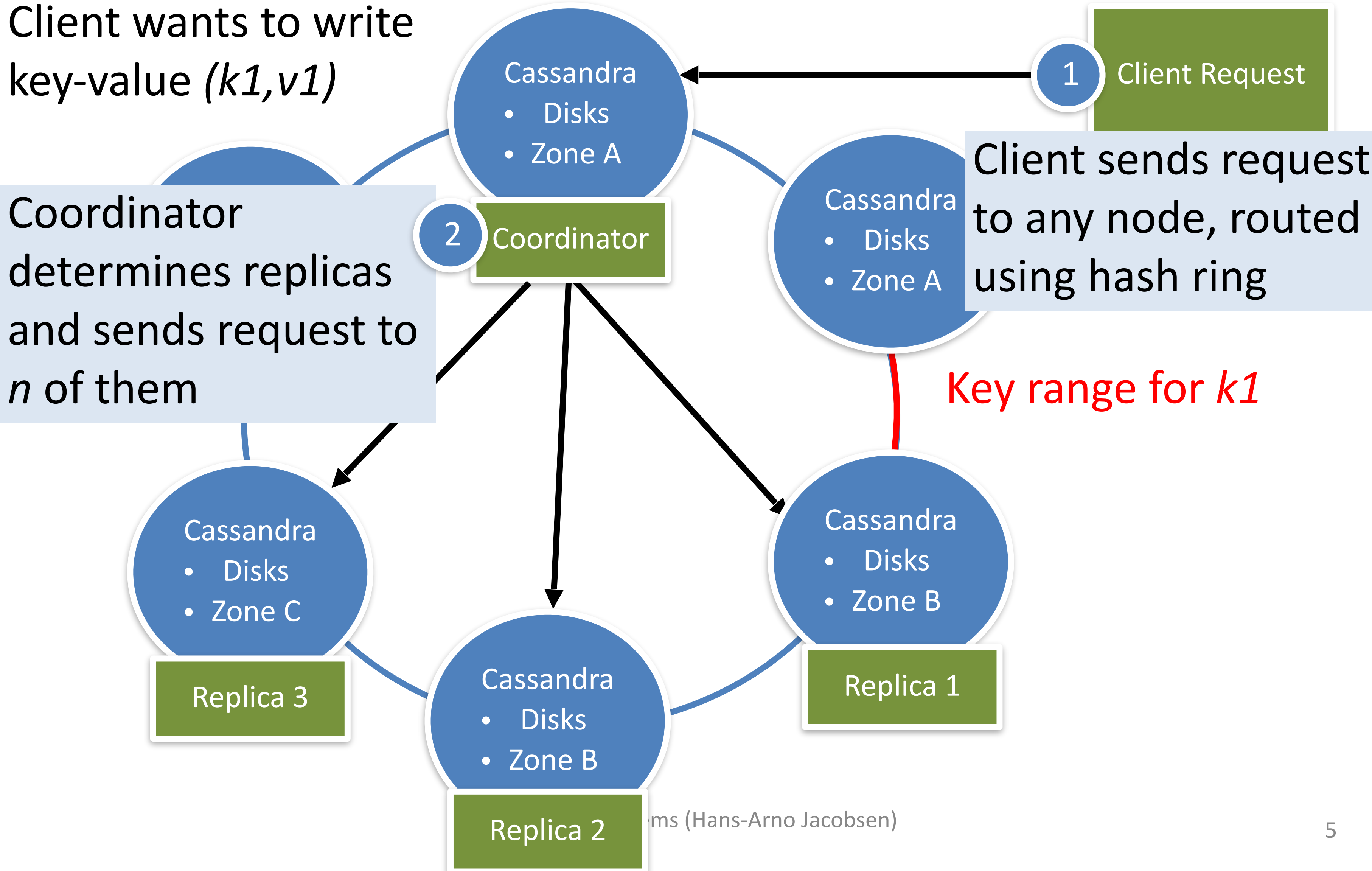
Cassandra global write-path

Client wants to write key-value *(k1,v1)*



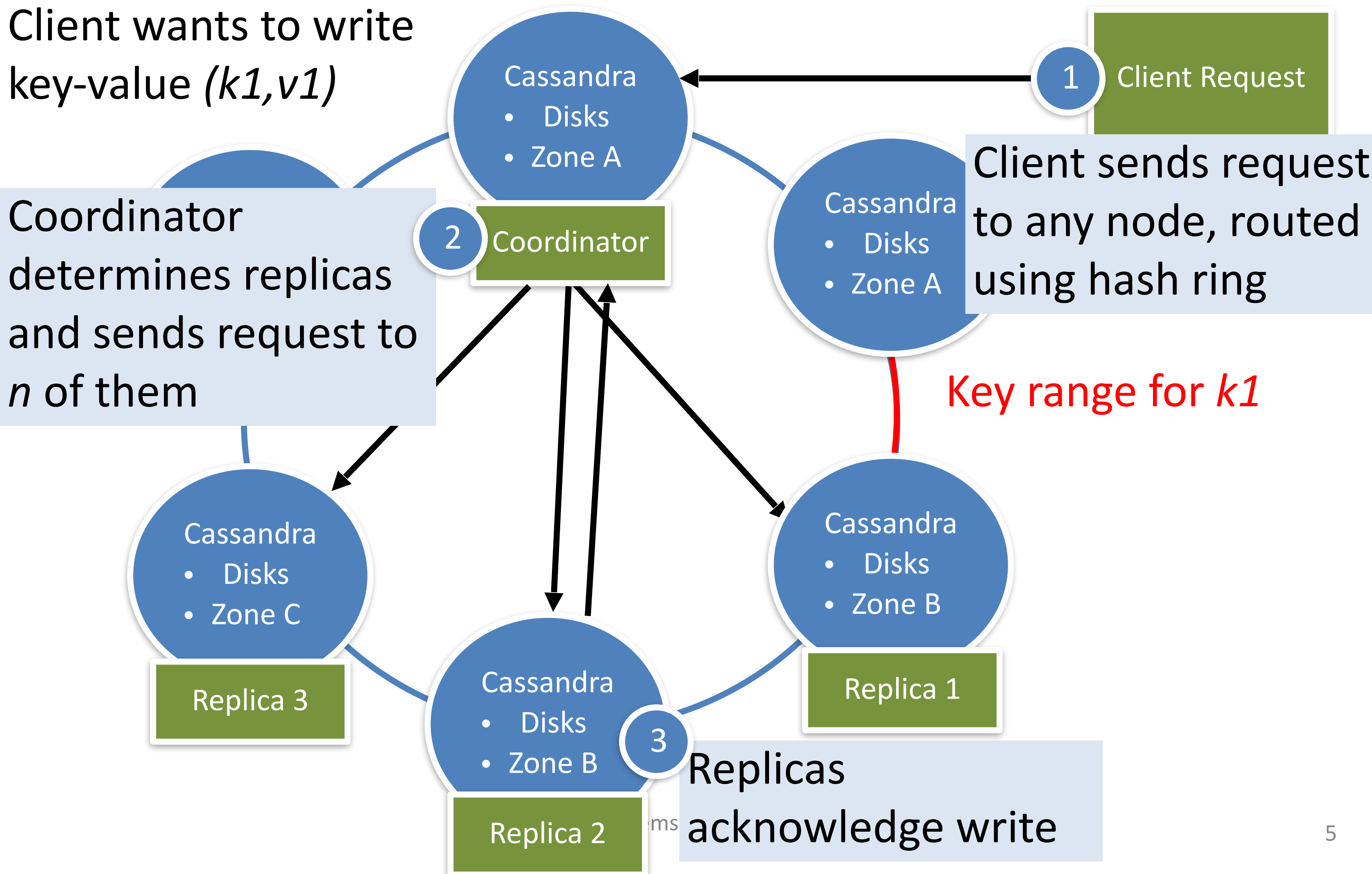
Cassandra global write-path

Client wants to write key-value *(k1,v1)*

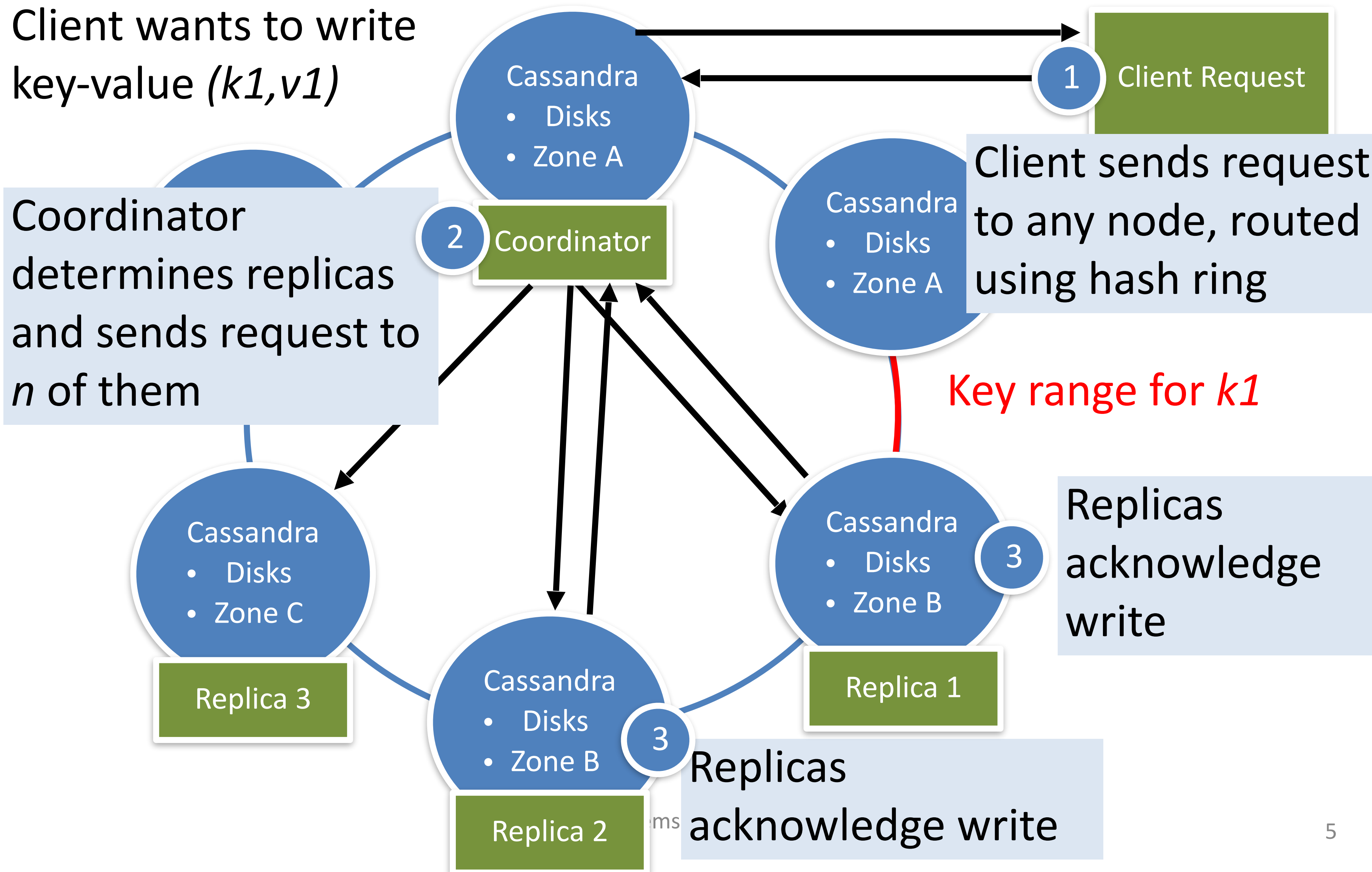


Cassandra global write-path

Client wants to write key-value *(k1,v1)*

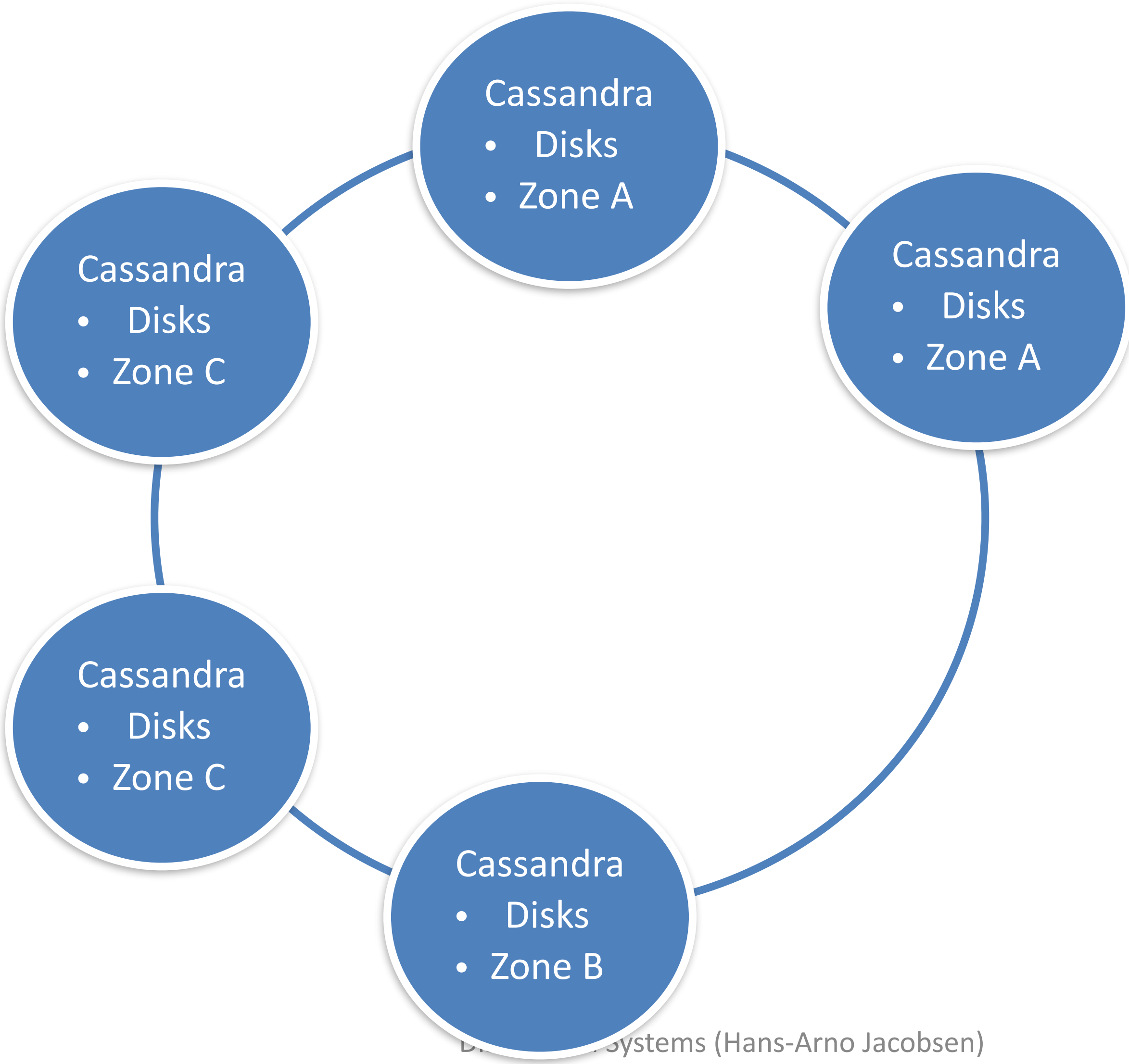


Cassandra global write-path



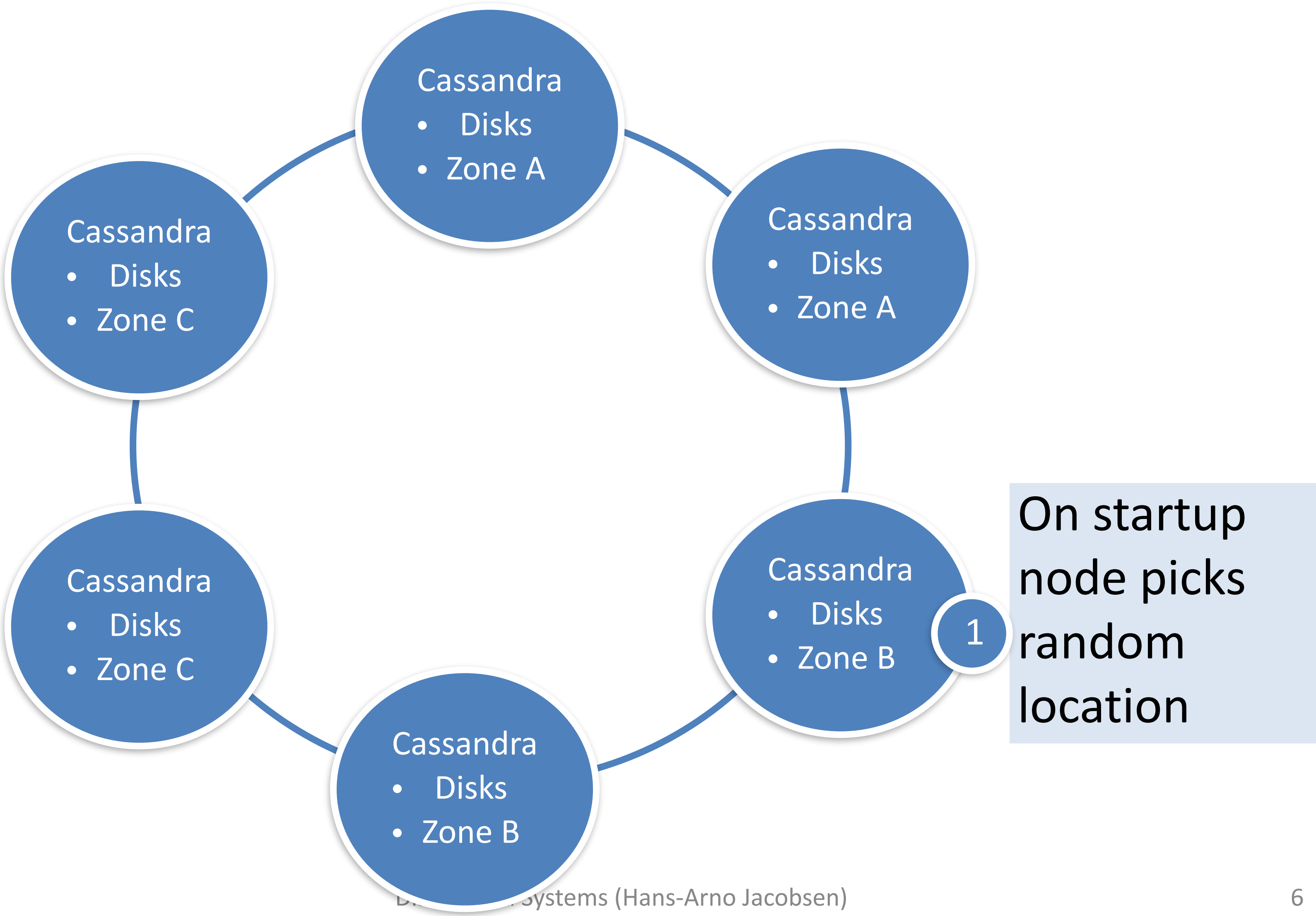
Incremental scaling in Cassandra

(i.e., adding a storage unit)



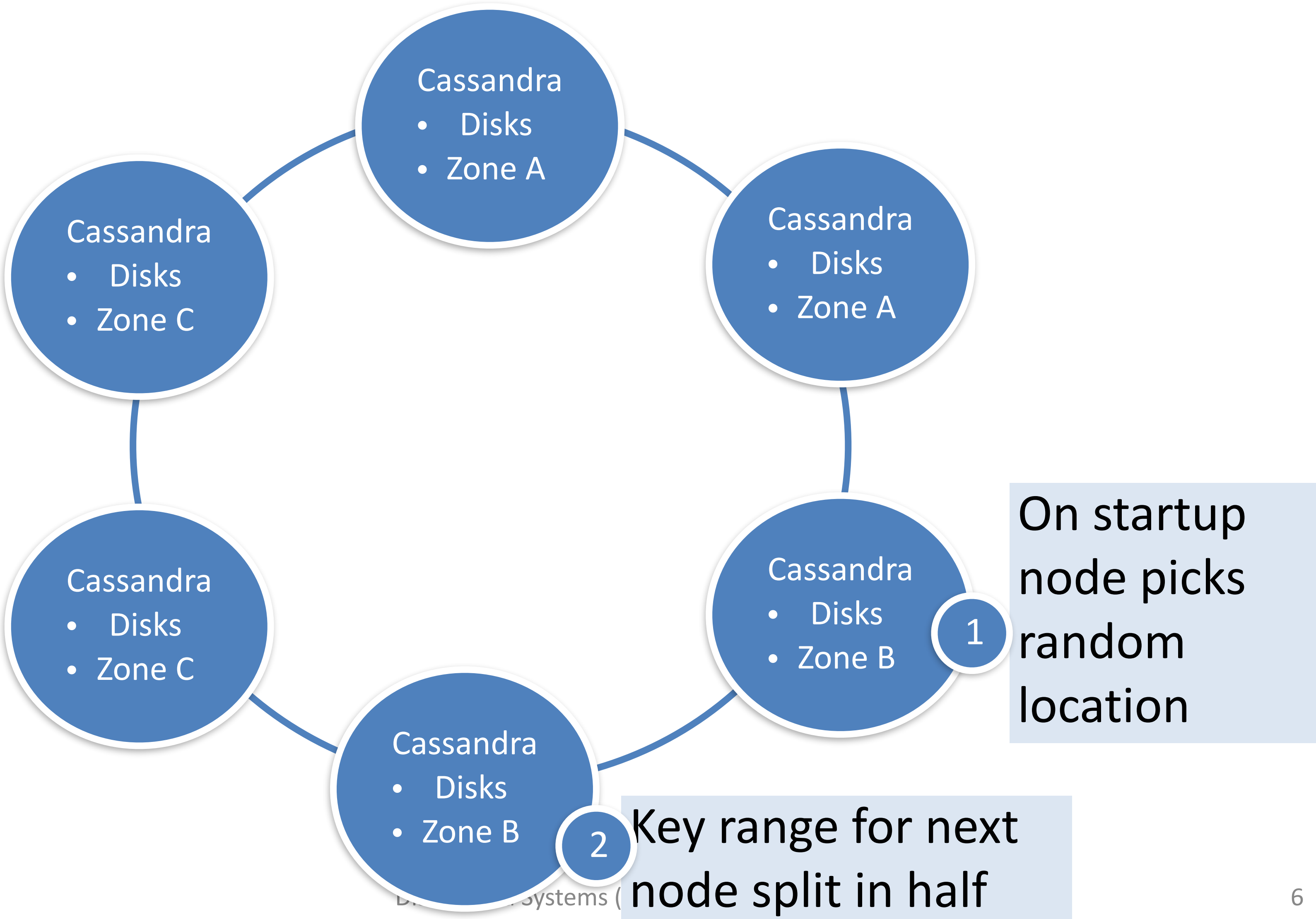
Incremental scaling in Cassandra

(i.e., adding a storage unit)



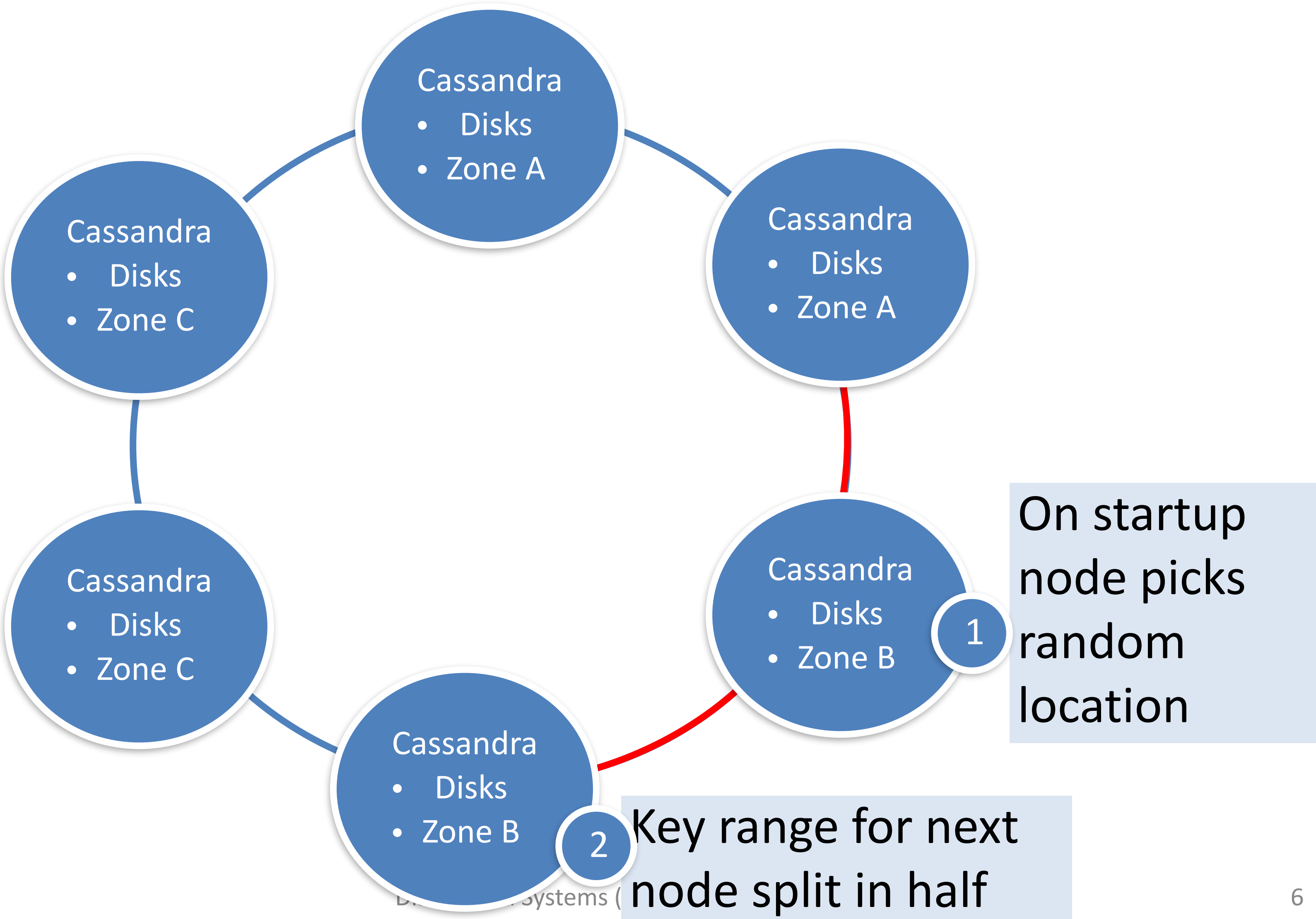
Incremental scaling in Cassandra

(i.e., adding a storage unit)

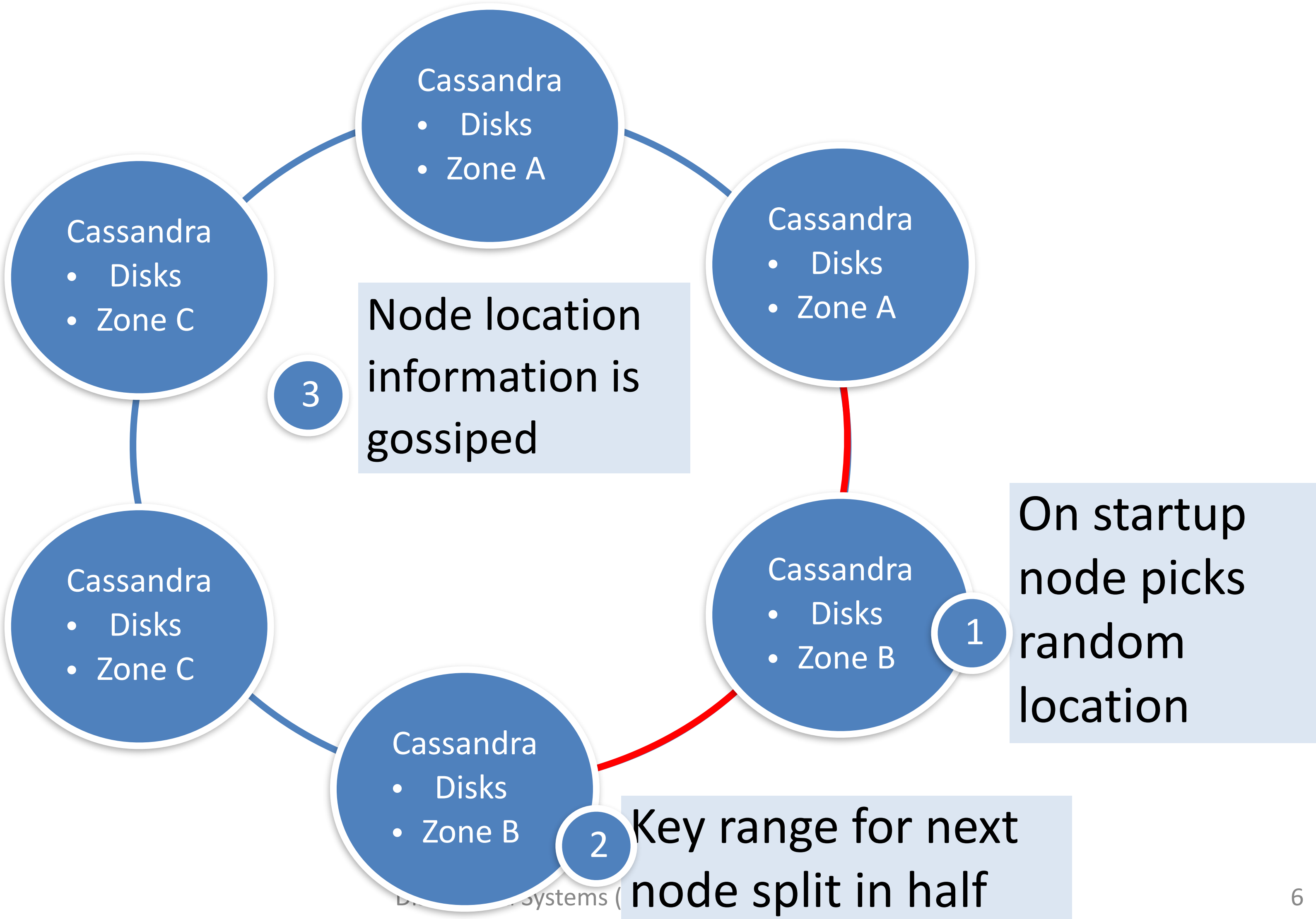


Incremental scaling in Cassandra

(i.e., adding a storage unit)

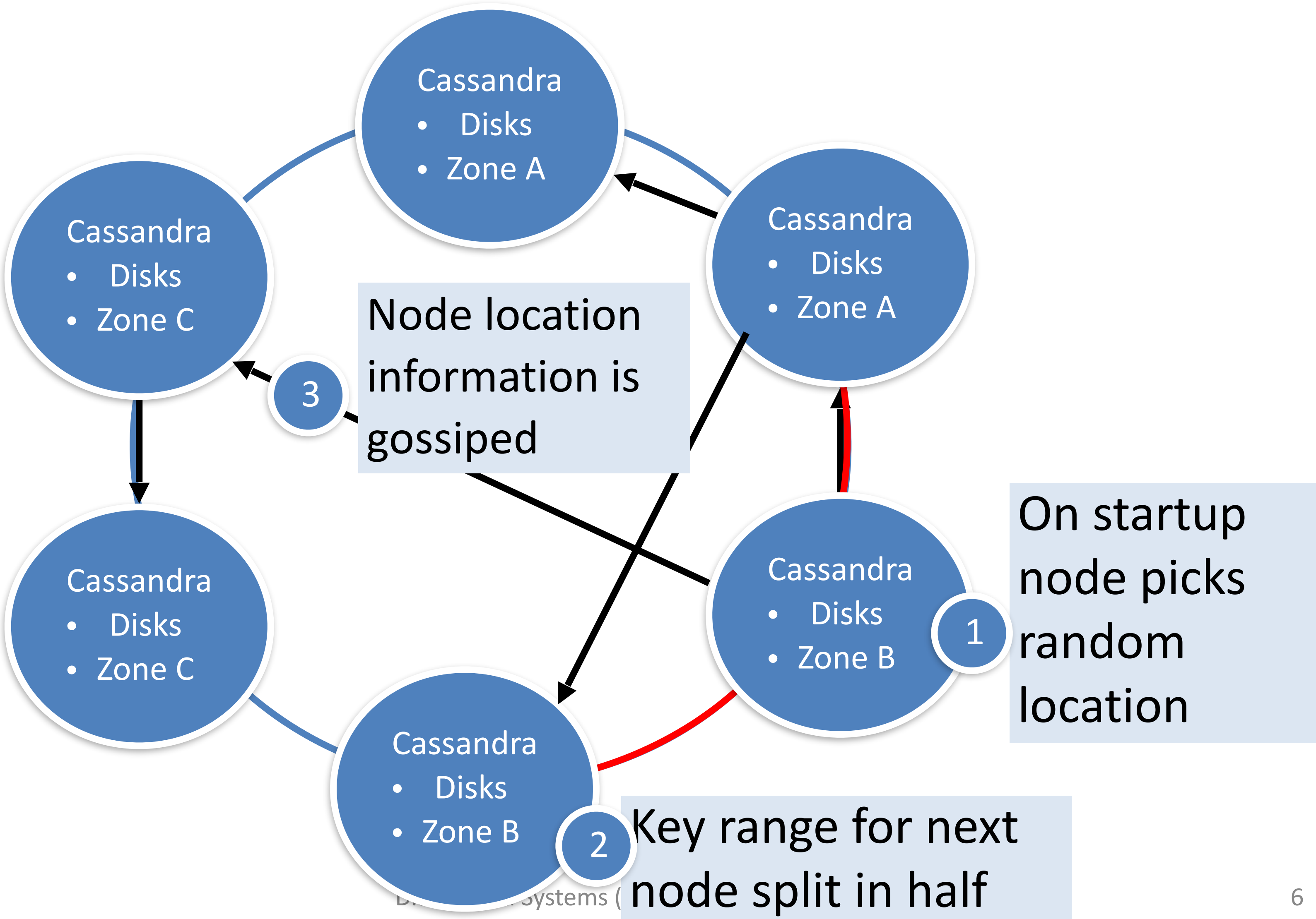


Incremental scaling in Cassandra
(i.e., adding a storage unit)

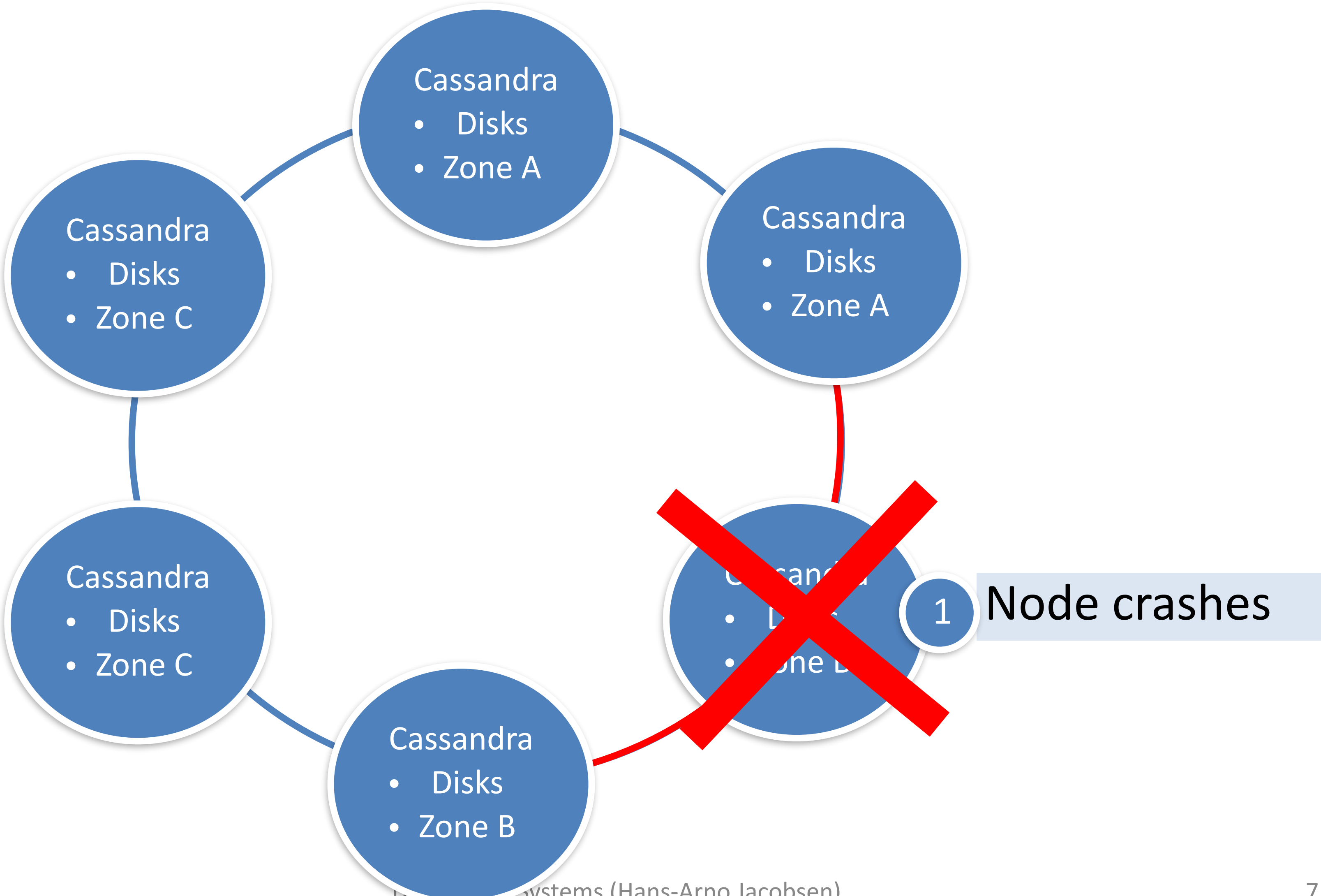


Incremental scaling in Cassandra

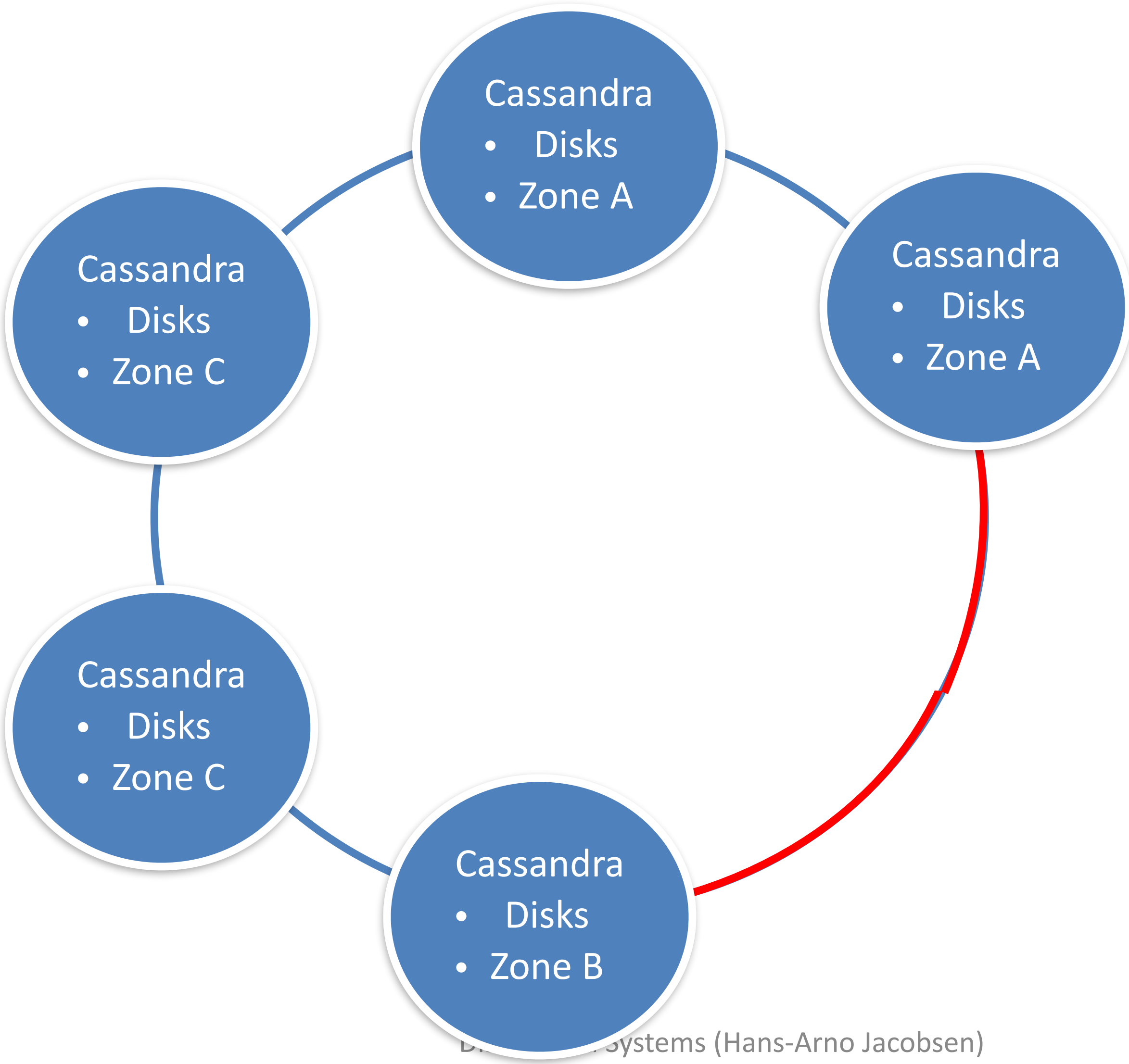
(i.e., adding a storage unit)



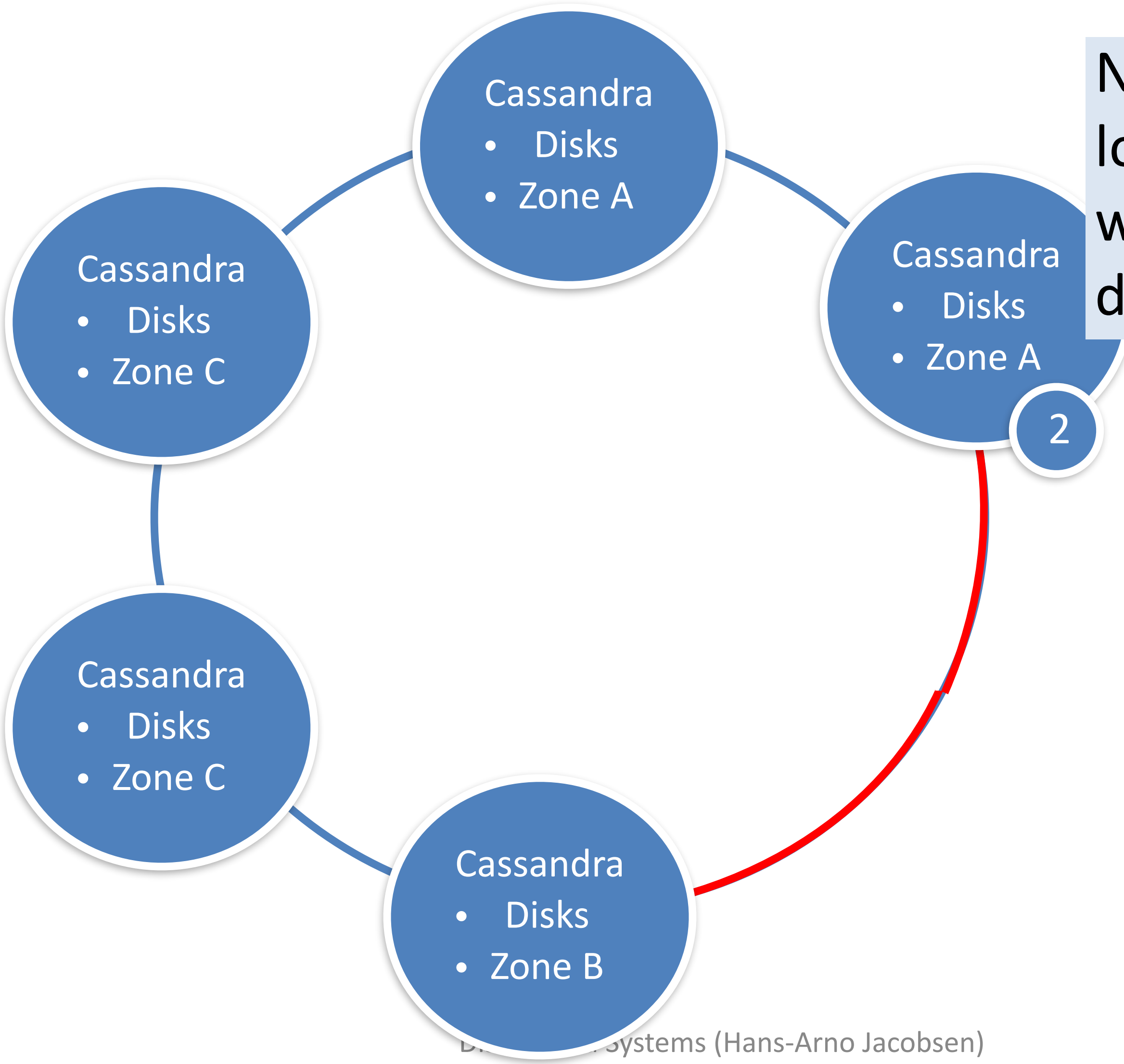
Storage unit failure



Storage unit failure

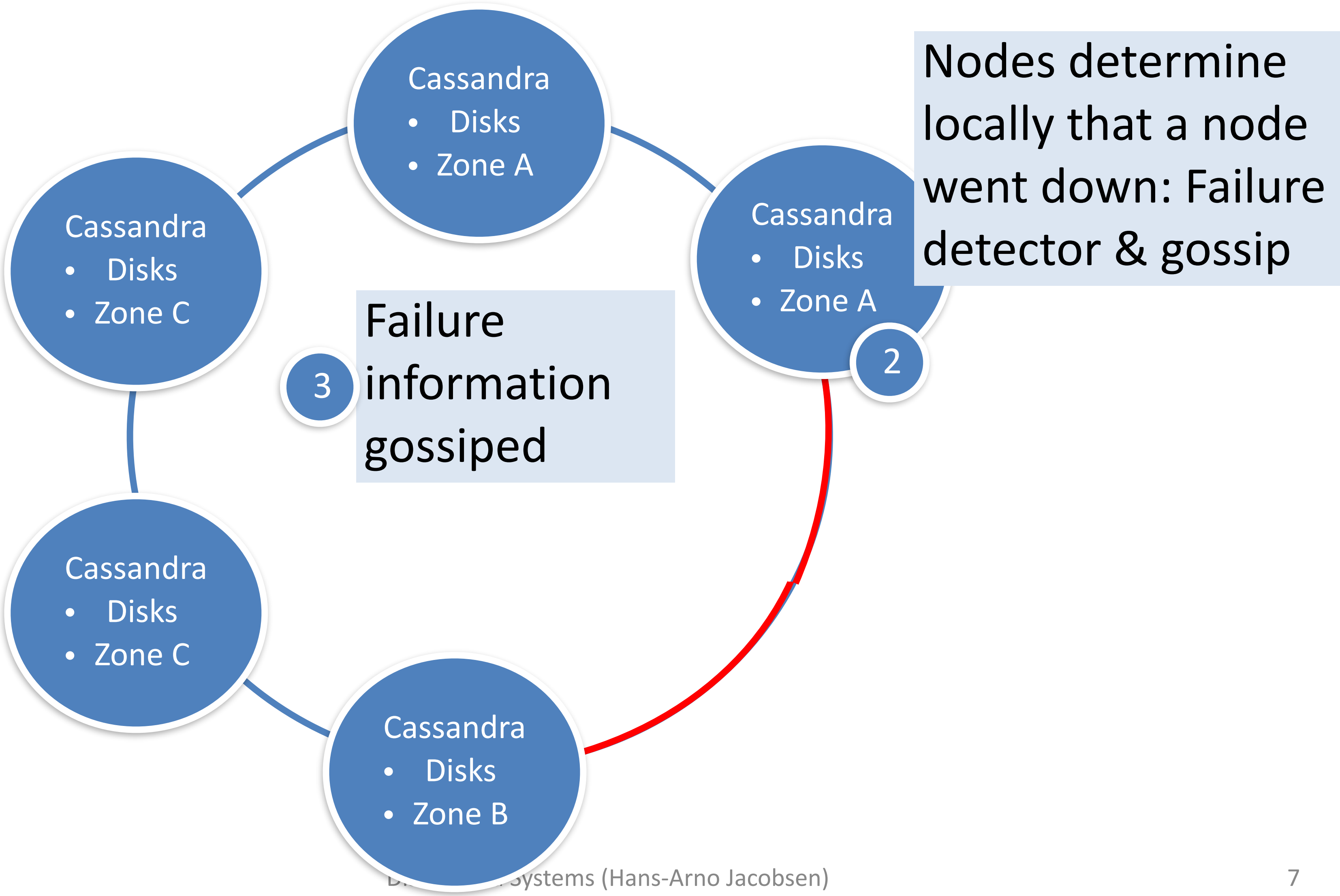


Storage unit failure



Nodes determine locally that a node went down: Failure detector & gossip

Storage unit failure



Core mechanisms

- Decentralized load balancing and scalability
  - *Cf. Consistent Hashing Lecture*
- Read/write reliability
  - *Cf. Replication Lecture*
- Membership management
  - *Cf. Gossip in Replication Lecture*
- Eventual consistency model
  - *Cf. Consistency Lecture*

Self-study questions

- Would the Dynamo/Cassandra architecture make sense given a distributed file system layer for storage, argue for or against?
- How does the life of a read and write request issued from a client to Dynamo/Cassandra differ from requests issued in BigTable?