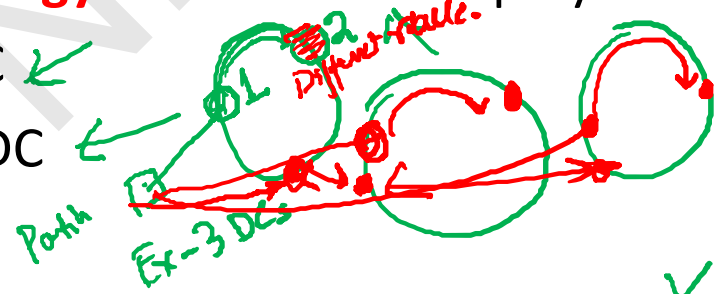
Data Placement Strategies

Data Placement Strategies

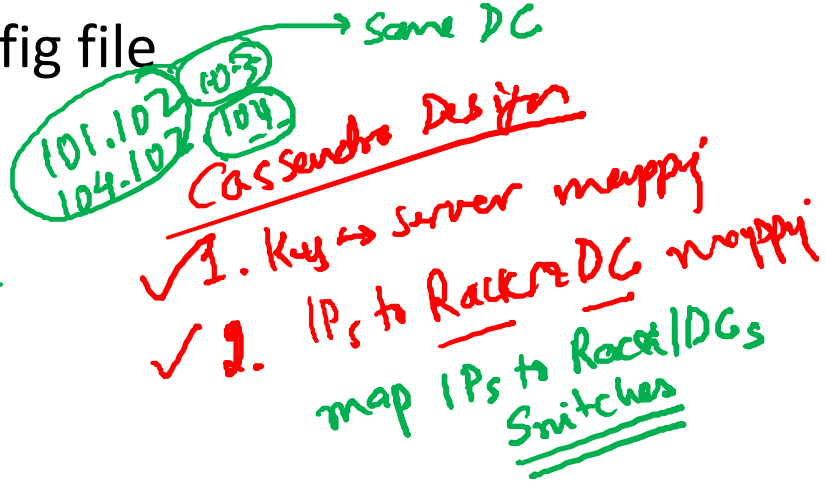
- **Replication Strategy:**
  1. *SimpleStrategy*
  2. *NetworkTopologyStrategy*

1. **SimpleStrategy:** uses the Partitioner, of which there are two kinds
   1. ***RandomPartitioner***: Chord-like hash partitioning
   2. ***ByteOrderedPartitioner***: Assigns ranges of keys to servers.
      - Easier for ***range queries*** (e.g., Get me all twitter users starting with [a-b])
2. **NetworkTopologyStrategy:** for multi-DC deployments
   - Two replicas per DC
   - Three replicas per DC
   - Per DC
     - First replica placed according to Partitioner
     - Then go clockwise around ring until you hit a different rack



Snitches

- **Maps:** IPs to racks and DCs. Configured in cassandra.yaml config file
- **Some options:**
  - **SimpleSnitch:** Unaware of Topology (Rack-unaware)
  - **RackInferring:** Assumes topology of network by octet of server's IP address
    - 101.102.103.104 = x.<DC octet>.<rack octet>.<node octet>
  - **PropertyFileSnitch:** uses a config file
  - **EC2Snitch:** uses EC2.
    - EC2 Region = DC
    - Availability zone = rack
- Other snitch options available



Writes

- Need to be lock-free and fast (no reads or disk seeks)
- Client sends write to one coordinator node in Cassandra cluster
  - Coordinator may be per-key, or per-client, or per-query
  - Per-key Coordinator ensures writes for the key are serialized
- Coordinator uses Partitioner to send query to all replica nodes responsible for key
- When X replicas respond, coordinator returns an acknowledgement to the client
  - X?

Writes (2)

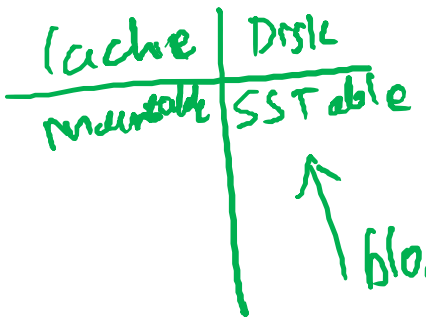
- **Always writable: Hinted Handoff mechanism**
  - If any replica is down, the coordinator writes to all other replicas, and keeps the write locally until down replica comes back up.
  - When all replicas are down, the Coordinator (front end) buffers writes (for up to a few hours).

- **One ring per datacenter**
  - Per-DC coordinator elected to coordinate with other DCs
  - Election done via Zookeeper, which runs a Paxos (consensus) variant

Writes at a replica node

On receiving a write

1. Log it in disk commit log (for failure recovery)
2. Make changes to appropriate memtables
   - **Memtable** = In-memory representation of multiple key-value pairs
   - *Typically append-only datastructure (fast)*
   - Cache that can be searched by key
   - Write-back aas opposed to write-through

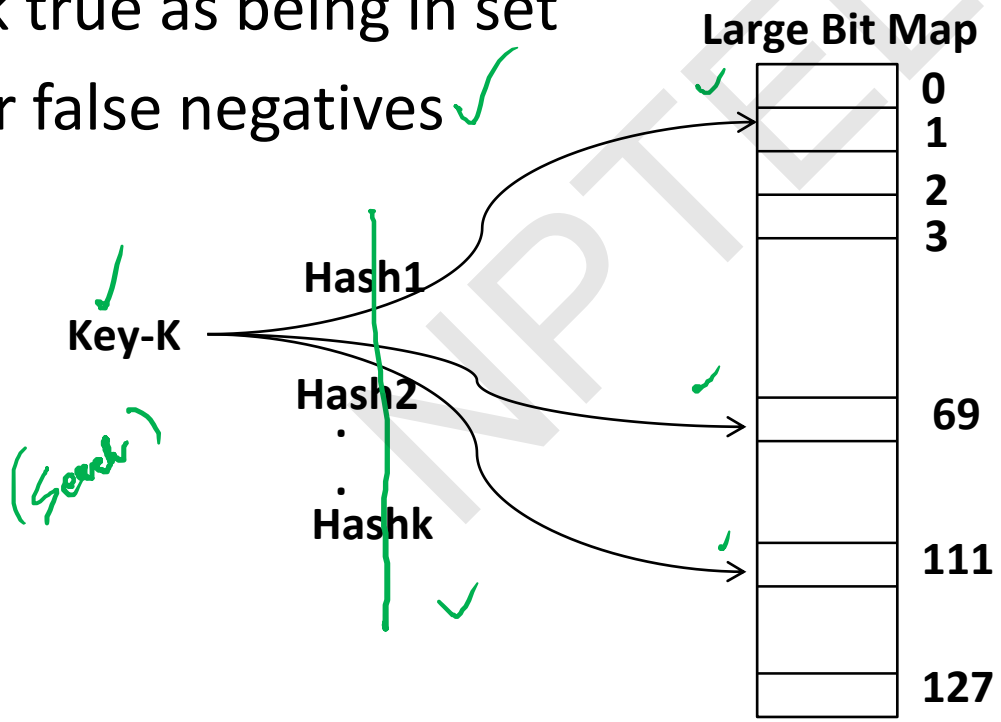


Later, when memtable is full or old, flush to disk

- Data File: An **SSTable** (Sorted String Table) – list of key-value pairs, sorted by key
- *SSTables are immutable (once created, they don't change)*
- Index file: An SSTable of (key, position in data sstable) pairs
- And a Bloom filter (for efficient search)

Bloom Filter

- Compact way of representing a set of items
- Checking for existence in set is cheap
- Some probability of false positives: an item not in set may check true as being in set
- Never false negatives



On insert, set all hashed bits.

On check-if-present, return true if all hashed bits set.

- False positives

False positive rate low

- m=4 hash functions
- 100 items
- 3200 bits
- FP rate = 0.02%

Compaction

Data updates accumulate over time and SStables and logs need to be compacted

- The process of compaction merges SSTables, i.e., by merging updates for a key
- Run periodically and locally at each server

Deletes

Delete: don't delete item right away

- Add a **tombstone** to the log
- Eventually, when compaction encounters tombstone it will delete item

Reads

Read: Similar to writes, except

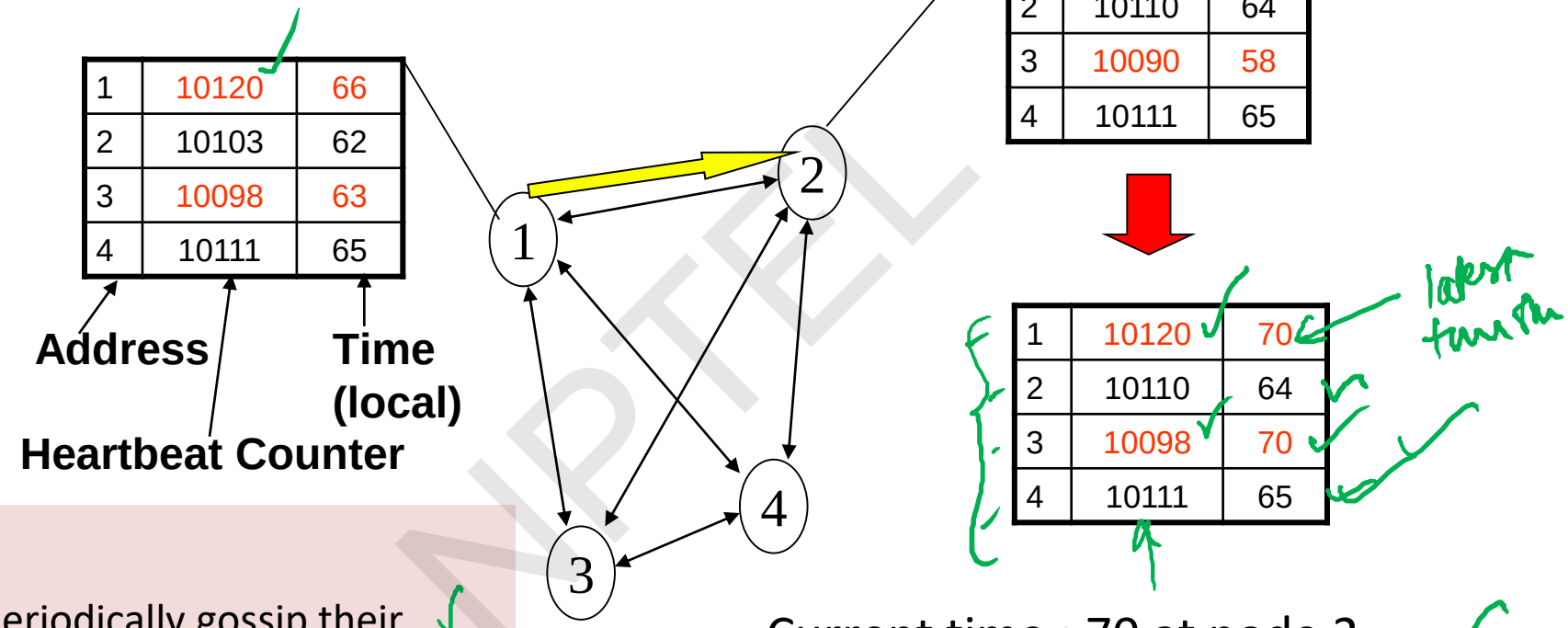
- **Coordinator can contact X replicas (e.g., in same rack)**
  - Coordinator sends read to replicas that have responded quickest in past
  - When X replicas respond, coordinator returns the latest-timestamped value from among those X
  - (X? We will check it later.)
- **Coordinator also fetches value from other replicas**
  - Checks consistency in the background, initiating a **read repair** if any two values are different
  - This mechanism seeks to eventually bring all replicas up to date
- **At a replica**
  - A row may be split across multiple SSTables => reads need to touch multiple SSTables => reads slower than writes (but still fast)

Membership

- Any server in cluster could be the coordinator
- So every server needs to maintain a list of all the other servers that are currently in the server
- List needs to be updated automatically as servers join, leave, and fail

Cluster Membership – Gossip-Style

Cassandra uses gossip-based cluster membership



Protocol:

- Nodes periodically gossip their membership list
- On receipt, the local membership list is updated, as shown
- If any heartbeat older than Tfail, node is marked as failed

Current time : 70 at node 2

(asynchronous clocks)

(Remember this?)

Suspicion Mechanisms in Cassandra

- Suspicion mechanisms to adaptively set the timeout based on underlying network and failure behavior
- **Accrual detector:** Failure Detector outputs a value (PHI) representing suspicion
- Applications set an appropriate threshold
- **PHI calculation for a member**
  - Inter-arrival times for gossip messages
  - PHI(t) =
    - log(CDF or Probability(t_now – t_last))/log 10
  - PHI basically determines the detection timeout, but takes into account historical inter-arrival time variations for gossiped heartbeats
- In practice, PHI = 5 => 10-15 sec detection time

Cassandra Vs. RDBMS

- MySQL is one of the most popular (and has been for a while)
- On > 50 GB data
- **MySQL**
  - Writes 300 ms avg
  - Reads 350 ms avg
- **Cassandra**
  - Writes 0.12 ms avg
  - Reads 15 ms avg
- Orders of magnitude faster
- What's the catch? What did we lose?

CAP Theorem

CAP Theorem

- **Proposed by Eric Brewer (Berkeley)**
- Subsequently proved by Gilbert and Lynch (NUS and MIT)
- In a distributed system you can satisfy atmost 2 out of the 3 guarantees:

1. **Consistency:** all nodes see same data at any time, or reads return latest written value by any client
2. **Availability:** the system allows operations all the time, and operations return quickly
3. **Partition-tolerance:** the system continues to work in spite of network partitions

CAP

CAP Out of 3 | atmost 2 guaranteed by system → Design issue →

Why is Availability Important?

- **Availability =** Reads/writes complete reliably and quickly.
- Measurements have shown that a 500 ms increase in latency for operations at Amazon.com or at Google.com can cause a 20% drop in revenue.
- At Amazon, each added millisecond of latency implies a $6M yearly loss.
- **User cognitive drift:** If more than a second elapses between clicking and material appearing, the user's mind is already somewhere else → Churn in customer in any business →
- SLAs (Service Level Agreements) written by providers predominantly deal with latencies faced by clients.

Why is Consistency Important?

- **Consistency =** all nodes see same data at any time, or reads return latest written value by any client.

- When you access your bank or investment account via multiple clients (laptop, workstation, phone, tablet), you want the updates done from one client to be visible to other clients.

- When thousands of customers are looking to book a flight, all updates from any client (e.g., book a flight) should be accessible by other clients.

Why is Partition-Tolerance Important?

- **Partitions** can happen across datacenters when the Internet gets disconnected
  - Internet router outages
  - Under-sea cables cut
  - DNS not working
- Partitions can also occur within a datacenter, e.g., a rack switch outage
- Still desire system to continue functioning normally under this scenario

Partition tolerance ensures the system is in face of possibility/network failure

CAP Theorem Fallout

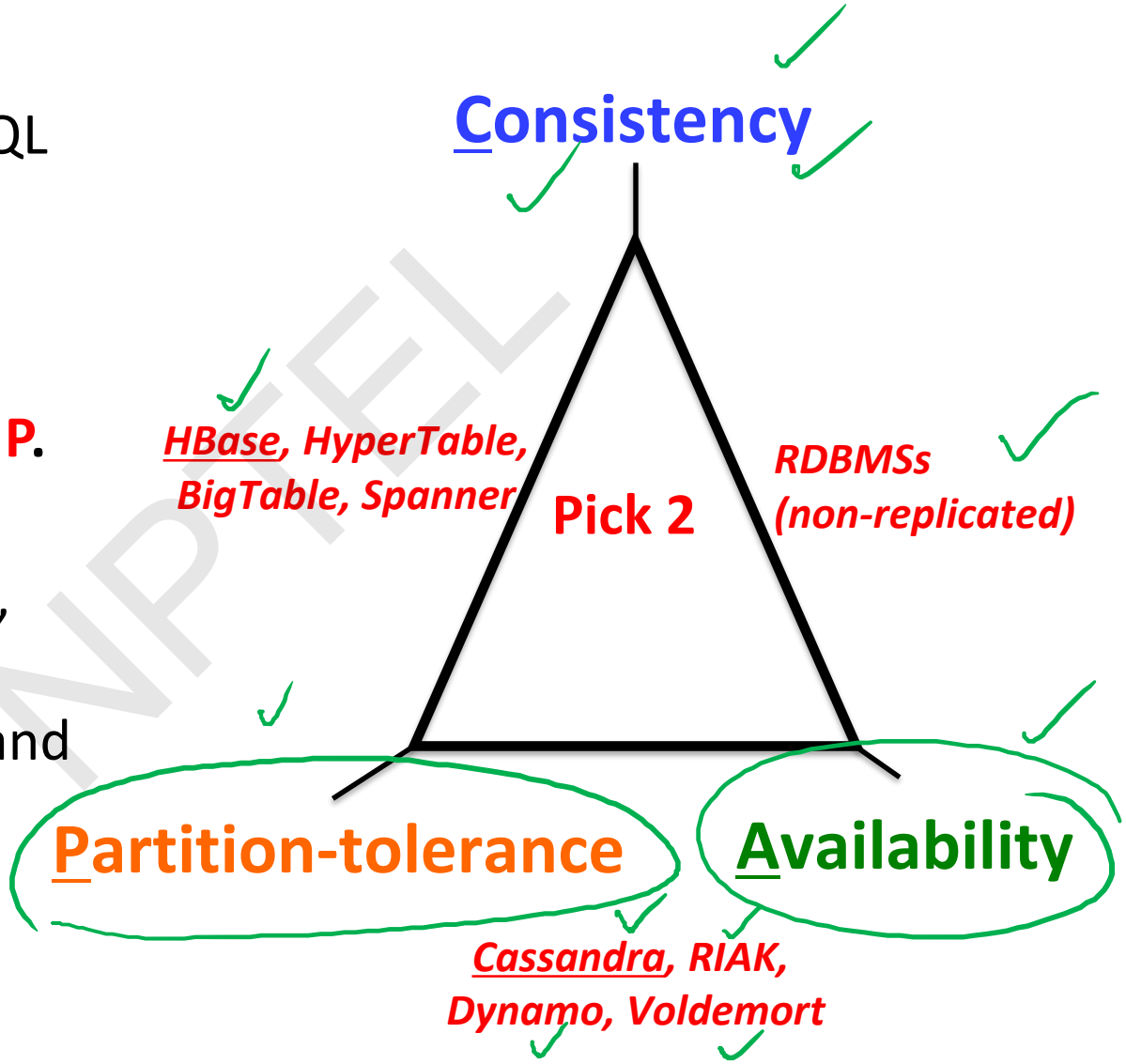
- Since partition-tolerance is essential in today's cloud computing systems, CAP theorem implies that a system has to choose between consistency and availability

- **Cassandra**
  - Eventual (weak) consistency, Availability, Partition-tolerance

- **Traditional RDBMSs**
  - Strong consistency over availability under a partition

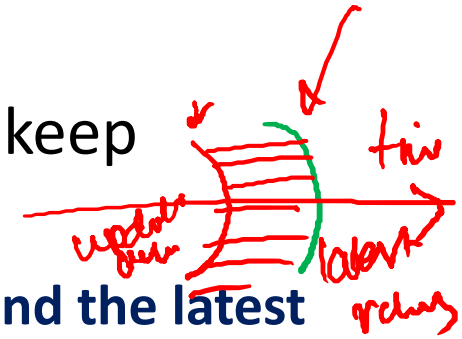
CAP Tradeoff

- Starting point for NoSQL Revolution
- A distributed storage system can achieve **at most two of C, A, and P.**
- When partition-tolerance is important, you have to choose between consistency and availability



Eventual Consistency

- If all writes stop (to a key), then all its values (replicas) will converge eventually.
- If writes continue, then system always tries to keep converging.
  - **Moving "wave" of updated values lagging behind the latest values sent by clients, but always trying to catch up.**
- May still return stale values to clients (e.g., if many back-to-back writes).
- But works well when there a few periods of low writes – system converges quickly.



RDBMS vs. Key-value stores

- While RDBMS provide **ACID**
  - **A**tomicity
  - **C**onsistency
  - **I**solation
  - **D**urability

- Key-value stores like Cassandra provide **BASE**
  - **B**asically **A**vailable **S**oft-state **E**ventual Consistency
  - Prefers Availability over Consistency

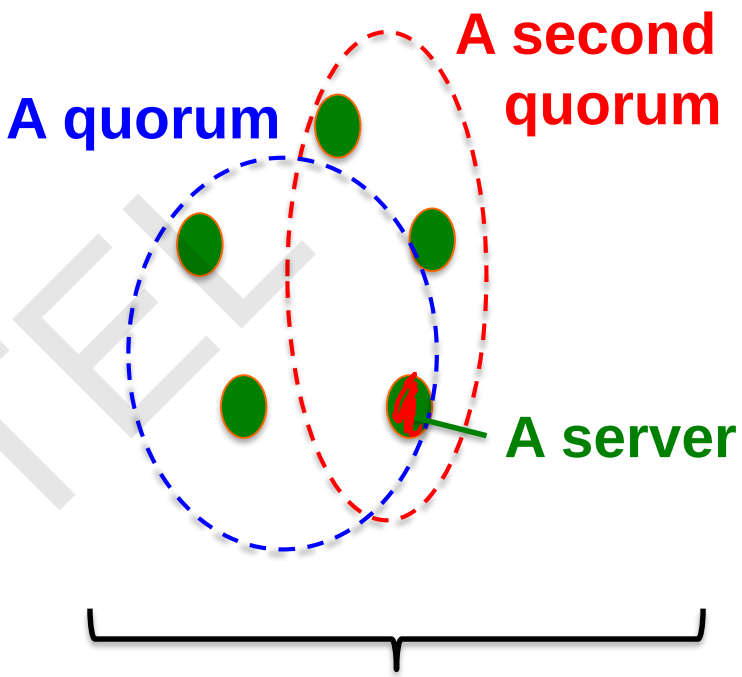
Consistency in Cassandra

- Cassandra has **consistency levels**
- Client is allowed to choose a consistency level for each operation (read/write)
  1. **ANY:** any server (may not be replica)
     - Fastest: coordinator caches write and replies quickly to client
  2. **ALL:** all replicas
     - Ensures strong consistency, but slowest
  3. **ONE:** at least one replica
     - Faster than ALL, but cannot tolerate a failure
  4. **QUORUM:** quorum across all replicas in all datacenters (DCs)
     - What?

Quorums for Consistency

In a nutshell:

- Quorum = majority
  - > 50%
- Any two quorums intersect
  - Client 1 does a write in red quorum
  - Then client 2 does read in blue quorum
- At least one server in blue quorum returns latest write
- Quorums faster than ALL, but still ensure strong consistency



Five replicas of a key-value pair

Quorums in Detail

- Several key-value/NoSQL stores (e.g., Riak and Cassandra) use quorums.
- **Reads**
  - Client specifies value of **R** (≤ N = total number of replicas of that key).
  - R = read consistency level.
  - Coordinator waits for R replicas to respond before sending result to client.
  - In background, coordinator checks for consistency of remaining (N-R) replicas, and initiates read repair if needed.

Quorums in Detail (Contd..)

- Writes come in two flavors
  - Client specifies W (≤ N)
  - W = write consistency level.
  - Client writes new value to W replicas and returns. Two flavors:
    - Coordinator blocks until quorum is reached.
    - Asynchronous: Just write and return.

Quorums in Detail (Contd.)

- R = read replica count, W = write replica count
- Two necessary conditions:
  1. **W+R > N**
  2. **W > N/2**
- Select values based on application
  - **(W=1, R=1):** very few writes and reads
  - **(W=N, R=1):** great for read-heavy workloads
  - **(W=N/2+1, R=N/2+1):** great for write-heavy workloads
  - **(W=1, R=N):** great for write-heavy workloads with mostly one client writing per key

Cassandra Consistency Levels (Contd.)

- Client is allowed to choose a consistency level for each operation (read/write)
  - ANY: any server (may not be replica)
    - Fastest: coordinator may cache write and reply quickly to client
  - ALL: all replicas
    - Slowest, but ensures strong consistency
  - ONE: at least one replica
    - Faster than ALL, and ensures durability without failures
  - **QUORUM:** quorum across all replicas in all datacenters (DCs)
    - Global consistency, but still fast
  - **LOCAL_QUORUM:** quorum in coordinator's DC
    - Faster: only waits for quorum in first DC client contacts
  - **EACH_QUORUM:** quorum in every DC
    - Lets each DC do its own quorum: supports hierarchical replies

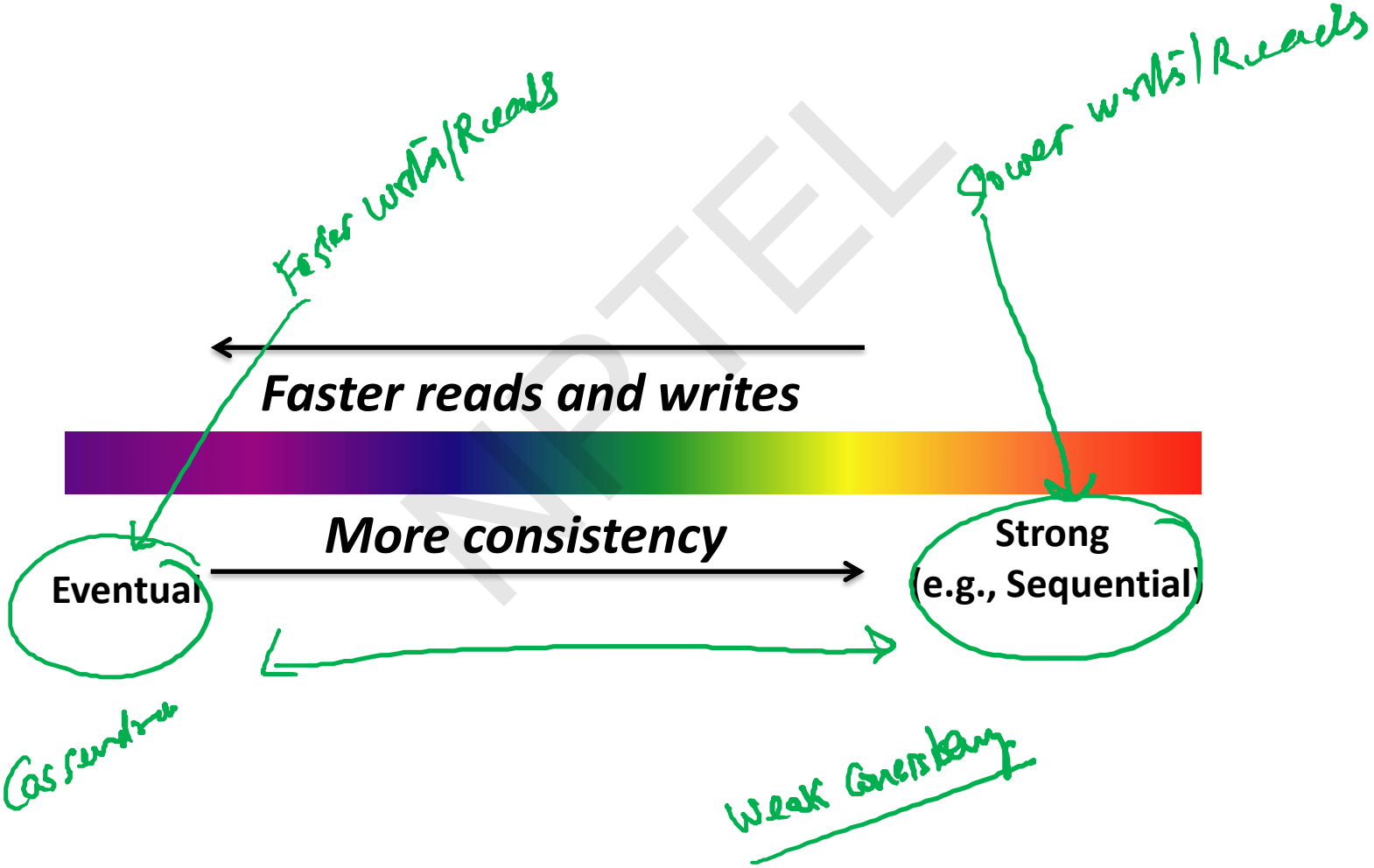
Types of Consistency

- Cassandra offers **Eventual Consistency**

- Are there other types of weak consistency models?

Consistency Solutions

Consistency Solutions

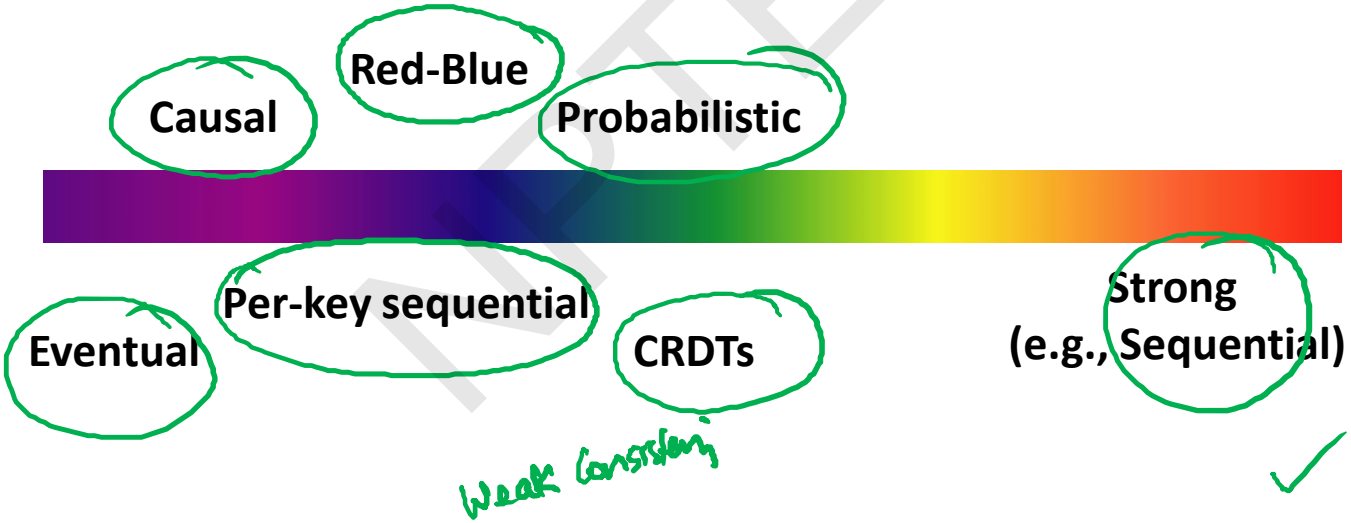


Eventual Consistency

- Cassandra offers **Eventual Consistency**
  - If writes to a key stop, all replicas of key will converge
  - Originally from Amazon's Dynamo and LinkedIn's Voldemort systems

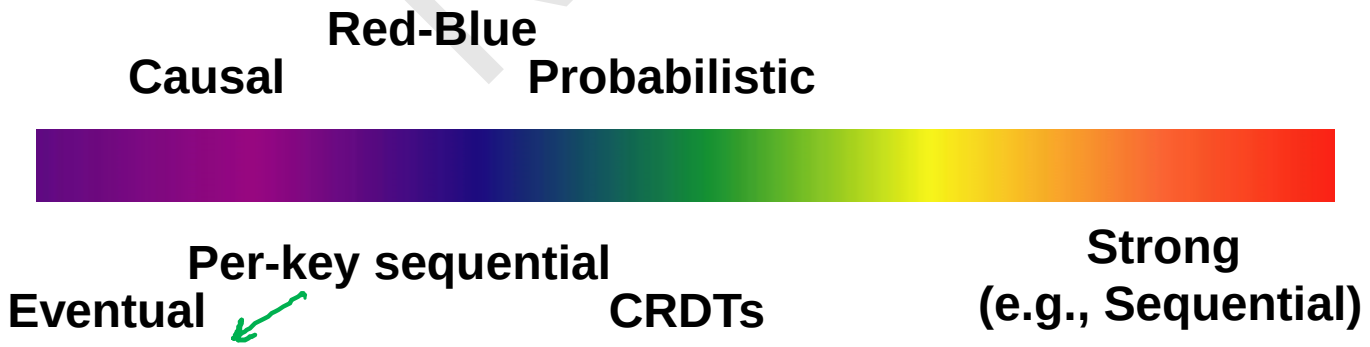
Newer Consistency Models

- Striving towards strong consistency
- While still trying to maintain high availability and partition-tolerance



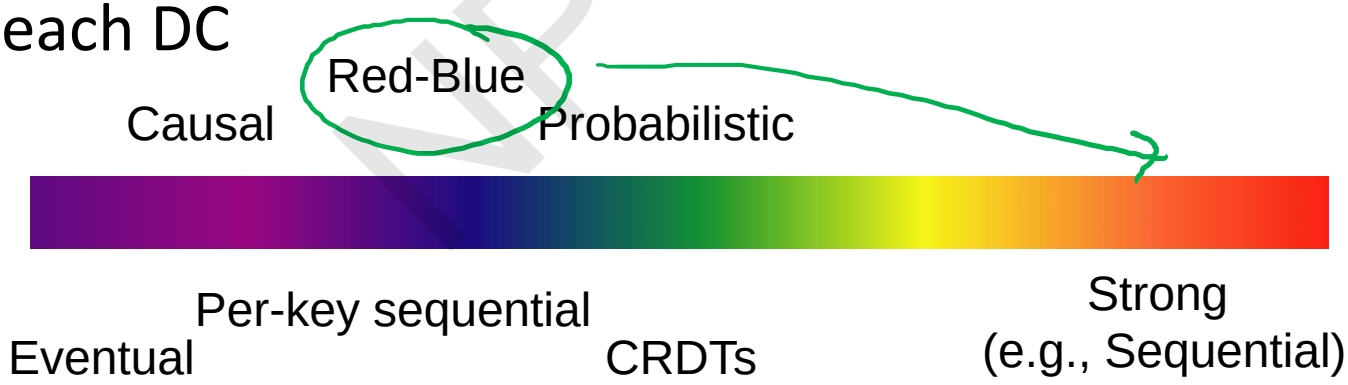
Newer Consistency Models (Contd.)

- **Per-key sequential**: Per key, all operations have a global order
- **CRDTs** (Commutative Replicated Data Types): Data structures for which commutated writes give same result [INRIA, France]
  - E.g., value == int, and only op allowed is +1
  - Effectively, servers don't need to worry about consistency



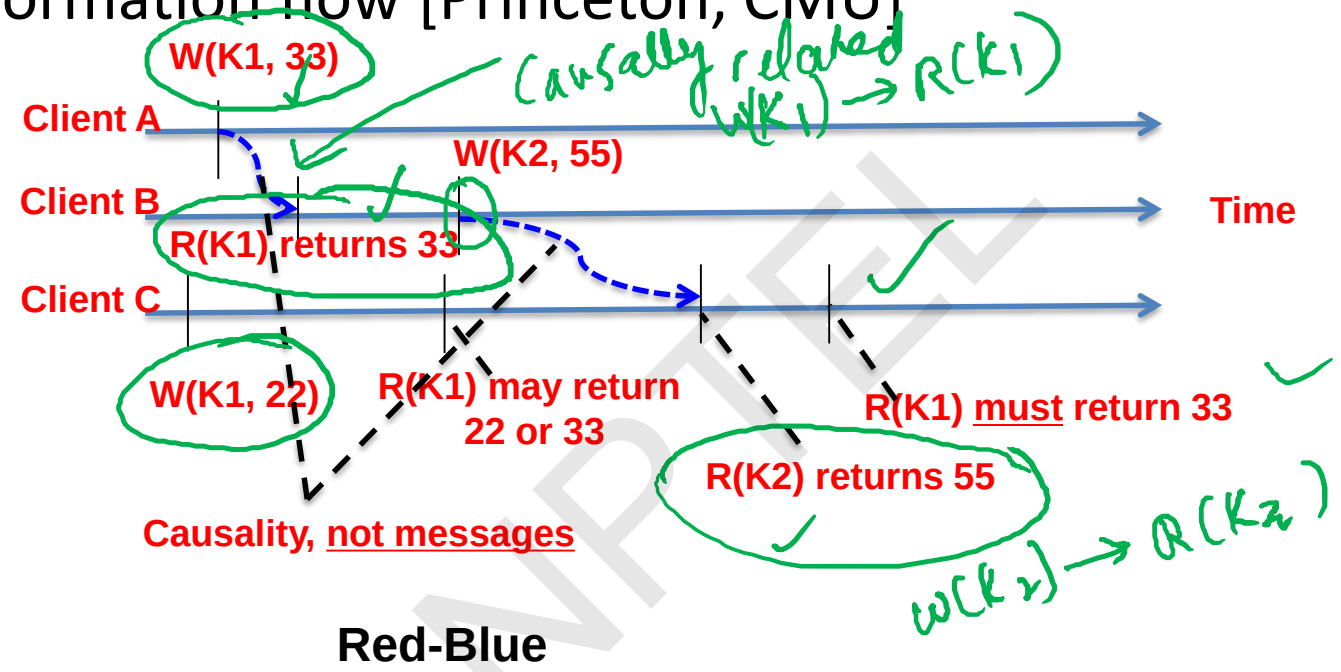
Newer Consistency Models (Contd.)

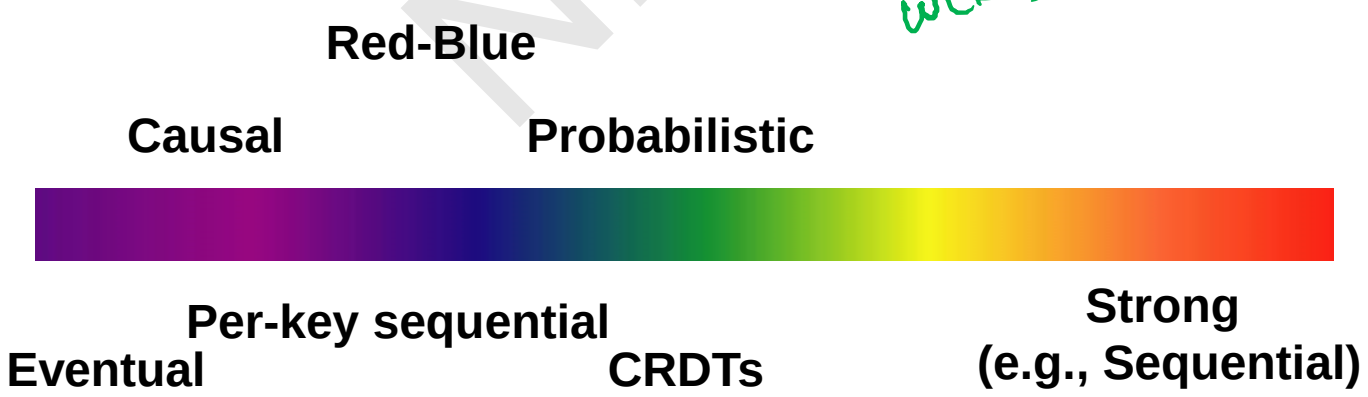
- **Red-blue Consistency**: Rewrite client transactions to separate operations into red operations vs. blue operations [MPI-SWS Germany]
  - Blue operations can be executed (commutated) in any order across DCs
  - Red operations need to be executed in the same order at each DC



Newer Consistency Models (Contd.)

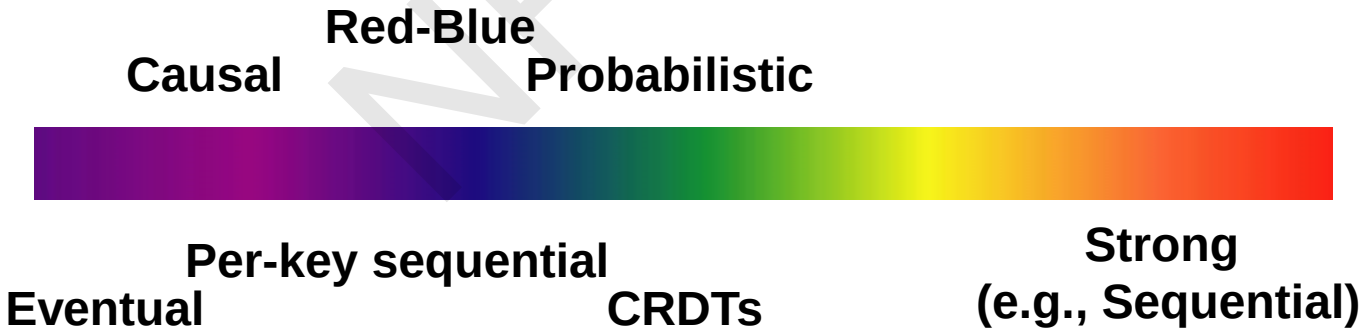
Causal Consistency: Reads must respect partial order based on information flow [Princeton, CMU]





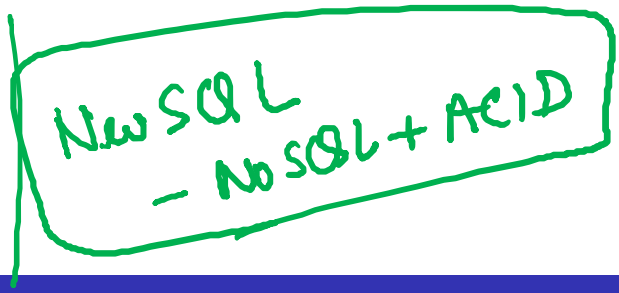
Which Consistency Model should you use?

- Use the lowest consistency (to the left) consistency model that is "correct" for your application
  - Gets you fastest availability



Strong Consistency Models

- **Linearizability**: Each operation by a client is visible (or available) **instantaneously** to all other clients
  - Instantaneously in real time
- **Sequential Consistency** [Lamport]:
  - *"... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*
  - After the fact, find a "reasonable" ordering of the operations (can re-order operations) that obeys sanity (consistency) at all clients, and across clients.
- Transaction ACID properties, example: newer key-value/NoSQL stores (sometimes called **"NewSQL")**
  - Hyperdex [Cornell]
  - Spanner [Google]
  - Transaction chains [Microsoft Research]



Conclusion

- Traditional Databases (RDBMSs) work with strong consistency, and offer ACID
- Modern workloads don't need such strong guarantees, but do need fast response times (availability)
- Unfortunately, CAP theorem
- **Key-value/NoSQL systems offer BASE**
  [Basically Available Soft-state Eventual Consistency]
  - Eventual consistency, and a variety of other consistency models striving towards strong consistency
- We have also discussed the design of Cassandra and different consistency solutions.

CQL
(Cassandra Query Language)

Dr. Rajiv Misra
Dept. of Computer Science & Engg.
Indian Institute of Technology Patna
rajivm@iitp.ac.in

Preface

Content of this Lecture:

- In this lecture, we will discuss CQL (Cassandra Query Language) Mapping to Cassandra's Internal Data Structure.

What Problems does CQL Solve?

- **The Awesomeness that is Cassandra:**

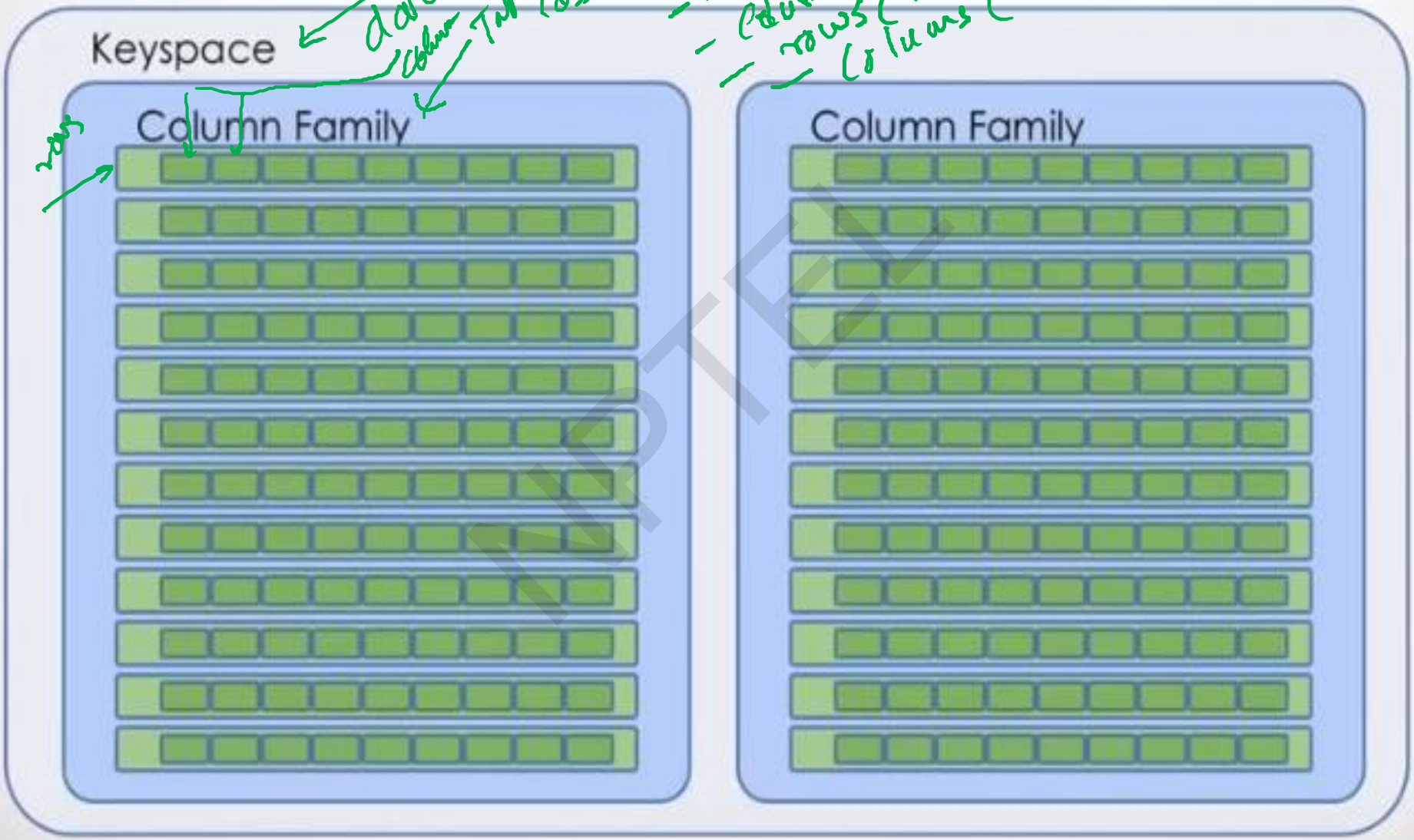
  - Distributed columnar data store
  - No single point of failure
  - Optimized for availability (through "Tunably" consistent
  - Optimized for writes
  - Easily maintainable
  - Almost infinitely scalable

What Problems does CQL Solve? (Contd.)

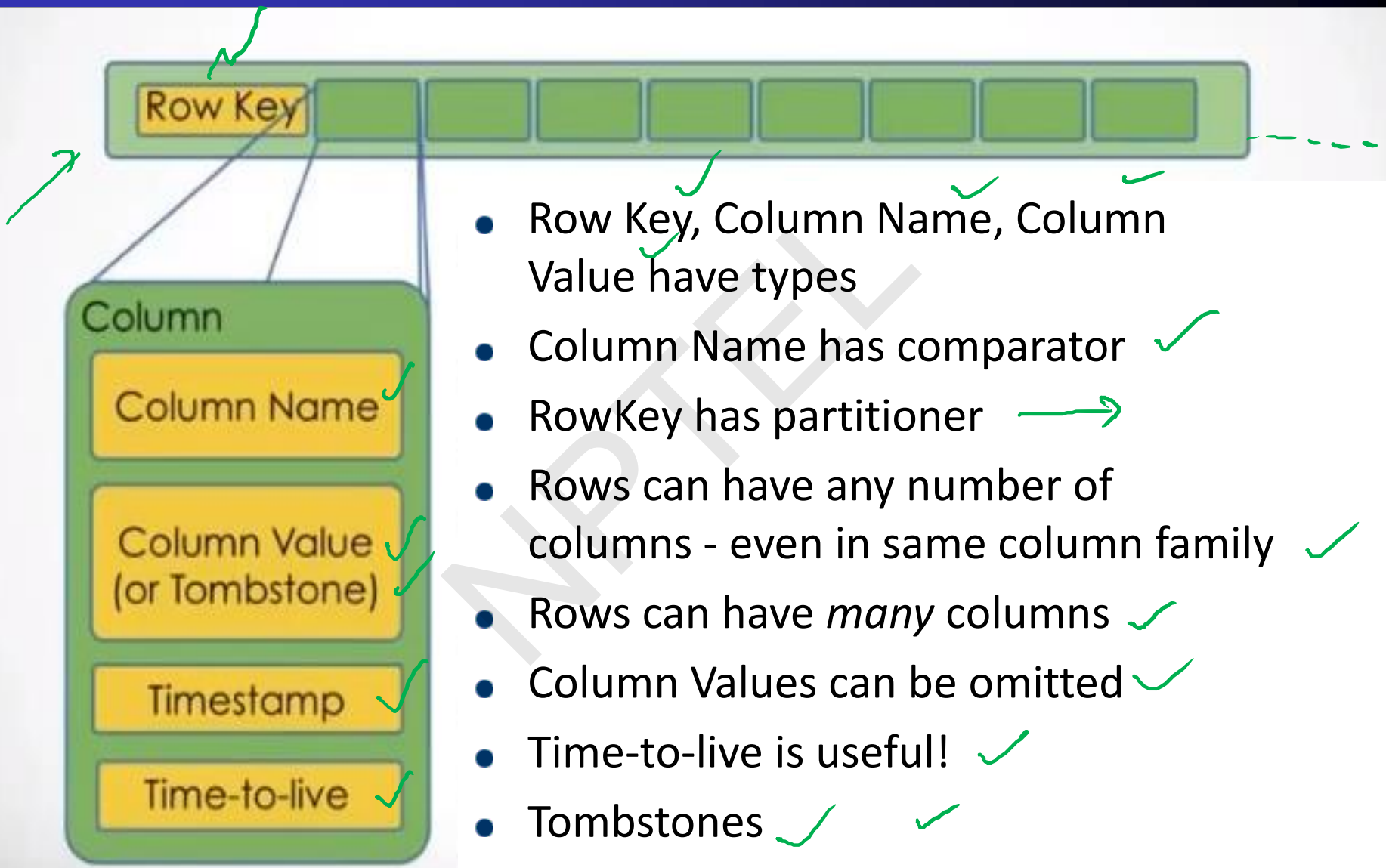
- **Cassandra's usability challenges**
  - NoSQL: "Where are my JOINS? No Schema? De-normalize!?"
  - BigTable: "Tables with millions of columns!?"

- **CQL Saves the day!**
  - A *best-practices* interface to Cassandra
  - Uses familiar SQL-Like language

C* Data Model

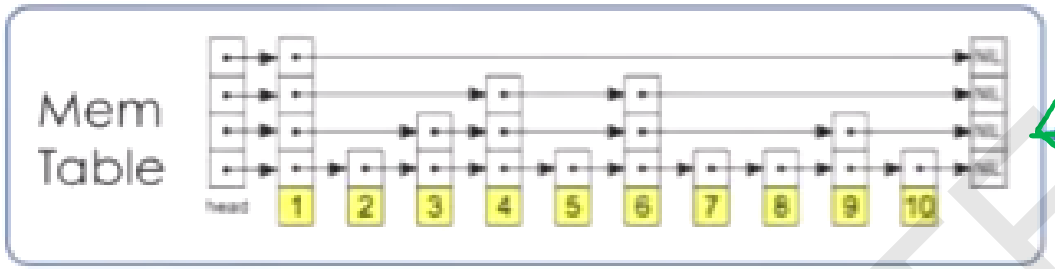


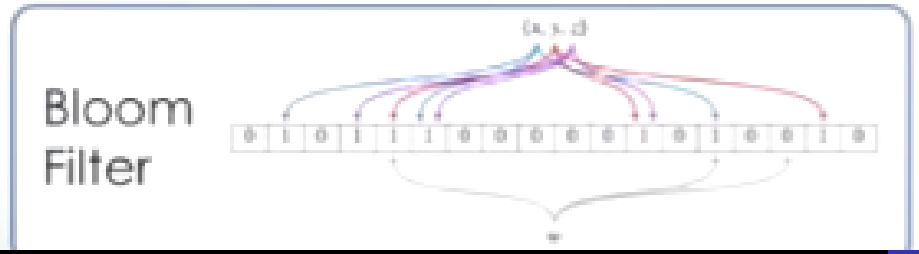
C* Data Model (Contd.)



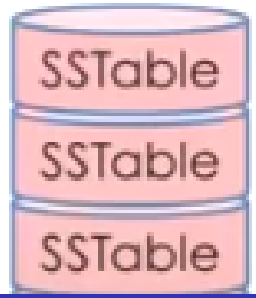
- Row Key, Column Name, Column Value have types
- Column Name has comparator
- RowKey has partitioner
- Rows can have any number of columns - even in same column family
- Rows can have *many* columns
- Column Values can be omitted
- Time-to-live is useful!
- Tombstones

C* Data Model: Writes

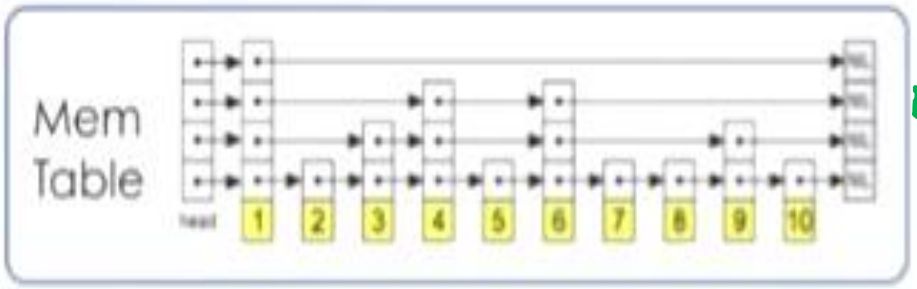


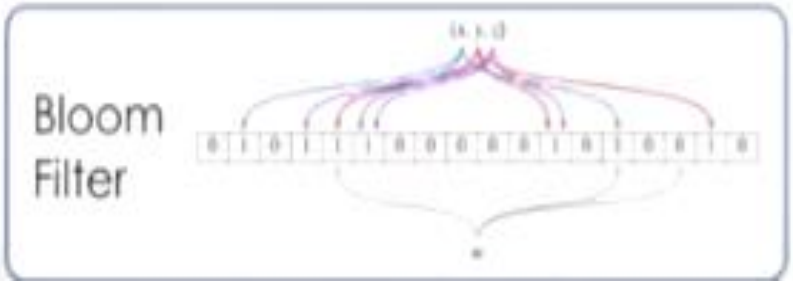


- Insert into MemTable
- Dump to CommitLog
- No read
- Very Fast!
- Blocks on CPU before O/I!

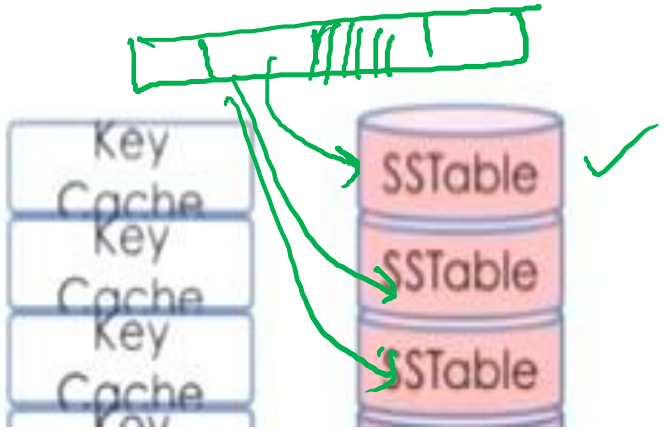


C* Data Model: Reads

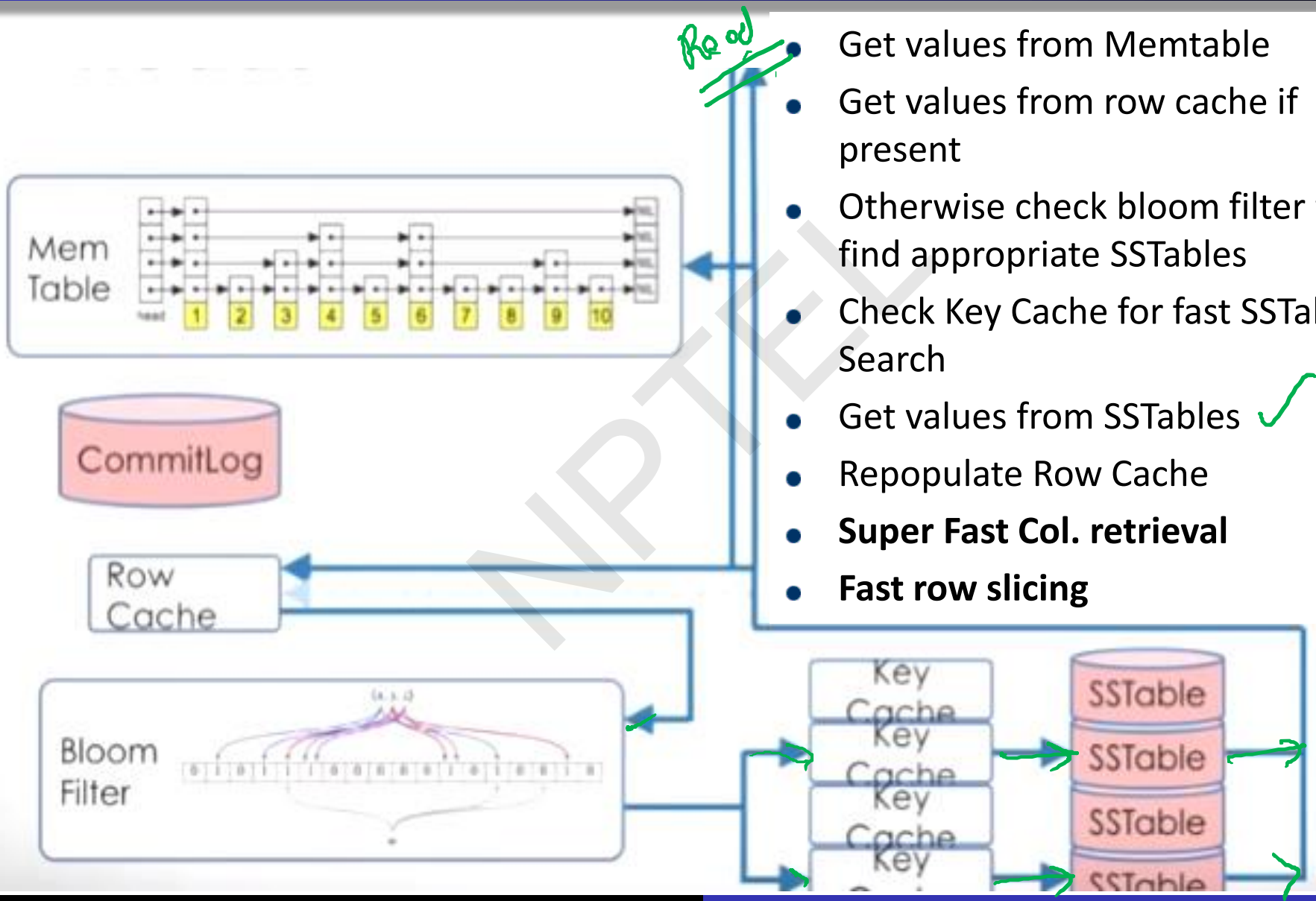




- Get values from Memtable
- Get values from row cache if present
- Otherwise check bloom filter to find appropriate SSTables
- Check Key Cache for fast SSTable Search
- Get values from SSTables
- Repopulate Row Cache
- **Super Fast Col. retrieval**
- **Fast row slicing**



C* Data Model: Reads (Contd.)



- Get values from Memtable
- Get values from row cache if present
- Otherwise check bloom filter to find appropriate SSTables
- Check Key Cache for fast SSTable Search
- Get values from SSTables
- Repopulate Row Cache
- **Super Fast Col. retrieval**
- **Fast row slicing**

Introducing CQL

- **CQL is a reintroduction of schema** so that you don't have to read code to understand the data model.
- **CQL creates a common language** so that details of the data model can be easily communicated.
- **CQL is a best-practices Cassandra interface** and hides the messy details.

Introducing CQL (Contd.)

```
CREATE TABLE users (
  id timeuuid PRIMARY KEY,
  lastname varchar,
  firstname varchar,
  dateOfBirth timestamp );

INSERT INTO users (id,lastname, firstname, dateofbirth)
  VALUES (now(),'Berryman','John','1975-09-15');

UPDATE users SET firstname = 'John'
  WHERE id = f74c0b20-0862-11e3-8cf6-b74c10b01fc6;

SELECT dateofbirth,firstname,lastname FROM users ;

 dateofbirth              | firstname | lastname
--------------------------+-----------+----------
 1975-09-15 00:00:00-0400 |      John | Berryman
```

Remember this:

- Cassandra finds rows fast ✓ (partition on row key)
- Cassandra scans columns fast
- Cassandra *does not* scan rows

The CQL/Cassandra Mapping

```
CREATE TABLE employees (
  name text PRIMARY KEY,
  age int,
  role text
);
```

```
name | age | role
-----+-----+-----
john | 37  | dev
eric | 38  | ceo
```

Rows are partitioned in Cassandra

	age	role
john	37	dev

row-key — Column values

	age	role
eric	38	ceo

row-key — Column values

The CQL/Cassandra Mapping

```
CREATE TABLE employees (
  company text,
  name text,
  age int,
  role text,
  PRIMARY KEY (company,name)
);
```

company	name	age	role
OSC	eric	38	ceo
OSC	john	37	dev
RKG	anya	29	lead
RKG	ben	27	dev
RKG	chad	35	ops

	eric:age	eric:role	john:age	john:role
OSC	38	dev	37	dev

	anya:age	anya:role	ben:age	ben:role	chad:age	chad:role
RKG	29	lead	27	dev	35	ops

The CQL/Cassandra Mapping (Contd.)

```
CREATE TABLE example (
  A text,
  B text,
  C text,
  D text,
  E text,
  F text,
  PRIMARY KEY ((A,B),C,D)
);
```

```
A | B | C | D | E | F
--+---+---+---+---+---
a | b | c | d | e | f
a | b | c | g | h | i
a | b | j | k | l | m
a | n | o | p | q | r
s | t | u | v | w | x
```

	c:d:E	c:d:F	c:g:E	c:g:F	j:k:E	j:k:F
a:b	e	f	h	i	l	m

	o:p:E	o:p:F
a:n	q	r

	u:v:E	u:v:F
s:t	w	x

CQL for Sets, Lists and Maps

- Collection Semantics
  - Sets hold list of unique elements
  - Lists hold ordered, possibly repeating elements
  - Maps hold a list of key-value pairs
- Uses same old Cassandra datastructure
- Declaring

```
CREATE TABLE mytable(
  X text,
  Y text,
  myset set<text>,
  mylist list<int>,
  mymap map<text, text>,
  PRIMARY KEY (X,Y)
);
```

Cannot be primary key

Collection fields can not be used in primary keys

Inserting

INSERT INTO mytable (row, **myset**)
VALUES (123, **{ 'apple', 'banana'}**);

INSERT INTO mytable (row, **mylist**)
VALUES (123, **['apple','banana','apple']**);

INSERT INTO mytable (row, **mymap**)
VALUES (123, **{1:'apple',2:'banana'}**)

Updating

UPDATE mytable SET **myset = myset + {‘apple’,‘banana’}**

WHERE row = 123;

UPDATE mytable SET **myset = myset - { ‘apple’ }**

WHERE row = 123;

UPDATE mytable SET **mylist = mylist + [‘apple’,‘banana’]**

WHERE row = 123;

UPDATE mytable SET **mylist = [‘banana’] + mylist**

WHERE row = 123;

UPDATE mytable SET **mymap[‘fruit’] = ‘apple’**

WHERE row = 123

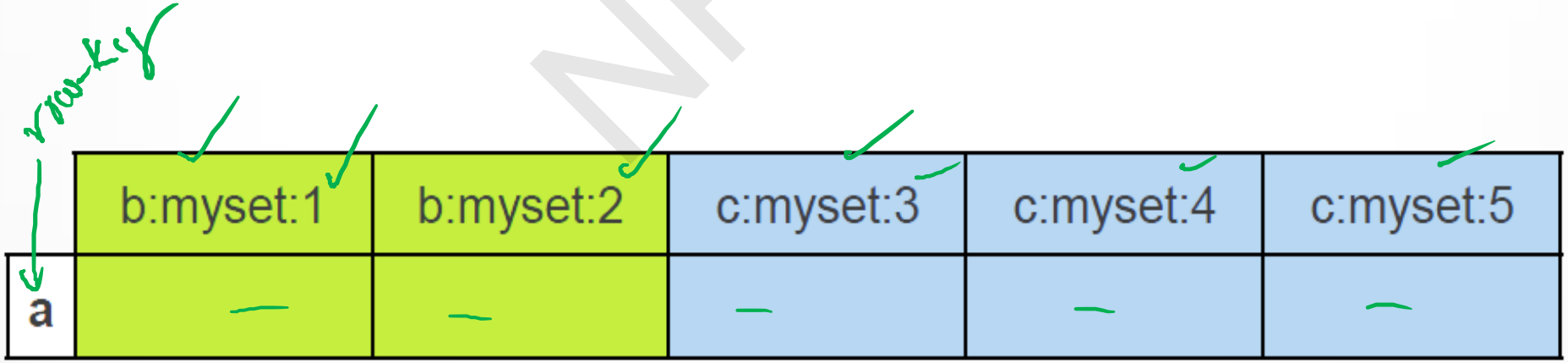
UPDATE mytable SET **mymap = mymap + { ‘fruit’:‘apple’}**

WHERE row = 123

SETS

```
CREATE TABLE mytable(
  X text,
  Y text,
  myset set<int>,
  PRIMARY KEY (X,Y)
);
```

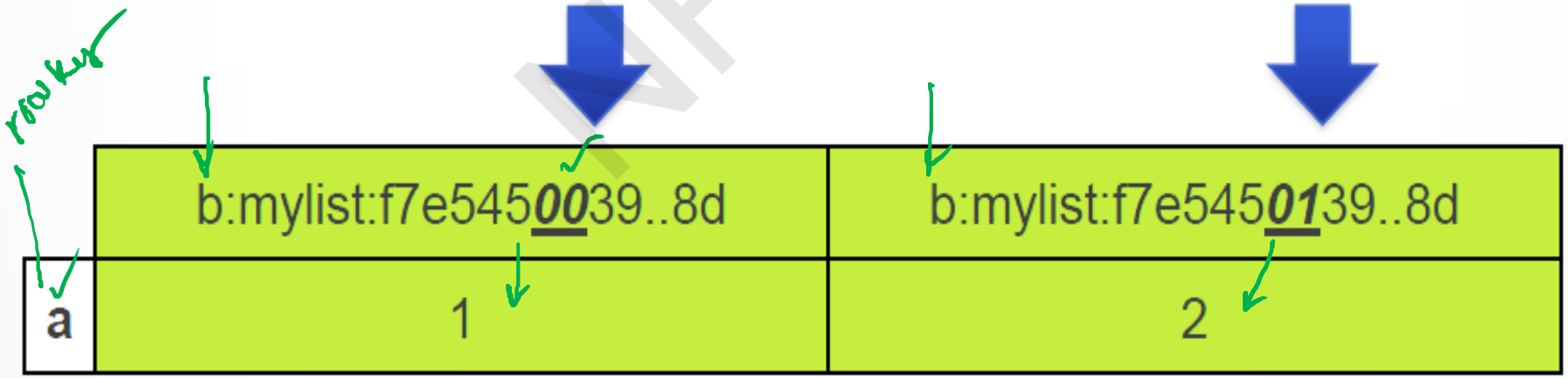
```
 X | Y |   myset
---+---+-----------
 a | b |   {1,2}
 a | c |   {3,4,5}
```



LISTS

```
CREATE TABLE mytable(
  X text,
  Y text,
  mylist list<int>,
  PRIMARY KEY (X,Y)
);
```

```
 X | Y |   mylist
---+---+-----------
 a | b |   [1,2]
```



MAPS

```
CREATE TABLE mytable(
  X text,
  Y text,
  mymap map<text,int>,
  PRIMARY KEY (X,Y)
);
```

```
 X | Y |   mymap
---+---+-----------
 a | b | {m:1,n:2}
 a | c |{n:3,p:4,q:5}
```

	b:mymap:m	b:mymap:n	c:mymap:n	c:mymap:p	c:mymap:q
a	1	2	3	4	5

Example

(in cqlsh)

```
CREATE KEYSPACE test WITH replication =
    {'class': 'SimpleStrategy', 'replication_factor': 1};
USE test;
CREATE TABLE stuff ( a int, b int, myset set<int>,
    mylist list<int>, mymap map<int,int>, PRIMARY KEY (a,b));
UPDATE stuff SET myset = {1,2}, mylist = [3,4,5], mymap = {6:7,8:9} WHERE a = 0
AND b = 1;
SELECT * FROM stuff;
```

(in cassandra-cli)

```
use test;
list stuff ;
```

(in cqlsh)

```
SELECT key_aliases,column_aliases from system.schema_columnfamilies WHERE
keyspace_name = 'test' AND columnfamily_name = 'stuff';
```

Conclusion

- CQL is a reintroduction of schema
- CQL creates a common data modeling language
- CQL is a best-practices Cassandra interface
- CQL let's you take advantage of the C* Data structure
- CQL protocol is binary and therefore interoperable with any language
- CQL is asynchronous and fast (Thrift transport layer is synchronous)
- CQL allows the possibility for prepared statements

Design of Zookeeper

Dr. Rajiv Misra
Dept. of Computer Science & Engg.
Indian Institute of Technology Patna
rajivm@iitp.ac.in

Preface

Content of this Lecture:

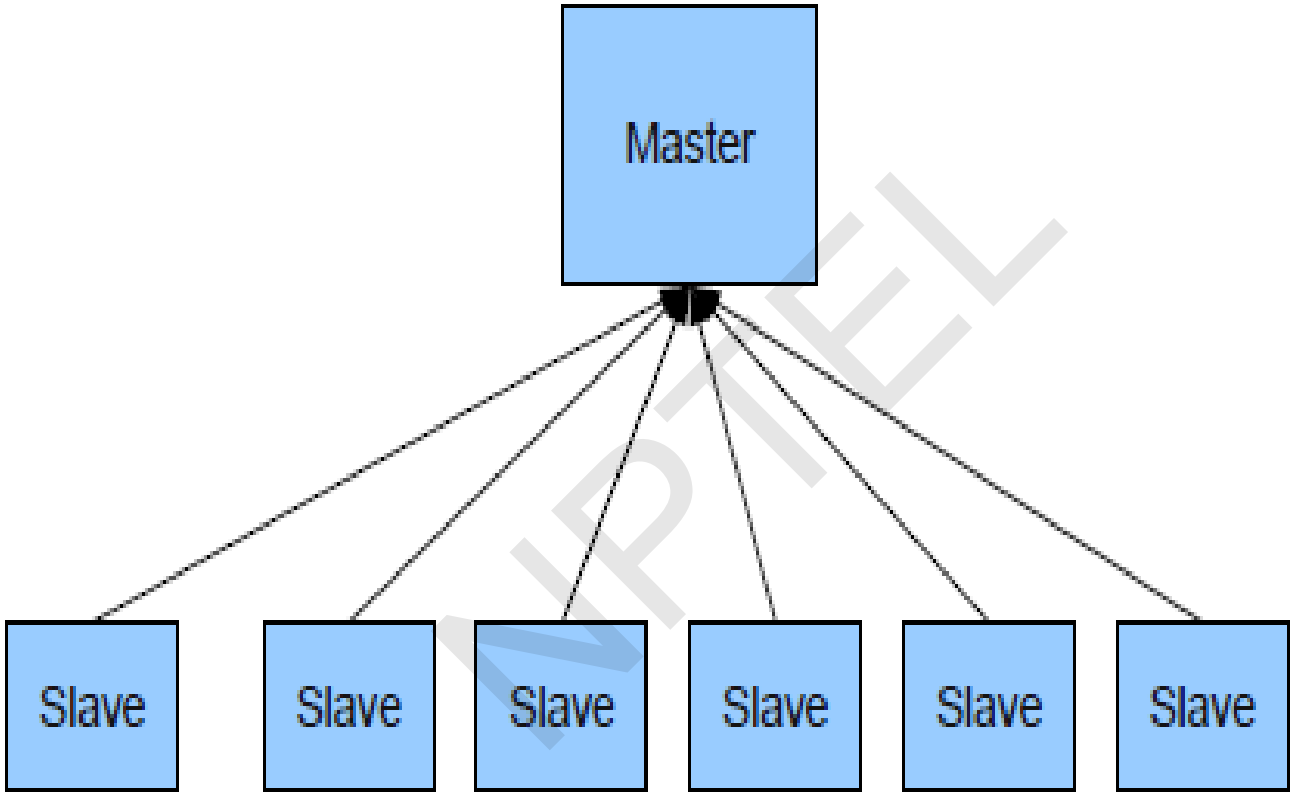
- In this lecture, we will discuss the '**design of ZooKeeper'**, which is a service for coordinating processes of distributed applications.
- We will discuss its basic fundamentals, design goals, architecture and applications.

https://zookeeper.apache.org/

ZooKeeper, why do we need it?

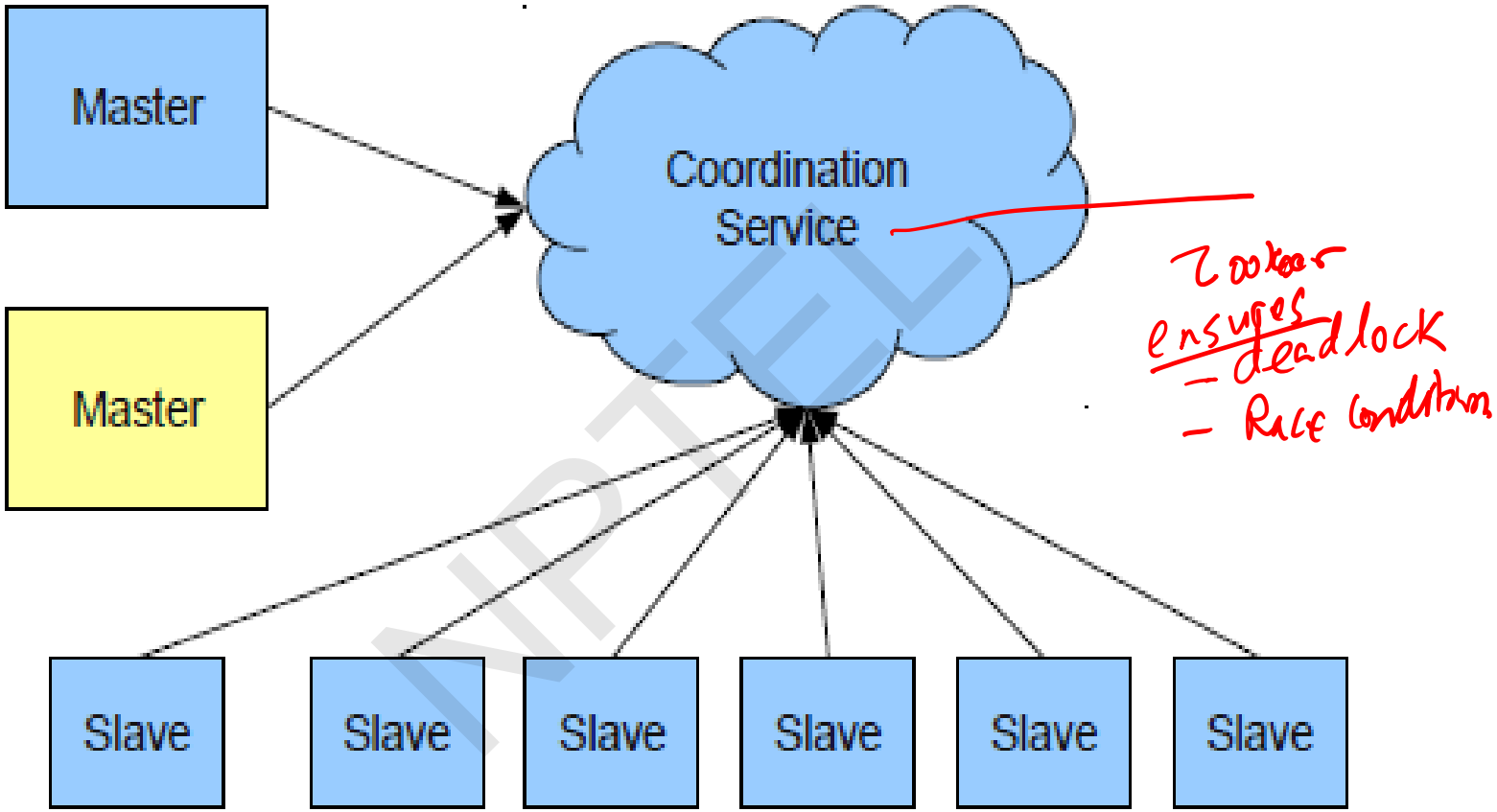
- **Coordination is important**

Classic Distributed System



- Most of the system like HDFS have one Master and couple of slave nodes and these slave nodes report to the master.

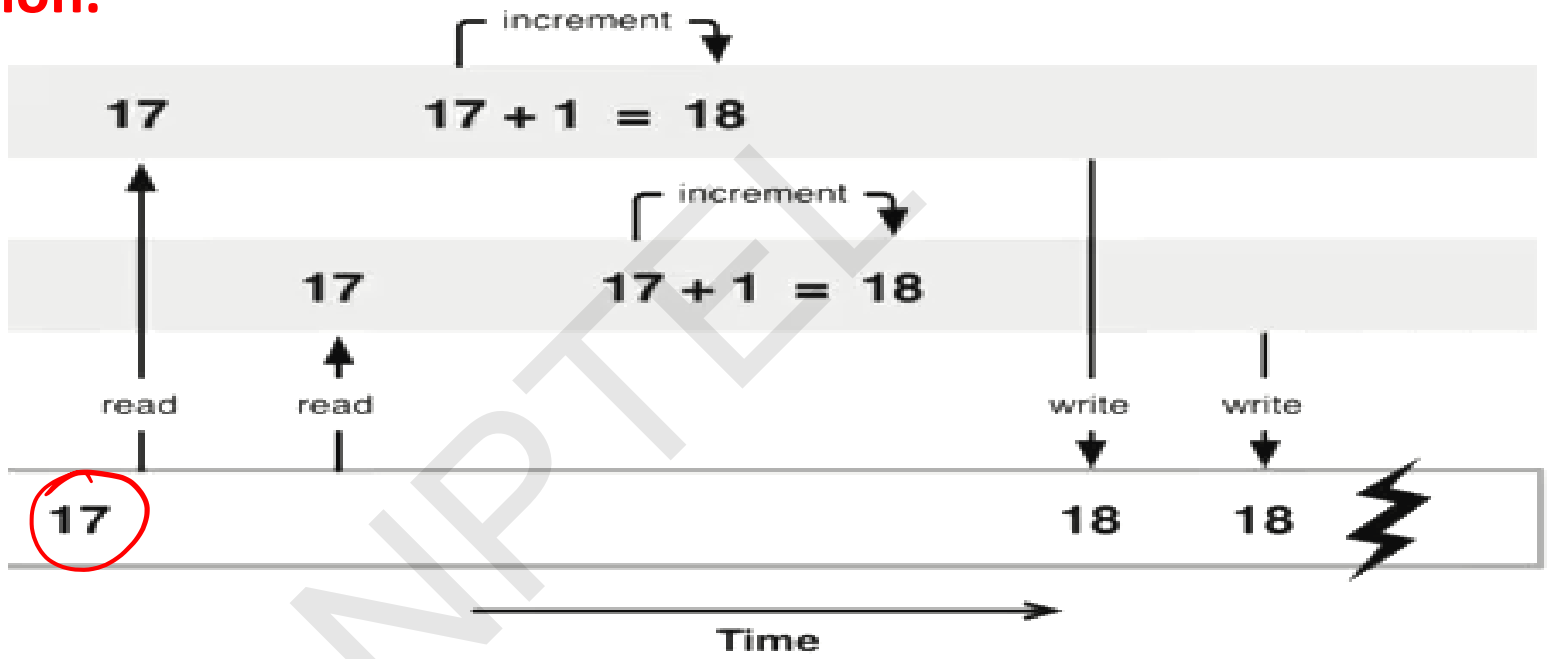
Fault Tolerant Distributed System



- Real distributed fault tolerant system have Coordination service, Master and backup master.
- If primary failed then backup works for it.

What is a Race Condition?

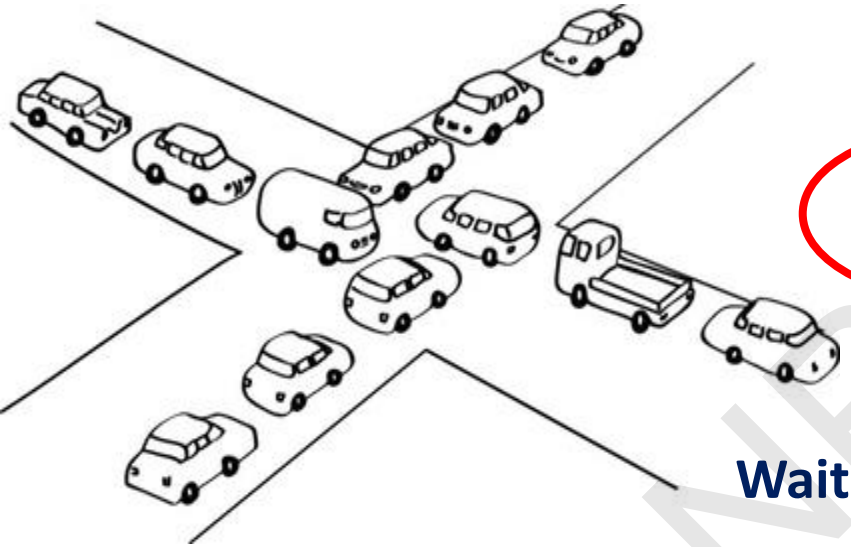
- When two processes are competing with each other causing **data corruption.**

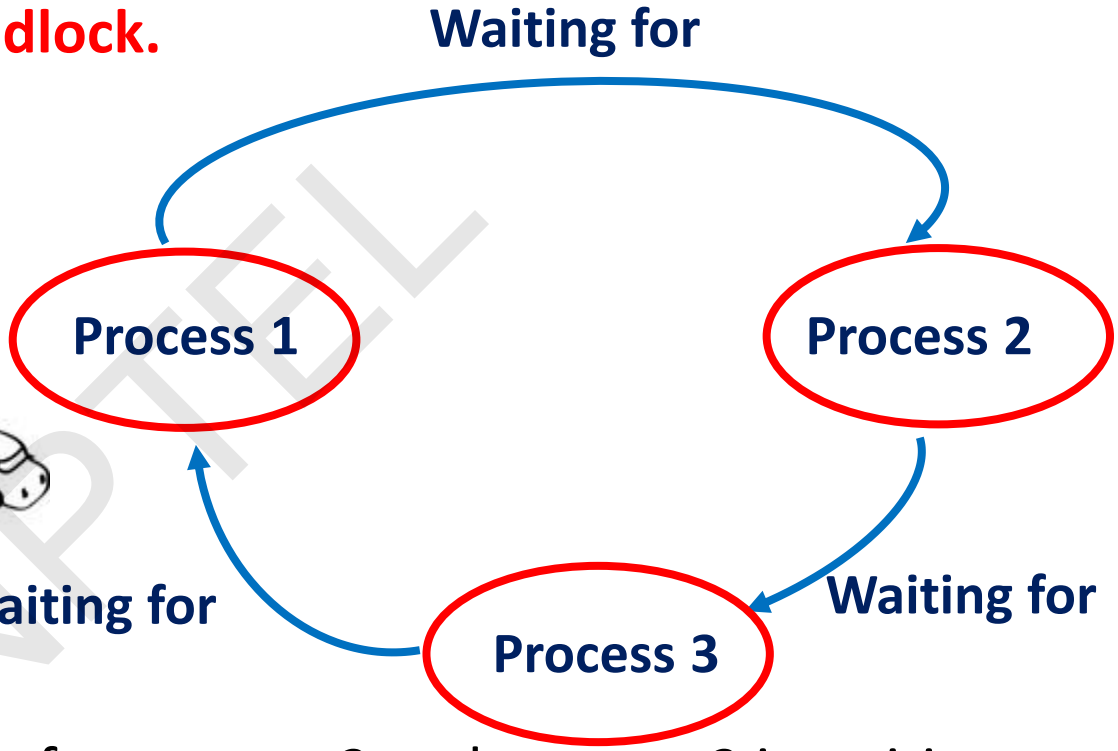


As shown in the diagram, two persons are trying to deposit 1 rs. online into the same bank account. The initial amount is 17 rs. Due to race conditions, the final amount in the bank is 18 rs. instead of 19.

What is a Deadlock?

- When two processes are waiting for each other directly or indirectly, it is called **deadlock.**

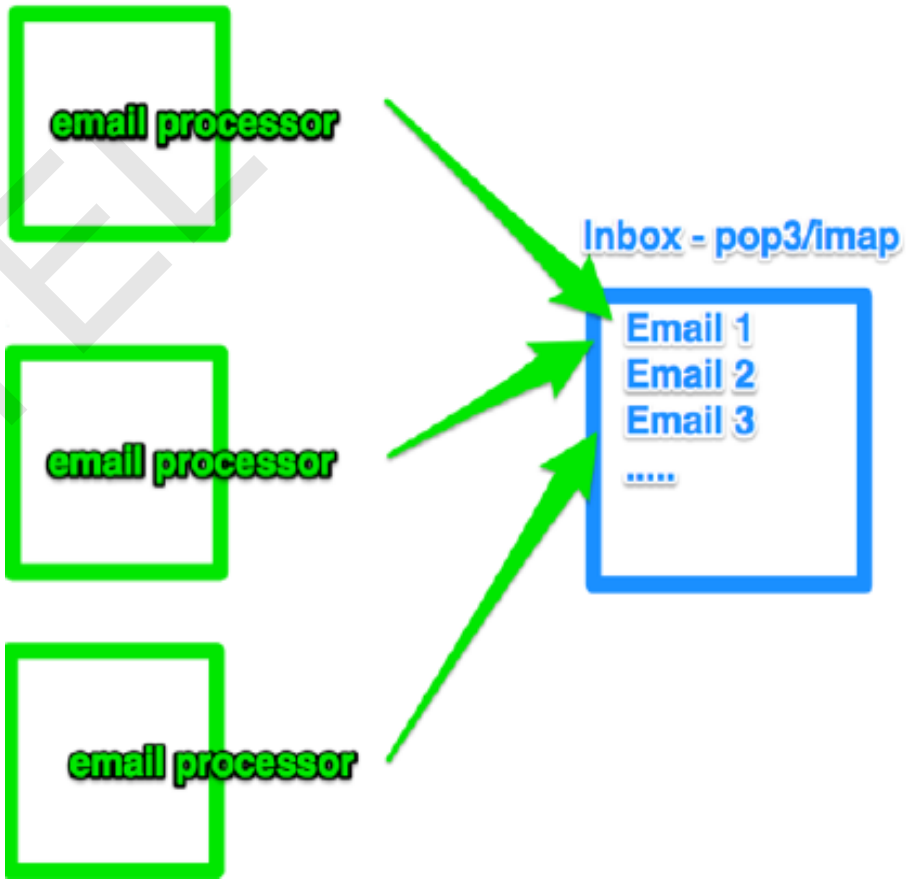




- Here, Process 1 is waiting for process 2 and process 2 is waiting for process 3 to finish and process 3 is waiting for process 1 to finish. All these three processes would keep waiting and will never end. This is called dead lock.

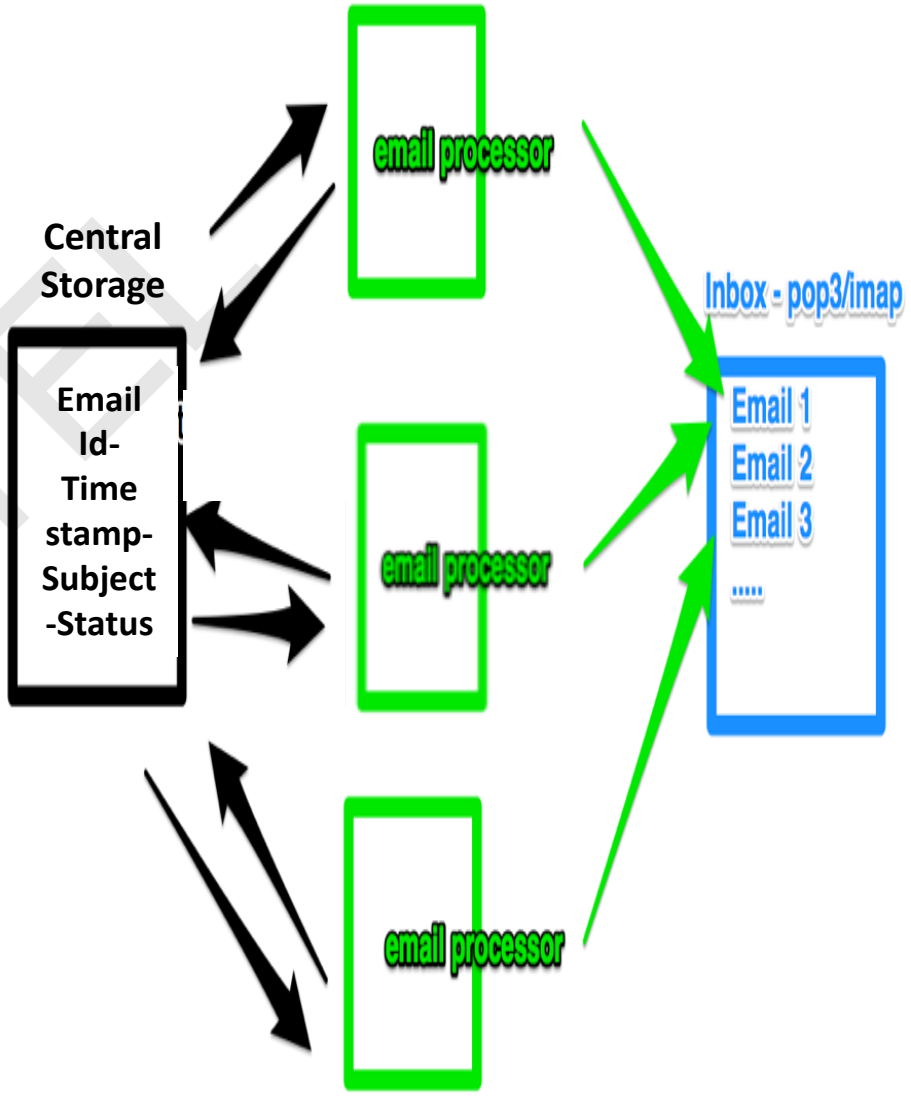
What is Coordination ?

- **How would Email Processors avoid reading same emails?**
- Suppose, there is an inbox from which we need to index emails.
- Indexing is a heavy process and might take a lot of time.
- Here, we have multiple machine which are indexing the emails. Every email has an id. You can not delete any email. You can only read an email and mark it read or unread.
- Now how would you handle the coordination between multiple indexer processes so that every email is indexed?



What is Coordination ?

- If indexers were running as multiple threads of a single process, it was easier by the way of using synchronization constructs of programming language.
- But since there are multiple processes running on multiple machines which need to coordinate, we need a central storage.
- This central storage should be safe from all concurrency related problems.
- **This central storage is exactly the role of Zookeeper.**



What is Coordination ?

- **Group membership:** Set of datanodes (tasks) belong to same group
- **Leader election:** Electing a leader between primary and backup
- **Dynamic Configuration:** Multiple services are joining, communicating and leaving (Service lookup registry)
- **Status monitoring:** Monitoring various processes and services in a cluster
- **Queuing:** One process is embedding and other is using
- **Barriers:** All the processes showing the barrier and leaving the barrier.
- **Critical sections:** Which process will go to the critical section and when?

What is ZooKeeper ?

- **ZooKeeper is a highly reliable distributed coordination kernel**, which can be used for distributed locking, configuration management, leadership election, work queues,….
- Zookeeper is a replicated service that holds the metadata of distributed applications.
- **Key attributed of such data**
  - **Small size**
  - **Performance sensitive**
  - **Dynamic**
  - **Critical**
- **In very simple words,** it is a central store of key-value using which distributed systems can coordinate. Since it needs to be able to handle the load, Zookeeper itself runs on many machines.

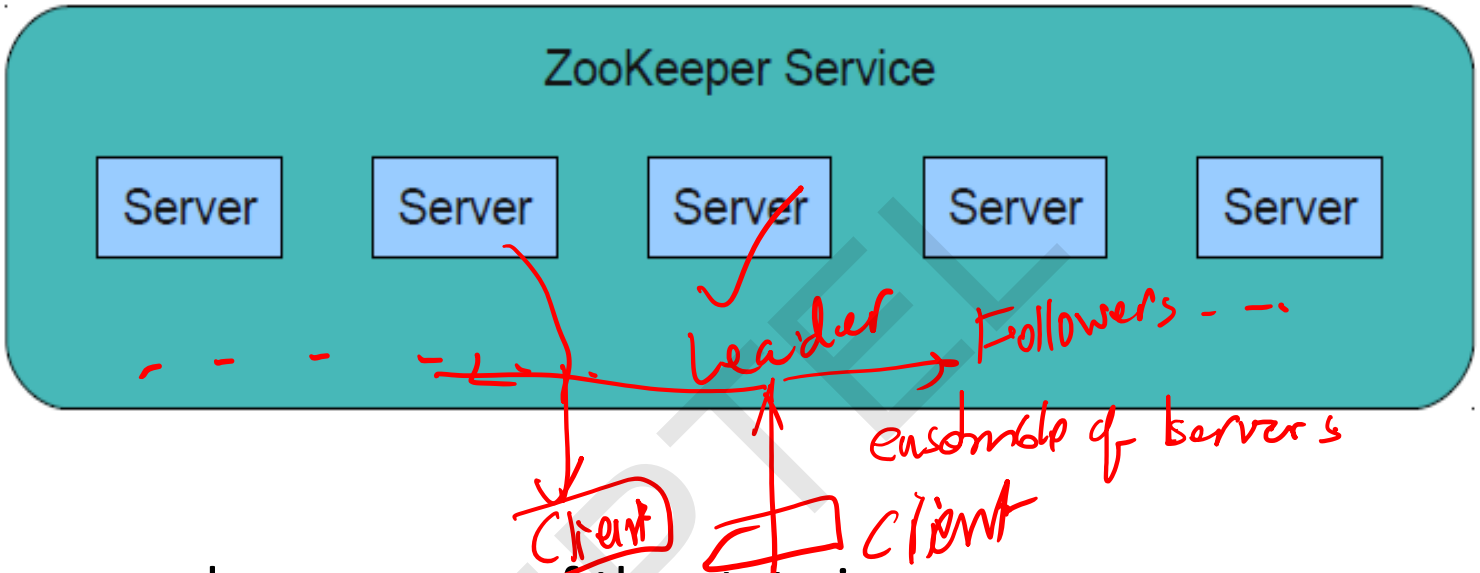
What is ZooKeeper ?

- Exposes a simple set of primitives
- Very easy to program
- Uses a data model like directory tree
- Used for
  - Synchronisation
  - Locking
  - Maintaining Configuration
- Coordination service that does not suffer from
  - Race Conditions
  - Dead Locks

Design Goals: 1. Simple

- A shared hierarchal namespace looks like standard file system
- The namespace has data nodes - znodes (similar to files/dirs)
- Data is kept in-memory
- Achieve high throughput and low latency numbers.
- **High performance**
  - Used in large, distributed systems
- **Highly available**
  - No single point of failure
- **Strictly ordered access**
  - Synchronisation

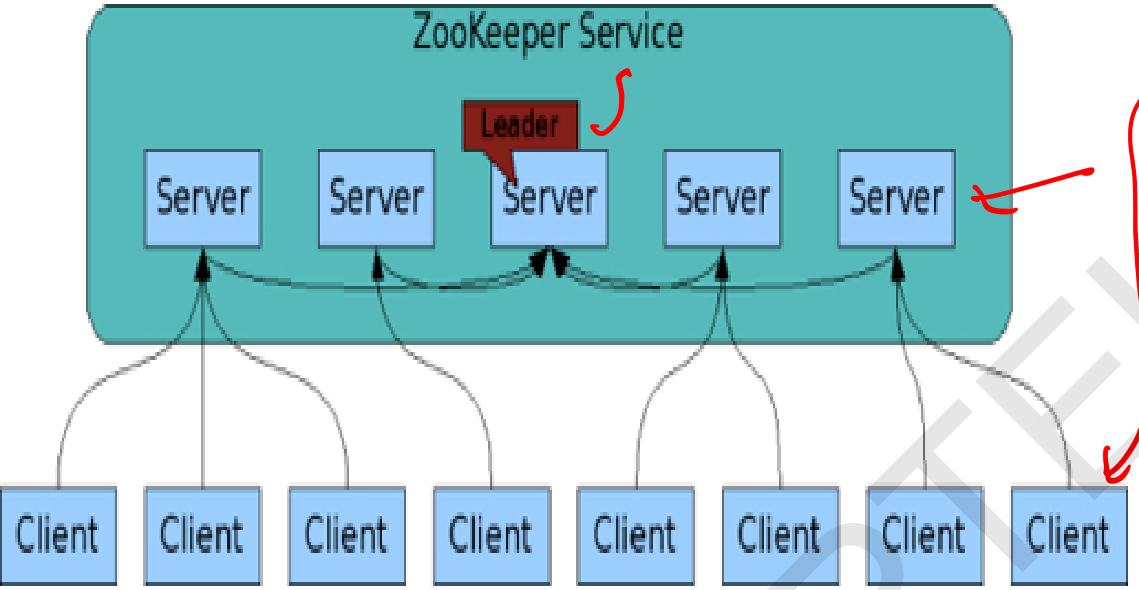
Design Goals: 2. Replicated



- All servers have a copy of the state in memory
- A leader is elected at startup
- Followers service clients, all updates go through leader
- Update responses are sent when a majority of servers have persisted the change

We need $2f+1$ machines to tolerate f failures

Design Goals: 2. Replicated



The client

- Keeps a TCP connection
- Gets watch events
- Sends heart beats.
- If connection breaks,
  - Connect to different server.

The servers

- Know each other
- Keep in-memory image of State
- Transaction Logs & Snapshots - persistent

Design Goals: 3. Ordered

- ZooKeeper stamps each update with a number

- **The number:**
  - Reflects the order of transactions.
  - used implement higher-level abstractions, such as synchronization primitives.

Design Goals: 4. Fast

Performs best where reads are more common than writes, at ratios of around 10:1.

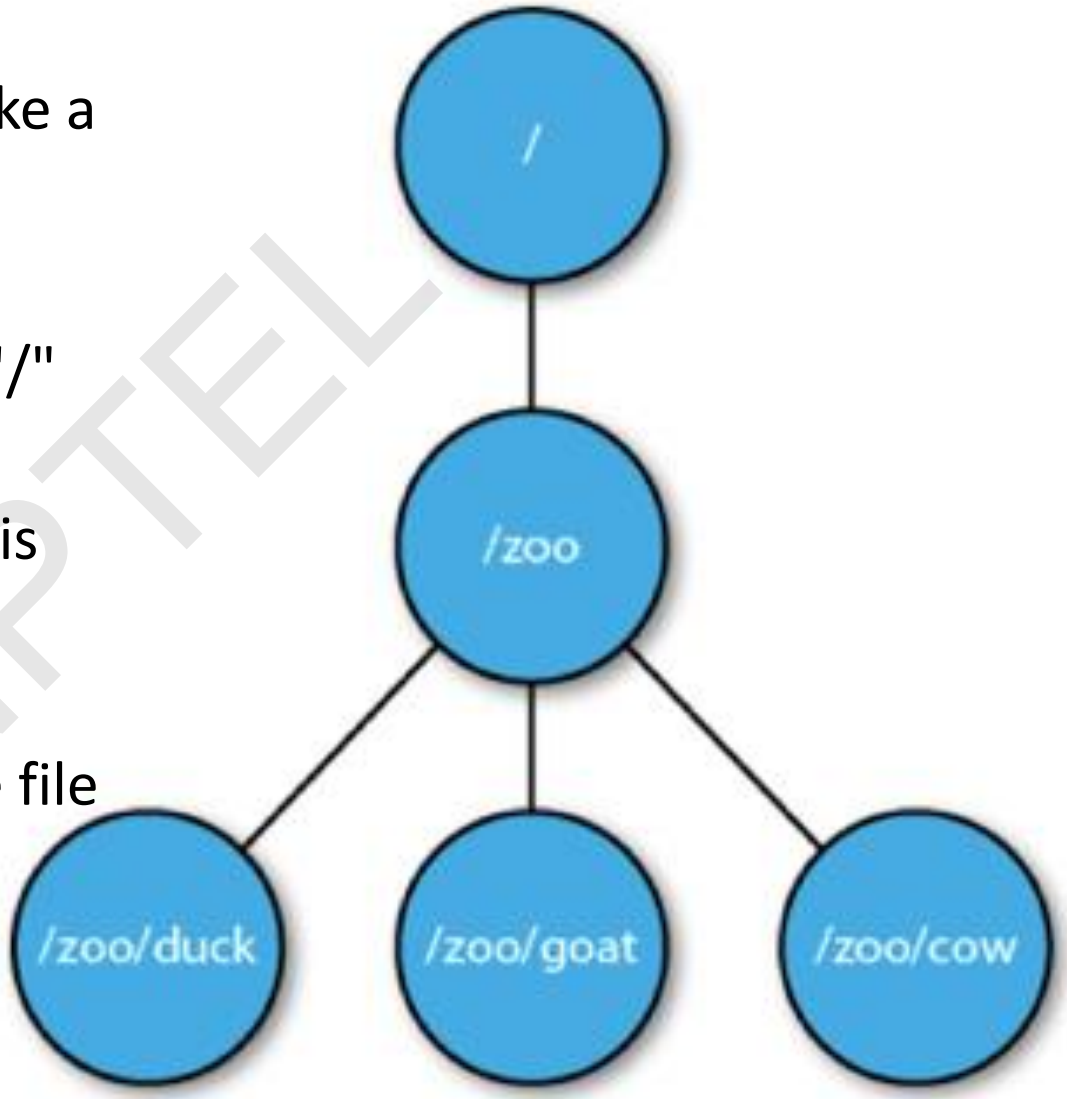
At Yahoo!, where it was created, the throughput for a ZooKeeper cluster has been benchmarked at over 10,000 operations per second for write-dominant workloads generated by hundreds of clients

Data Model

- The way you store data in any store is called **data model.**
- **Think of it as highly available fileSystem:** In case of zookeeper, think of data model as if it is a highly available file system with little differences.
- **Znode:** We store data in an entity called znode.
- **JSON data:** The data that we store should be in JSON format which Java script object notation.
- **No Append Operation:** The znode can only be updated. It does not support append operations.
- **Data access (read/write) is atomic:** The read or write is atomic operation meaning either it will be full or would throw an error if failed. There is no intermediate state like half written.
- **Znode:** Can have children

Data Model Contd…

- So, znodes inside znodes make a tree like heirarchy.
- The top level znode is "/".
- The znode "/zoo" is child of "/" which top level znode.
- duck is child znode of zoo. It is denoted as /zoo/duck
- Though "." or ".." are invalid characters as opposed to the file system.



Data Model – Znode - Types

- **Persistent**
  - Such kind of znodes remain in zookeeper untill deleted. This is the default type of znode. To create such node you can use the command: create /name_of_myznode "mydata"

- **Ephemeral**
  - Ephermal node gets deleted if the session in which the node was created has disconnected. Though it is tied to client's session but it is visible to the other users.

  - An ephermal node can not have children not even ephermal children.

Data Model – Znode - Types

- **Sequential**
  - Creates a node with a sequence number in the name
  - The number is automatically appended.

```
create -s /zoo v
Created /zoo0000000008
```

```
create -s /zoo/ v
Created /zoo/0000000003
```

```
create -s /xyz v
Created /xyz0000000009
```

```
create -s /zoo/ v
Created /zoo/0000000004
```

- The counter keeps increasing monotonically
- Each node keeps a counter

Architecture

- **Zookeeper can run in two modes: (i) Standalone and (ii) Replicated.**

(i) Standalone:

- In standalone mode, it is just running on one machine and for practical purposes we do not use stanalone mode.
- This is only for testing purposes.
- It doesn't have high availability.

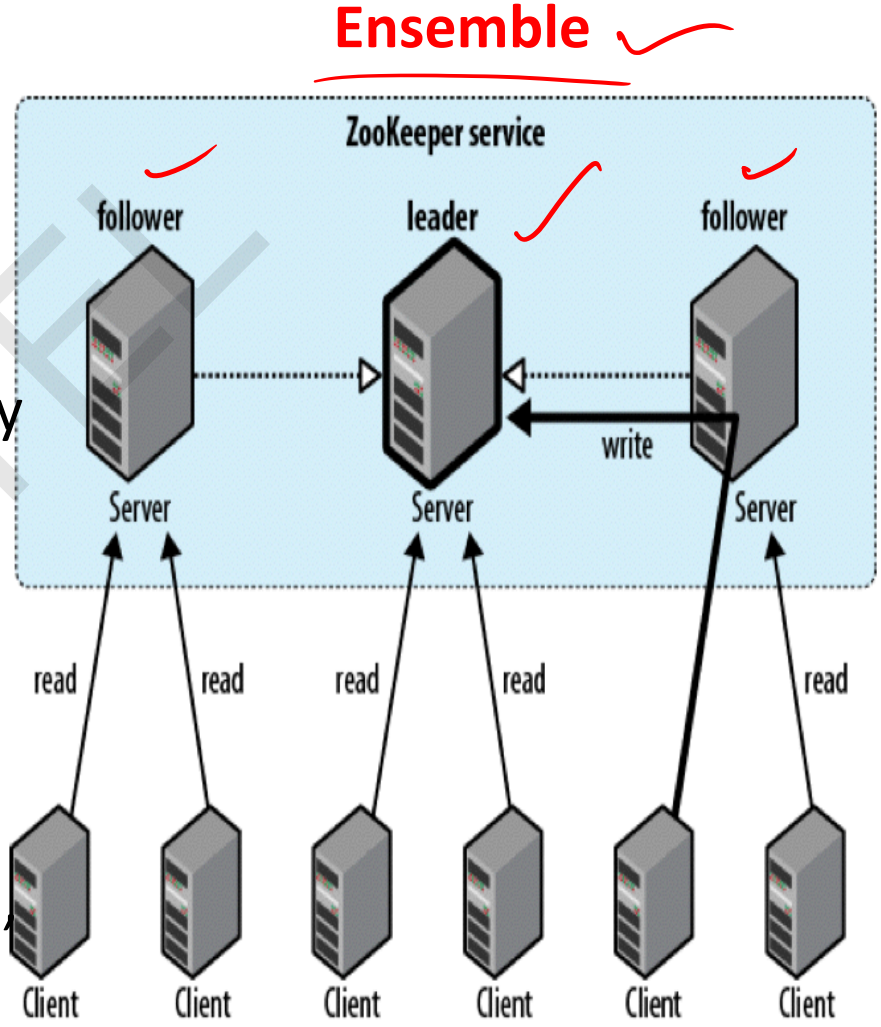
(ii) Replicated:

- Run on a cluster of machines called an ensemble.
- High availability
- Tolerates as long as majority.

Architecture: Phase 1

Phase 1: Leader election (Paxos Algorithm)

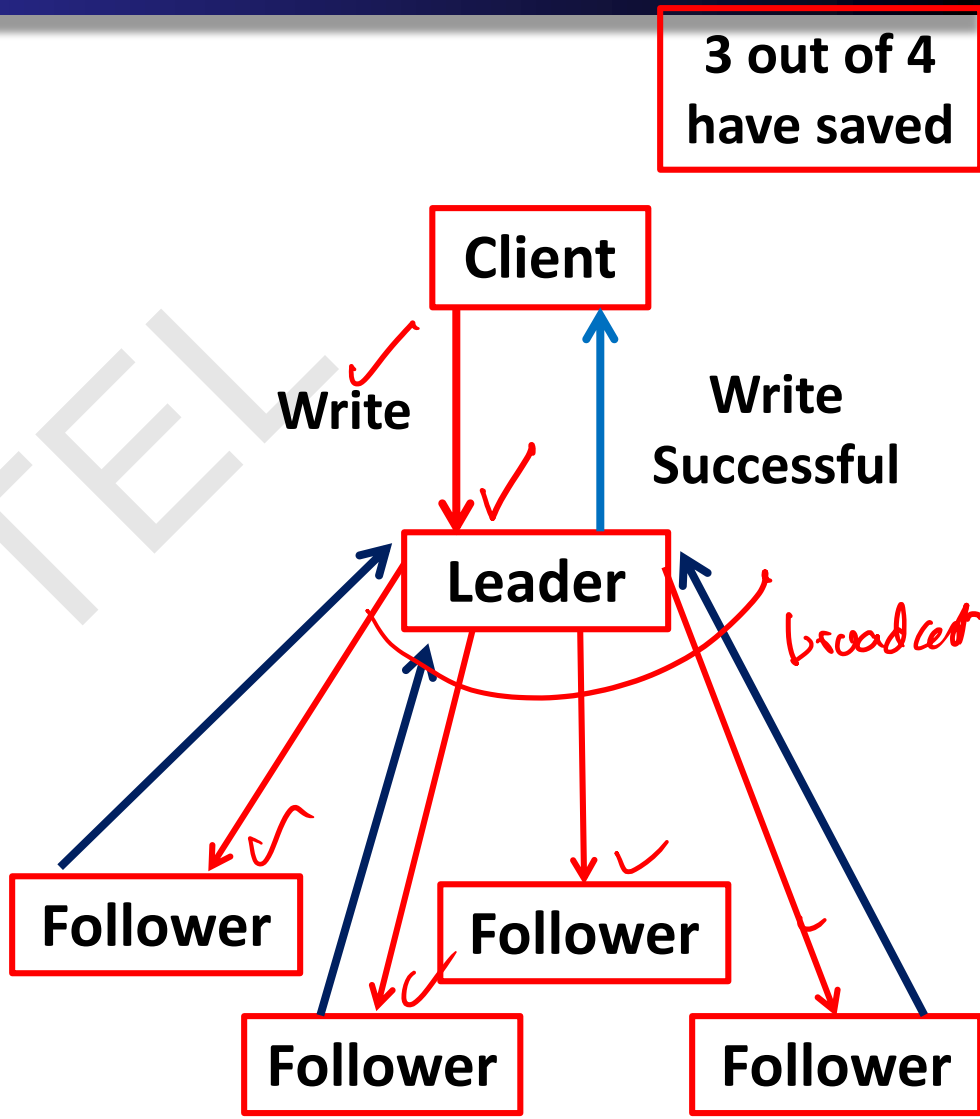
- The machines elect a distinguished member - leader.
- The others are termed followers.
- This phase is finished when majority sync their state with leader.
- If leader fails, the remaining machines hold election. takes 200ms.
- If the majority of the machines aren't available at any point of time, the leader automatically steps down.



Architecture: Phase 2

Phase 2: Atomic broadcast

- All write requests are forwarded to the leader,
- Leader broadcasts the update to the followers
- When a majority have persisted the change:
  - The leader commits the up-date
  - The client gets success response.
- The protocol for achieving consensus is atomic like two-phase commit.
- Machines write to disk before in-memory



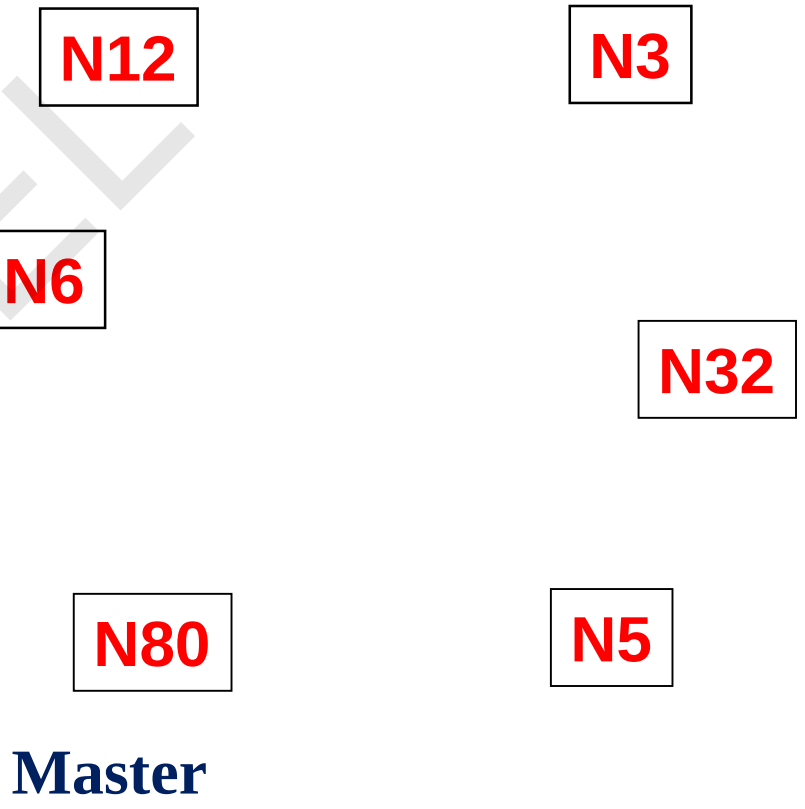
Election in Zookeeper

- Centralized service for maintaining configuration information
- **Uses a variant of Paxos called Zab (Zookeeper Atomic Broadcast)**
- Needs to keep a leader elected at all times

Reference: **http://zookeeper.apache.org/**

Election in Zookeeper (2)

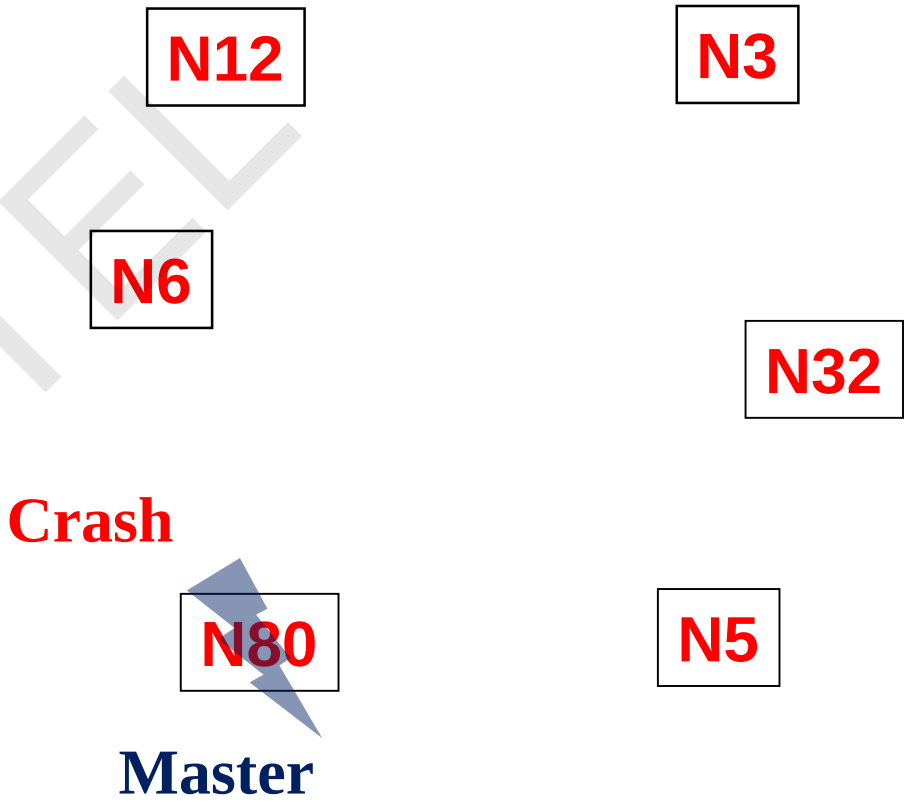
- Each server creates a new **sequence number** for itself
  - Let's say the sequence numbers are **ids**
  - Gets highest id so far (from ZK(zookeeper) file system), creates next-higher id, writes it into ZK file system
- Elect the highest-id server as leader.



Election in Zookeeper (3)

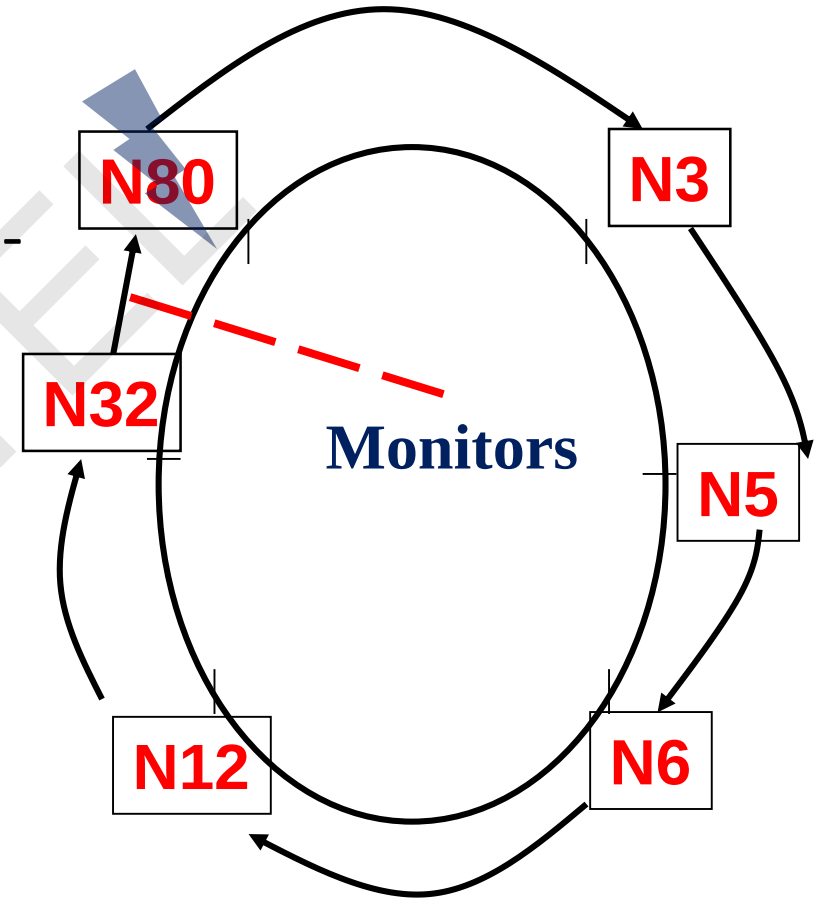
- **Failures:**

- One option: everyone monitors current master (directly or via a failure detector)
  - On failure, initiate election
  - Leads to a flood of elections
  - Too many messages



Election in Zookeeper (4)

- **Second option:** (implemented in Zookeeper)
  - Each process monitors its next-higher id process
  - **if** that successor was the leader and it has failed
    - Become the new leader
  - **else**
    - wait for a timeout, and check your successor again.



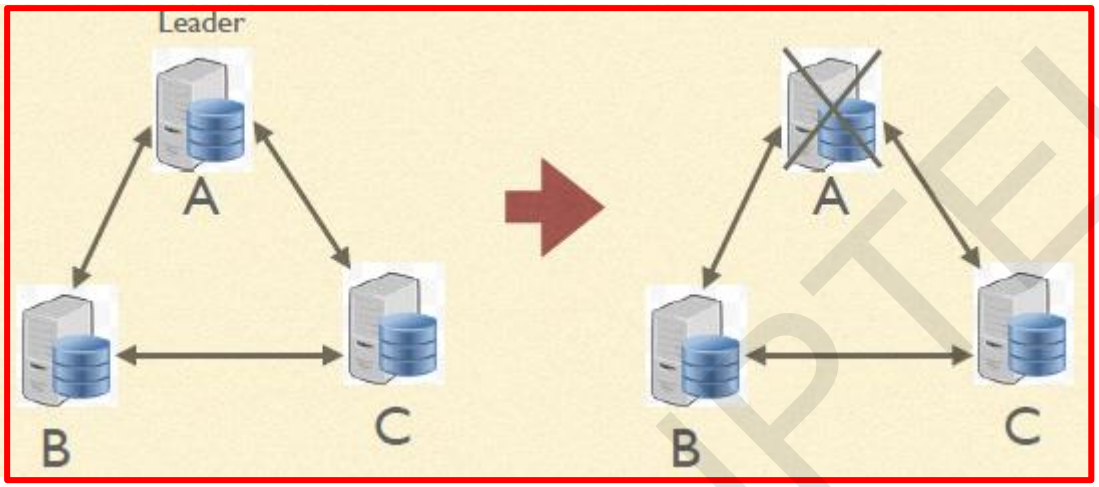
Election in Zookeeper (5)

- What about id conflicts? What if leader fails during election ?

- To address this, Zookeeper uses a ***two-phase commit*** (run after the sequence/id) protocol to commit the leader
  - Leader sends NEW_LEADER message to all
  - Each process responds with ACK to at most one leader, i.e., one with highest process id
  - Leader waits for a majority of ACKs, and then sends COMMIT to all
  - On receiving COMMIT, process updates its leader variable
- Ensures that safety is still maintained

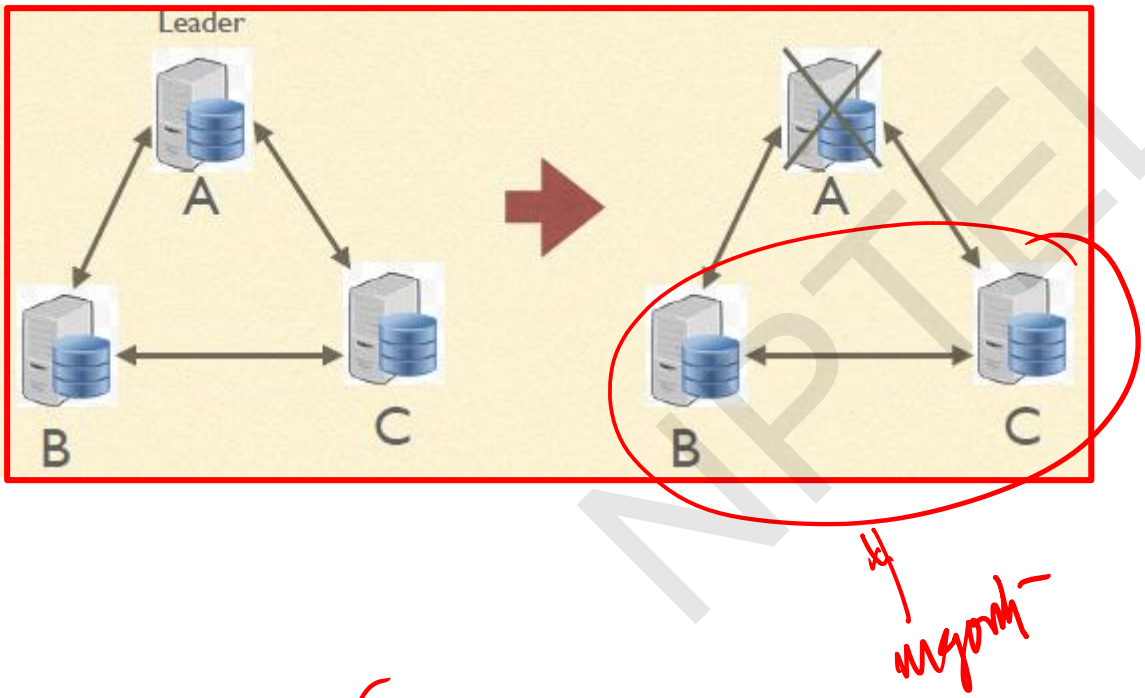
Election Demo

- If you have three nodes A, B, C with A as Leader. And A dies. Will someone become leader?

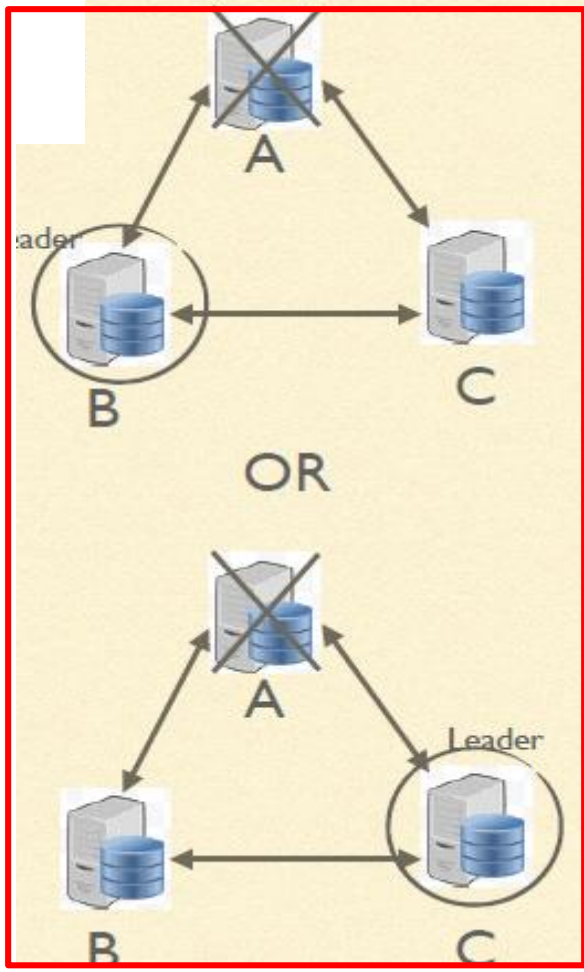


Election Demo

- If you have three nodes A, B, C with A as Leader. And A dies. Will someone become leader?

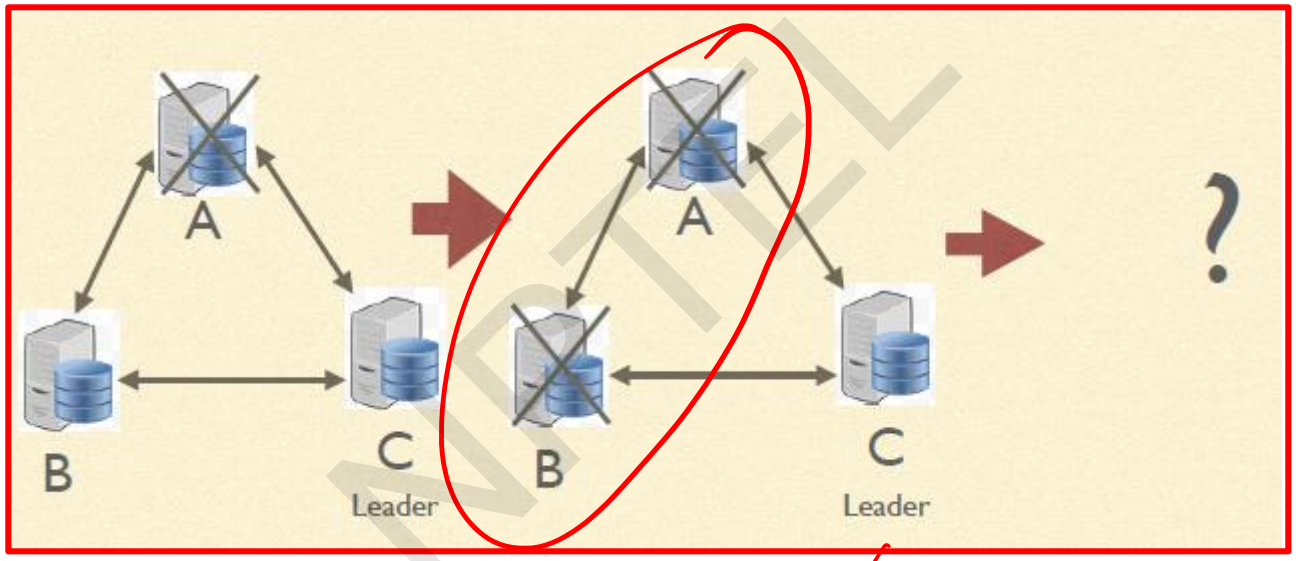


Yes. Either B or C.



Election Demo

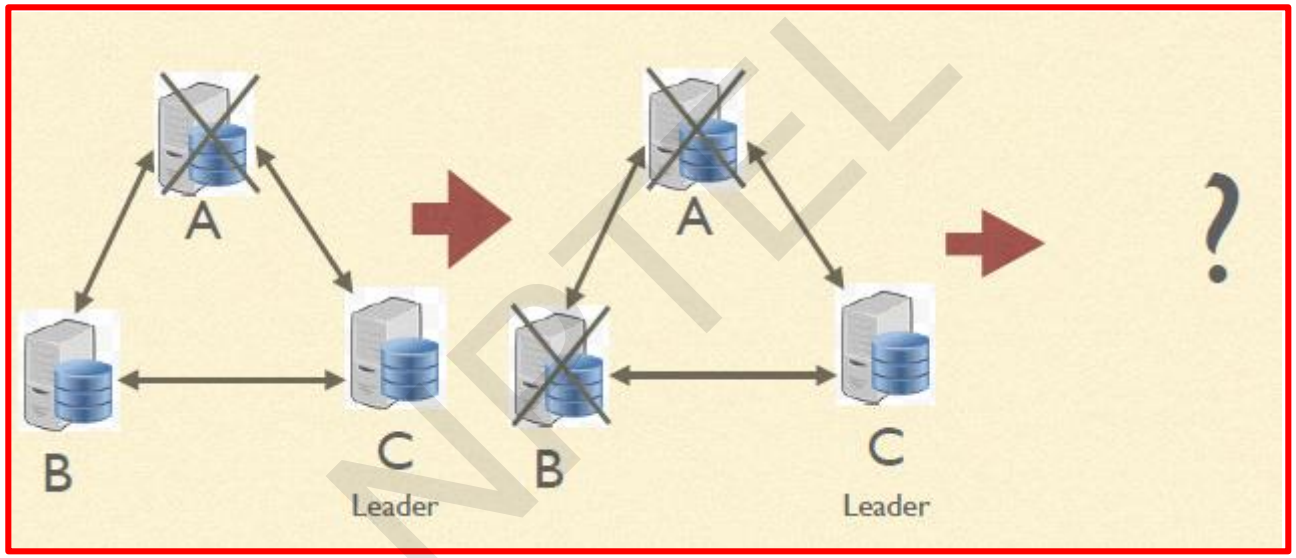
- If you have three nodes A, B, C And A and B die. Will C become Leader?



1/3 not majority

Election Demo

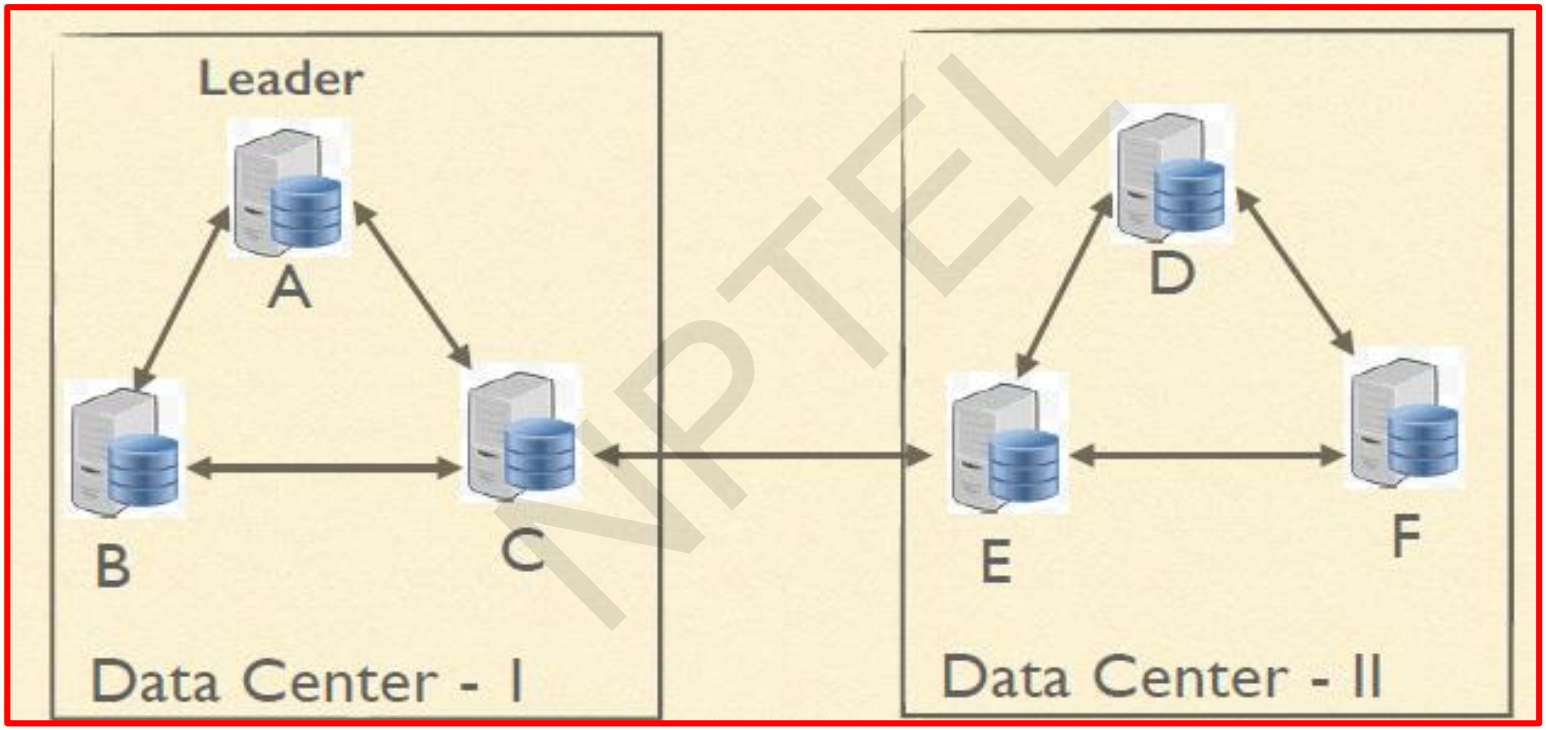
- If you have three nodes A, B, C And A and B die. Will C become Leader?



No one will become Leader.
C will become Follower.
Reason: Majority is not available.

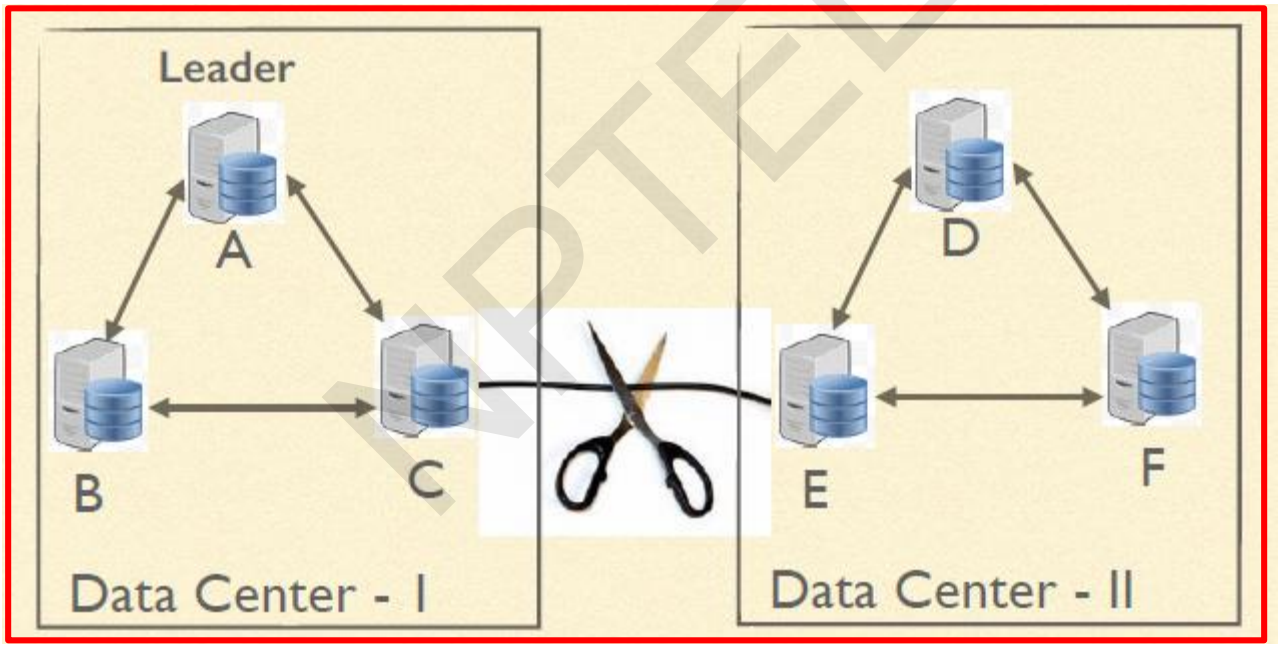
Why do we need majority?

- **Imagine:** We have an ensemble spread over two data centres.



Why do we need majority?

- **Imagine:** The network between data centres got disconnected. If we did not need majority for electing Leader,
- **What will happen?**

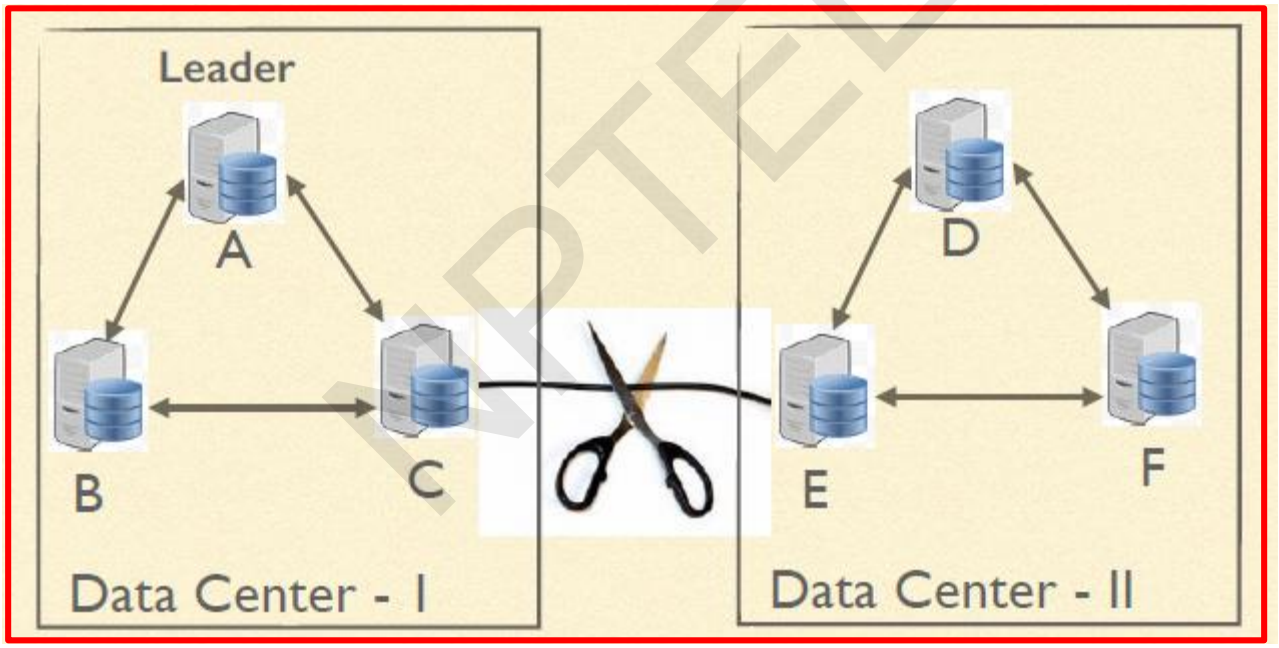


Why do we need majority?

Each data centre will have their own Leader.

No Consistency and utter Chaos.

That is why it requires majority.



Sessions

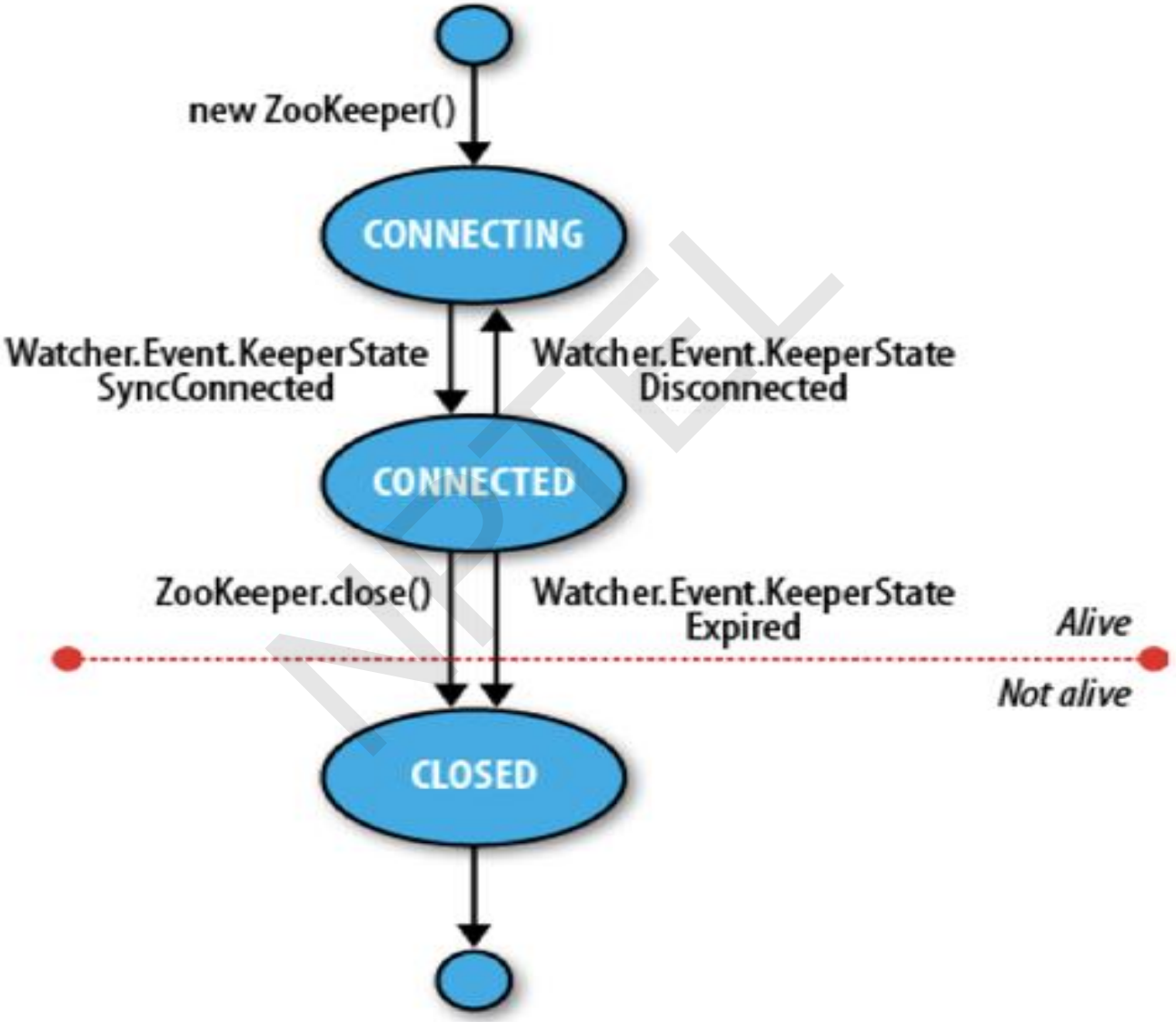
- Lets try to understand **how do the zookeeper decides to delete ephermals nodes** and takes care of session management.

- A client has list of servers in the ensemble
- It tries each until successful.
- Server creates a new session for the client.
- A session has a timeout period - decided by caller

Contd…

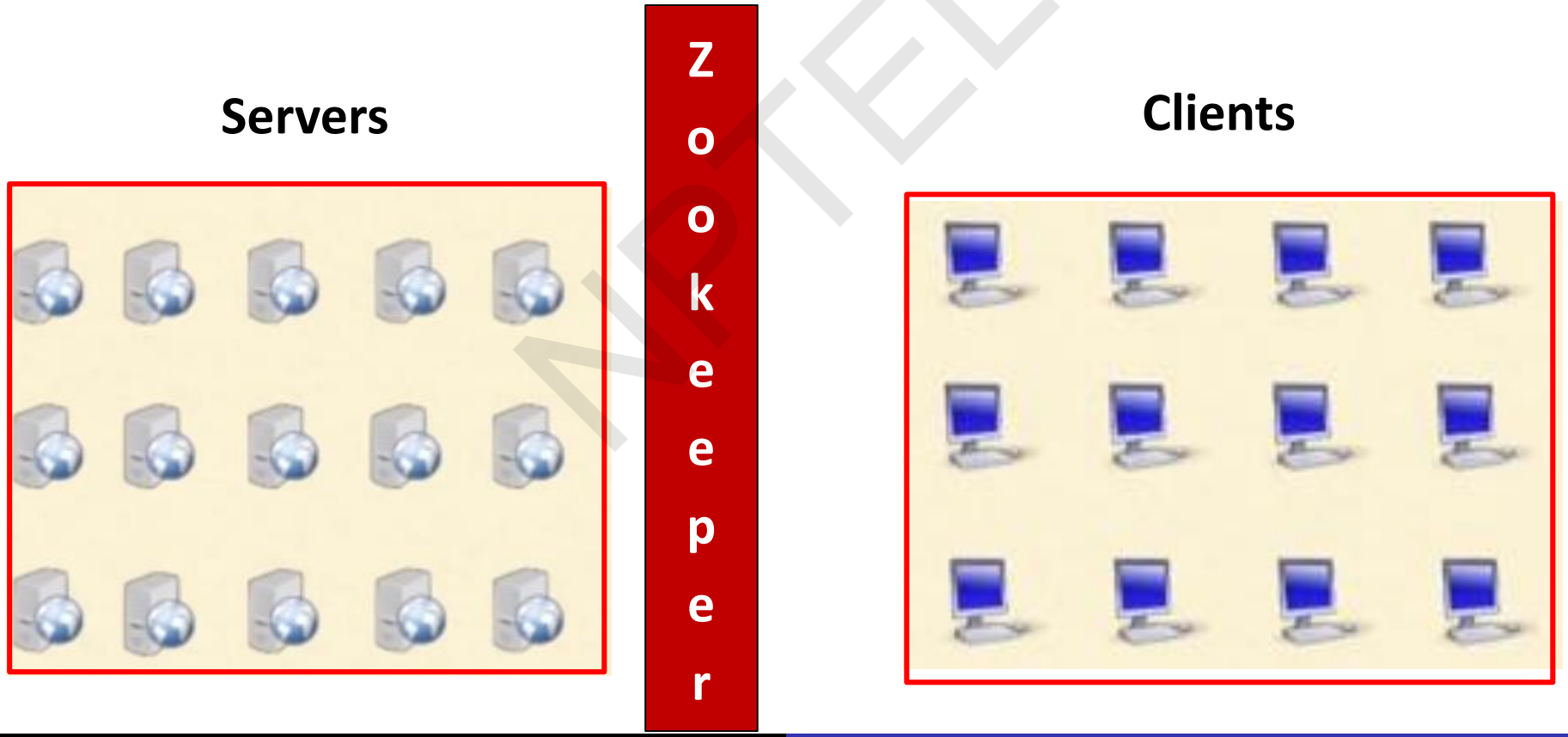
- If the server hasn’t received a request within the timeout period, it may expire the session.
- On session expire, ephermal nodes are lost
- To keep sessions alive client sends pings (heartbeats)
- Client library takes care of heartbeats
- Sessions are still valid on switching to another server
- Failover is handled automatically by the client
- Application can't remain agnostic of server reconnections - because the operations will fail during disconnection.

States



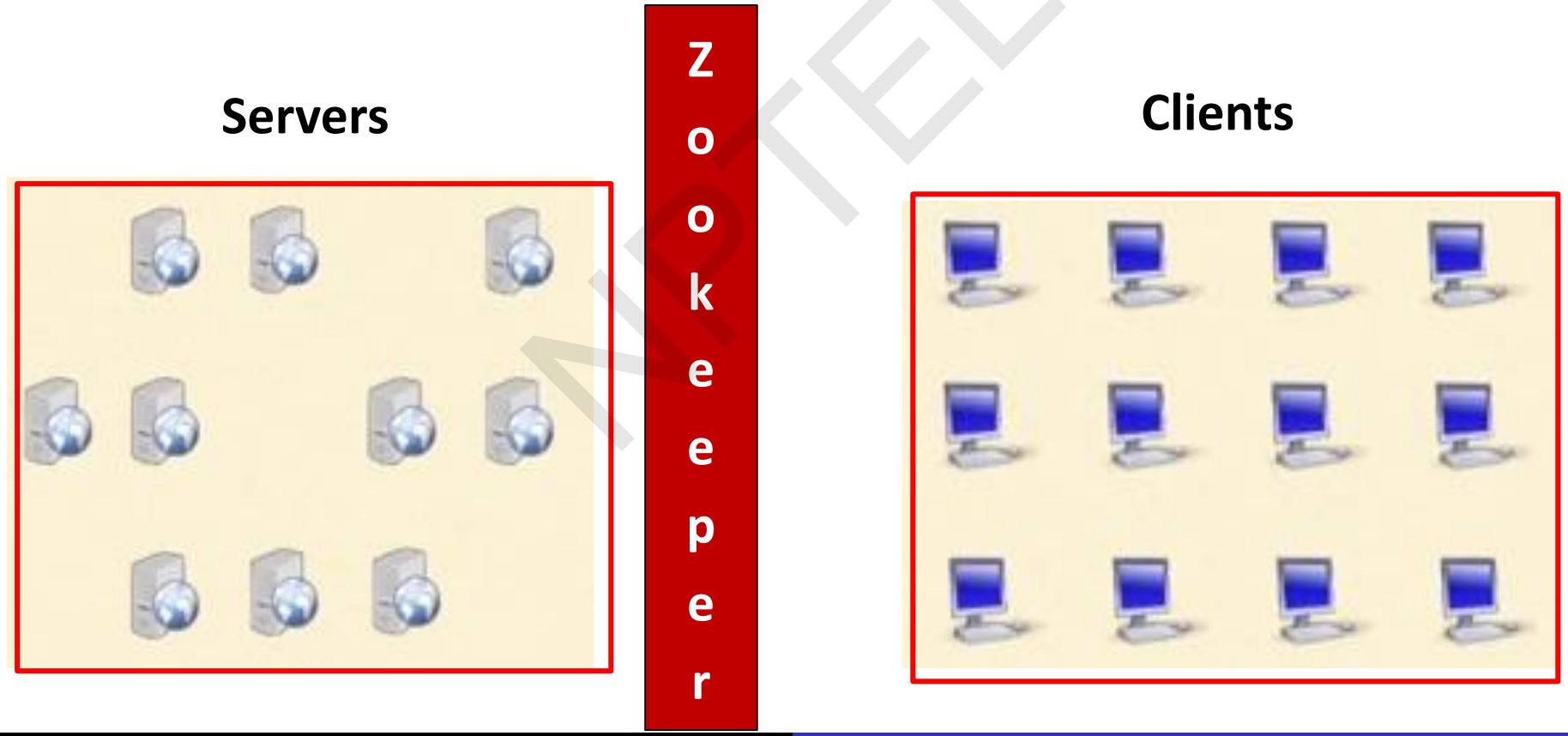
Use Case: Many Servers How do they Coordinate?

- Let us say there are many servers which can respond to your request and there are many clients which might want the service.



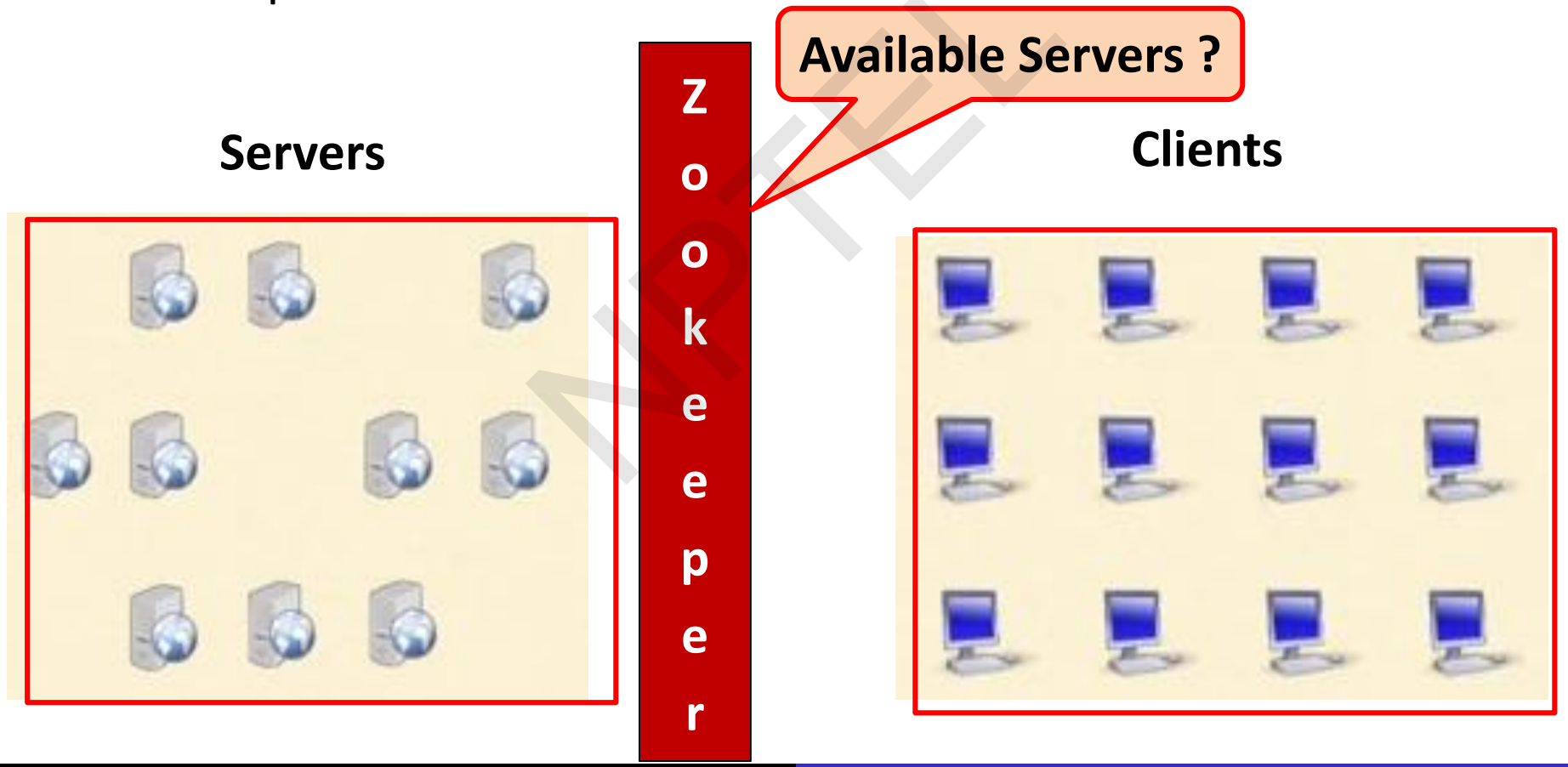
Use Case: Many Servers How do they Coordinate?

- From time to time some of the servers will keep going down. How can all of the clients can keep track of the available servers?



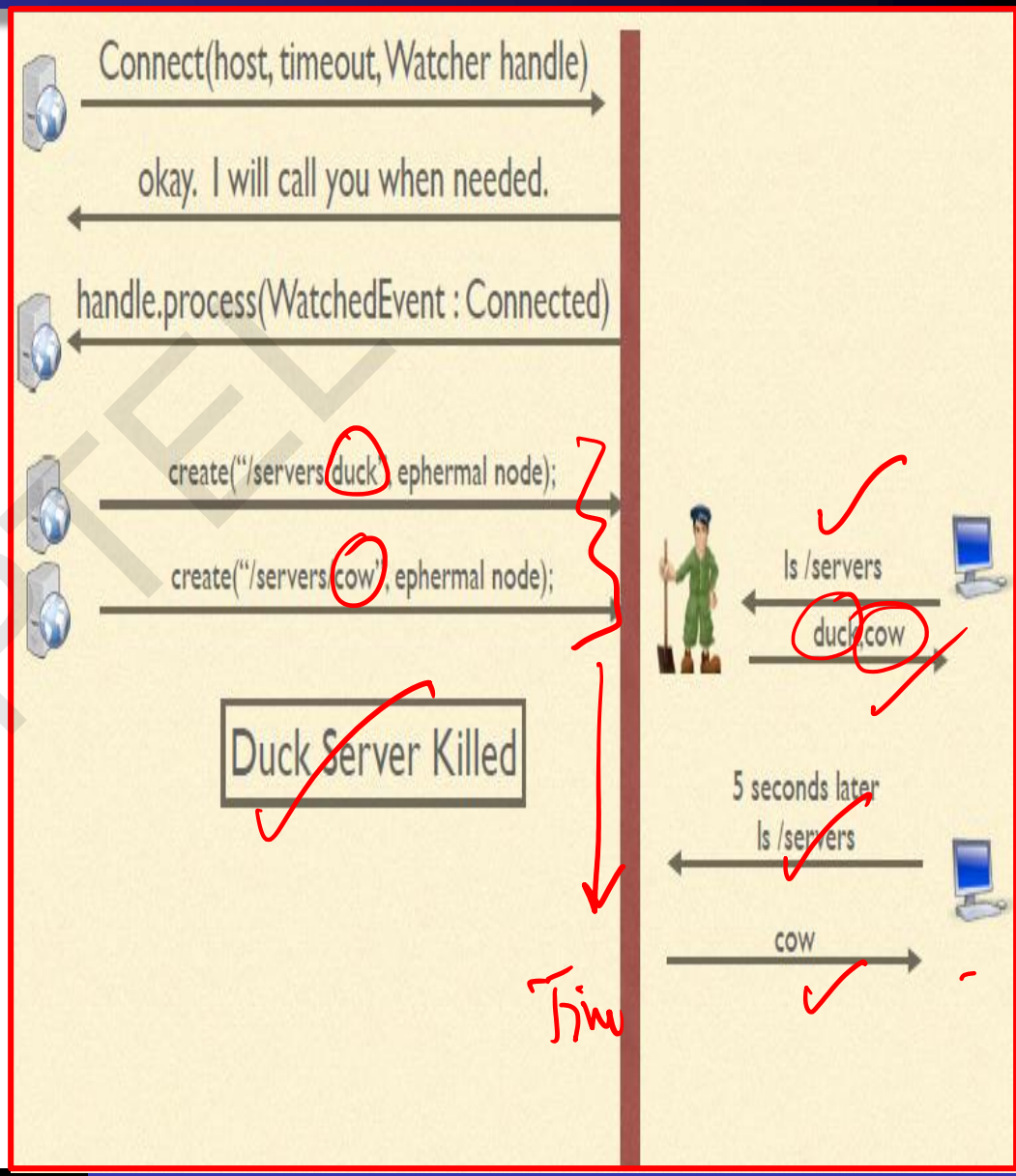
Use Case: Many Servers How do they Coordinate?

- It is very easy using zookeeper as a centeral agency. Each server will create their own ephermal znode under a particular znode say "/servers". The clients would simply query zookeeper for the most recent list of servers.



Use Case: Many Servers How do they Coordinate?

- Lets take a case of two servers and a client. The two server duck and cow created their ephermal nodes under "/servers" znode. The client would simply discover the alive servers cow and duck using command ls /servers.
- Say, a server called "duck" is down, the ephermal node will disappear from /servers znode and hence next time the client comes and queries it would only get "cow". So, the coordinations has been made heavily simplified and made efficient because of ZooKeeper.



Guarantees

- **Sequential consistency**
  - Updates from any particular client are applied in the order
- **Atomicity**
  - Updates either succeed or fail.
- **Single system image**
  - A client will see the same view of the system, The new server will not accept the connection until it has caught up.
- **Durability**
  - Once an update has succeeded, it will persist and will not be undone.
- **Timeliness**
  - Rather than allow a client to see very stale data, a server will shut down,

Operations

OPERATION	DESCRIPTION
create	Creates a znode (parent znode must exist)
delete	Deletes a znode (mustn't have children)
exists/ls	Tests whether a znode exists & gets metadata
getACL, setACL	Gets/sets the ACL for a znode getChildren/ls Gets a list of the children of a znode
getData/get, setData	Gets/sets the data associated with a znode
sync	Synchronizes a client's view of a znode with ZooKeeper

Multi Update

- Batches together multiple operations together
- Either all fail or succeed in entirety
- Possible to implement transactions
- Others never observe any inconsistent state

APIs

- Two core: Java & C
- contrib: perl, python, REST
- For each binding, sync and async available

Synch:

Public Stat exists (String path, Watcher watcher) throws KeeperException, InterruptedException

Asynch:

Public void exists (String path, Watcher watcher, StatCallback cb, Object ctx

Watches

- Clients to get notifications when a znode changes in some way
- Watchers are triggered only once
- For multiple notifications, re-register

Watch Triggers

- The read operations exists, getChildren, getData may have watches
- Watches are triggered by write ops: create, delete, setData
- **ACL (Access Control List)** operations do not participate in watches

WATCH OF …ARE TRIGGERED	WHEN ZNODE IS…
exists	created, deleted, or its data updated.
getData	deleted or has its data updated.
getChildren	deleted, or its any of the child is created or deleted

ACLs - Access Control Lists

ACL Determines who can perform certain operations on it.

- **ACL is the combination**
  - authentication scheme,
  - an identity for that scheme,
  - and a set of permissions

- **Authentication Scheme**
  - **digest -** The client is authenticated by a username & password.
  - **sasl -** The client is authenticated using Kerberos.
  - **ip -** The client is authenticated by its IP address.

Use Cases

Building a reliable configuration service

- A Distributed lock service

  Only single process may hold the lock

When Not to Use?

1. To store big data because:
   - The number of copies == number of nodes
   - All data is loaded in RAM too
   - Network load of transferring all data to all Nodes

2. Extremely strong consistency

ZooKeeper Applications: The Fetching Service

- **The Fetching Service:** Crawling is an important part of a search engine, and Yahoo! crawls billions of Web documents. The Fetching Service (FS) is part of the Yahoo! crawler and it is currently in production. Essentially, it has master processes that command page-fetching processes.

- The master provides the fetchers with configuration, and the fetchers write back informing of their status and health. The main advantages of using ZooKeeper for FS are recovering from failures of masters, guaranteeing availability despite failures, and decoupling the clients from the servers, allowing them to direct their request to healthy servers by just reading their status from ZooKeeper.

- Thus, FS uses ZooKeeper mainly to manage configuration metadata, although it also uses Zoo- Keeper to elect masters (leader election).

ZooKeeper Applications: Katta

- **Katta:** It is a distributed indexer that uses Zoo- Keeper for coordination, and it is an example of a non- Yahoo! application. Katta divides the work of indexing using shards.
- A master server assigns shards to slaves and tracks progress. Slaves can fail, so the master must redistribute load as slaves come and go.
- The master can also fail, so other servers must be ready to take over in case of failure. Katta uses ZooKeeper to track the status of slave servers and the master (**group membership**), and to handle master failover (**leader election**).
- Katta also uses ZooKeeper to track and propagate the assignments of shards to slaves (**configuration management**).

ZooKeeper Applications: Yahoo! Message Broker

- **Yahoo! Message Broker: (YMB)** is a distributed publish-subscribe system. The system manages thousands of topics that clients can publish messages to and receive messages from. The topics are distributed among a set of servers to provide scalability.
- Each topic is replicated using a primary-backup scheme that ensures messages are replicated to two machines to ensure reliable message delivery. The servers that makeup YMB use a shared-nothing distributed architecture which makes coordination essential for correct operation.
- YMB uses ZooKeeper to manage the distribution of topics (configuration metadata), deal with failures of machines in the system (failure detection and group membership), and control system operation.

ZooKeeper Applications: Yahoo! Message Broker

- Figure, shows part of the znode data layout for YMB.
- Each broker domain has a znode called nodes that has an ephemeral znode for each of the active servers that compose the YMB service.
- Each YMB server creates an ephemeral znode under nodes with load and status information providing both group membership and status information through ZooKeeper. of YMB.

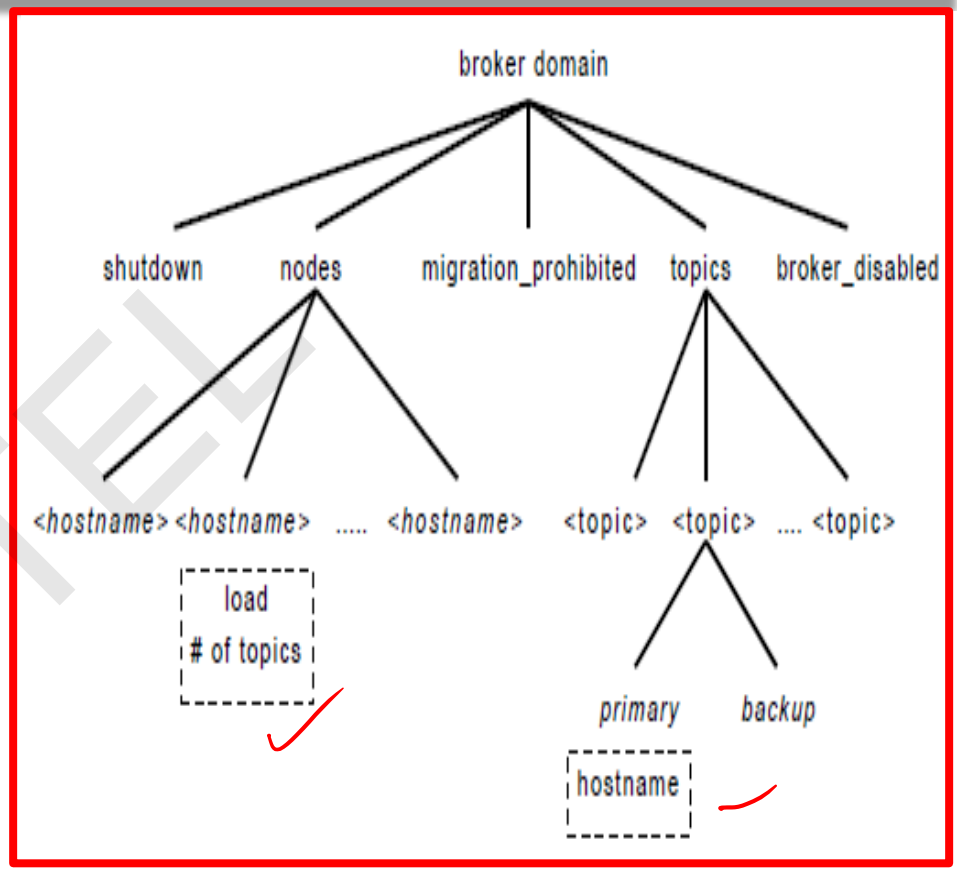


Figure: The layout of Yahoo! Message Broker (YMB) structures in ZooKeeper

ZooKeeper Applications: Yahoo! Message Broker

- The topics directory has a child znode for each topic managed by YMB.
- These topic znodes have child znodes that indicate the primary and backup server for each topic along with the subscribers of that topic.
- The primary and backup server znodes not only allow servers to discover the servers in charge of a topic, but they also manage leader election and server crashes.

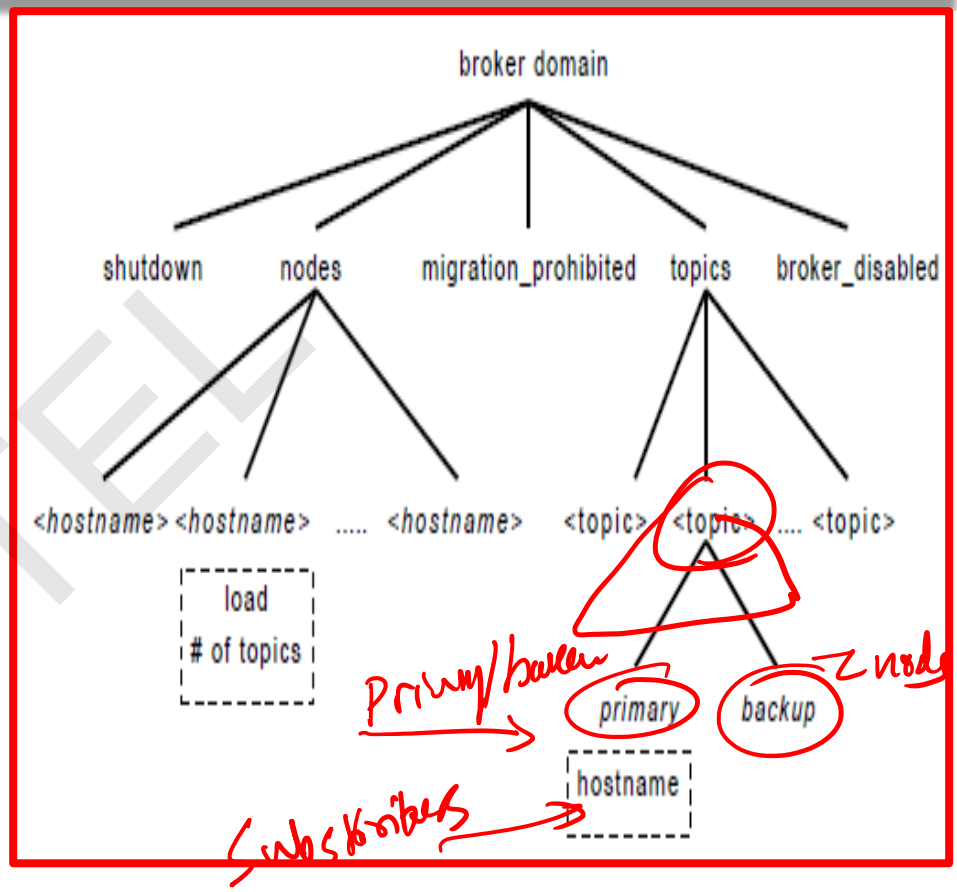


Figure: The layout of Yahoo! Message Broker (YMB) structures in ZooKeeper

More Details

ZooKeeper: Wait-free coordination for Internet-scale systems

Patrick Hunt and Mahadev Konar
Yahoo! Grid
{phunt,mahadev}@yahoo-inc.com

Flavio P. Junqueira and Benjamin Reed
Yahoo! Research
{fpj,breed}@yahoo-inc.com

See: https://zookeeper.apache.org/

Conclusion

- **ZooKeeper** takes a wait-free approach to the problem of coordinating processes in distributed systems, by exposing wait-free objects to clients.

- **ZooKeeper** achieves throughput values of hundreds of thousands of operations per second for read-dominant workloads by using fast reads with watches, both of which served by local replicas.

- In this lecture, we have discussed the basic fundamentals, design goals, architecture and applications of ZooKeeper.