Week 6
Spyros Mastorakis

Outline

- CAP Theorem
- Consistent Hashing

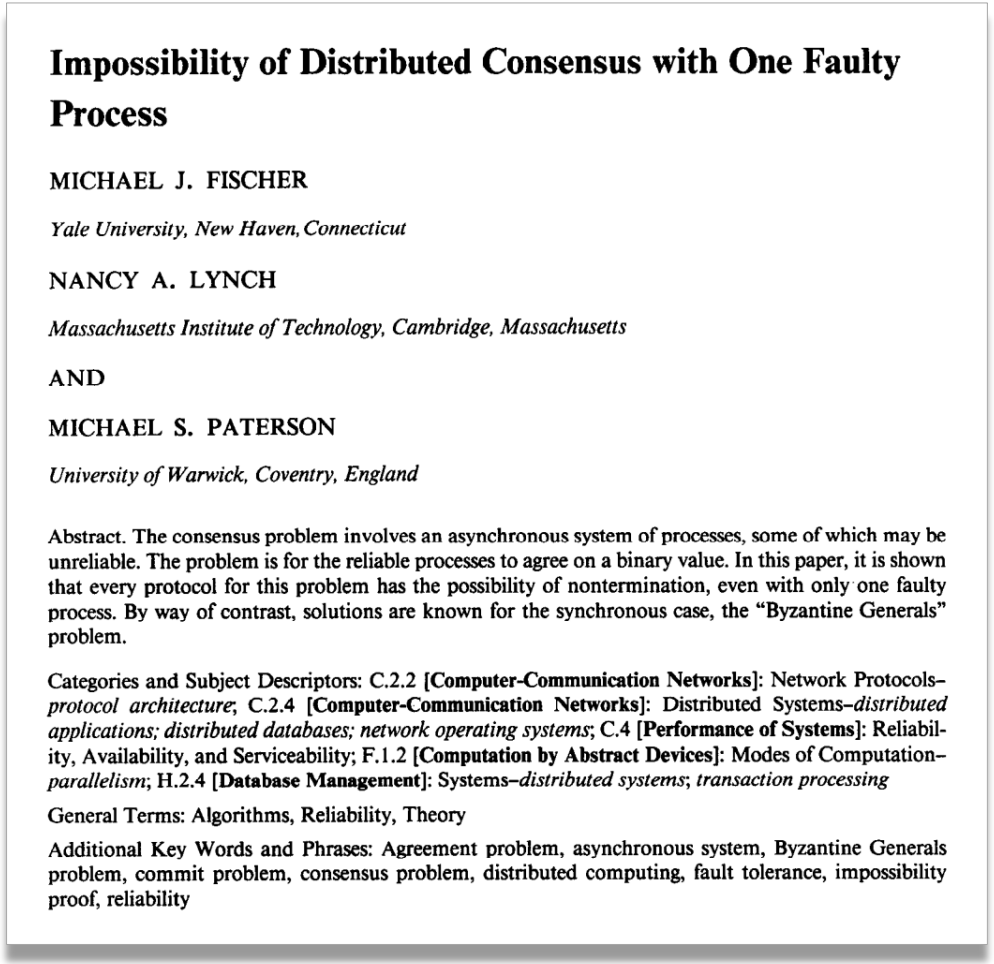
CAP Theorem

Replicated State Machines

- Lamport/vector clock-based RSM
  - Cannot progress if any replica is unavailable
- Primary backup replication
  - Can replace primary/backup upon failure
  - Unavailable until failed replica is replaced
- Paxos-based RSM
  - Available as long as majority is in the same partition

FLP Impossibility Result

- In asynchronous model, distributed consensus is impossible if even one process *may* fail
- Holds even for "weak" consensus (i.e., only some process must learn)



Impossibility of Distributed Consensus with One Faulty Process

MICHAEL J. FISCHER

Yale University, New Haven, Connecticut

NANCY A. LYNCH

Massachusetts Institute of Technology, Cambridge, Massachusetts

AND

MICHAEL S. PATERSON

University of Warwick, Coventry, England

Abstract. The consensus problem involves an asynchronous system of processes, some of which may be unreliable. The problem is for the reliable processes to agree on a binary value. In this paper, it is shown that every protocol for this problem has the possibility of nontermination, even with only one faulty process. By way of contrast, solutions are known for the synchronous case, the "Byzantine Generals" problem.

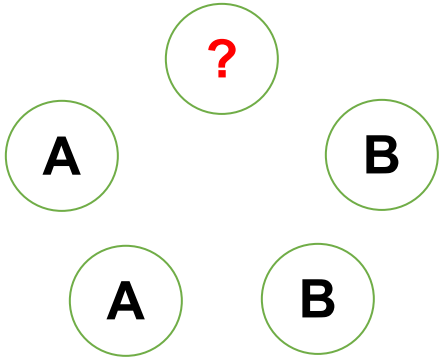
Categories and Subject Descriptors: C.2.2 **[Computer-Communication Networks]**: Network Protocols–*protocol architecture*; C.2.4 **[Computer-Communication Networks]**: Distributed Systems–*distributed applications; distributed databases; network operating systems*; C.4 **[Performance of Systems]**: Reliability, Availability, and Serviceability; F.1.2 **[Computation by Abstract Devices]**: Modes of Computation–*parallelism*; H.2.4 **[Database Management]**: Systems–*distributed systems*; *transaction processing*

General Terms: Algorithms, Reliability, Theory

Additional Key Words and Phrases: Agreement problem, asynchronous system, Byzantine Generals problem, commit problem, consensus problem, distributed computing, fault tolerance, impossibility proof, reliability

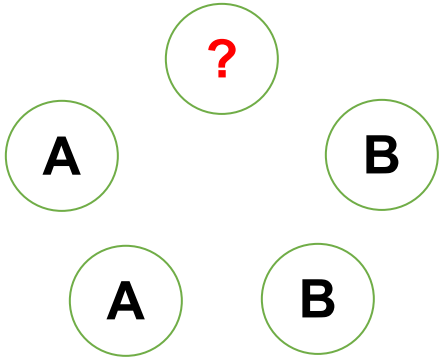
FLP Impossibility Result

- Intuition: cannot distinguish from slowness
  - May not hear from process that has deciding vote
- Implication: choose safety or liveness



FLP Impossibility Result

- Intuition: cannot distinguish from slowness
  - May not hear from process that has deciding vote

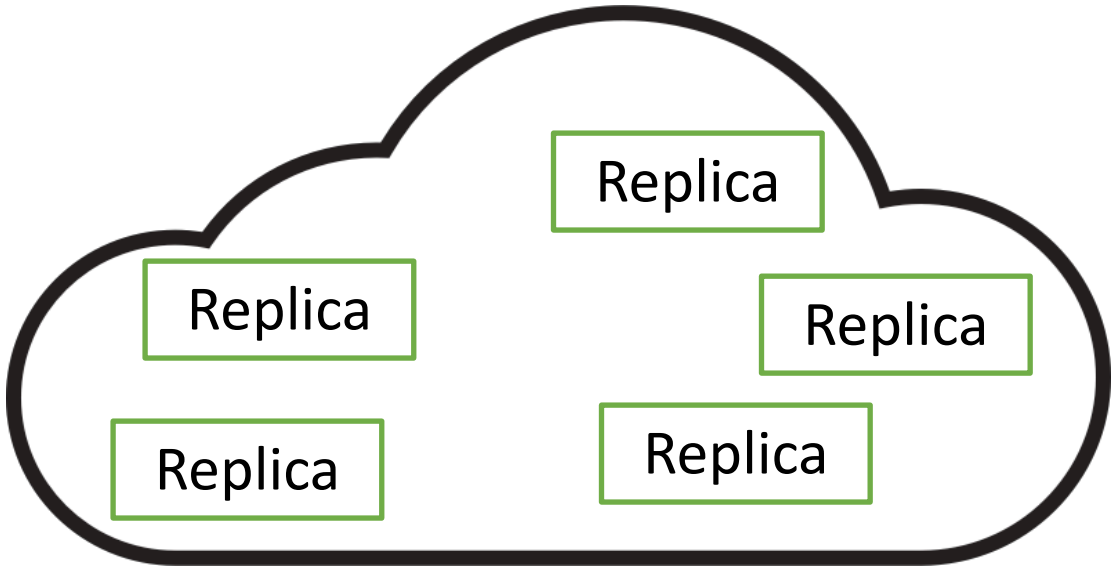


- Implication: choose safety or liveness

- How to get both safety and liveness?
  - Need failure detectors (partial synchrony)

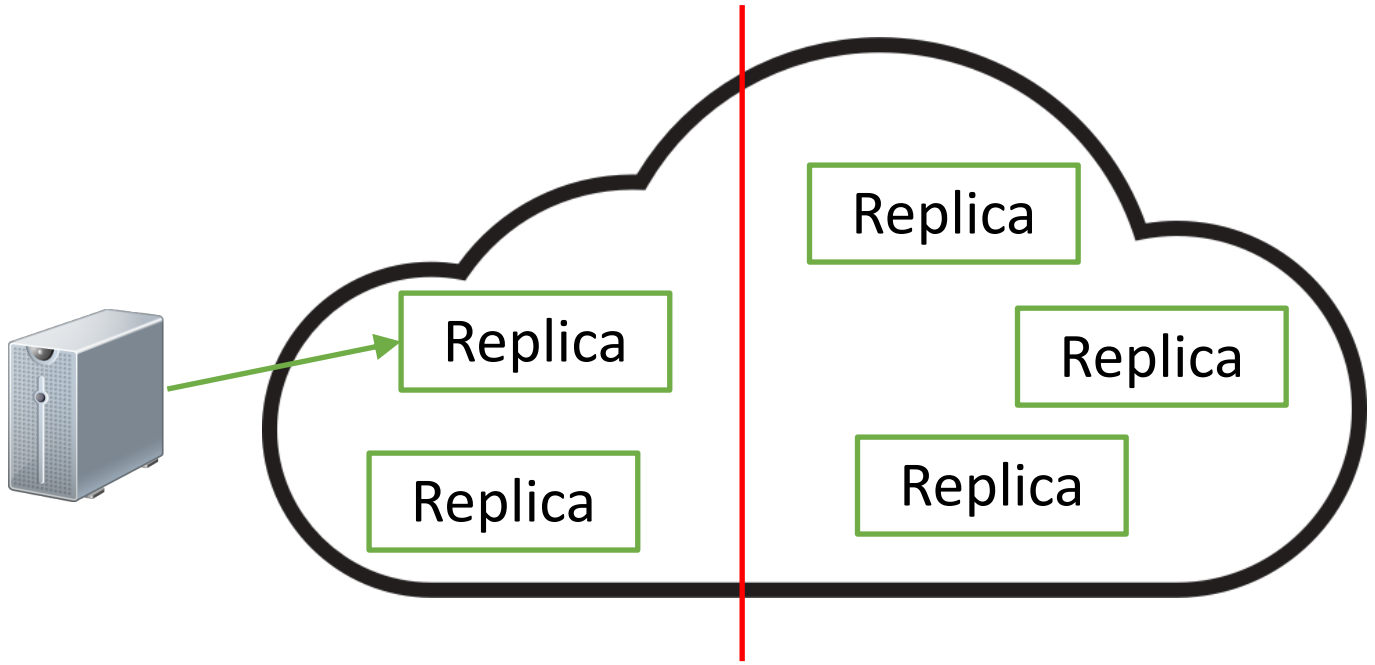
CAP Theorem

- Pick any two: consistency, availability, and partition-tolerance
  - In practice, choose between CP and AP



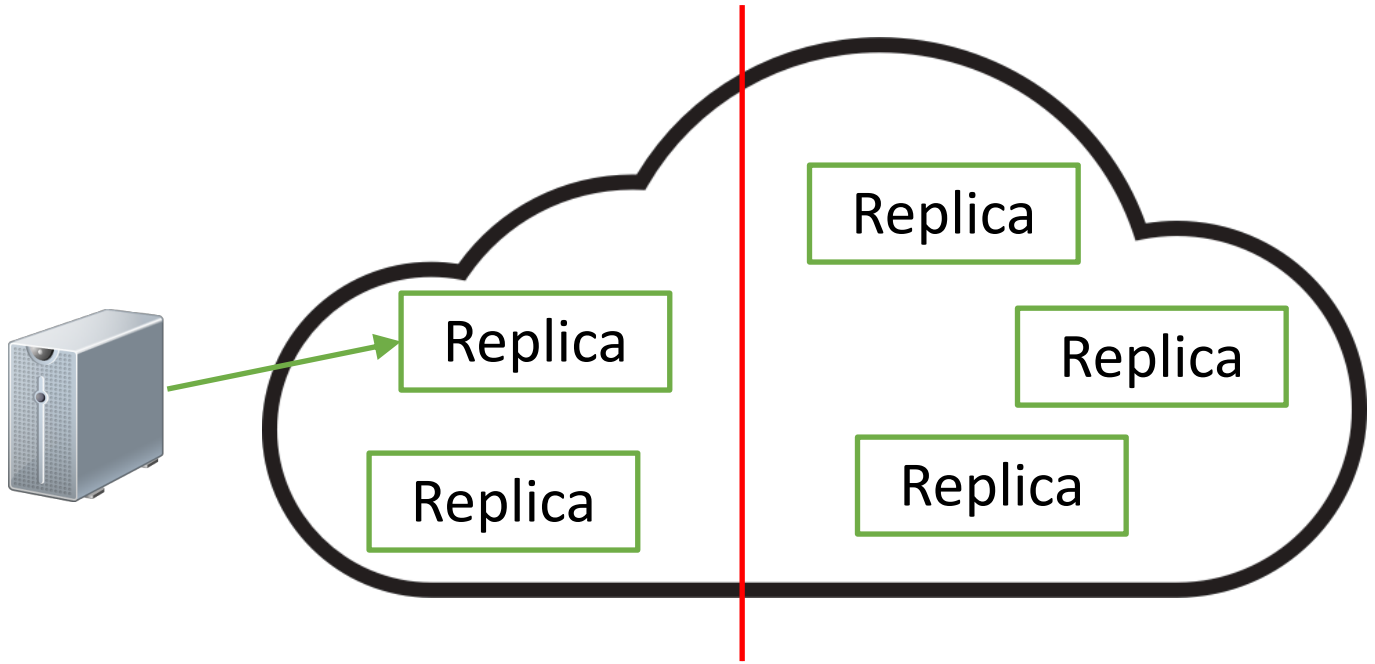
CAP Theorem

- Pick any two: consistency, availability, and partition-tolerance
  - In practice, choose between CP and AP



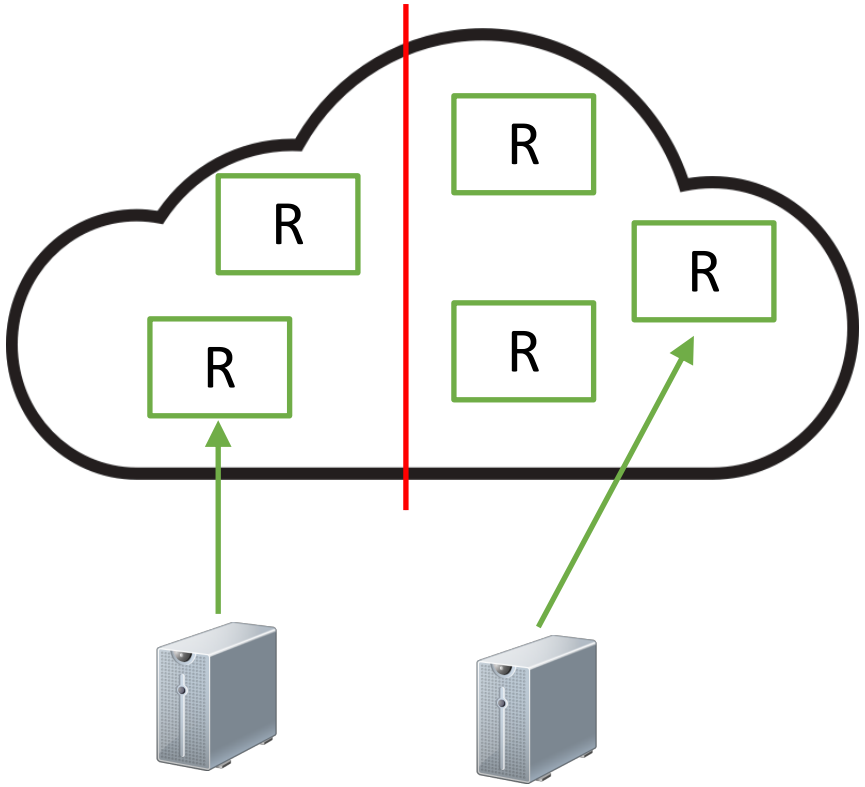
CAP Theorem

- Pick any two: consistency, availability, and partition-tolerance
  - In practice, choose between CP and AP

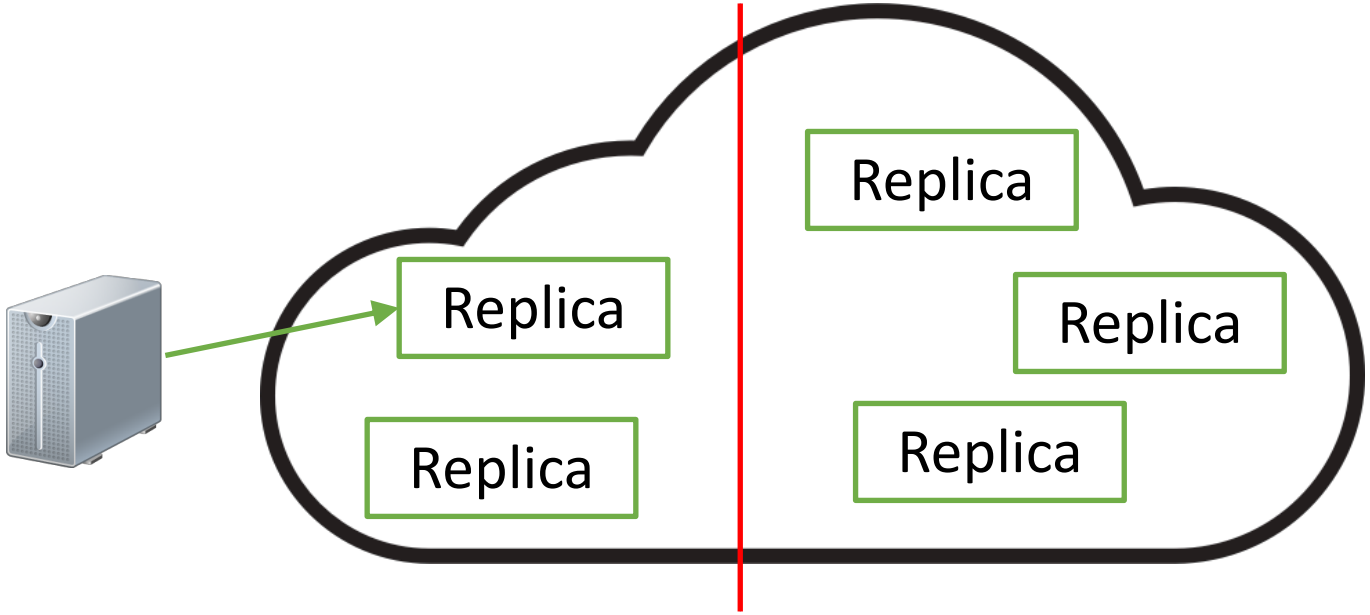


PACELC

- If partition,
  - Choose availability vs. consistency
- Else,
  - Choose latency vs. consistency

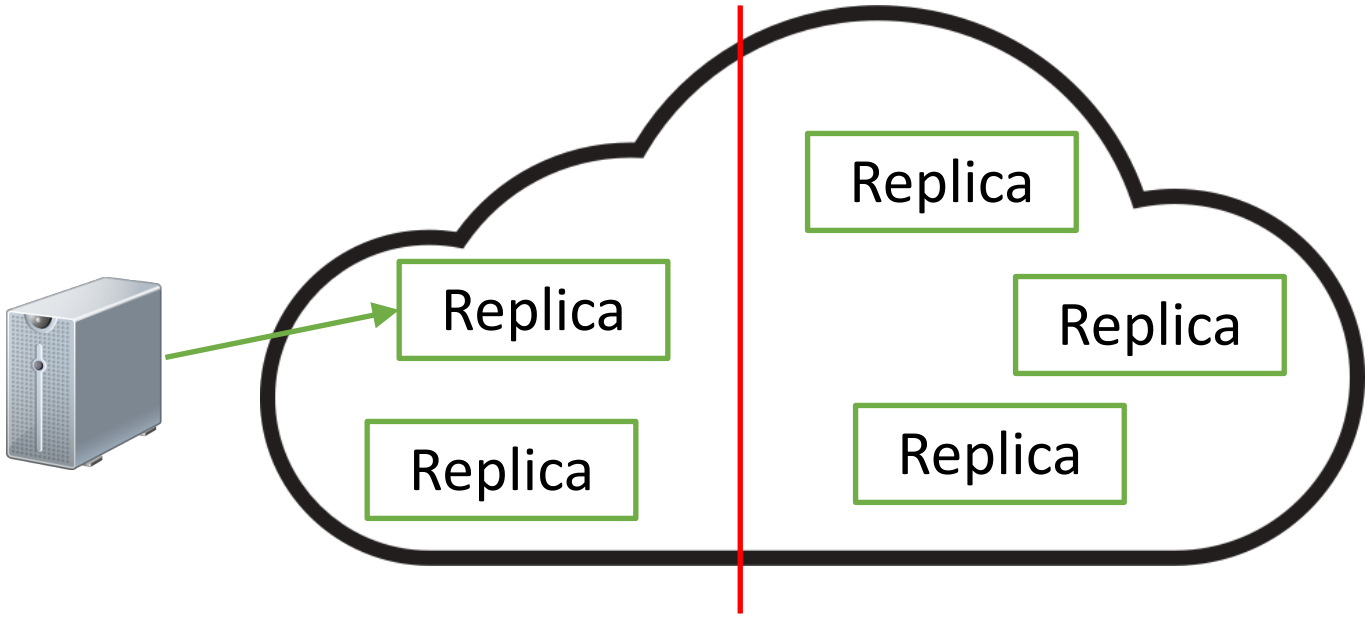


Consistency vs. Availability



- Replica can execute client requests only if in the same partition as majority

Consistency vs. Availability



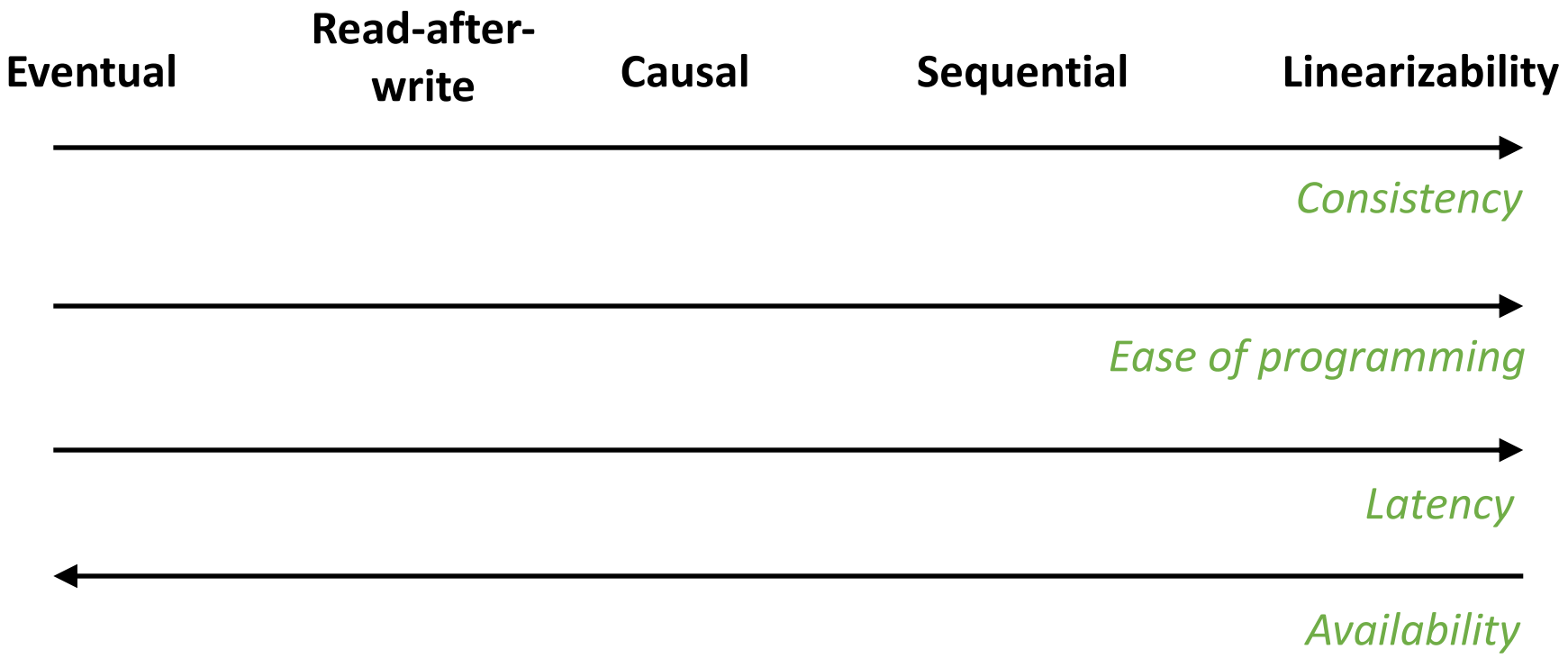
- Replica can execute client requests only if in the same partition as majority
- What consistency can we offer if we want **any** replica to **always** serve client requests?

Eventual Consistency

- At some point, all the replicas will be consistent…
  - If no new updates, all replicas eventually converge to same state

- Never know whether some write from the past may yet reach your node
  - Nodes need to maintain log of changes
  - All entries in log must be tentative forever
  - All nodes must store entire log forever

Consistency Spectrum



Consistent Hashing

How to make our system scale?

- Assumption so far: all replicas have entire state
  - Example: every replica has value for every key
- What we need instead:
  - Partition state
  - Map partitions to servers

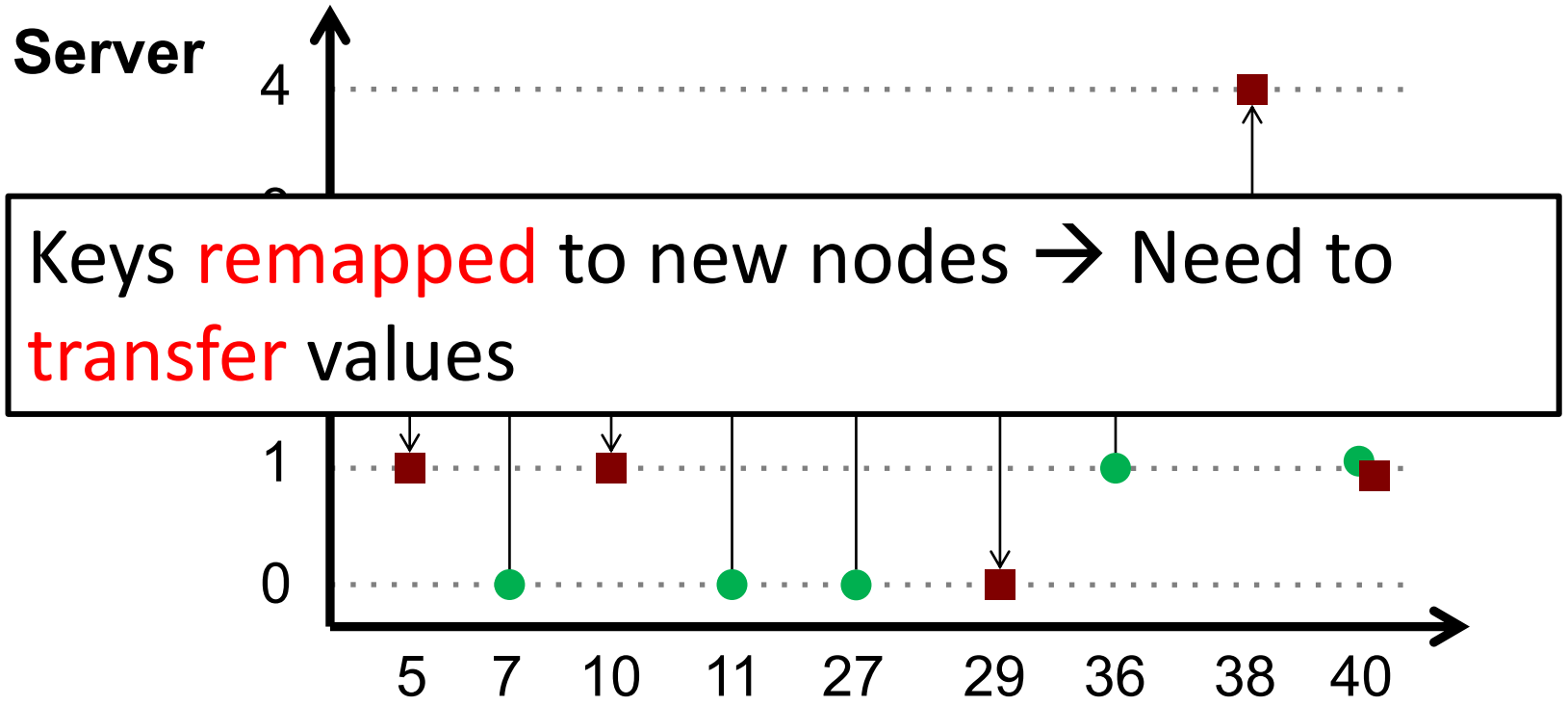
Partitioning state

- Modulo hashing
  - Apply hash function to key
  - Compute modulo to # of servers (N)
  - Store (key, value) pair at *hash(key) mod N*
- Example
  - Store student's transcripts across 4 servers
  - Hash function = (Year of birth) mod 4
  - Hash function = (Date of birth) mod 4
- Problem: skew in load across servers

Problem for modulo hashing: changing number of servers

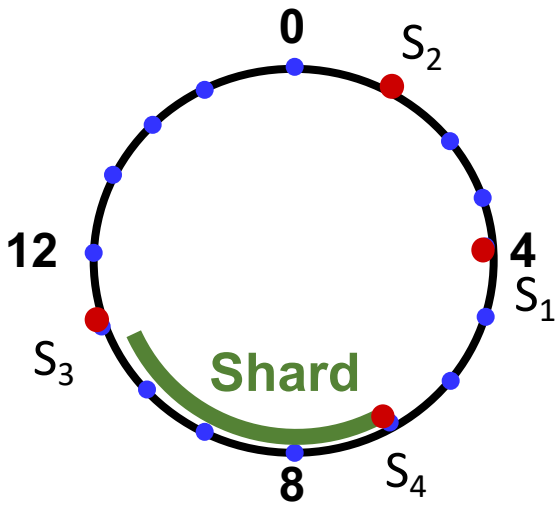
$h(x) = x + 1 \pmod 4$

Add one machine: $h(x) = x + 1 \pmod 5$



Consistent Hashing

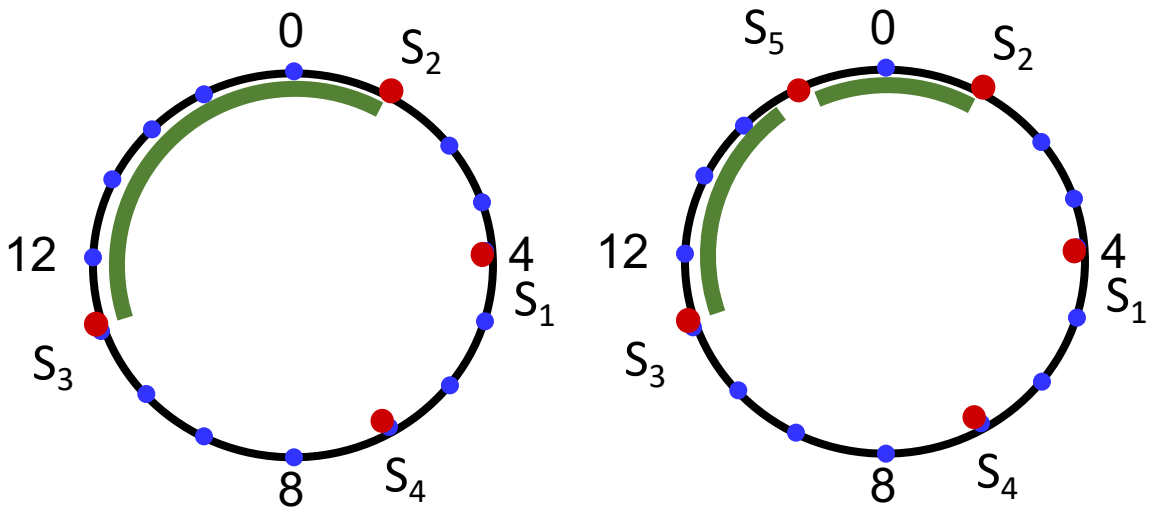
- Represent hash space as a circle



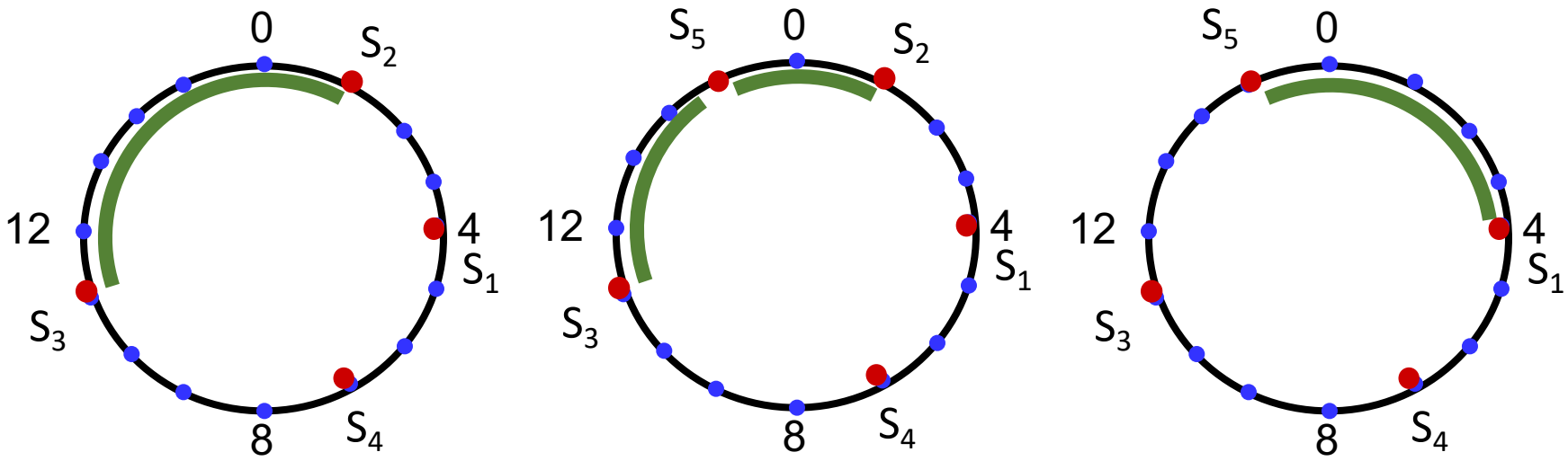
- Partition keys across servers
  - Assign every server a random ID
  - Hash server ID
  - Server responsible for keys between predecessor and itself

- How to map a key to a server?
  - Hash key and execute read/write at successor

Adding/Removing Nodes



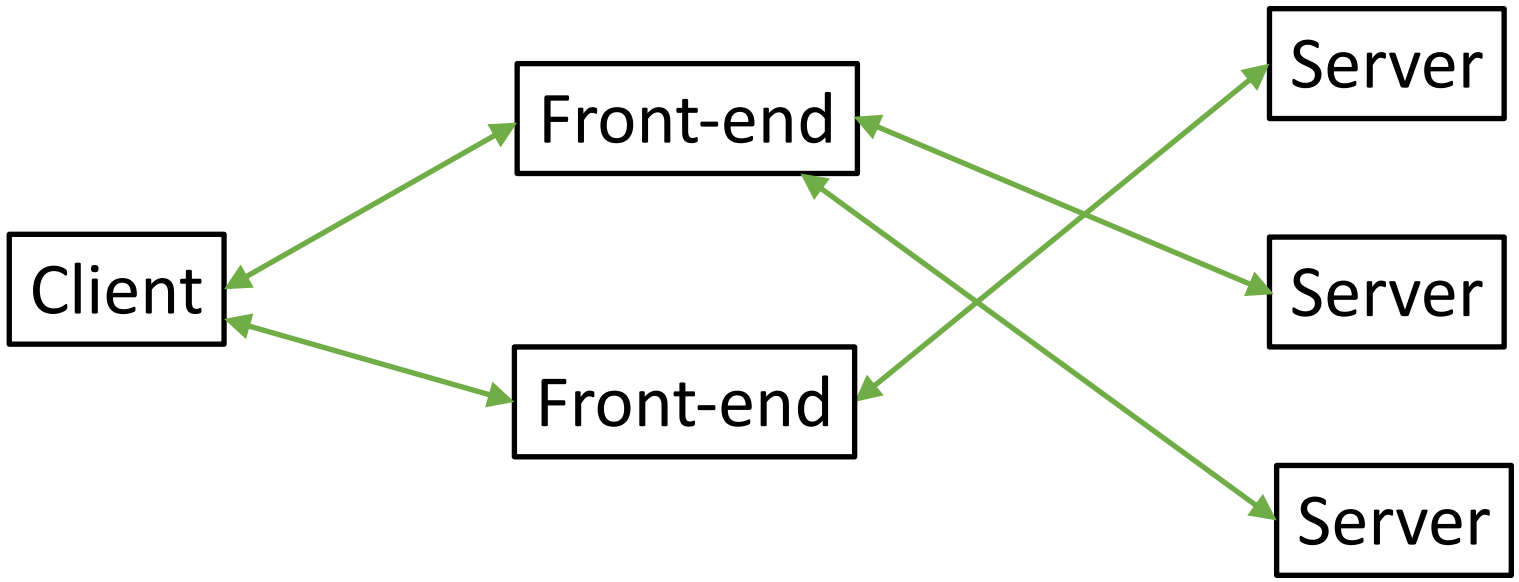
Adding/Removing Nodes



- Minimizes migration of state upon change in set of servers
  - Server addition: new server splits successor's shard
  - Server removal: successor takes over shard

Using Consistent Hashing

How does client map keys to servers?



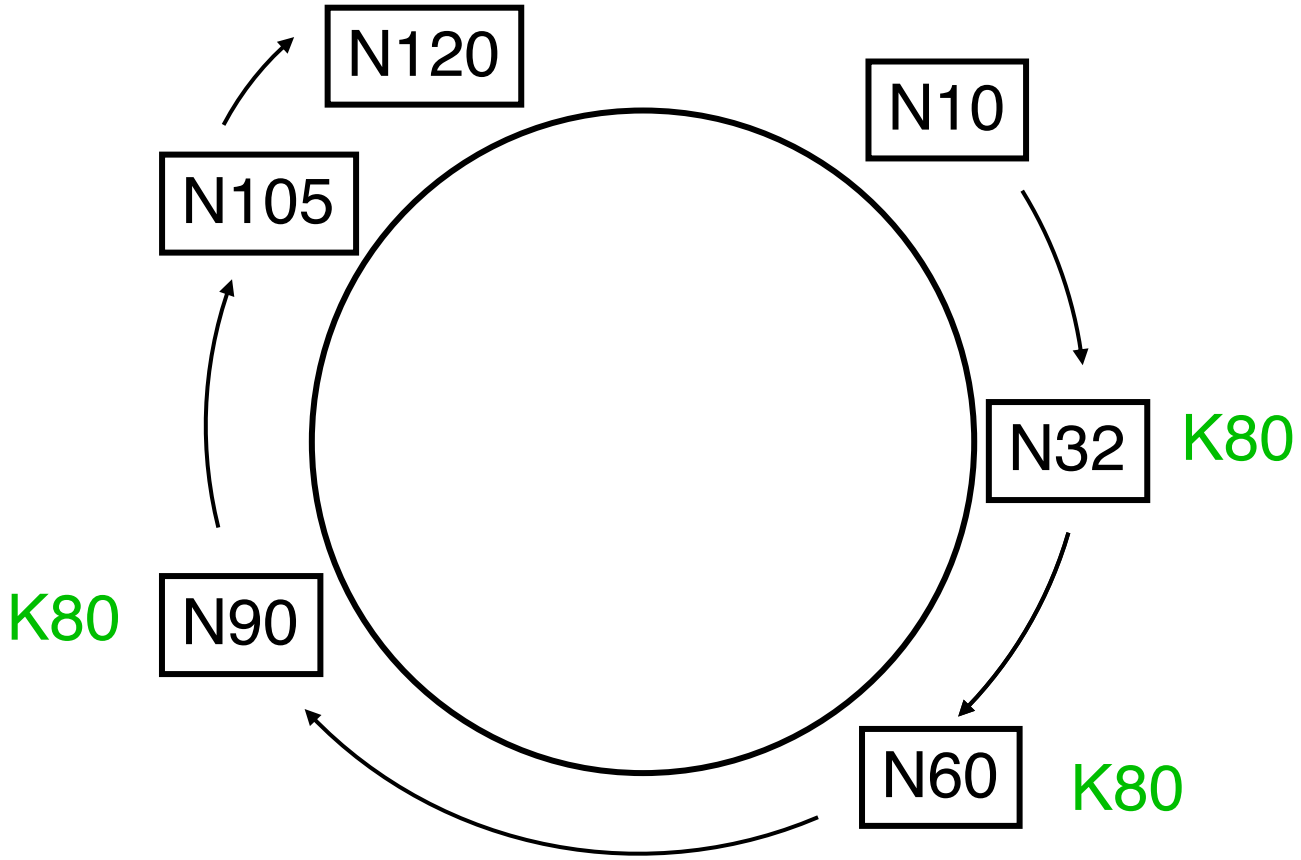
Front-ends must agree on set of active servers

Distributed Hash Table

- Scalable lookup of node responsible for any key
  - Scale to thousands (or even millions) of nodes
  - No one node knows all nodes in the system

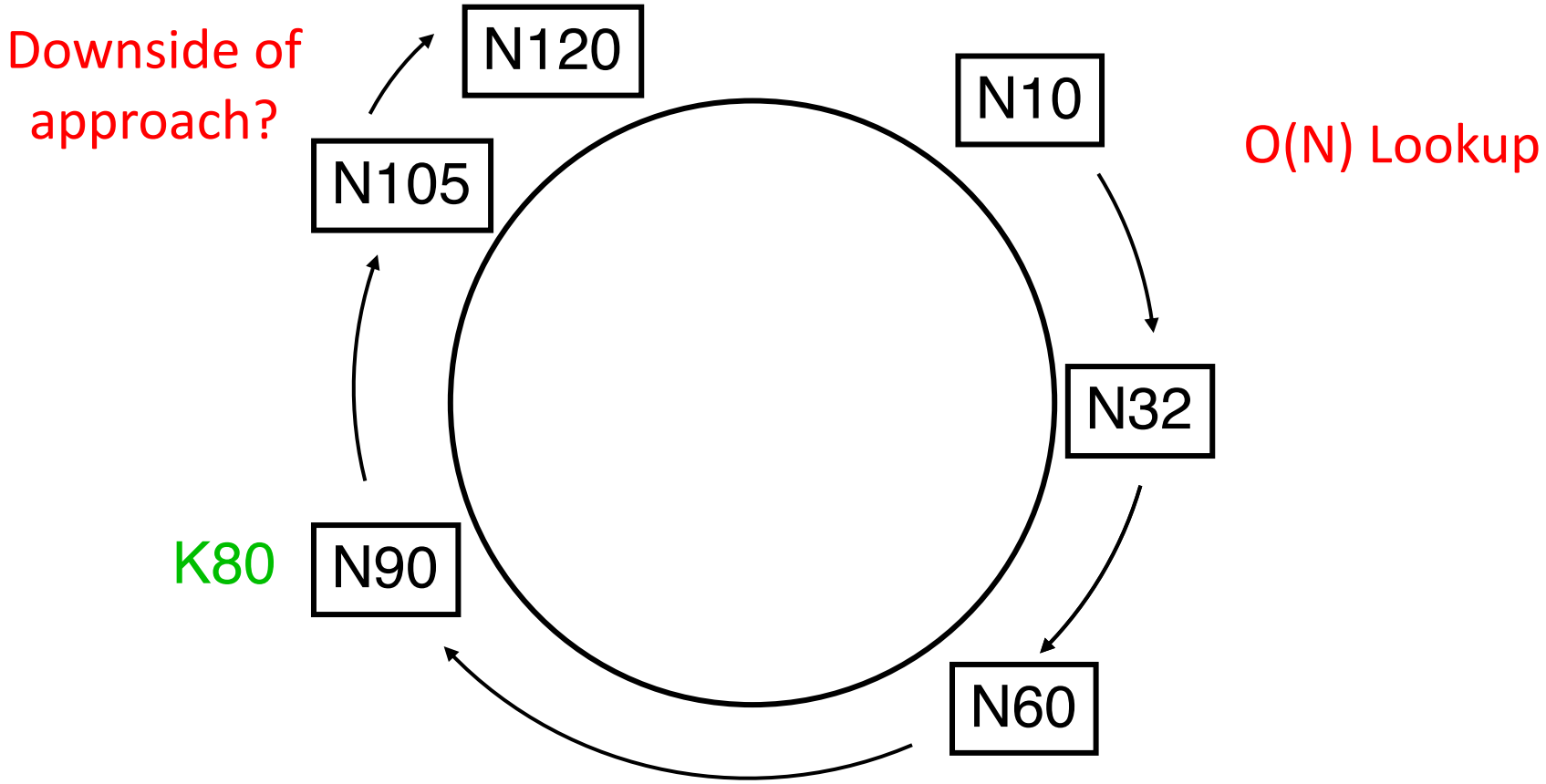
- Example usage:
  - Trackerless BitTorrent
  - Key = file content hash
  - Value = IP addresses of nodes that have file content

Successor Pointers



- If you don’t have value for key, forward to successor

Successor Pointers



- If you don’t have value for key, forward to successor

Efficient Lookups

- What's required to enable O(1) lookups?
  - Every node must know all other nodes
- Need to convert linear search to binary search!
- Idea: maintain log(N) pointers to other nodes
  - Called finger table
  - Pointer to node ½-way across hash space
  - Pointer to node ¼-way across hash space
  - …

Is log(N) lookup fast or slow?

- For a million nodes, it's 20 hops
- If each hop takes 50ms, lookups take 1 second
- If each hop has 10% chance of failure, it's a couple of timeouts
- So log(N) is better than O(N), but it is not great!

Handling Churn in Nodes

- Need to update finger tables upon addition or removal of nodes
- Hard to preserve consistency in the face of these changes