Distributed System : Introduction

Houmin Wei

Electronics Engineering & Computer Science
Peking University

北京大学
PEKING UNIVERSITY

March 6, 2018

Outline I

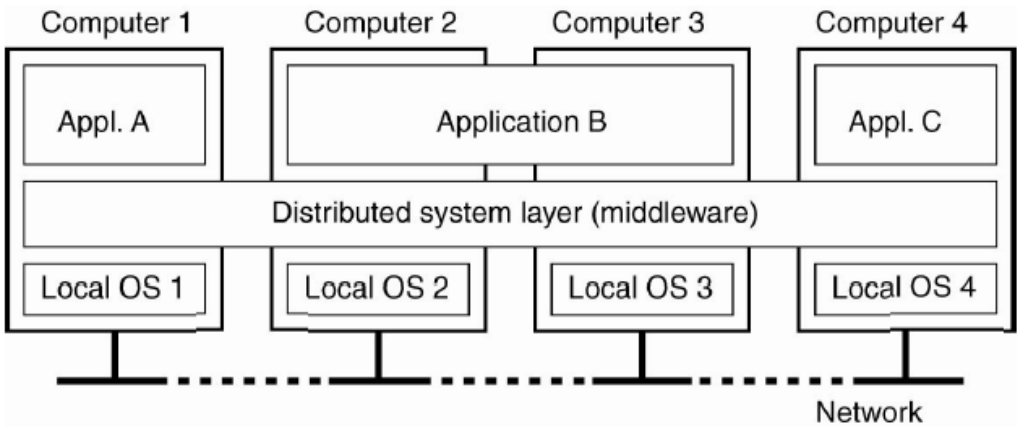
Why Distribution?

- Economics
  - Much better price/performance ratio
- Reliablity
  - One node fails, but the service goes on
- Enhanced performance
  - Tasks can be executed concurrently
- Easier modular expansion
  - Hardware and software resources can be easily added without replacing existing resources.
- Resource Sharing
  - Only one printer, share it over the network

What is a Distributed system?

Definition

A distributed system consists of a collection of autonomous computers, connected through a network and distribution middleware, which enables computers to coordinate their activities and to share the resources of the system, so that users perceive the system as single, integrated computing facility.



- Resource Sharing
- Transparency
- Scalability
- Concurrency
- Fault Tolerance

Typical Distributed Systems I

- Distributed Storage System
  - Structured Storage Systems
    - MySQL, PostgreSQL
    - Structured data
    - Strong consistency
    - Random access
    - Expansion not good
  - No-Structed Storage Systems
    - GFS, HDFS
    - Manage data using metadata by master
    - Big chunks(like 64MB), replicated copy
    - Fault tolerant automatically.
    - No random access, typically append
    - Not good for real-time system
  - Semi-Structure Storage Systems
    - NoSQL(Bigtable, Dynamo, Habse)
    - Good Expansion

Typical Distributed Systems II

- Random access(update, read)
- Key-Value Store, No SQL, No ACID

- In-memory Storage Systems
  - memcached, redis
  - based on memory, not disk.
- NewSQL
  - Spanner
  - Use atomic clock to realize syncronization
  - both expansion and SQL

- Distributed Computing System
  - MapReduce like: MapReduce(Hadoop), Spark
  - Graph: GraphLab, Pregel
  - Streaming: Storm

- ...

Topic: Performance

The dream: scalable throughput

Nx servers -> Nx total throughput via parallel CPU/disk/net.
So handling more load only requires buying more computers.

Scaling gets harder as N grows:

- Load im-balance, stragglers.
- Non-parallelizable code: initialization, interaction.
- Bottlenecks from shared resources, e.g. network.

Topic: Fault Tolerance

- 1000s of servers, complex net -> always something broken.
- We'd like to hide these failures from the application.
- We ofen want:
  - Availability – app can keep using its data despite failures.
  - Durability – app's data will come back to life when failures are repaired.
- Big idea: replicated servers.
  If one server crashes, client can proceed using the other(s).

Topic: Consistency

- General-purpose infrastructure needs well-defined behavior.
  E.g. 'Get(k) yields the value from the most recent Put(k,v)'
- Achieving good behavior is hard!
  - 'Replica' servers are hard to keep identical.
  - Clients may crash midway through multi-step update.
  - Servers crash at awkward moments. e.g. after executing but before replying.
  - Network may make live servers look dead.
- Consistency and performance are enemies.
  - Consistency requires communication, e.g. to get latest Put().
  - Strong Consistency often leads to slow systems.
  - High performance often imposes weak consistency on applications.

Later...

- Lamport's Logic Clock
- P2P Systems
- Leader Election
- Concensus Algorithm
- ...

Clock Syncronization

- Need for time syncronization
- Time syncronization techniques
- Lamport Clocks
- Vector Clocks

Inherent Limitations of a Distributed System

- Absence of Global clock
  - difficult to make temporal order of events.
  - difficult to collect up-to-date information on the state of the entire system.
- Absence of Shared Memory
  - no up-to-date state of the entire system to any individual process as there's no shared memory.
  - coherent view – all observations of different process(computers) are made at the same physical time.
  - complete view(global state) – local views(local states) + message in transit difficult to obtain coherent global state.

Physical Clocks

Problem

Sometimes we simply need the exact time, not just an ordering.

Solution: Universal Coordinated Time(UTC)

- Based on the number of transitions/sec of the cesium 133 atom.
- At present, the real time is taken as the average of some 50 cesium-clocks around the world.
- Introduces a leap second from time to time to compensate that days are getting longer.

Note

UTC is broadcast through short wave radio and satellite. Satellites can give an accuracy of about $\pm 0.5ms$.

Physical Clocks

Problem

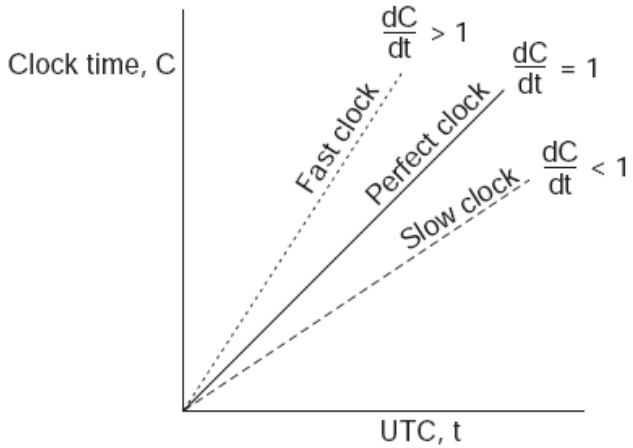
Suppose we have an distributed system with a UTC-receiver
Somewhere in it => we still have to distributed its time to each machine.

Basic Principle

- Every machine has a timer that generates an interrupt H times per second.
- There is a clock in machine p that ticks on each timer interrupt. Denote the value of that clock by $C_p(t)$, where t is UTC time.
- Ideally, we have that for each machine p, $C_p(t) = t$, or in other words, $\frac{dC}{dt} = 1$.

Physical Clocks

In practice: $1 - r \leq \frac{dC}{dt} \leq 1 + r$.



Goal

Never let 2 clocks in any system differ by more than δ time units $=>$ Syncronize at least every $\delta/(2r)$ seconds.

Physical Clocks

Clock Syncronization Principle

Principle I

Every machine asks a timer server for the accurate time at least every $\delta/(2r)$ seconds(Network Time Protocol).

Principle II

Let the server scan all machines periodically, calculate an average, and inform each machine how it should adjust its time relative to its present time.

Logic Clock

Why not physical clock?

- Nodes may differ on real time at ms level using NTP.
- It's not necessary. If two processes do not interact, it is not necessary that their clocks be syncronized because the lack of synchronization would not be observable and thus could not cause problem.
- What usually matters is not that all processes agree on exactly what time it is, but rather that they agree on the order in which events occur.

Partial Order

Definition

Orders are special binary relations. Suppose that P is a set and that $\leq$ is a relation on P. Then $\leq$ is a partial order if it is reflexive, antisymmetric, and transitive. i.e., for all a, b and c in P, we have that:

$$a \leq a (reflexive) \tag{1}$$

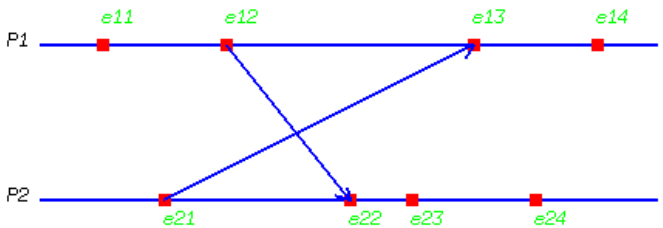
$$\text{if } a \leq b \text{ and } b \leq a \text{ then } a = b (antisymmetric) \tag{2}$$

$$\text{if } a \leq b \text{ and } b \leq c \text{ then } a \leq c (transitive) \tag{3}$$

Happens Before

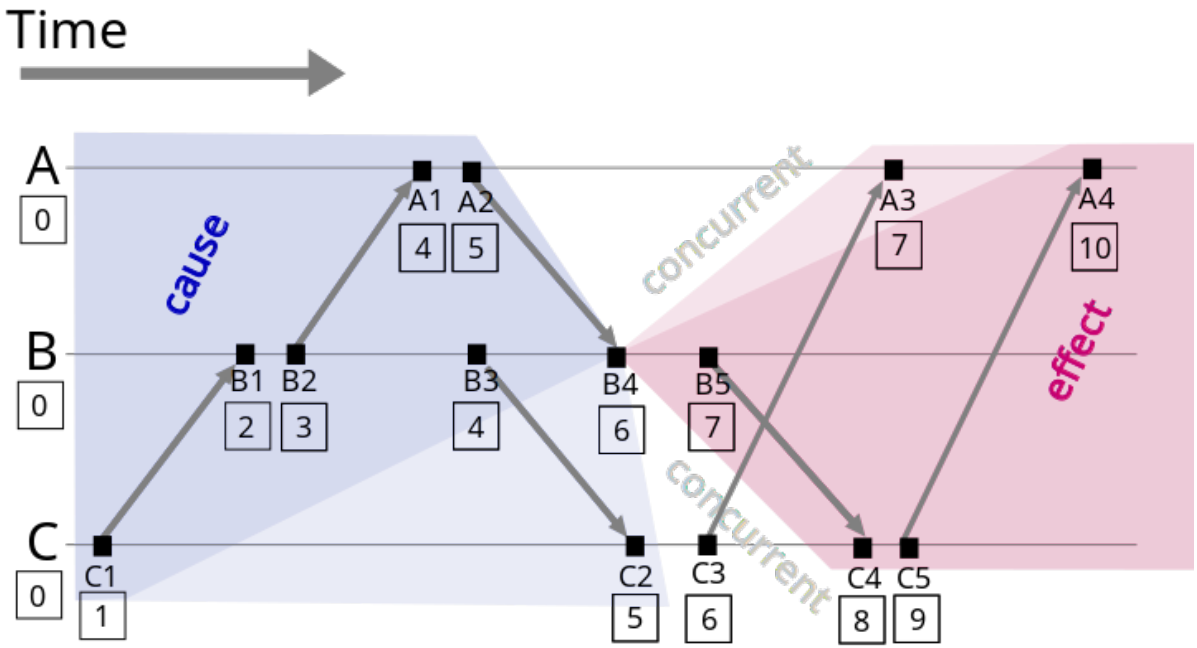
The happens before $a \rightarrow$ relation

- $a \rightarrow b$, if a and b are events in the same process and a occurred before b.
- $a \rightarrow b$, if a is the event of sending a message m in a process and b is the event of receipt of the same message by another process
- if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$ (transitive).
- concurrent: $a||b$ if $!(a \rightarrow b)$ and $!(b \rightarrow a)$.
- for any 2 events in a system, either $a \rightarrow b$ or $b \rightarrow a$ or $a||b$.



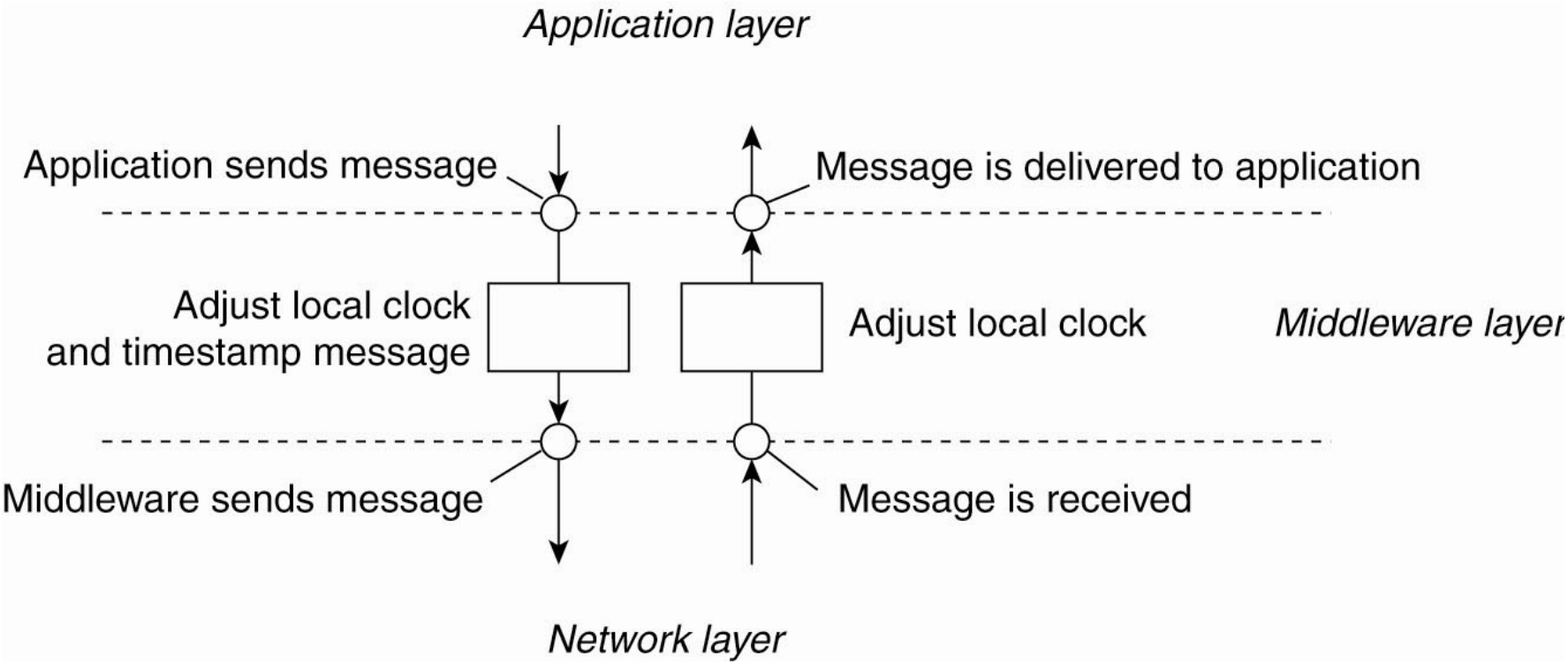
$e11 \rightarrow e12$, $e12 \rightarrow e22$
$e21 \rightarrow e13$, $e14 \parallel e24$

Lamport Clock



- 每个事件对应一个 Lamport 时间戳，初始值为 0
- 如果为节点内发生事件，时间戳加 1
- 如果为发送事件，时间戳加 1 并在消息中带上该时间戳
- 如果为接收事件，时间戳 =Max(本地时间戳，消息中的时间戳)+1

Positioning of Lamport Clock in DS



Lamport Clock

Limitation of Lamport-Clock

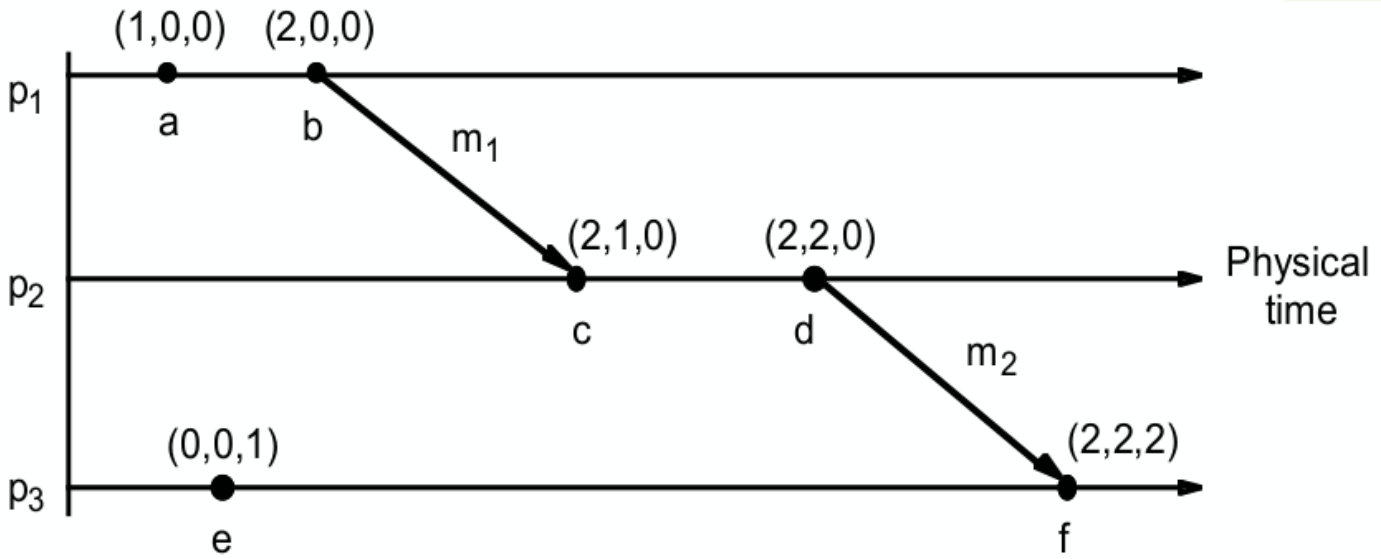
$C_i(a)$ – timestamp of event a at P_i
if $a \rightarrow b$, then $C(a) < C(b)$
but $C(a) < C(b)$ does not necessarily imply $a \rightarrow b$

- Vector Clock overcome the shortcoming of Lamport Clock
- Goal
  - Want ordering that matches causality.
  - $C(a) < C(b)$ if and only if $a \rightarrow b$.
- Method: label each event by vector V(e) [c1, c2, ..., cn].

Vector Clock

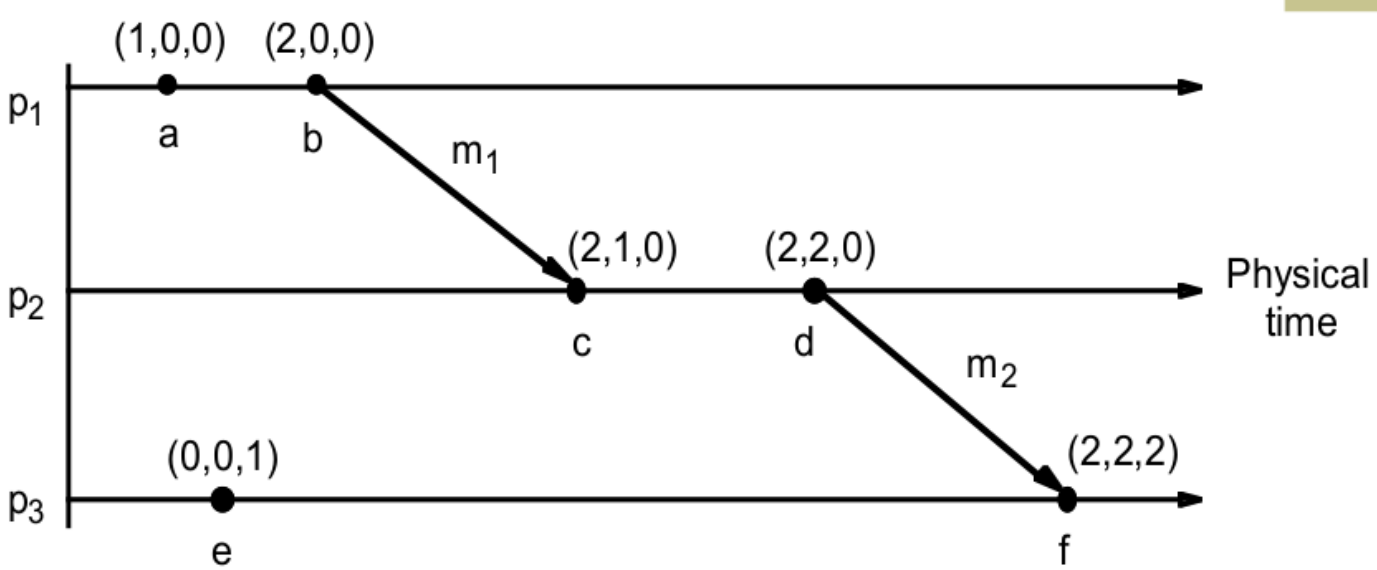
- Initially, all vectors [0,0,…,0]
- For event on process i, increment own c_i
- Label message sent with local vector
- When process j receives message with vector $[d_1, d_2, \ldots, d_n]$:
  - Set local each local entry k to $\max(c_k, d_k)$
  - Increment value of c_j

Vector Clock



- At p_1
  - *a occurs at* (1,0,0); *b* occurs at (2,0,0)
  - piggyback (2,0,0) on m_1
- At p_2 on receipt of m_1 use *max* ((0,0,0), (2,0,0)) = (2, 0, 0) and add 1 to own element = (2,1,0)
- Meaning of =, <=, max etc for vector timestamps
  - compare elements pairwise

Vector Clock



- Note that e $\rightarrow$ e' implies V(e)<V(e'). The converse is also true
- Can you see a pair of parallel events?
  - *c* || *e* (parallel) because neither *V*(c) <= *V*(*e*) nor *V*(*e*) <= *V*(*c*)

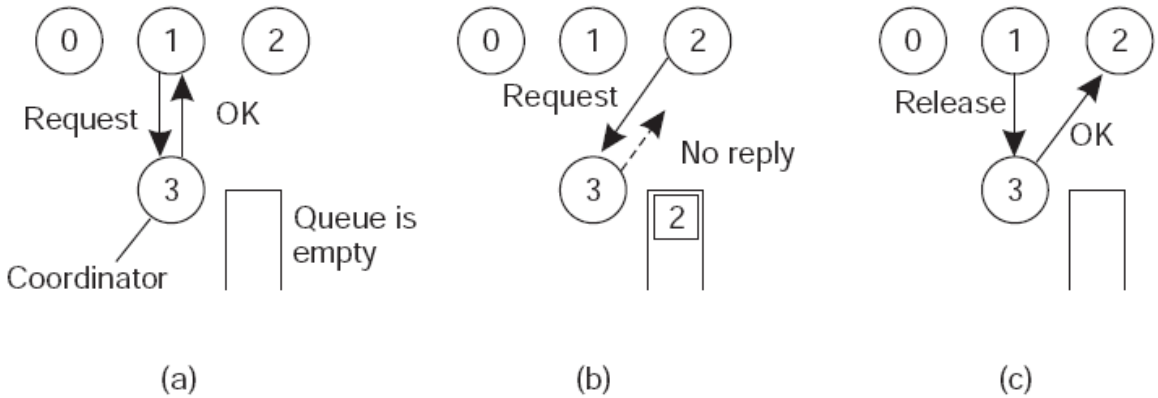
Review

- Clock Syncronization
  - Rely on a time-stamped network messages
  - Estimate delay for message transmission
  - Can synchronize to UTC or to local source
  - Clocks never exactly syncronized
  - Often inadequate for distributed system
- Logical Clocks
  - Encode causality relationship
  - Lamport clocks provide only one-way encoding
  - Vector clocks provide exact causality information

Election Algorithms

A Centralized Algorithm

- One process is elected as the coordinator.
- Whenever a process wants to access a shared-resource, it sends request to the coordinator to ask for permission.
- Coordinator may queue requests.



Election Algorithm

Principle

An algorithm requires that some process acts as a coordinator. The question is how to select this special process dynamically.

Election by bullying

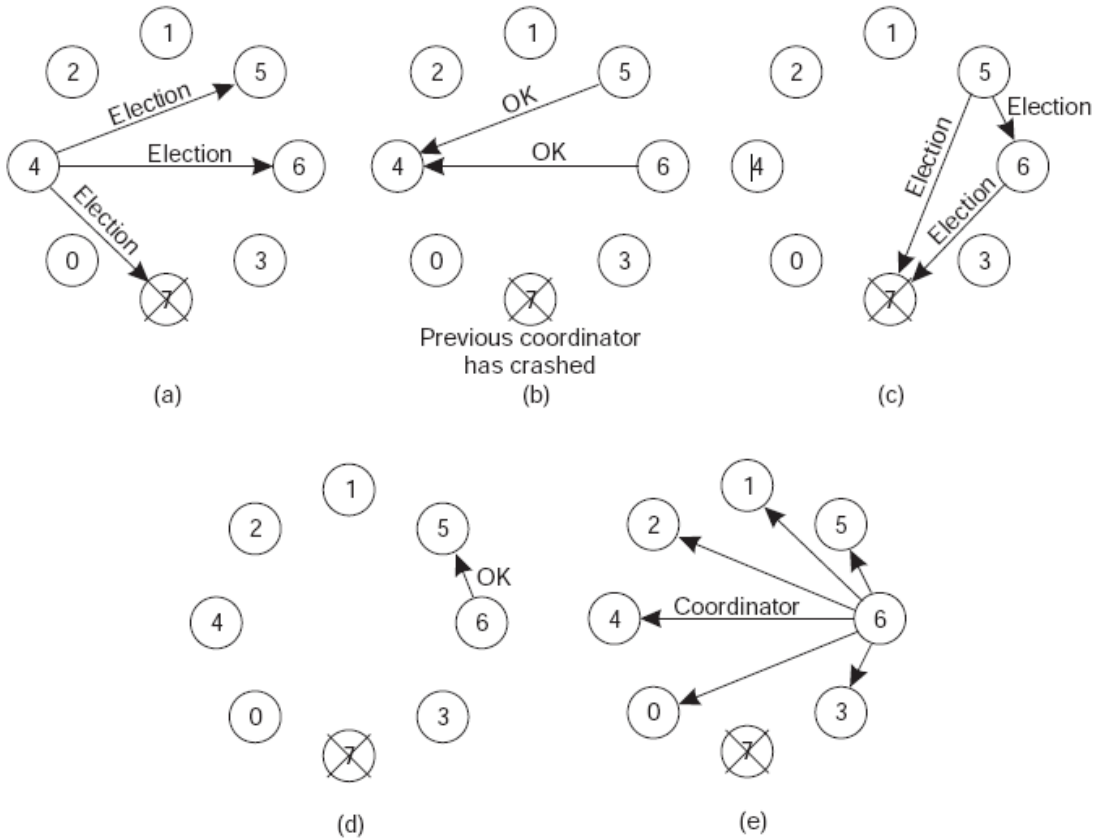
Principle

Each process has an associated priority(weight). The process with the highest priority should always be elected as the coordinator.

Issue: How do we find the heaviest process?

- Any process can just start an election by sending an election message to all other processes with higher numbers.
- If a process P_{heavy} receives an election message from a lighter process P_{light}, it sends a take-over message to P_{light}. P_{light} is out of the race.
- If a process doesn't get a take-over message back, it wins, and sends a victory message to all other processes.

Election by bullying



(a) Process 4 holds an election.
(b) Processes 5 and 6 responded, telling 4 to stop.
(c) Now 5 and 6 hold an election.
(d) Process 6 tells 5 to stop.

Election by bullying

Issue

Suppose crashed nodes comes back online:

- Sends a new election message to higher numbered processes.
- Repeat until only one process left standing.
- Announces victory by sending message saying that it is coordinator(if not already coordinator)
- Existing(lower numbered) coordinator yields.

Hence the term 'bully'

Election in a ring

Principle

Process priority is obtained by organizing process into a (logical) ring. Process with the highest priority should be elected as coordinator.

- Any process can start an election by sending an election message to its successor. If a successor is down, the message is passed on to the next successor.
- If a message is passed on, the sender adds itself to the list. When it gets back to the initiator, everyone had a chance to make its presence known.
- The initiator sends a coordinator message around the ring containing a list of all living processes. The one with the highest priority is elected as coordinator.

Definition of P2P

A P2P(Peer to Peer) System exhibits the following characteristics:

- High degree of autonomy from central servers
- Exploits resources at the edge of the network
  - Storage, CPU cycles, human presence
- Individual nodes have itermittent connectivity

Not strict requirements, instead typical characteristics
Above characteristics allow us to distinguish P2P systems from other similar systems.

Applications of P2P

- P2P File Sharing and content distribution:
  BitTorrent, Napster, Gnutella, KaZaA
- P2P Communication:
  Typical instant messaging setup: Skype
- P2P Computation
- P2P Collaboration

Napster: Overview

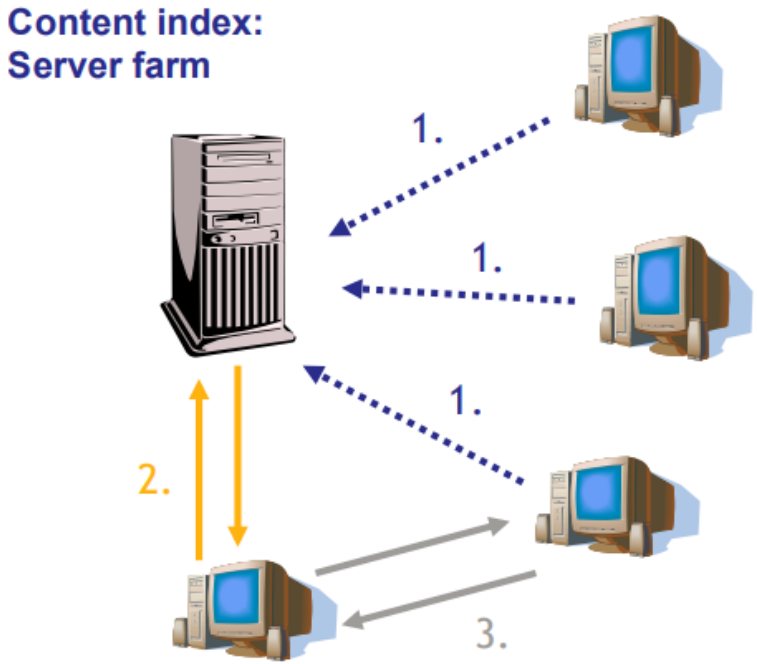
- The first P2P file sharing application(MP3 only)
- Made the term 'peer-to-peer' known(1999, Shawn Fanning)
- Based on central index server(actually a server farm)
- User registers with the central server
  - Give list of files to be shared
  - Central server know all the peers and files in network
- Searching based on keywords
- Search results were a list of files with information about the file and the peer sharing it
  - For example, encoding rate, size of file, peer's bandwidth
  - Some information entered by the user, hence unreliable

Napster: Framework

Original Napster design

1. Peers register with central server, give list of file to be shared.
2. Peers send queries to central server which has content index of all files.
3. File transfers happen directly between peers.

Last point is common to all P2P networks and is their main strength as it allows them to scale well.

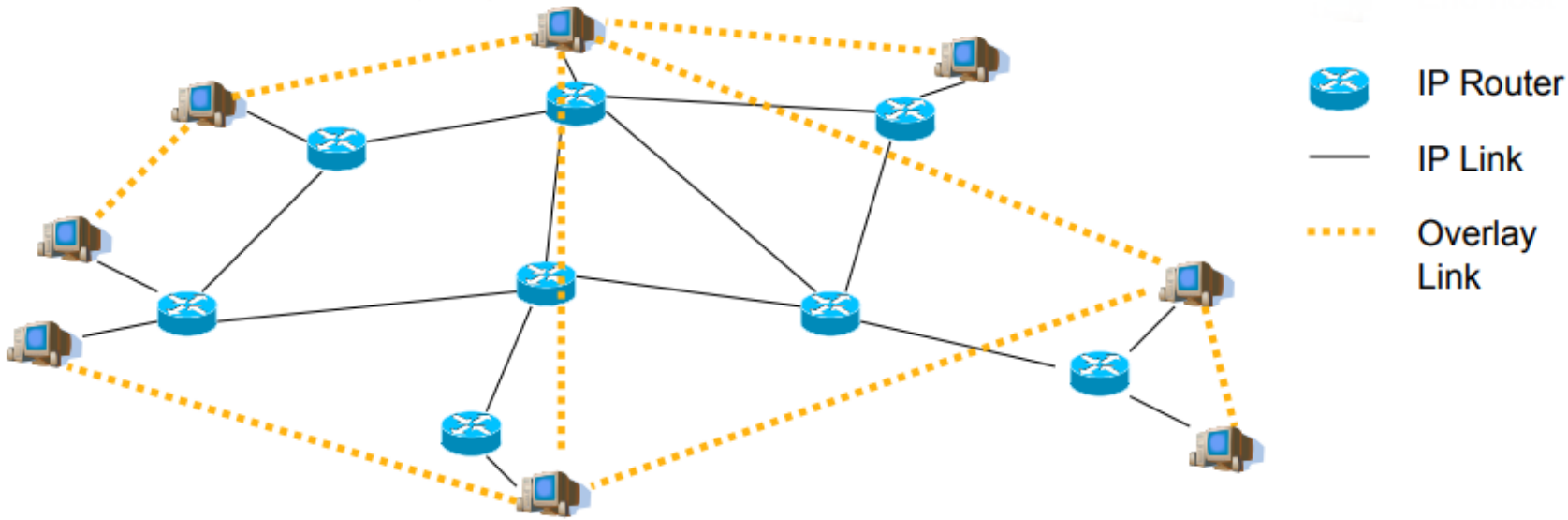


Napster: Discussion

- Pros
  - Consistent view of the network
    Central server always knows who is there and who is not.
  - Fast and efficient searching, Search scope is O(1)
    Central server always knows all available files.
  - Answer guaranteed to be corrent
    Nothing found means none of the current on-line peers in the network has the file.
- Cons
  - Single point of failure
  - Server needs enough computation power to handle all queries
  - Server maintains O(N) State

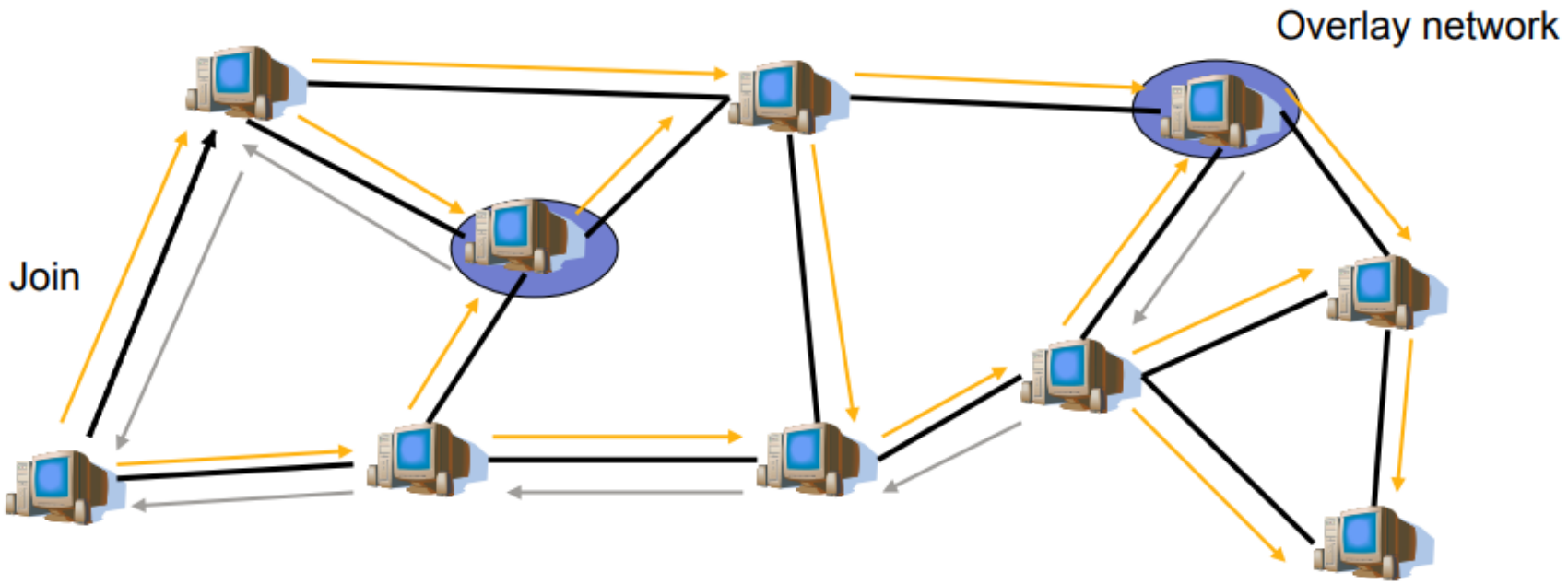
Gnutella: Overview

- Napster is centralize, Gnutella is fully distributed.
- Based on overlay network



  A virtual network on top of underlying IP network
- All peers are fully equal, called servents(server + client)

Gnutella: Framework



- To join, peer needs address of one member, learn others
- Queries are sent to neighbors
- Neighbors forword queries to their neighbors(flooding)
- Replies routed back via query path to querying peer

Guntella: Discussion

- Pros:
  - Fully de-centralized
  - Search cost distributed
- Cons:
  - Search scope is O(N)
  - Nodes leave often, network unstable
  - Periodic Ping/Pong consume lots of resources

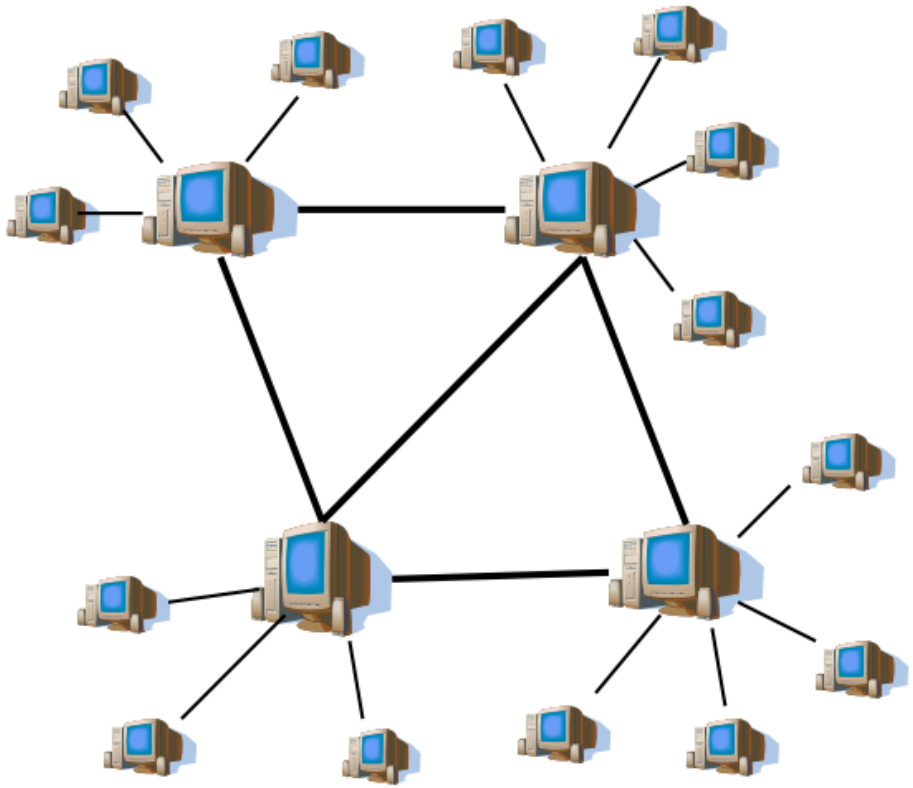
KaZaA: Overview

- Created in 2001
- Two kinds of nodes in KaZaA: Ordinary Nodes, SuperNodes
- ON is a normal peer run by a user
- SN is also a peer run by a user, but with more resources and responsibilities
- KaZaA forms a two-tier hierachy
  top level has only SN, lower level only ON
- ON belongs to one SN
- SN acts as a Napster-like hub for all its ON-children
  keeps track of files in those peers

KaZaA: Framework

Smart Query Flooding:

- Join: on startup, client contacts a SN
- Publish: send list of files to SN
- Search: send query to SN, SN flood query amongst themselves
- Fetch: get the file directly from peers



KaZaA: Discussion

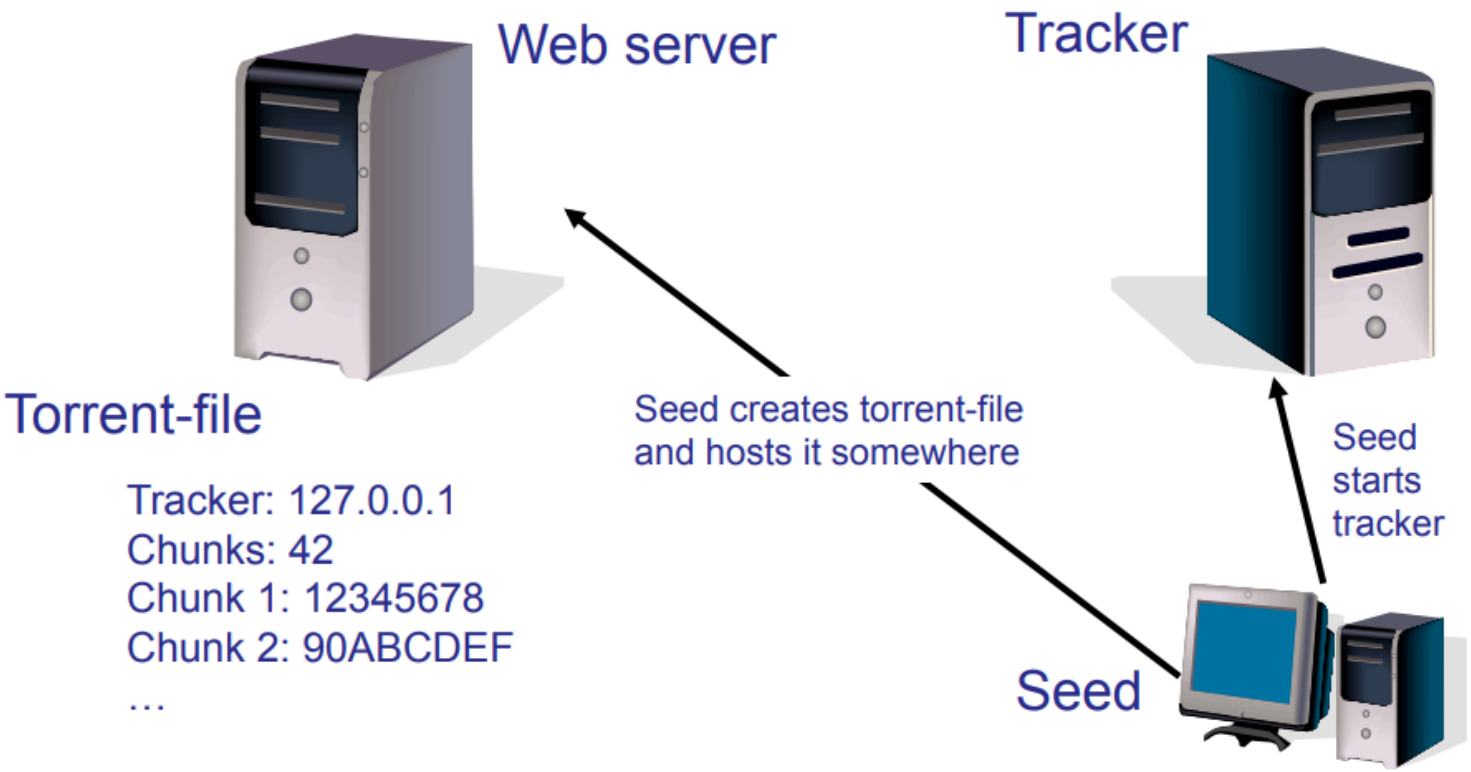
- Pros:
  - Efficient searching under each SN
  - Flooding restricted to SN only
  - Efficient searching with 'low' resource usage
- Cons:
  - Still no real guarantees on search scope or search time

BitTorrent: Overview

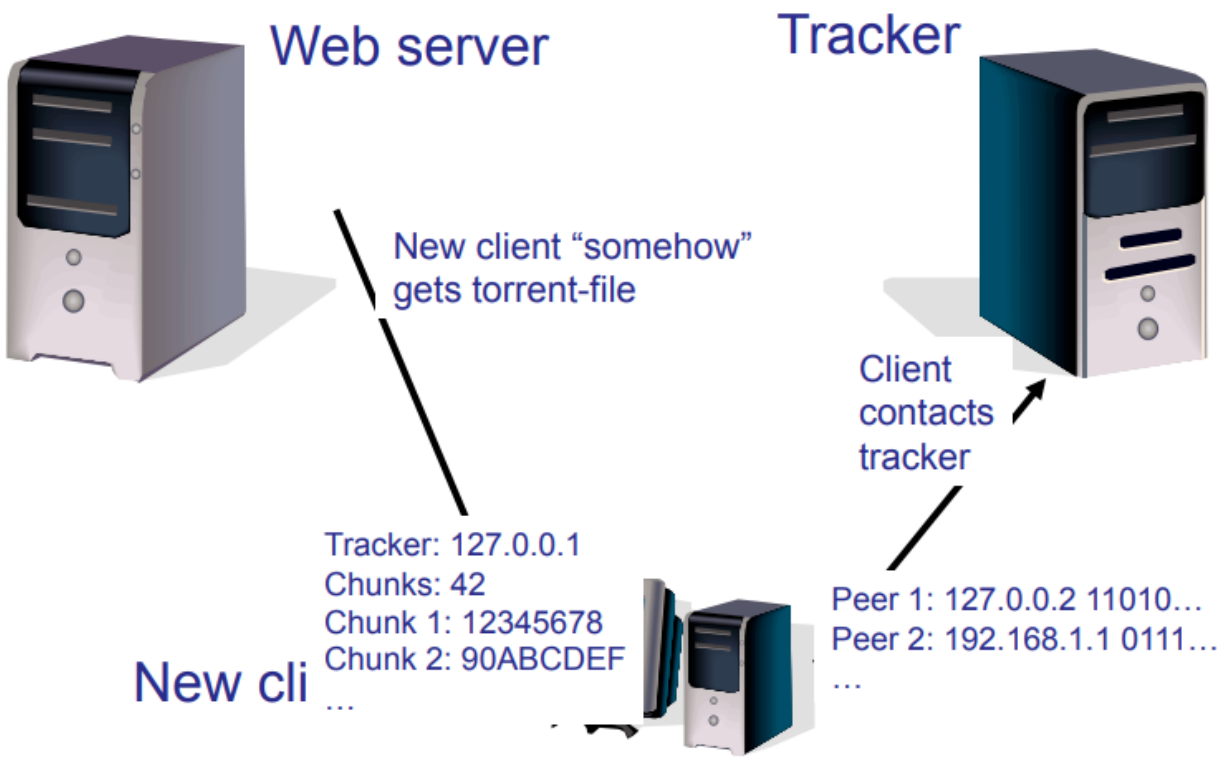
2 basic ways to find objects:

- Search for them with keywords that match objects's description
- Address them using their unique name(cf. URLs in Web)
- Swarming:
  - Join: contact centralized tracker server, get a list of peers.
  - Publish: Run a tracker server.
  - Search: Out-of-band, E.g. use Google to find a tracker for the file you want.
  - Fetch: Download chunks of the file from your peers. Upload chunks you have to them.

BitTorrent: Framework



BitTorrent: Framework



BitTorrent: Tit-for-Tat

- A is downloading from some other people
  A will let the fastest N of those download from him.
- Be optimistic: occasionally let freeloaders download
  Otherwise no one would ever start!

BitTorrent: Discussion

- Pros:
  - Works reasonably well in pratice
  - Gives peers incentive to share resources, avoids freeriders
- Cons:
  - Central tracker server need to bootstrap swarm.
  - What if tracker server fails?

Distributed Hash Tables

In BitTorrent version 4.2.0, BitTorrent introduce Trackerless torrent using DHT.

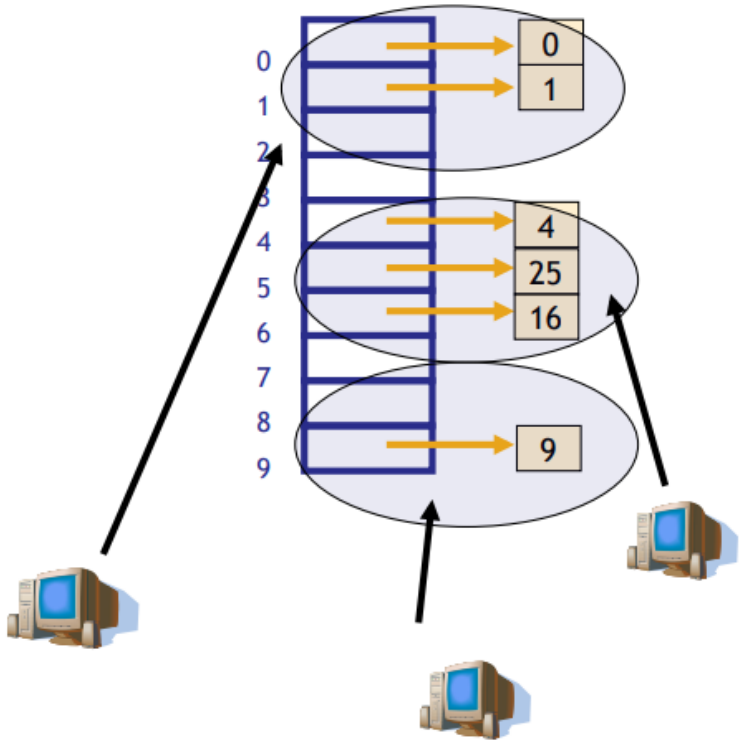
- Actual file transfer process in P2P network is scalable
  File transfers directly between peers
- Searching does not scale in same way
- Put another way: Use addressing instead of searching
- Original motivation for DHTs: More efficient searching and object location in P2P networks
- For a special resource, the tracker record the nodes/peers associated with the resource.
- If the tracker fails, we can lookup the DHT for the nodes/peers info.

Recall Hash Table

- allow insertions, deletions, and lookup in constant time.
- fixed-size array, elements of array also called hash buckets.
- Hash funtion maps keys to elements in the array.
- Properties of good hash functions
  - Fast to compute
  - Good distribution of keys into hash table
  - Example: SHA-1 algorithm

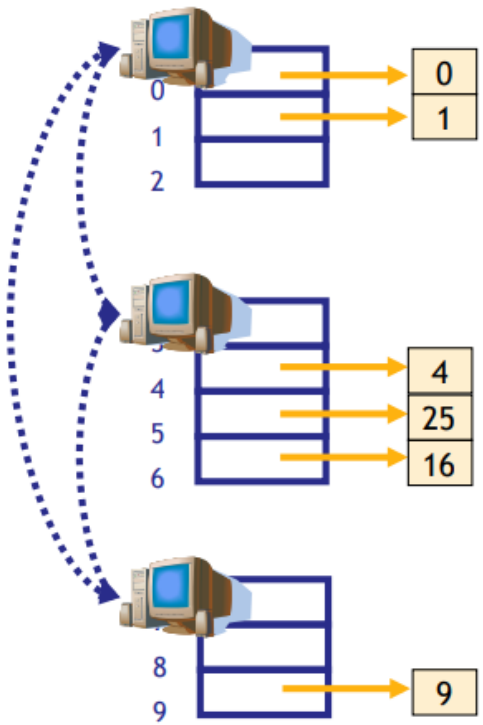
DHT: Idea

- Hash tables are fast for lookups.
- Idea: Distribute hash buckets to peers.
- Result is Distributed Hash Table.
- Need efficient mechanism for finding which peer is responsible for which bucket and routing between them.



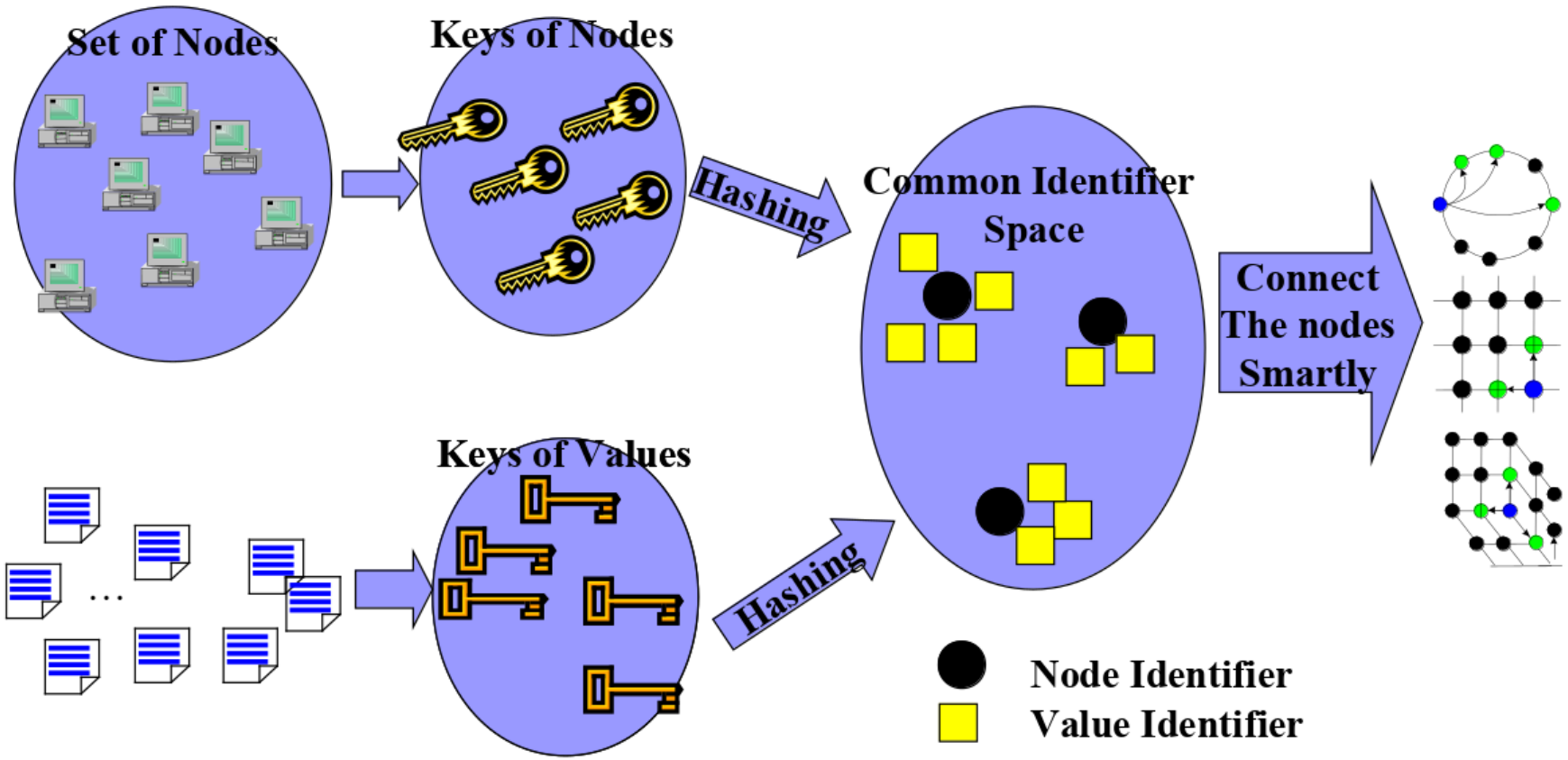
DHT: Principle

- In a DHT, each node is responsible for one or more hash buckets.
  As nodes join and leave, the responsibilities change.
- Nodes communicate among themselves to find the responsible node.
  Scalable Communications make DHTs efficient.
- Hash buckets distributed over nodes.
- Nodes form an overlay network.
  Route messages in overlay to find responsible node.



Structured Overlay Networks/DHTs

Chord, Pastry, Tapestry, CAN, Kademlia, P-Grid, Viceroy



DHT: Overview

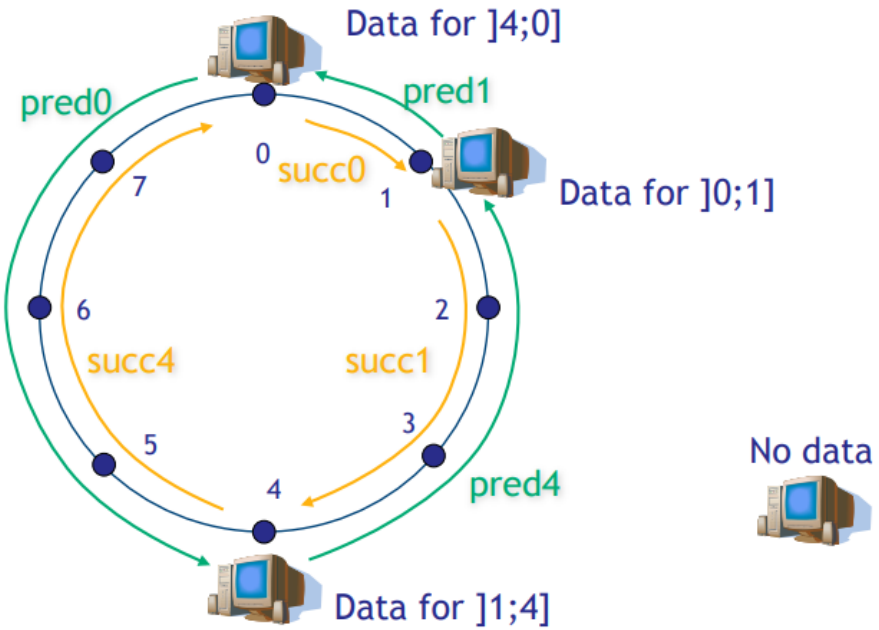
- All DHTs provide the same abstraction
  - Put(key, value)
  - value = Get(key)
- Difference is in overlay routing scheme
  - Chord => ring
  - Kademlia => tree
  - CAN, Tapstry, Pastry ...

Chord: Basics

- Chord use SHA-1 hash function
  - Results in a 160-bit object/node indentifier
  - Same hash function for obejects and nodes
- Node ID hashed from IP address
- Object ID hashed from object name
- SHA-1 gives a 160-bit indentifier space
- organized in a ring which wraps around
  - Nodes keep track of predecessor and successor
  - Node responsible for objects between its predecessor and itself
  - Overlay is ofen called Chord Ring

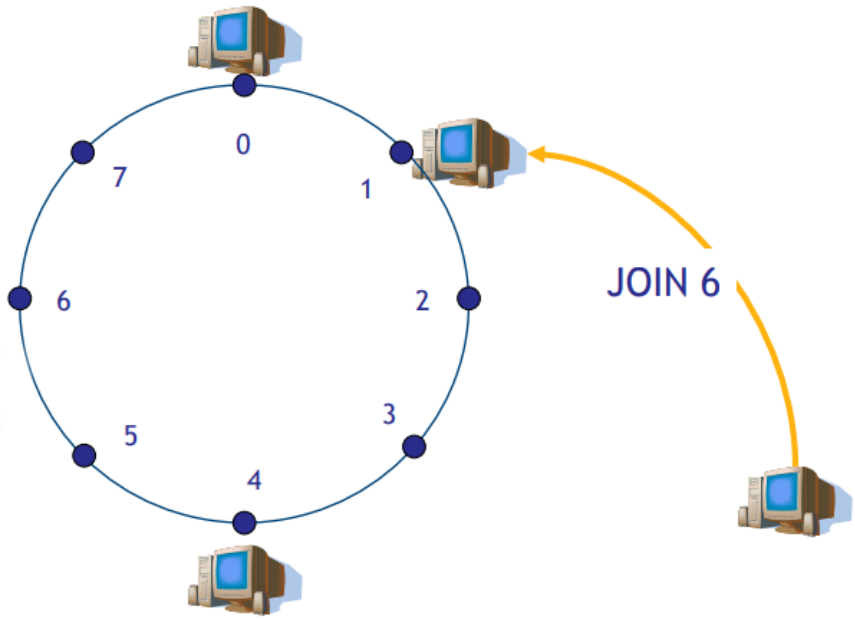
Node Join

- Existing network with nodes on 0,1 and 4
- Hash of new node to join: 6
- Known node in network: Node 1
- Contact Node1

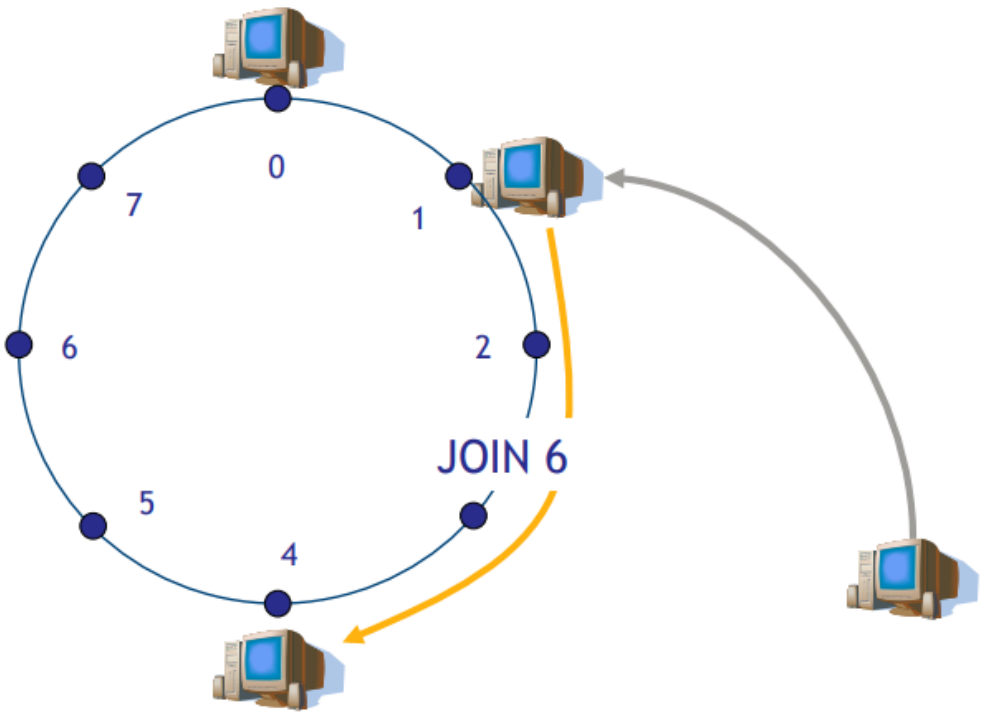


Node Join: Contact Known node

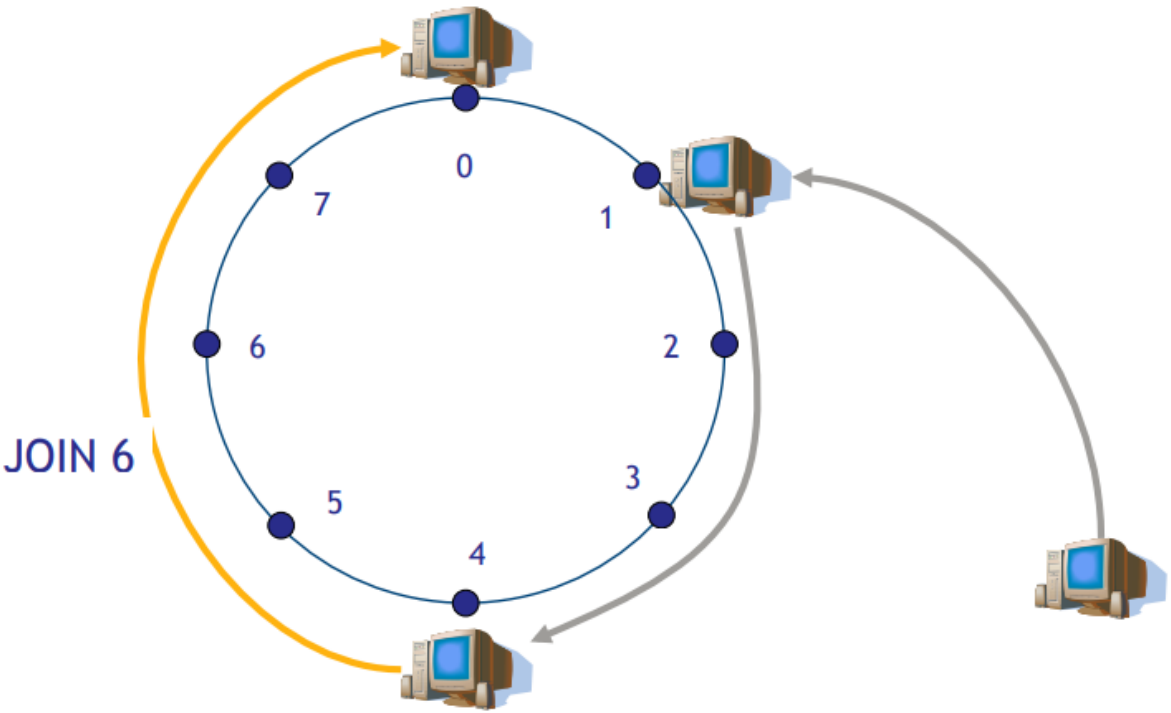
- Existing network with nodes on 0,1 and 4
- Hash of new node to join: 6
- Known node in network: Node 1
- Contact Node1



Node Join: Contact Known node

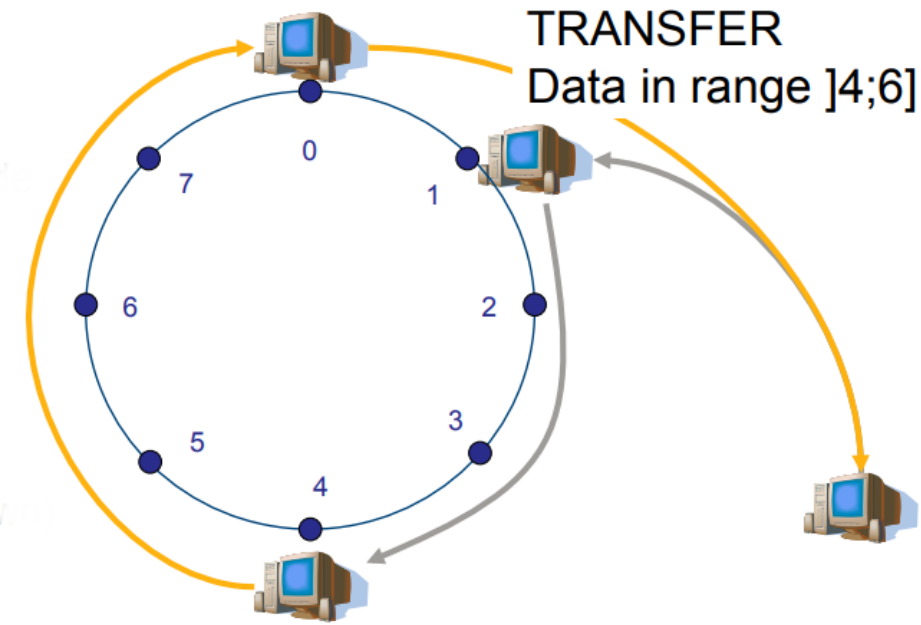


Node Join: Contact Known node

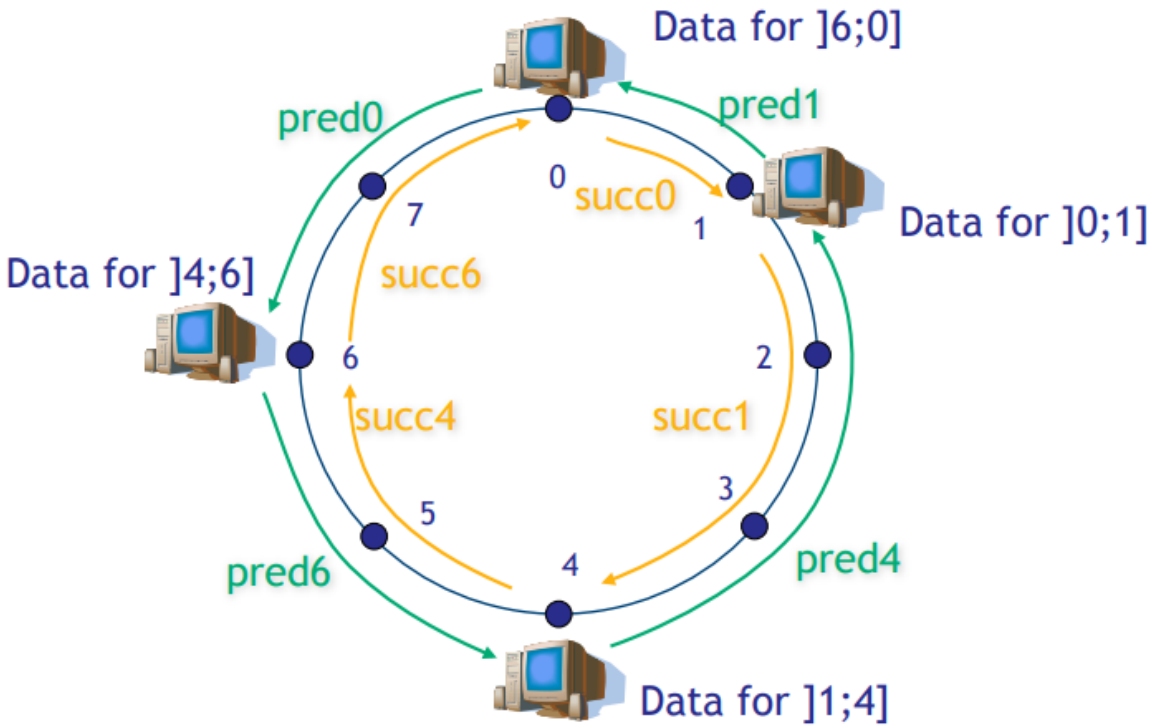


Node Join: Join Successful + Transfer

- Joining is successful
- Old responsible node transfer data that should be in new node
- New node informs Node4 about new successor

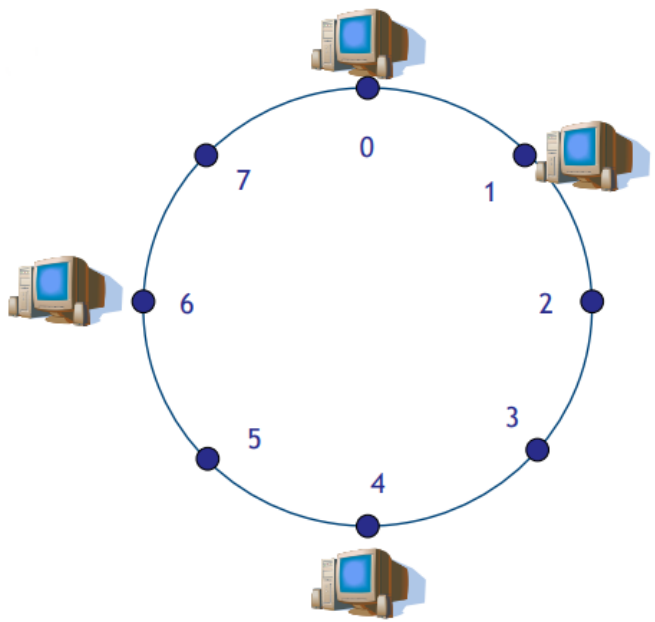


Node Join: All is Done

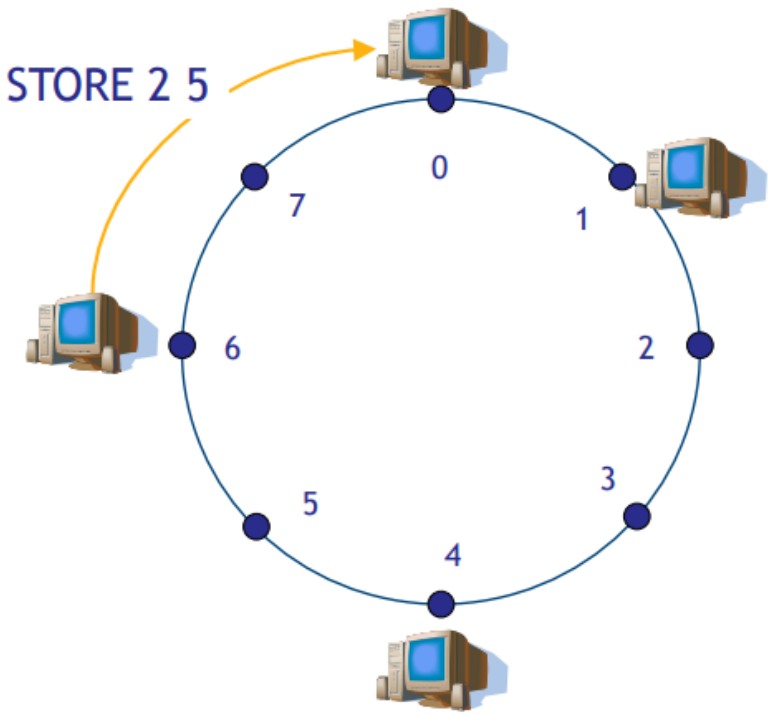


Storing a Value

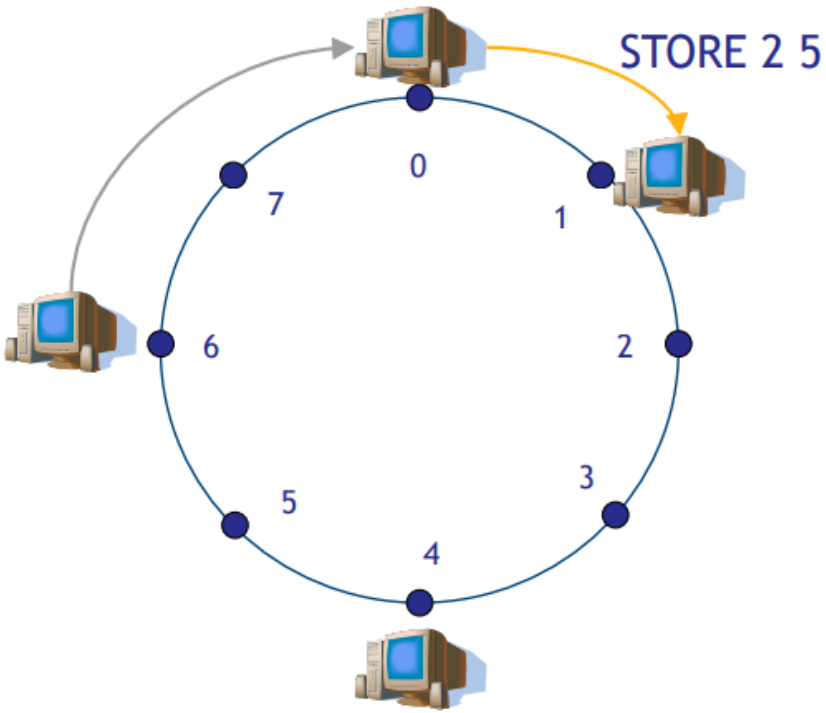
- Node6 wants to strore object with name 'FOO' and value 5
- hash(Foo) = 2



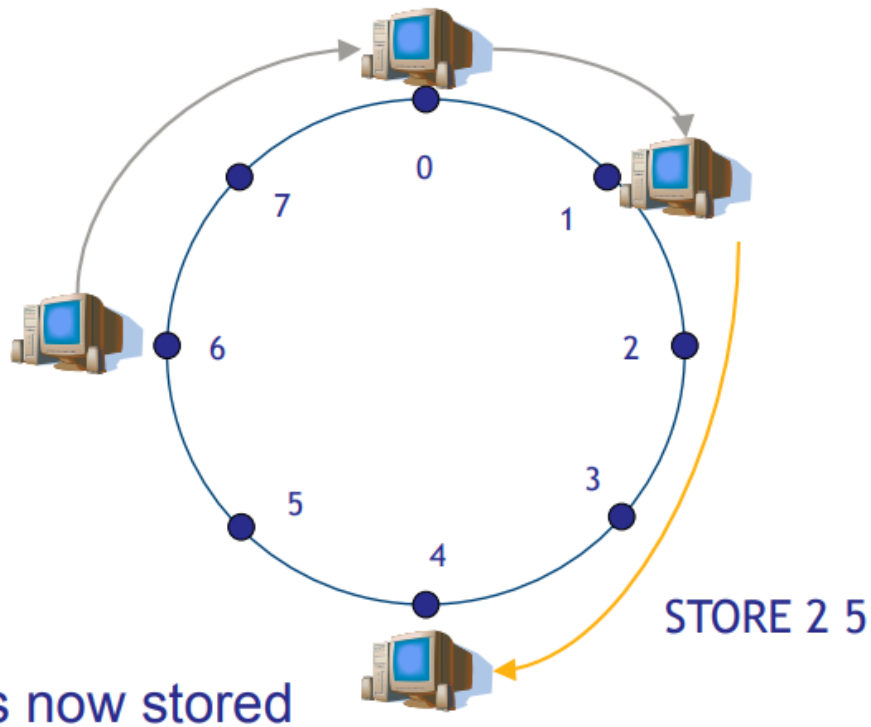
Storing a Value



Storing a Value

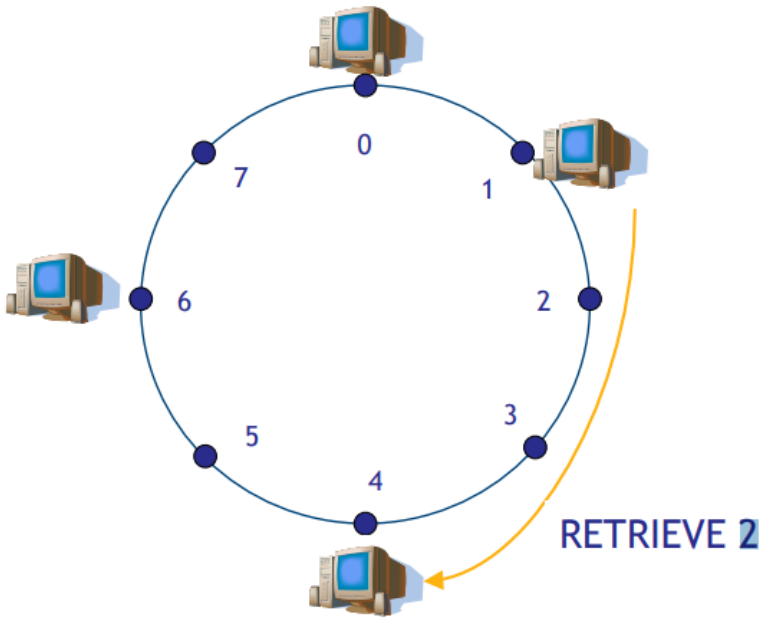


Storing a Value

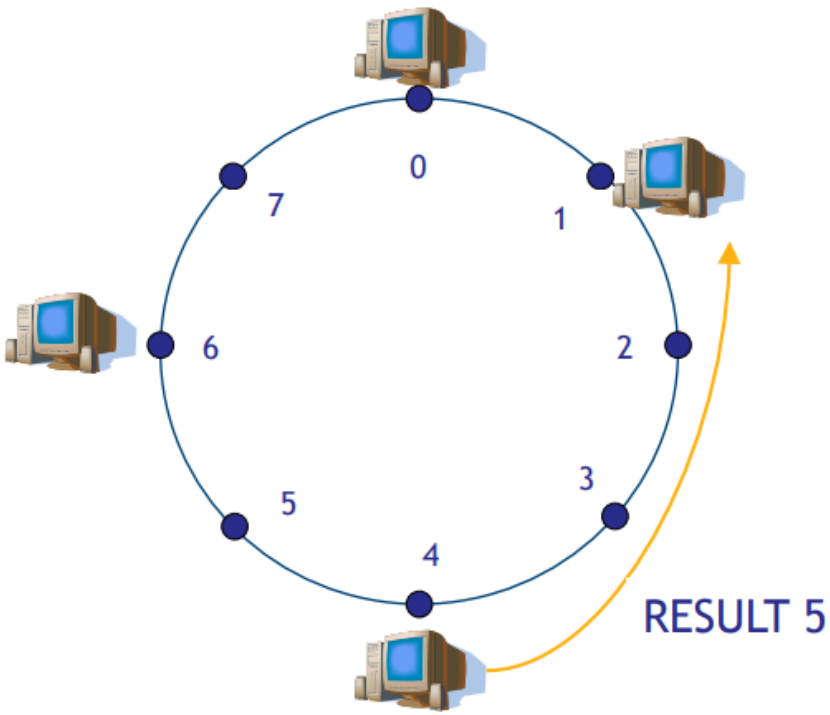


Retrieving a Value

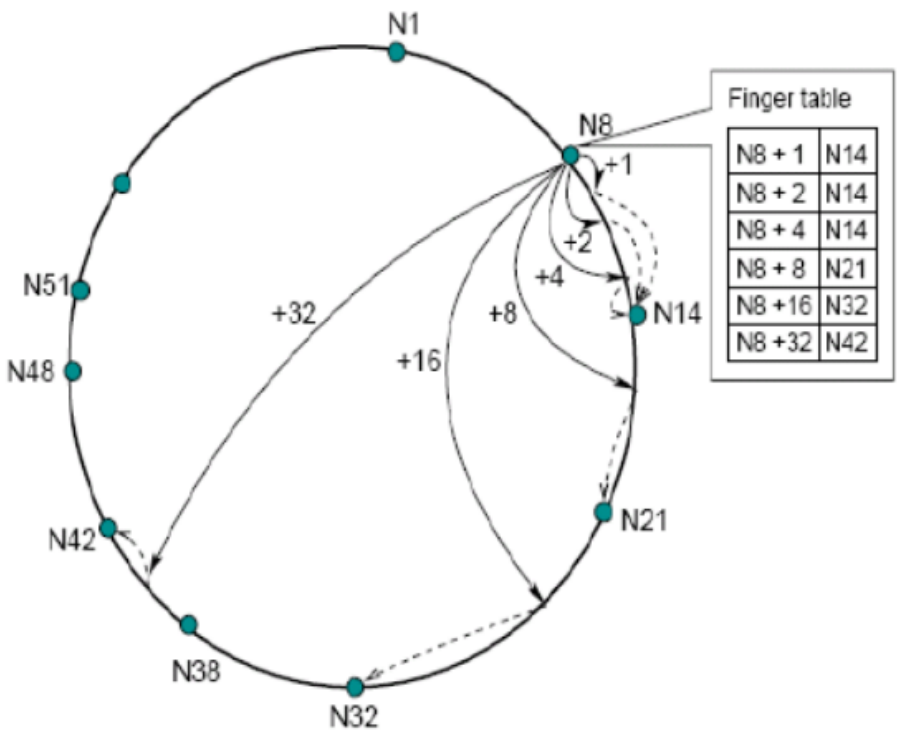
- Node1 wants to get object with name 'FOO'
- hash(Foo) = 2
- Foo is stored on Node4

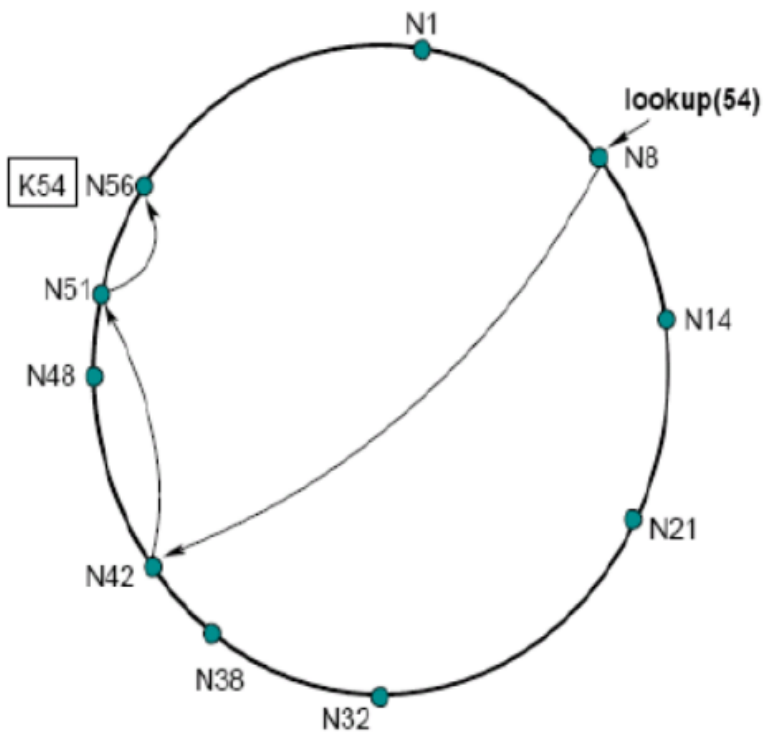


Retrieving a Value



Scalable Key Location: Finger Tables





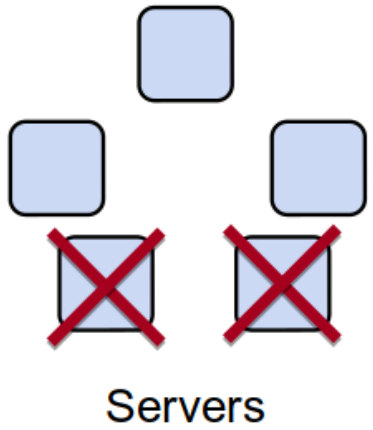
Row i in finger table at node n contains first node s that succeeds n by least 2^{i-1} on the ring. First finger is the successor.

P2P system ends here.

Let's go back to distributed system.

What is Consensus?

- Agreememt on shared state(single system image)
- Recovers from server failure autonomously
  - Minority of servers fail: no problem
  - Majority fail: lose availability, retain consistency

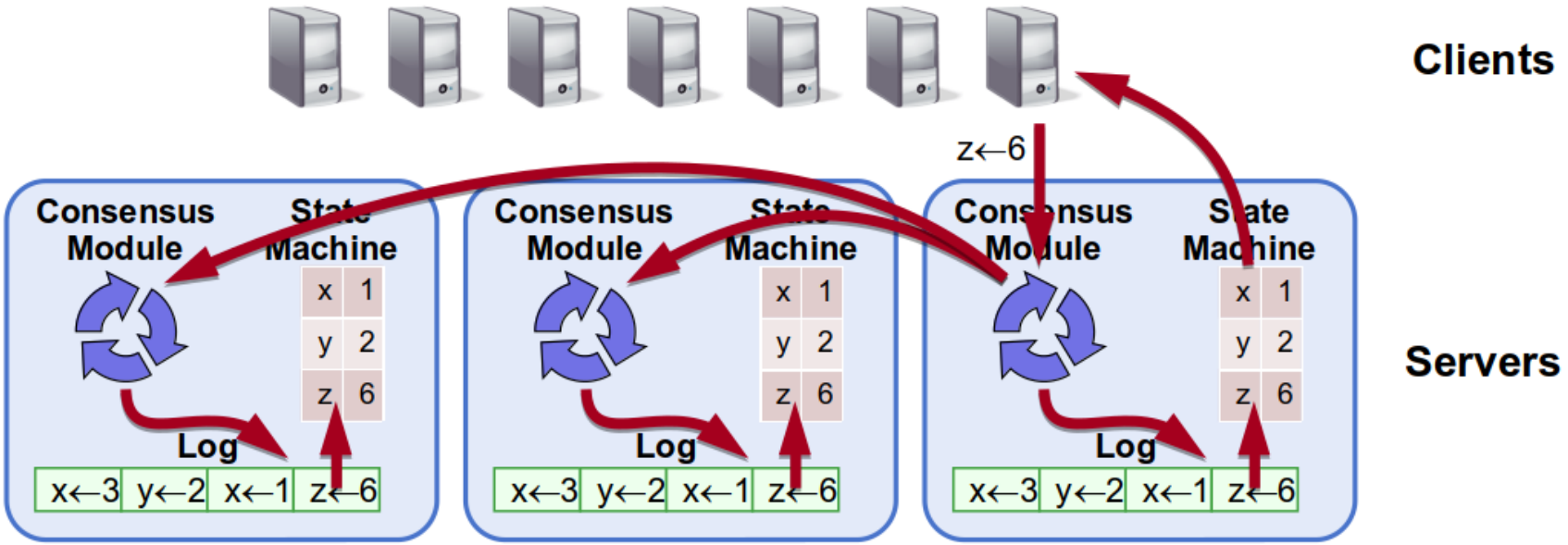


- Key to building consistent storage systems.

To eliminate single point of failure: Replication

- Network partition or server down
- Consensus
  - Allows collection of machines to work as coherent group
  - Continuous service, even if some machines fail
- A concensus algorithm(built-in or library)
  - Paxos(1990) has dominated discussion for 25 years, hard for engineer.
  - Raft(2013) is easier to understand.
  - ...
- A concensus service
  - Google Chubby
  - Apache ZooKeeper
  - ...

Replicated State Machine(RSM)



- Replicated log => All servers execute same commands in same order.
- Consensus module ensures proper log replication.
- System makes progress as long as any majority of servers are up.
- Failure model: fail-stop(not Byzantine), delayed/lost messages.

Raft Overview

- Leader election
  - Select one of servers to act as cluster leader
  - Detect crashes, choose new leader
- Log replication
  - Leader takes commands from clients, appends them to its log.
  - Leader replicates its log to other servers(overwriting inconsistencies)
- Safety: Only a server with an up-to-date log can become leader.

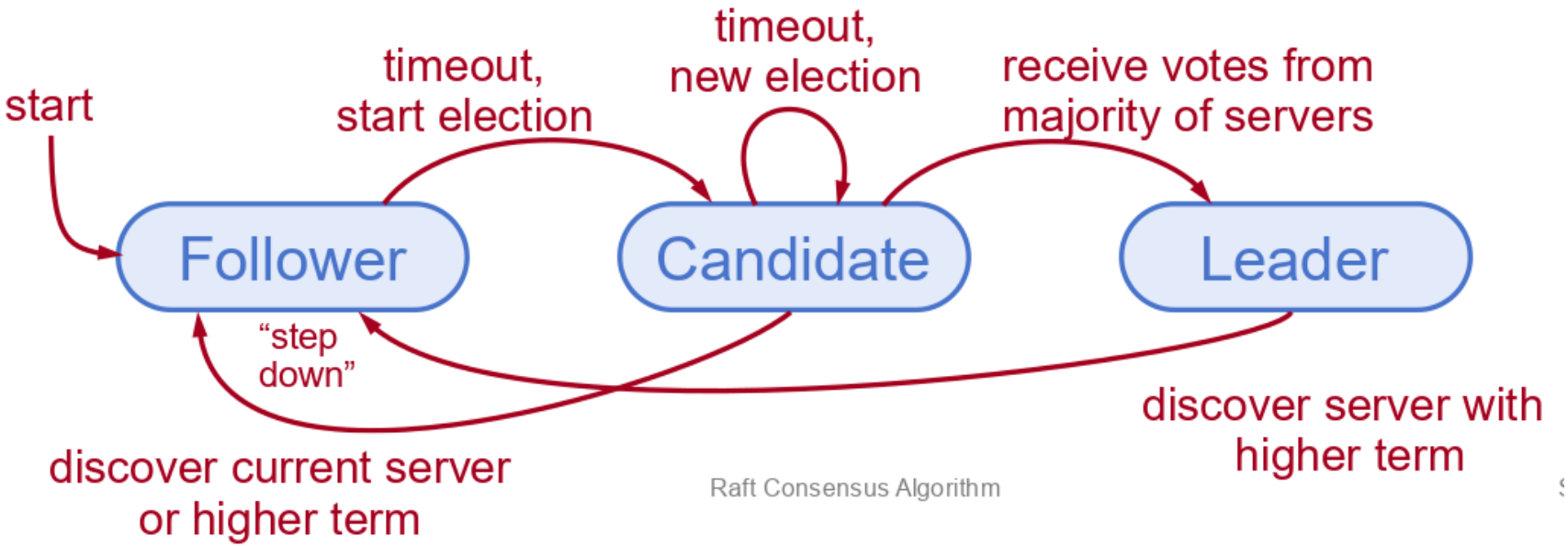
Raft Visualization

Core Raft Overview

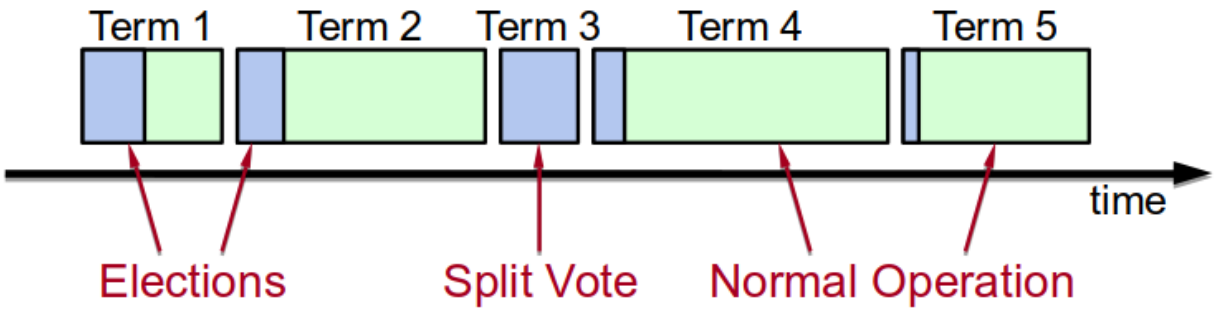
- Leader election
  - Heartbeats and timeouts to detect crashes
  - Randomized timeouts to avoid split votes
  - Majority voting to guarantee at most one leader per term
- Log replication
  - Leader takes commands from clients, appends them to its log.
  - Leader replicates its log to other servers(overwriting inconsistencies)
- Safety
  - Only elect leaders with all committed entries in their logs.
  - New leader defers committing entries from prior terms.

Server States

- **At any given time, each server is either:**
  - Leader: handles all client interactions, log replication
  - Followers: completely passive(issue no RPCs, responds to incoming RPCs)
  - Candidate: used to elect a new leader
- **Normal operation: 1 leader, N-1 followers**



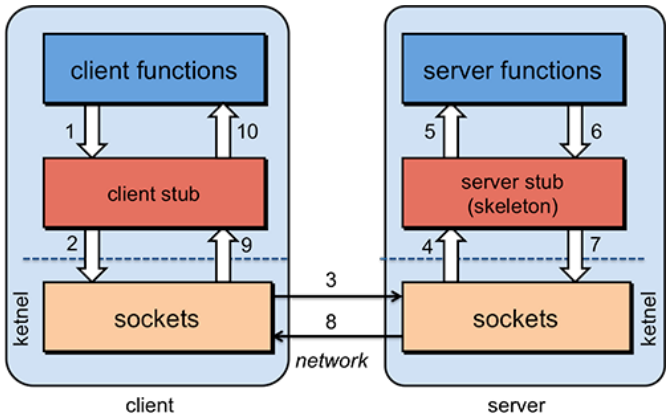
Terms



- **Time divided into terms:**
  - Election
  - Normal operation under a single leader
- **At most 1 leader per term**
- **Some terms have no leader(failed election)**
- **Each server maintains current term value**
- **Key role of terms: identify obsolete information**

Raft Remote Procedure Calls(RPCs)

- **Raft servers using RPCs to communicate**
  - RPC is a key piece of distribute system machinery
  - RPC ideally make net communication look just like function call



- **RequestVote RPC**
  - Invoked by candidate to gather votes
- **AppendEntries RPC**
  - Invoked by leader to replicate log entries
  - Also used as heartbeats

RequestVote RPC

RequestVote RPC

Invoked by candidates to gather votes (§5.2).

Arguments:

term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry (§5.4)
lastLogTerm	term of candidate's last log entry (§5.4)

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§5.2, §5.4)

AppendEntries RPC

AppendEntries RPC

Invoked by leader to replicate log entries (§5.3); also used as heartbeat (§5.2).

Arguments:

term	leader's term
leaderId	so follower can redirect clients
prevLogIndex	index of log entry immediately preceding new ones
prevLogTerm	term of prevLogIndex entry
entries[]	log entries to store (empty for heartbeat; may send more than one for efficiency)
leaderCommit	leader's commitIndex

Results:

term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§5.3)
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§5.3)
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

Heartbeats and Timeouts

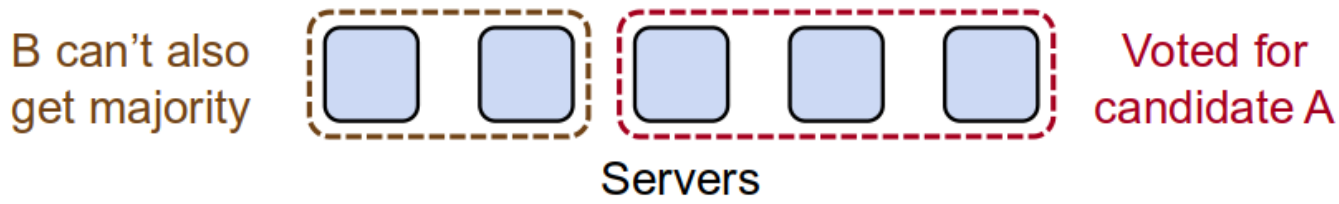
- **Servers start up as followers**
- **Followers expect to receive RPCs from leaders or candidates**
- **Leaders must send heartbeats(empty AppendEntries RPCs) to maintains authority**
- **If election Timeout elapses with no RPCs:**
  - Follower assumes leader has crashed
  - Follower starts new election
  - Timeouts typically 100-500 ms

Election Basics

- **Increment current term**
- **Change to Candidate state**
- **Vote for self**
- **Send RequestVote RPCs to all other servers, retry until either:**
  - Receive votes from majority of servers:
    - Become leader
    - Send AppendEntries heartbeats to all other servers
  - Receive RPC from valid leader:
    - Return to follower state
  - No-one wins election(election timeout elapses):
    - Increment term, start new election

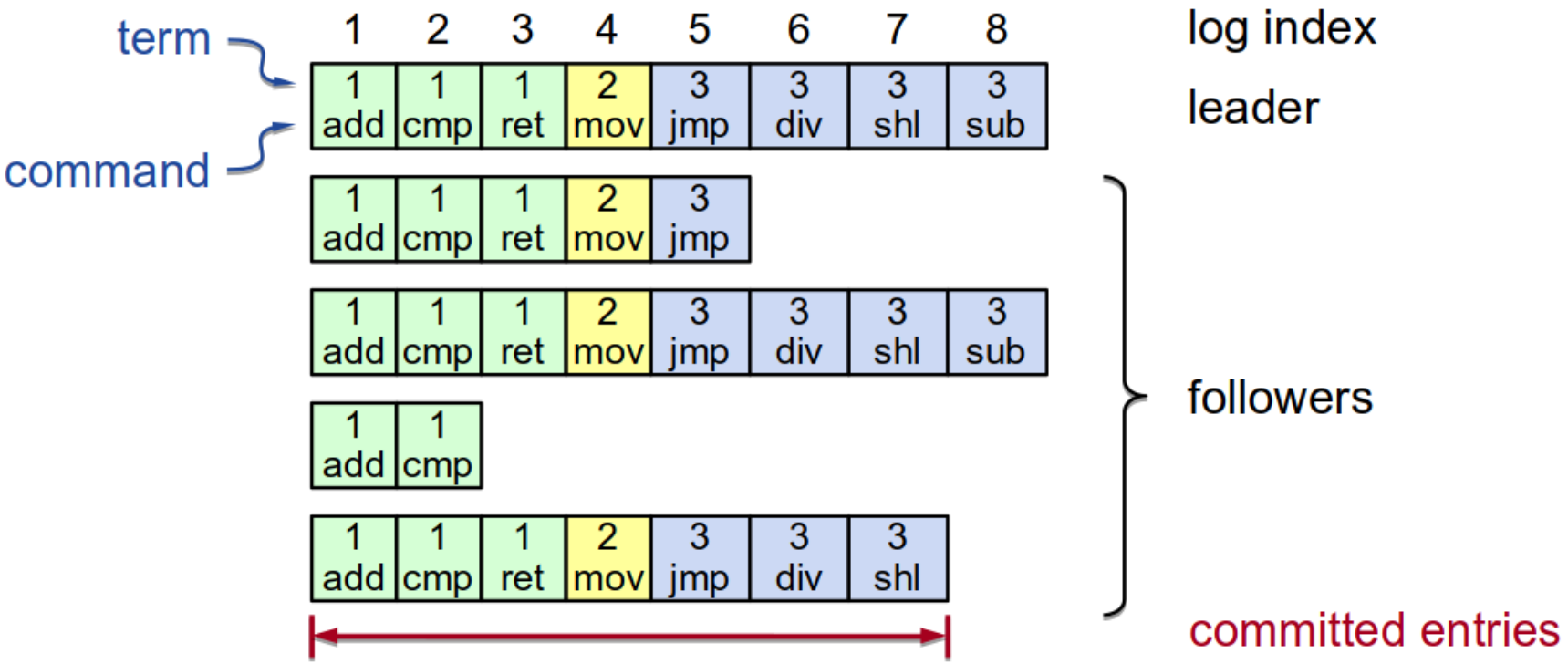
Elections, cont'd

- **Safety: allow at most one winner per term**
  - Each server gives out only one vote per term(persist on disk)
  - Two different candidates can't accumulate majorities in same term



- **Liveness: some candidate must eventually win**
  - Choose election timeouts randomly in [T, 2T]
  - One sever usually times out and wins election before others wake up
  - Works well if T $\gg$ broadcast time

Log Structure



- Log entry = index, term, command
- Log stored on stable storage(disk); survives crashes
- Entry committed if known to be stored on majority of servers
  - Durable, will eventually be executed by state machines

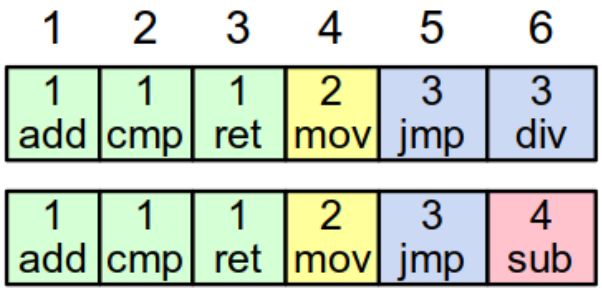
Normal Operation

- **Client sends command to leader**
- **leader appends command to its log**
- **Leader sends AppendEntries RPCs to followers**
- **Once new entry committed:**
  - Leader passes command to its state machine, returns result to client
  - Leader notifies followers of committed entries in subsequent AppendEntries RPCs
  - Followers pass committed commands to their state machines
- **Crashed/slow followers?**
  - Leader retries RPCs until they succeed
- **Performance is optimal in common case:**
  - One successful RPC to any majority of servers

Log Consistency

High level of conherency between logs:

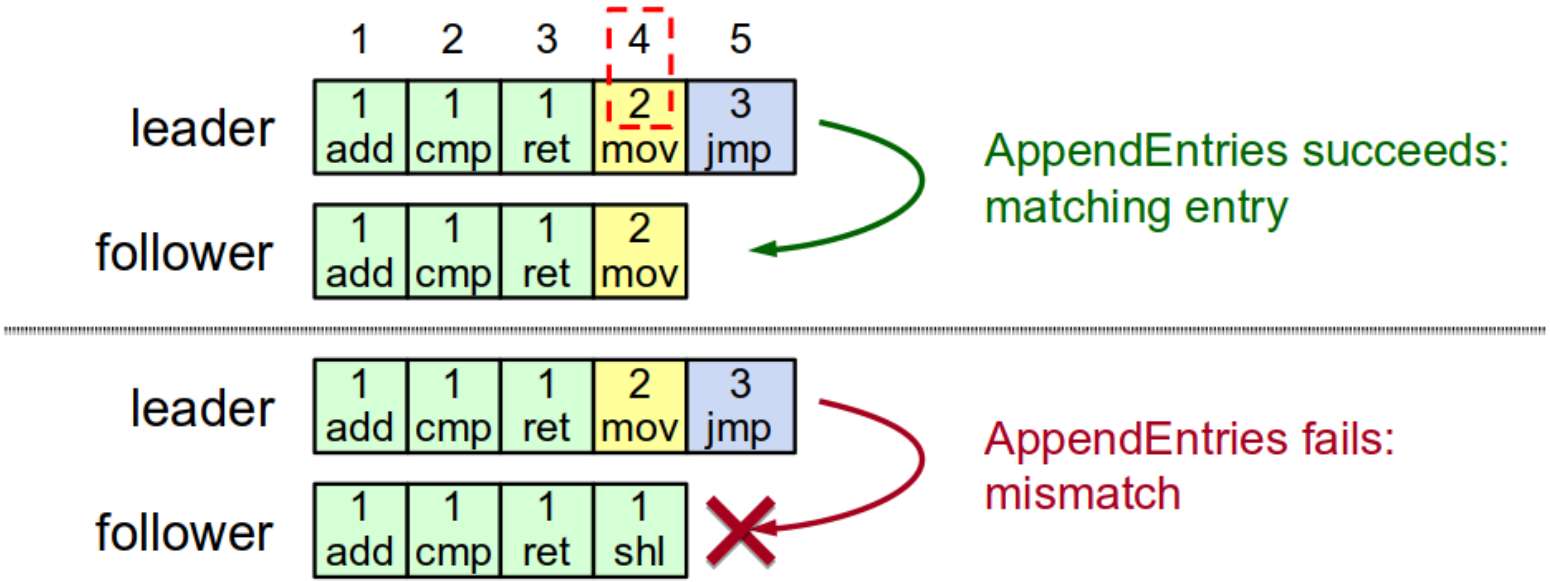
- If log entries on different servers have same index and term:
  - They store the same command
  - The logs are identitcal in all preceding entries



- If a given entry is committed, all preceding entries are also committed

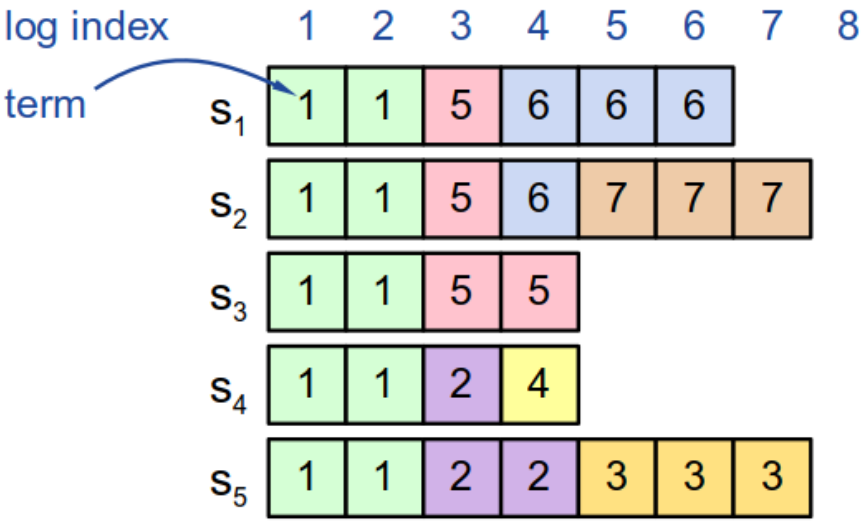
AppendEntries Consistency Check

- Each AppendEntries RPC contains index, term of entry preceding new ones
- Follower must contain matching entry; otherwise it rejects request
- Implements an induction step, ensures coherency



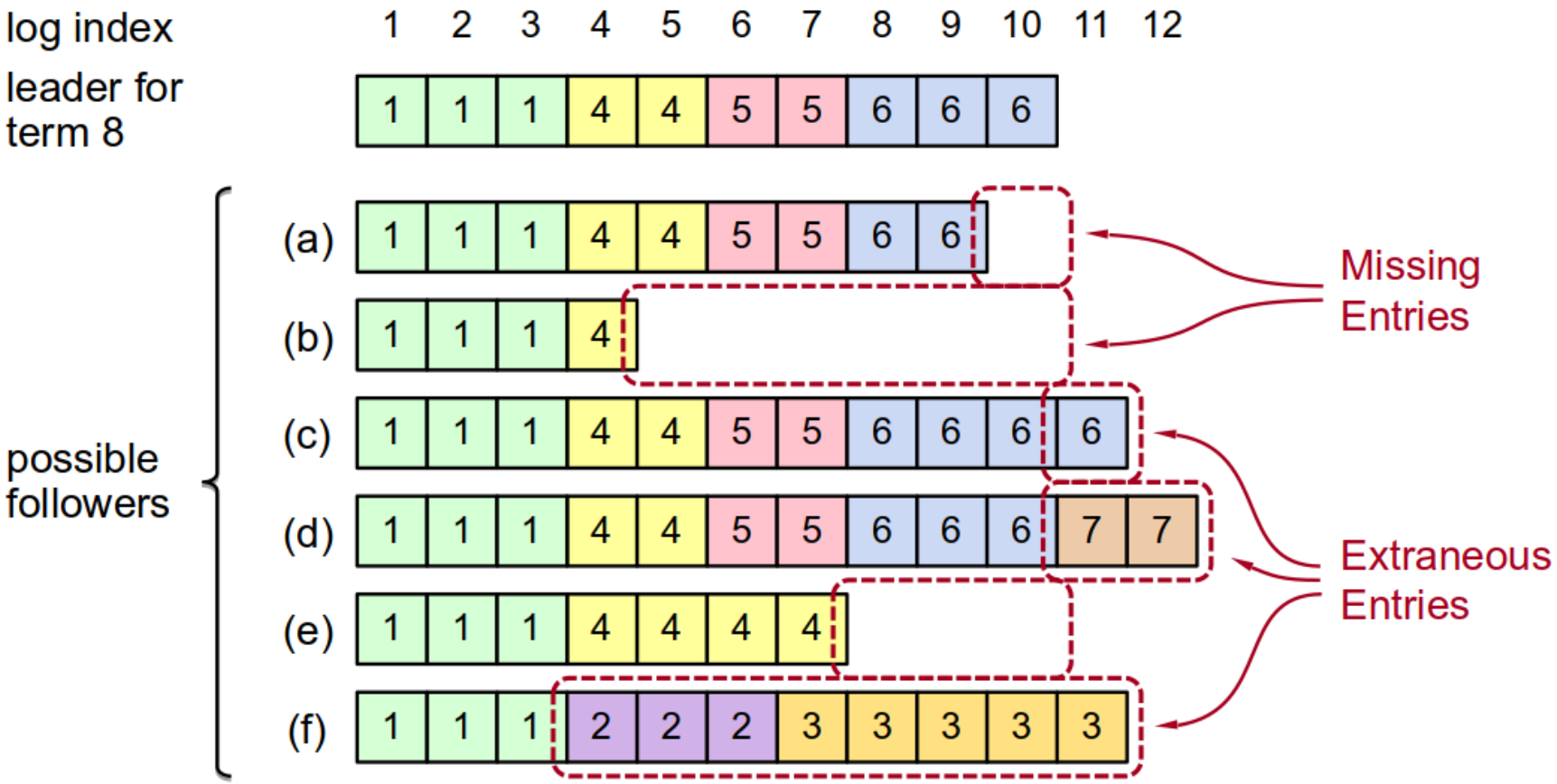
Leader Changes

- At begining of new leader's term:
  - Old leader may have left entries partially replicated
  - No special steps by new leader: just start normal operation
  - Leader's log is the truth
  - Will eventually make follower's logs identical to leader's
  - Multiple crashes can leave many extraneous log entries:



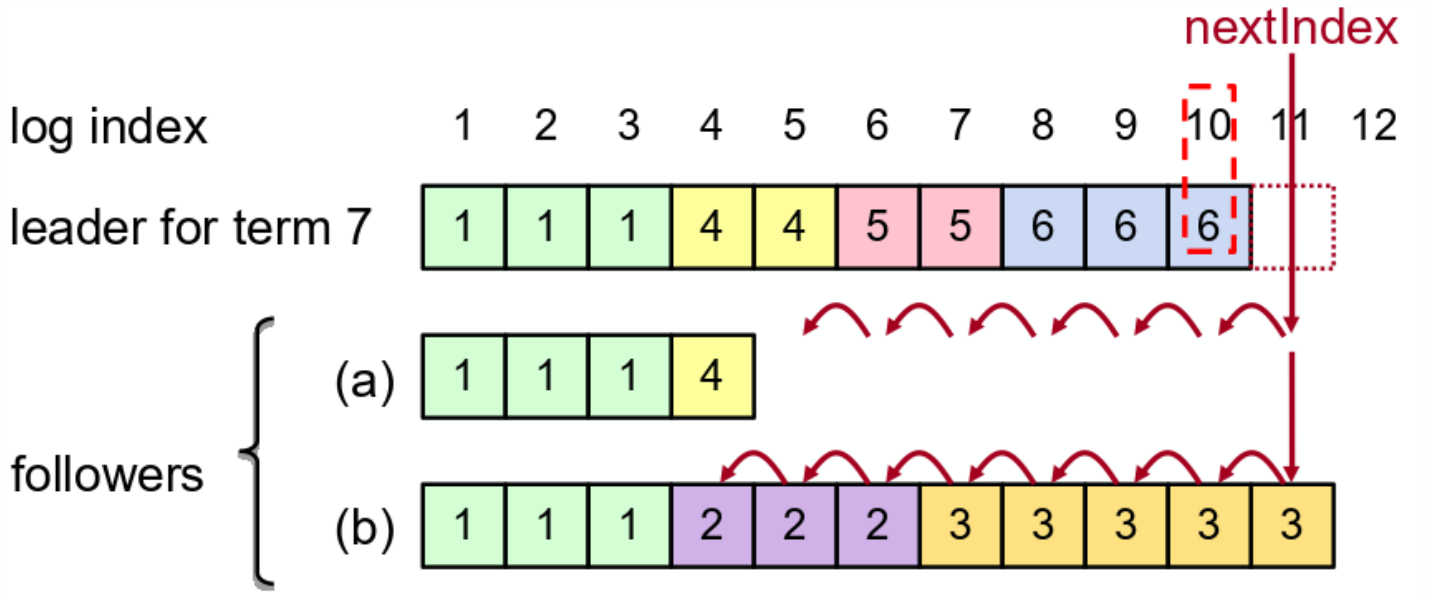
Log Inconsistencies

Leader changes can result in log inconsistencies:



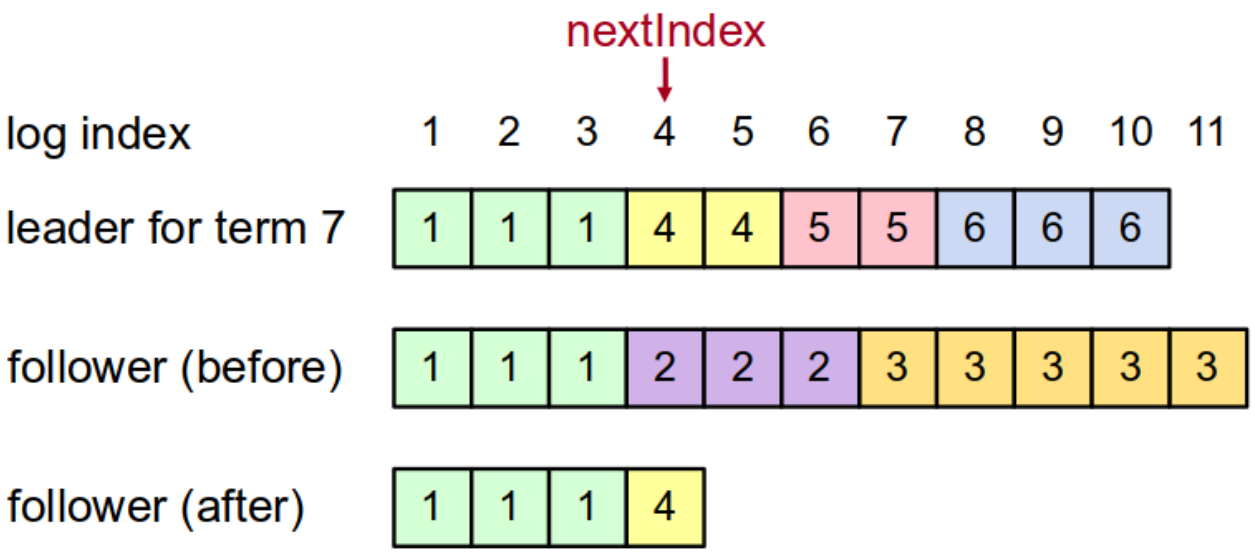
Repairing Follower Logs

- **New leader must make follower logs consistent with its own**
  - Delelte extraneous entries
  - Fill in missing entries
- **Leader keeps nextIndex for each follower:**
  - Index of next log entry to send to that follower
  - Initialized to (1 + leader's last index)
- **When AppendEntries Consistency Check fails, decrement nextIndex and try again:**



Repairing Logs, cont'd

- **When follower overwrites inconsistent entry, it deletes all subsequent entries:**



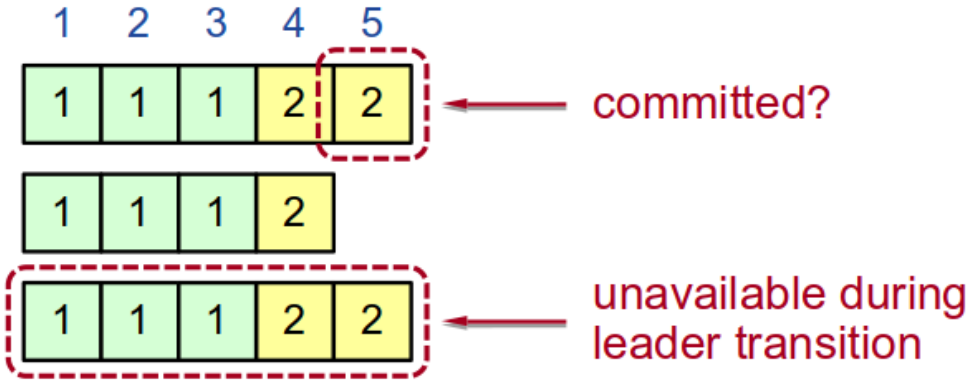
Safety Requirement

Once a log entry has been applied to a state machine, no other state machine must apply a different value for that log entry

- **Raft safety property:**
  - If a leader has decided that a log entry is committed, that entry will be present in the logs of all future leaders.
- **This guarantees the safety requirement**
  - Leaders never overwrite entries in their logs
  - Only entries in the leader's log can be committed
  - Entries must be committed before applying to state machine

Picking the Best Leader

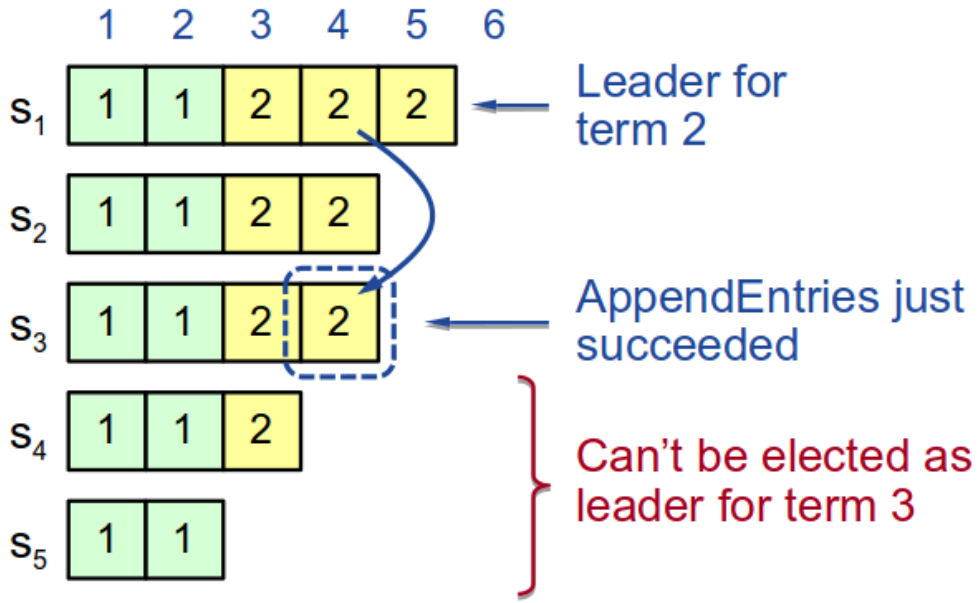
- **Can't tell which entries are committed!**



- **During elections, choose candidate with log most likely to contain all committed entries**
  - Candidates include log info in RequestVote RPCs: (index & term of last log entry)
  - Voting server V denies vote if its log is 'more complete':
    $(lastTerm_V > lastTerm_C) \|$
    $(lastTerm_V == lastTerm_C) \&\& (lastIndex_V > lastIndex_C)$
  - Leader will have 'most complete' log among electing majority

Committing Entry from Current Term

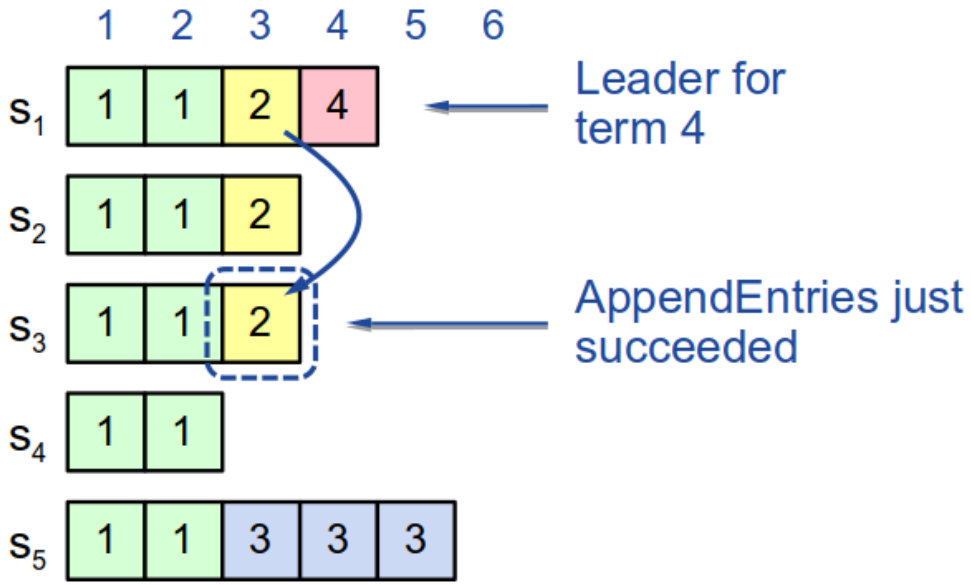
- **Case 1/2: Leader decides entry in current term is committed**



- **Safe: leader for term 3 must contain entry 4**

Committing Entry from Earlier Term

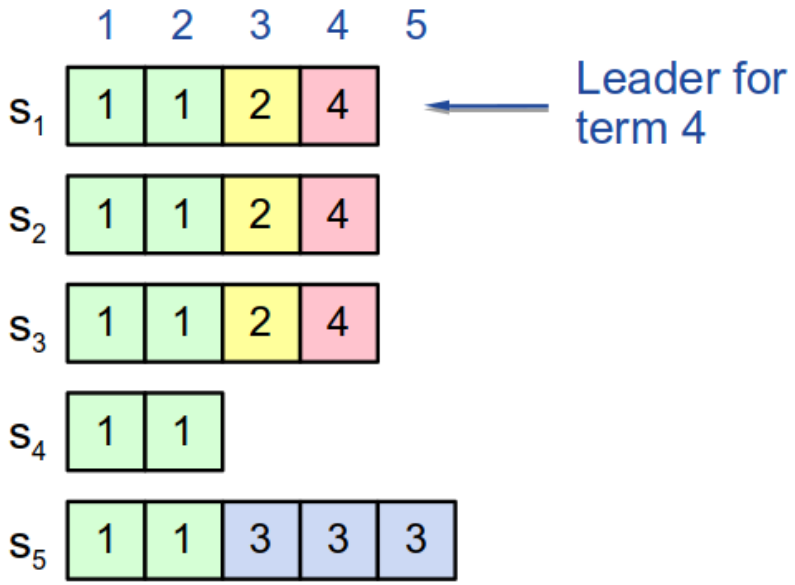
- **Case 2/2: Leader is trying to finish committing entry from an earlier term**



- **Entry 3 not safely committed:**
  - S_5 can be elected as leader for term 5
  - If elected, it will overwrite entry 3 on S_1, S_2, and S_3!

New Commitment Rules

- **For a leader to decide an entry is committed:**
  - Must be stored on a majority of servers
  - At least one new entry from leader's term must also be stored on majority of servers
- **Once entry 4 committed:**
  - S_5 cannot be elected leader for term 5
  - Entries 3 and 4 both safe



Combination of election rules and Commitment rules makes Raft Safe

Neutralizing Old Leaders

- **Deposed leader may not be dead:**
  - Temporarily disconnected from network
  - Other servers elect a new leader
  - Old leader becomes reconnected, attempts to commit log entries
- **Terms used to detect stale leaders(and candidates)**
  - Every RPC contains term of sender
  - If sender's term is older, RPC is rejected, sender reverts to follower and updates its term
  - If receiver's term is older, it reverts to follower, updates its term, then processes RPC normally
- **Election updates terms of majority of servers**
  - Deposed server cannot commit new log entries

Client Protocol

- **Send commands to leader**
  - If leader unknown, contact any server
  - If contacted server not leader, it will redirected to leader
- **Leader does not respond until command has been logged, committed, and executed by leader's state machine**
- **If request times out(e.g., leader crash)**
  - Client reissues command to some other server
  - Eventually redirected to new leader
  - Retry request with new leader

Client Protocol, cont'd

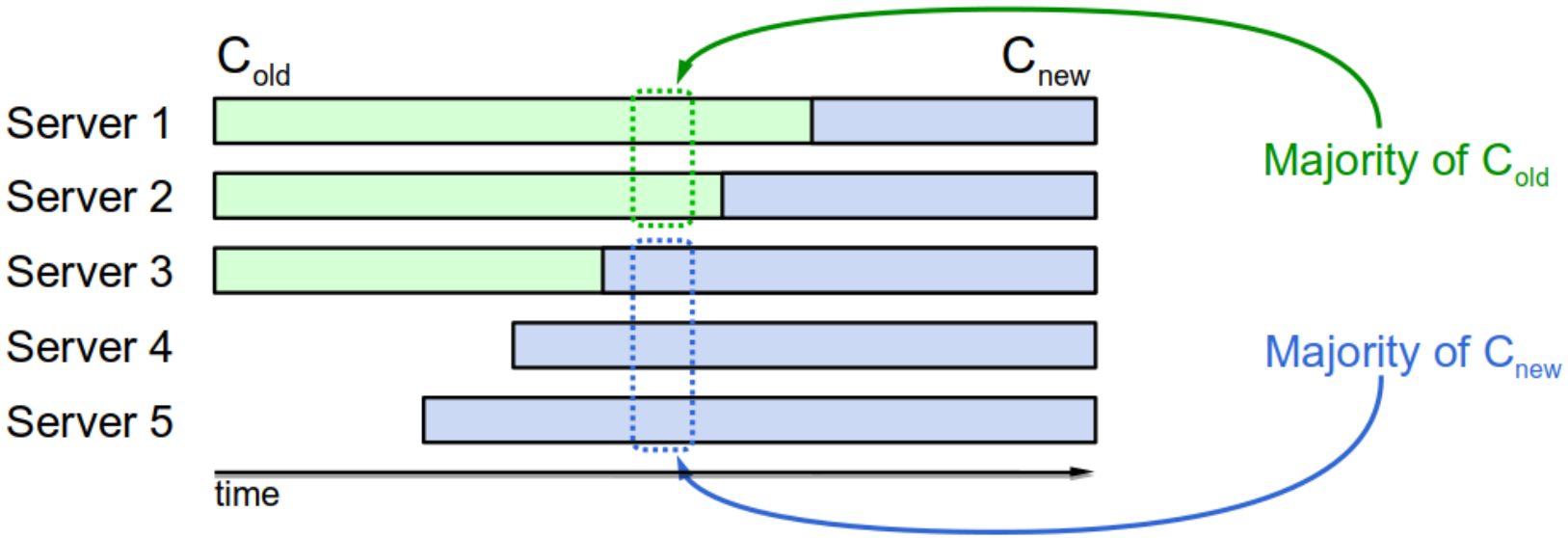
- **What if leader crashes after executing command, but before responding?**
  - Must not execute command twice
- **Solution: client embeds a unique id in each command**
  - Server includes id in log entry
  - Before accepting command, leader checks its log for entry with that id
  - if id found in log, ignore new command, return response from old command
- **Result: exactly-once semantics as long as client doesn't crash**

Configuration Changes

- **System configuration:**
  - ID, address for each server
  - Determines what constitutes a majority
- **Consensus mechanism must support changes in the configuration:**
  - Replace failed machine
  - Change degree of replication

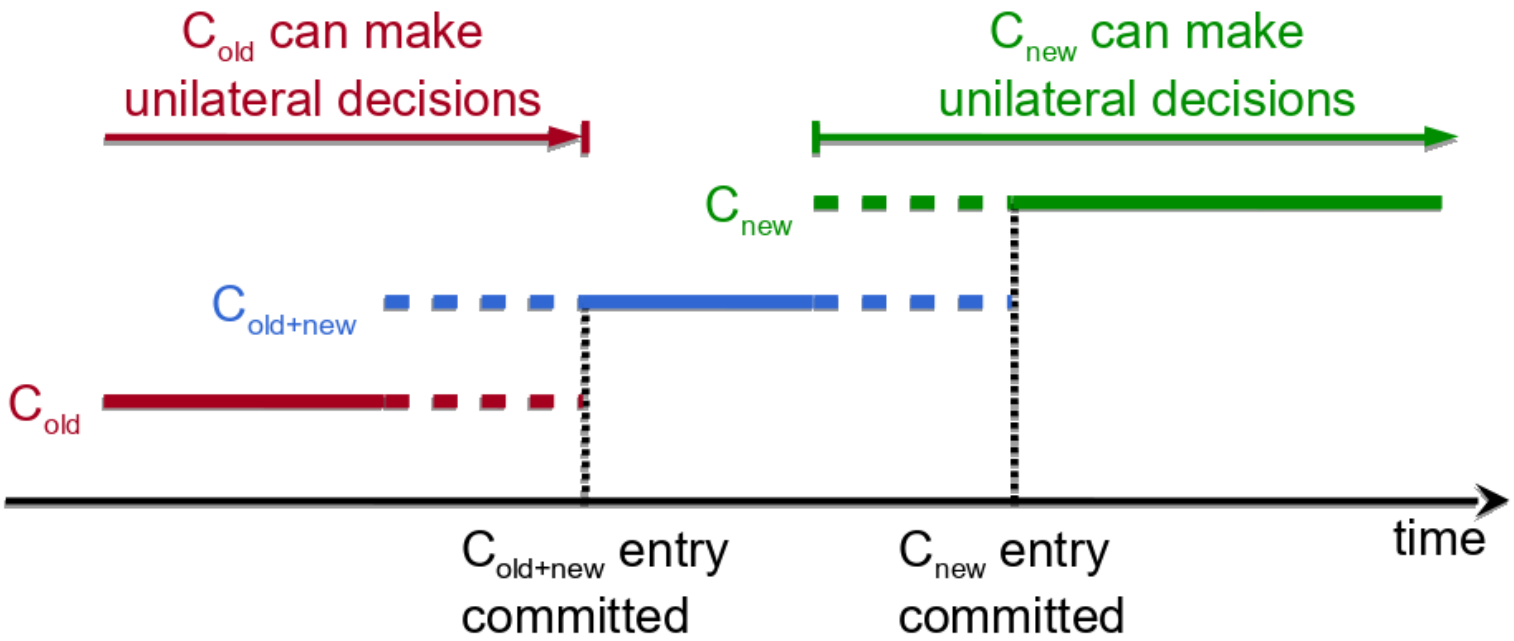
Configuration Changes, cont'd

Cannot switch directly from one configuration to anther: conflicting majorities could rise



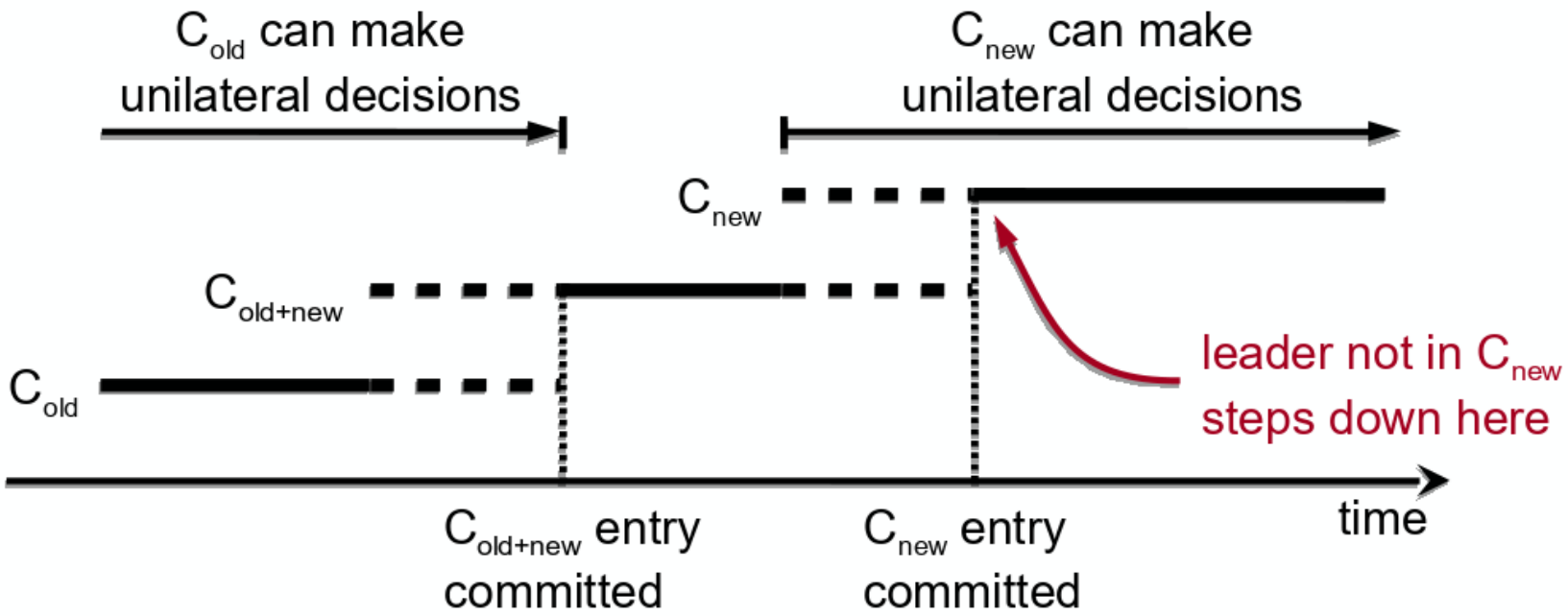
Joint Consensus

- **Raft uses a 2-phase approach**
  - Intermediate phase uses joint consensus(need majority of both old and new configurations for elections, Commitment)
  - Configuration changes is just a log entry; applied immediately on receipt(committed or not)
  - Once joint consensus is committed, begin replicating log entry for final configuration



Joint Consensus, cont'd

- **Additional details:**
  - Any server from either configuration can serve as leader
  - If current leader is not in C_{new}, must step down once C_{new} is committed



Paxos Algorithm

- Leslie Lamport, 1990
- Nearly synonymous with consensus

The Chubby Author

There is only one consensus protocol, and that's Paxos, all other approaches are just broken versions of Paxos.

The Chubby Author

There are significant gaps between the description of the Paxos algorithm and the needs of a real-world system...the final system will based on an unproven protocol.