Peer to Peer Systems in Cloud Computing

Dr. Rajiv Misra
Associate Professor
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 Peer to Peer (P2P) techniques in cloud computing systems.

- We will study some of the widely-deployed P2P systems such as: **Napster, Gnutella, Fasttrack and BitTorrent** and P2P Systems with provable properties such as: **Chord, Pastry and Kelips.**

Need of Peer to Peer Systems

- First distributed systems that seriously focused on **scalability with respect to number of nodes**
- P2P techniques be abundant in cloud computing systems
  - Key-value stores (e.g., Cassandra) use Chord p2p hashing

P2P Systems

Widely-deployed P2P Systems:

1. Napster
2. Gnutella
3. Fasttrack
4. BitTorrent

P2P Systems with Provable Properties:

1. Chord
2. Pastry
3. Kelips

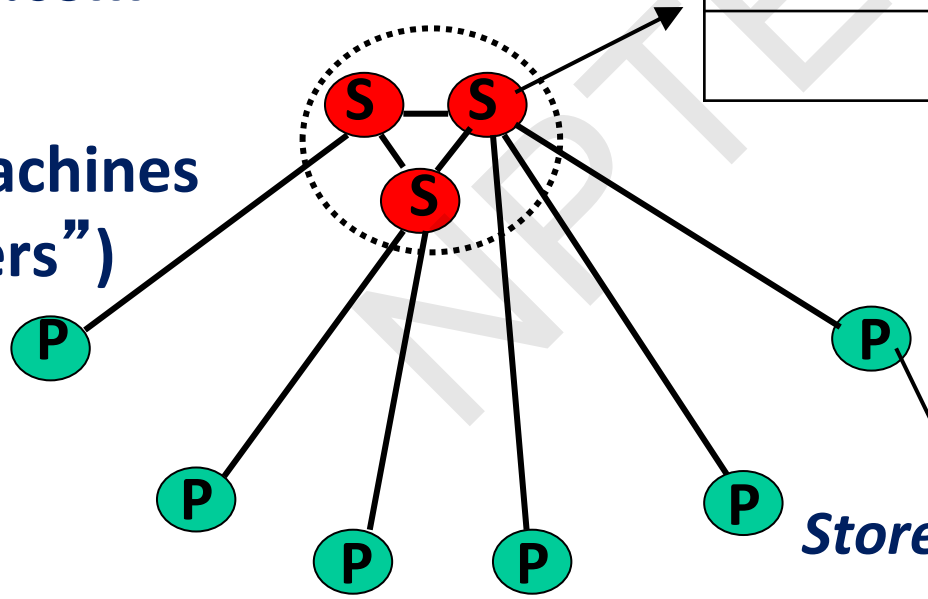
Napster Structure

Store a directory, i.e., filenames with peer pointers

Filename	Info about
Public enemy.mp3	Beatles, @123.34.12.32: 1003

napster.com Servers

Client machines ("Peers")



Store their own files

Napster Structure

Client

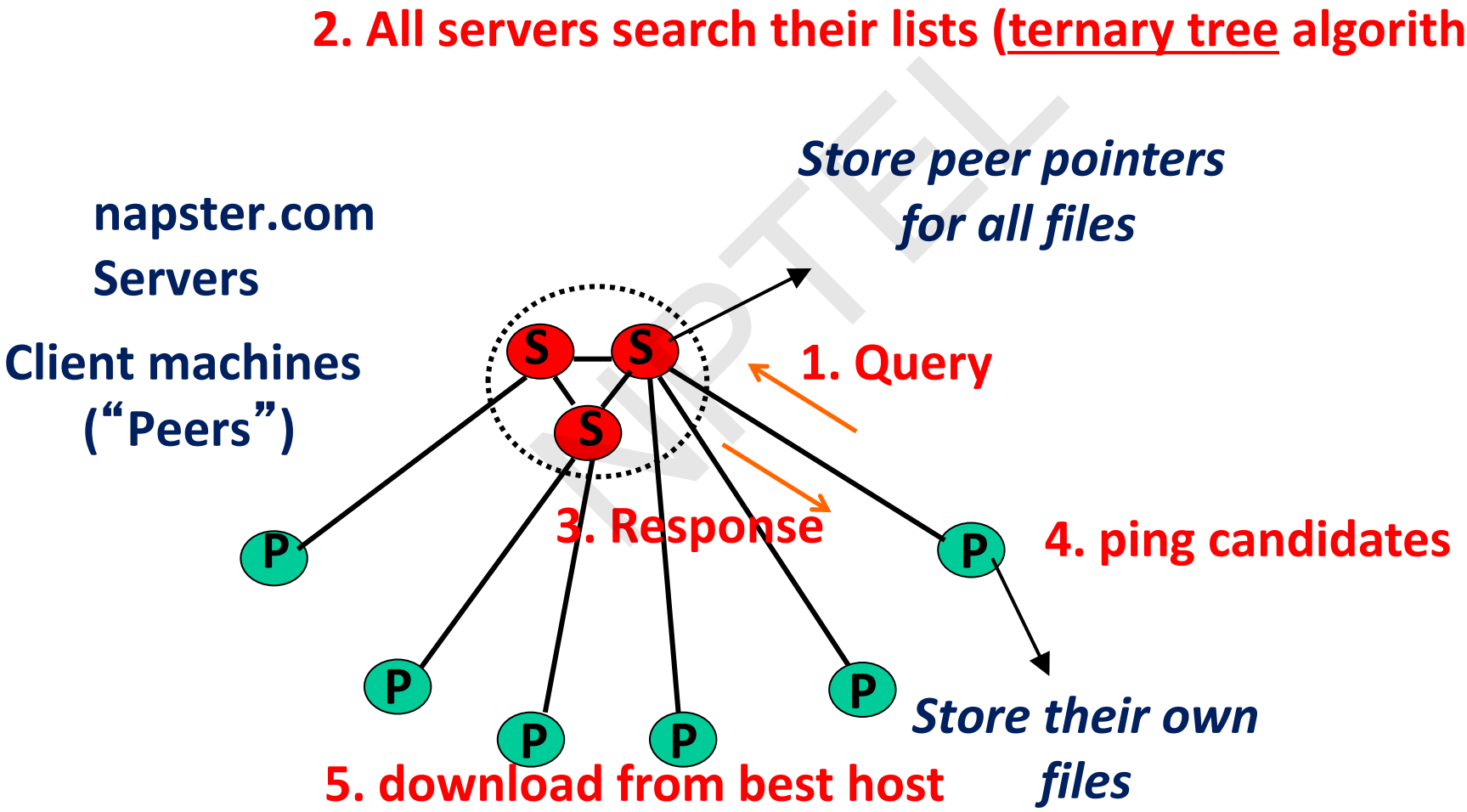
- Connect to a Napster server:
  - Upload list of music files that you want to share
  - Server maintains list of <filename, ip_address, portnum> tuples. **Server stores no files.**

Napster Operations

Client (contd.)

- **Search**
  - Send server keywords to search with
  - (Server searches its list with the keywords)
  - Server returns a list of hosts - <ip_address, portnum> tuples - to client
  - Client pings each host in the list to find transfer rates
  - Client fetches file from best host
- **All communication uses TCP (Transmission Control Protocol)**
  - Reliable and ordered networking protocol

Napster Search



Nodes Joining a P2P system

- **Can be used for any p2p system**

  - Send an http request to well-known url for that P2P service.
  - Message routed (after lookup in DNS=Domain Name system) to introducer, a well known server that keeps track of some recently joined nodes in p2p system
  - Introducer initializes new peers' neighbor table

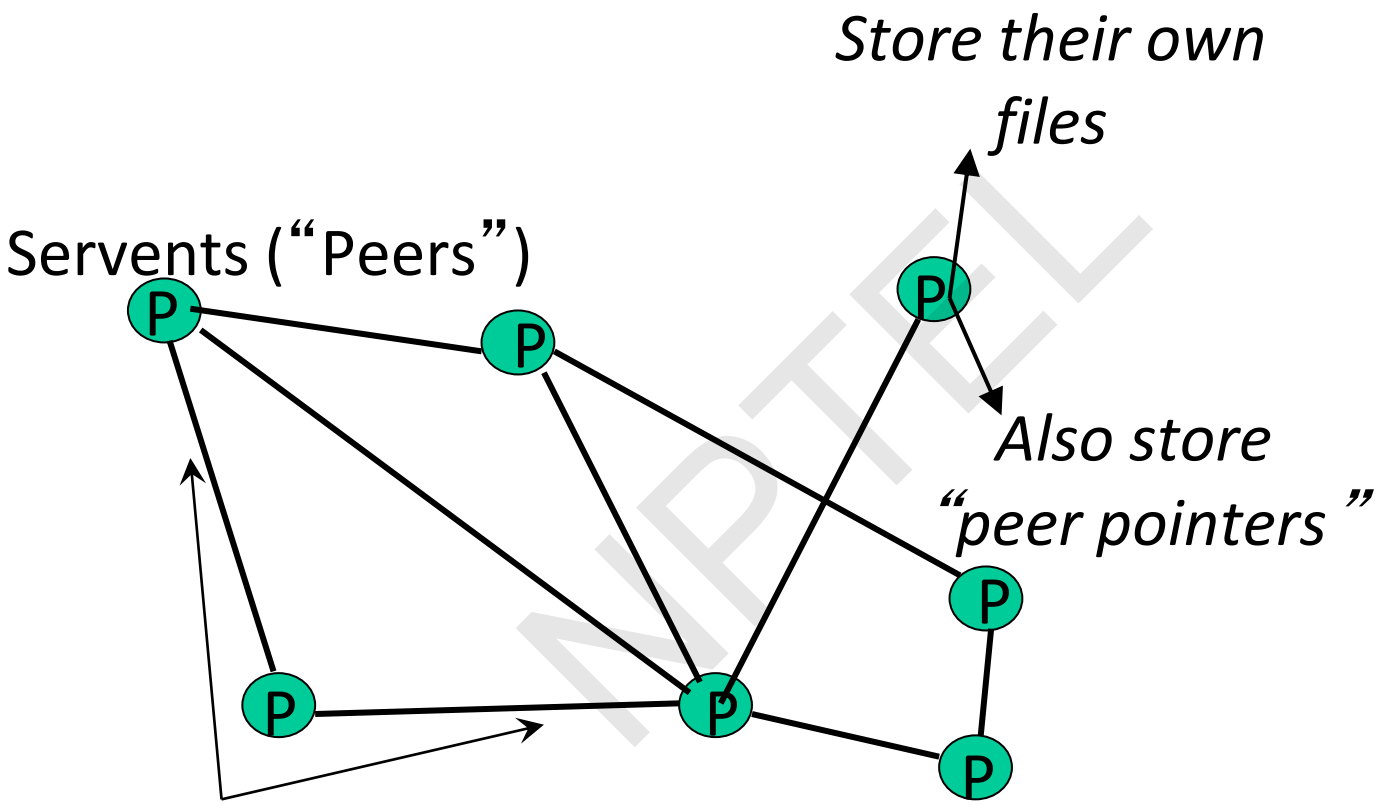
Issues with Napster

- Centralized server a source of congestion
- Centralized server single point of failure
- No security: plaintext messages and passwords
- **napster.com** declared to be responsible for users' copyright violation
  - "Indirect infringement"
  - **Next P2P system: Gnutella**

Gnutella

- Eliminate the servers
- Client machines search and retrieve amongst themselves
- Clients act as servers too, called **servents**
- **Gnutella** (possibly by analogy with the GNU Project) is a large peer-to-peer network. It was the first decentralized peer-to-peer network of its kind.
- [Mar 2000] release by AOL, immediately withdrawn, but 88K users by [Mar 2003]
- Original design underwent several modifications

Gnutella



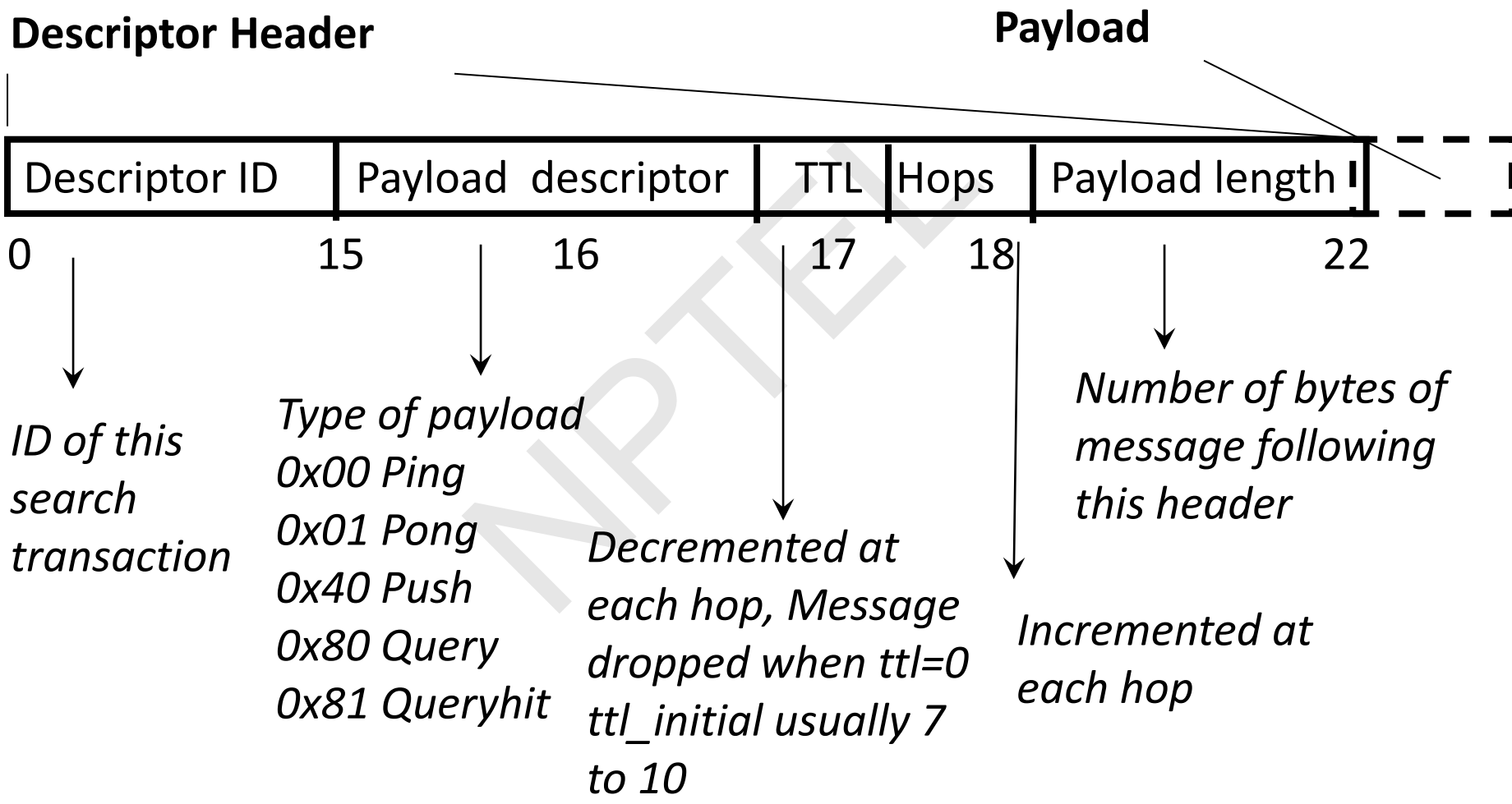
Connected in an **overlay graph**

(== each link is an implicit Internet path)

How do I search for a particular file?

- Gnutella ***routes*** different messages within the overlay graph
- Gnutella protocol has 5 main message types
  1. **Query (search)**
  2. **QueryHit (response to query)**
  3. **Ping (to probe network for other peers)**
  4. **Pong (reply to ping, contains address of another peer)**
  5. **Push (used to initiate file transfer)**
- Into the message structure and protocol
  - All fields except IP address are in little-endian format
  - Ox12345678 stored as 0x78 in lowest address byte, then 0x56 in next higher address, and so on.

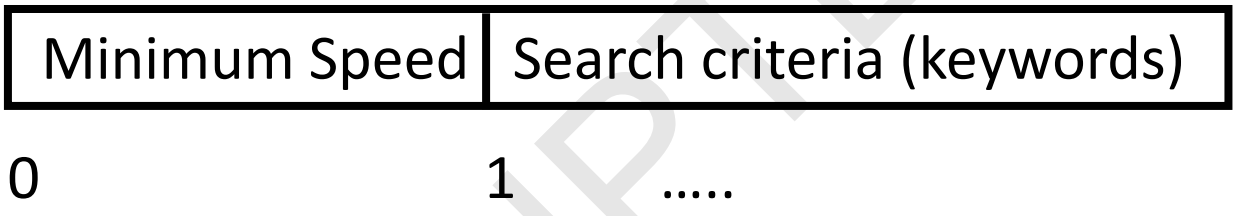
How do I search for a particular file?



Gnutella Message Header Format

How do I search for a particular file?

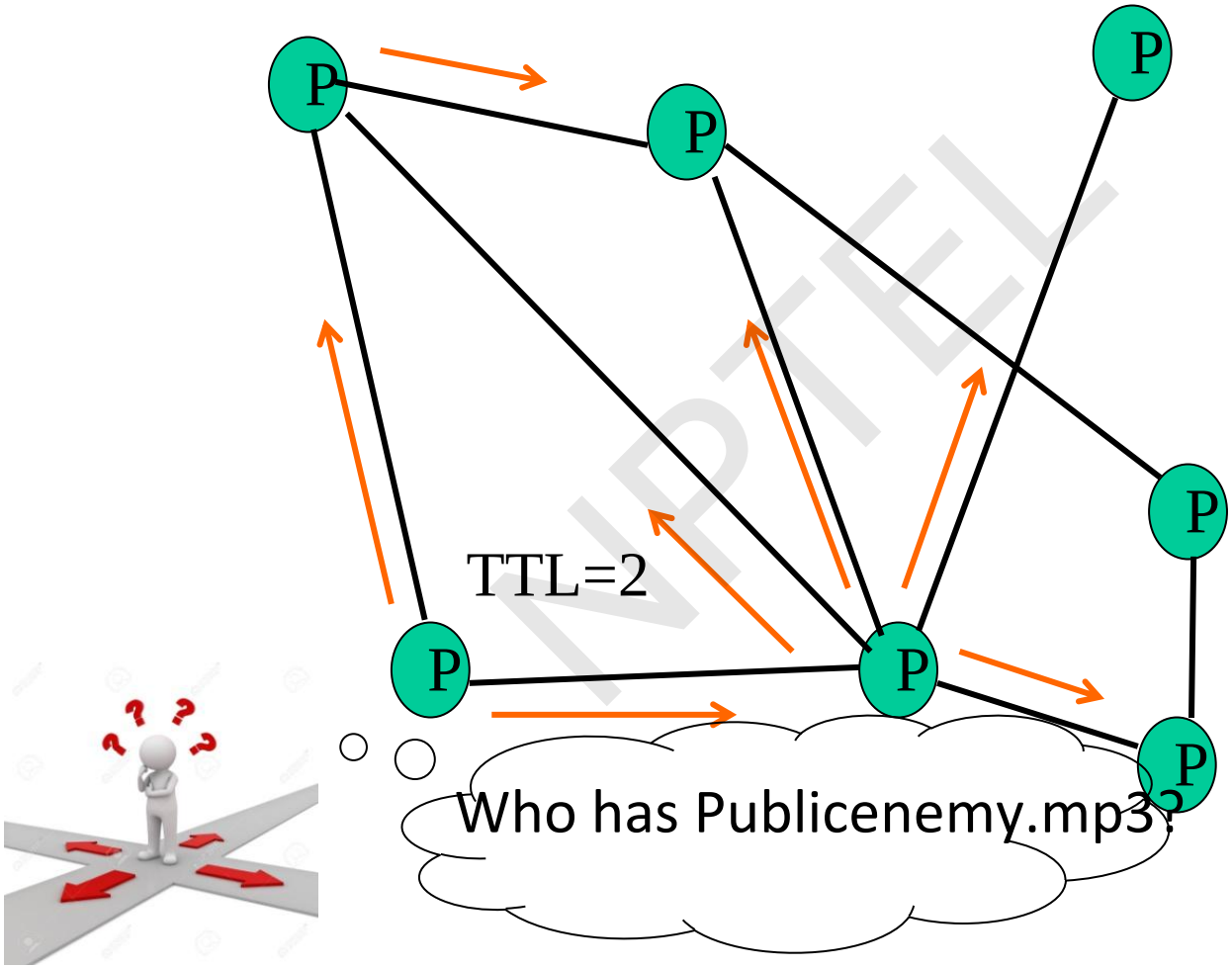
Query (0x80)



Payload Format in Gnutella Query Message

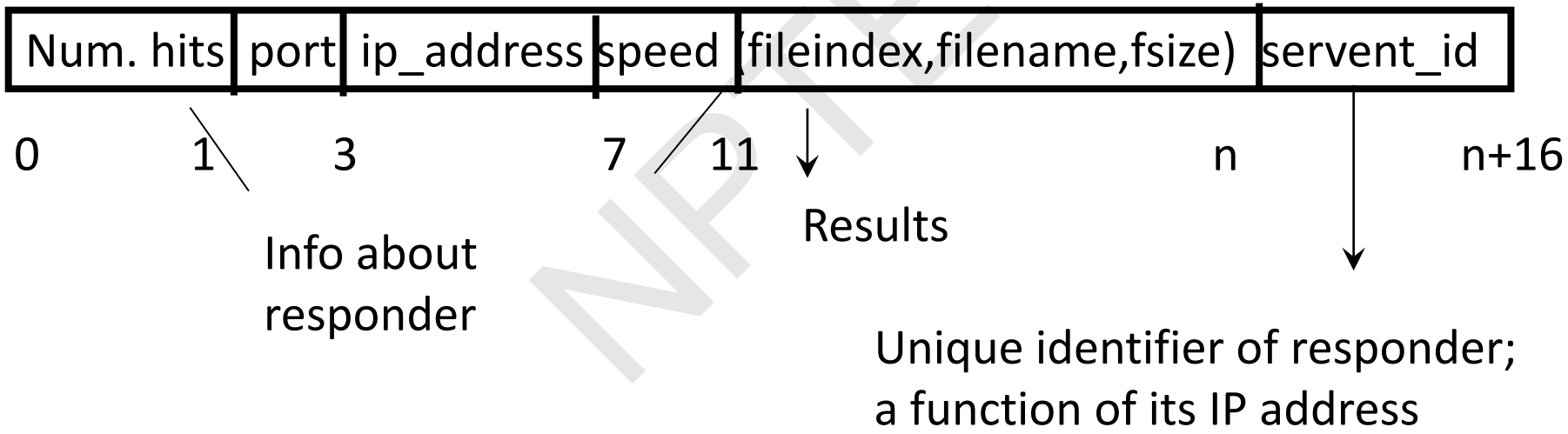
Gnutella Search

Query's flooded out, ttl-restricted, forwarded only once



Gnutella Search

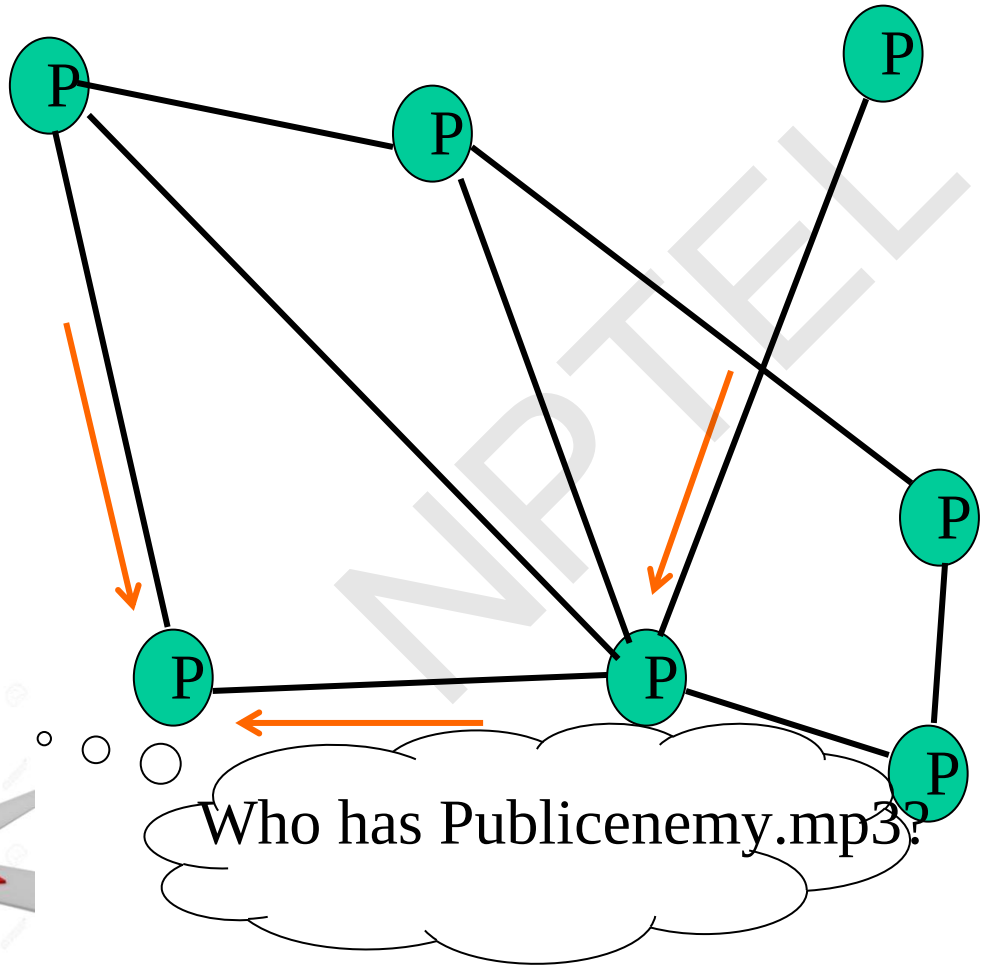
QueryHit (0x81) : successful result to a query



Payload Format in Gnutella QueryHit Message

Gnutella Search

Successful results QueryHit's routed on reverse path



Avoiding excessive traffic

- To avoid duplicate transmissions, each peer maintains a list of recently received messages
- Query forwarded to all neighbors except peer from which received
- Each Query **(identified by DescriptorID)** forwarded only once
- **QueryHit** routed back only to peer from which Query received with same DescriptorID
- Duplicates with same DescriptorID and Payload descriptor (msg type, e.g., Query) are dropped
- QueryHit with DescriptorID for which Query not seen is dropped

After receiving QueryHit messages

- **Requestor chooses "best" QueryHit responder**
  - Initiates HTTP request directly to responder's ip+port

GET /get/<File Index>/<File Name>/HTTP/1.0\r\n

Connection: Keep-Alive\r\n

Range: bytes=0-\r\n

User-Agent: Gnutella\r\n

\r\n

- **Responder then replies with file packets after this message:**

HTTP 200 OK\r\n

Server: Gnutella\r\n

Content-type:application/binary\r\n

Content-length: 1024 \r\n

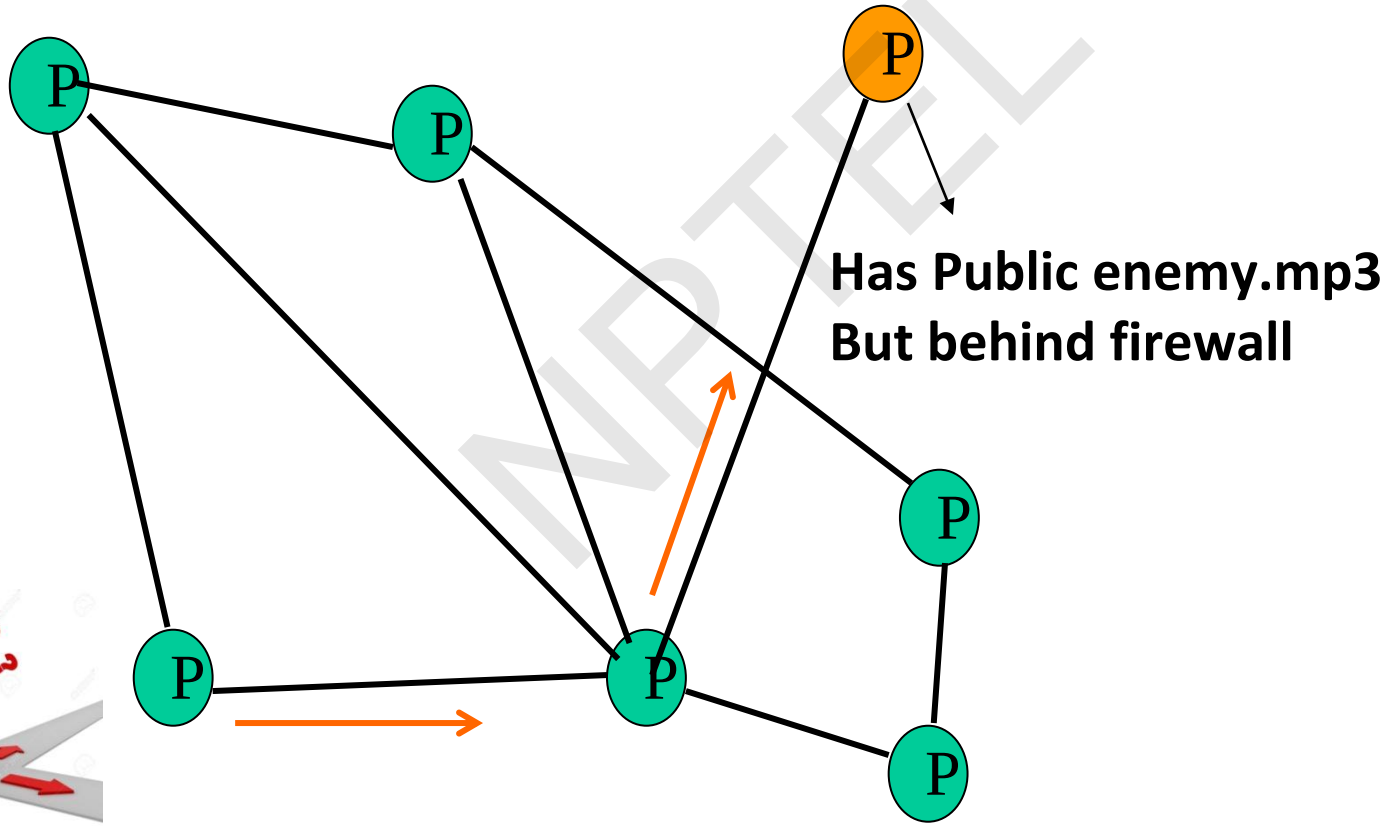
\r\n

After receiving QueryHit messages (2)

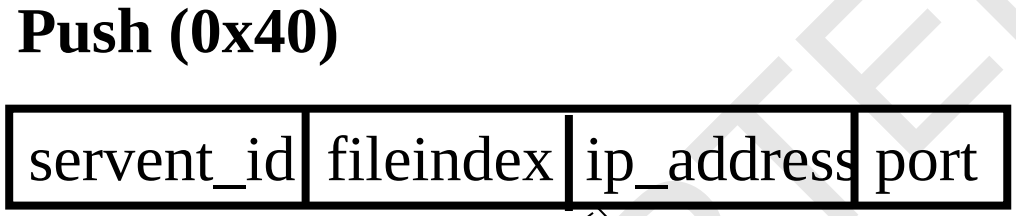
- HTTP is the file transfer protocol. Why?
  - Because it's standard, well-debugged, and widely used.
- Why the "range" field in the GET request?
  - To support partial file transfers.
- What if responder is behind firewall that disallows incoming connections?

Dealing with Firewalls

Requestor sends Push to responder asking for file transfer



Dealing with Firewalls



same as in received QueryHit

Address at which requestor can accept incoming connections

Dealing with Firewalls

- Responder establishes a TCP connection at ip_address, port specified. Sends

  GIV <File Index>:<Servent Identifier>/<File Name>\n\n

- Requestor then sends GET to responder (as before) and file is transferred as explained earlier

- What if requestor is behind firewall too?
  - Gnutella gives up
  - Can you think of an alternative solution?

Ping-Pong

Ping (0x00)

no payload

Pong (0x01)

Port	ip_address	Num. files shared	Num. KB shared

- Peers initiate Ping's periodically
- Pings flooded out like Querys, Pongs routed along reverse path like QueryHits
- Pong replies used to update set of neighboring peers
  - to keep neighbor lists fresh in spite of peers joining, leaving and failing

Summary: Gnutella

- No servers
- Peers/servents maintain "neighbors", this forms an overlay graph
- Peers store their own files
- Queries flooded out, ttl restricted
- QueryHit (replies) reverse path routed
- Supports file transfer through firewalls
- Periodic Ping-pong to continuously refresh neighbor lists
  - List size specified by user at peer : heterogeneity means some peers may have more neighbors
  - Gnutella found to follow **power law** distribution:

$$P(\#links = L) \sim L^{-k} \quad (k \text{ is a constant})$$

Problems

- Ping/Pong constituted 50% traffic
  - **Solution:** Multiplex, *cache* and reduce frequency of pings/pongs
- Repeated searches with same keywords
  - **Solution:** *Cache* Query, QueryHit messages
- Modem-connected hosts do not have enough bandwidth for passing Gnutella traffic
  - **Solution:** use a central server to act as proxy for such peers
  - **Another solution:**

    ➔FastTrack System

Problems (Contd...)

- Large number of ***freeloaders***
  - 70% of users in 2000 were freeloaders
  - Only download files, never upload own files

- Flooding causes excessive traffic
  - Is there some way of maintaining meta-information about peers that leads to more intelligent routing?

    ➔ Structured Peer-to-peer systems

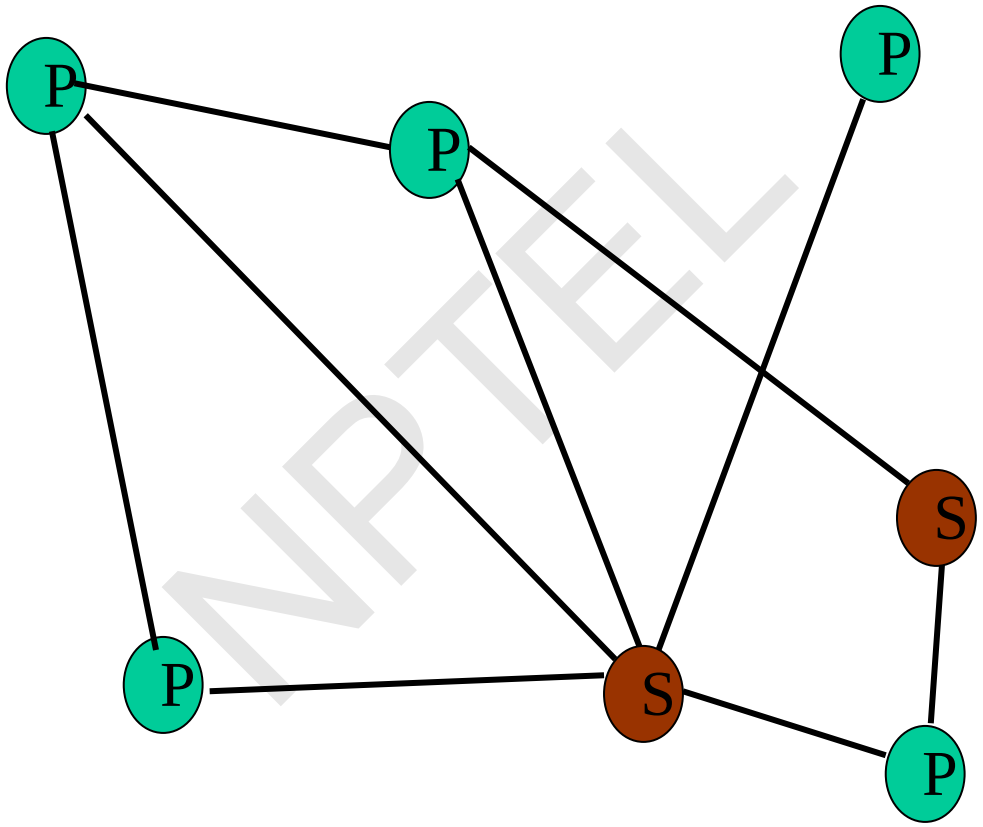
    Example: Chord System

FastTrack

- Hybrid between Gnutella and Napster
- Takes advantage of "healthier" participants in the system
- Underlying technology in Kazaa, KazaaLite, Grokster
- Proprietary protocol, but some details available
- Like Gnutella, but with some peers designated as ***supernodes***

A FastTrack-like System

Peers

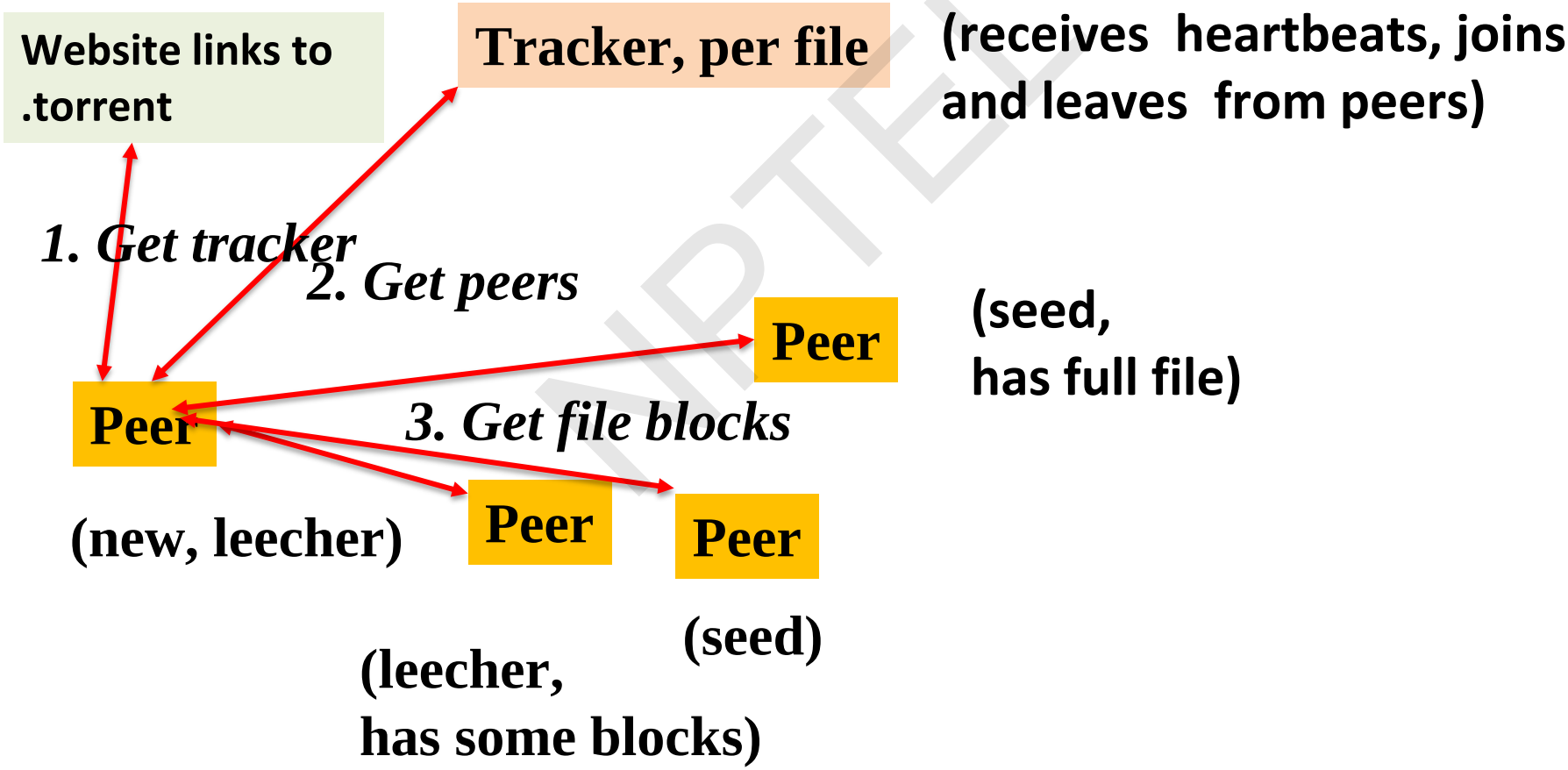


Supernodes

FastTrack (Contd...)

- A supernode stores a directory listing a subset of nearby (<filename,peer pointer>), similar to Napster servers
- Supernode membership changes over time
- Any peer can become (and stay) a supernode, provided it has earned enough ***reputation***
  - **Kazaalite:** participation level (=reputation) of a user between 0 and 1000, initially 10, then affected by length of periods of connectivity and total number of uploads
  - More sophisticated Reputation schemes invented, especially based on economics
- A peer searches by contacting a nearby supernode

BitTorrent


Website links to .torrent
Tracker, per file
(receives heartbeats, joins and leaves from peers)
1. Get tracker
2. Get peers
Peer
(seed, has full file)
Peer
3. Get file blocks
(new, leecher)
Peer
Peer
(seed)
(leecher, has some blocks)

BitTorrent (2)

- File split into blocks (32 KB – 256 KB)
- Download **Local Rarest First** block policy: prefer early download of blocks that are least replicated among neighbors
  - Exception: New node allowed to pick one random neighbor: helps in bootstrapping
- **Tit for tat** bandwidth usage: Provide blocks to neighbors that provided it the best download rates
  - Incentive for nodes to provide good download rates
  - Seeds do the same too
- **Choking:** Limit number of neighbors to which concurrent uploads <= a number (5), i.e., the "best" neighbors
  - Everyone else choked
  - Periodically re-evaluate this set (e.g., every 10 s)
  - **Optimistic unchoke:** periodically (e.g., ~30 s), unchoke a random neigbhor – helps keep unchoked set fresh

DHT (Distributed Hash Table)

- A **hash table** allows you to insert, lookup and delete objects with keys
- A ***distributed* hash table** allows you to do the same in a distributed setting (objects=files)
- **Performance Concerns:**
  - Load balancing
  - Fault-tolerance
  - Efficiency of lookups and inserts
  - Locality
- **Napster, Gnutella, FastTrack are all DHTs** (sort of)
- So is **Chord, a structured peer to peer system**

Comparative Performance

	Memory	Lookup Latency	#Messages for a lookup
Napster	O(1) (O(N)@server)	O(1)	O(1)
Gnutella	O(N)	O(N)	O(N)
Chord	**O(log(N))**	**O(log(N))**	**O(log(N))**

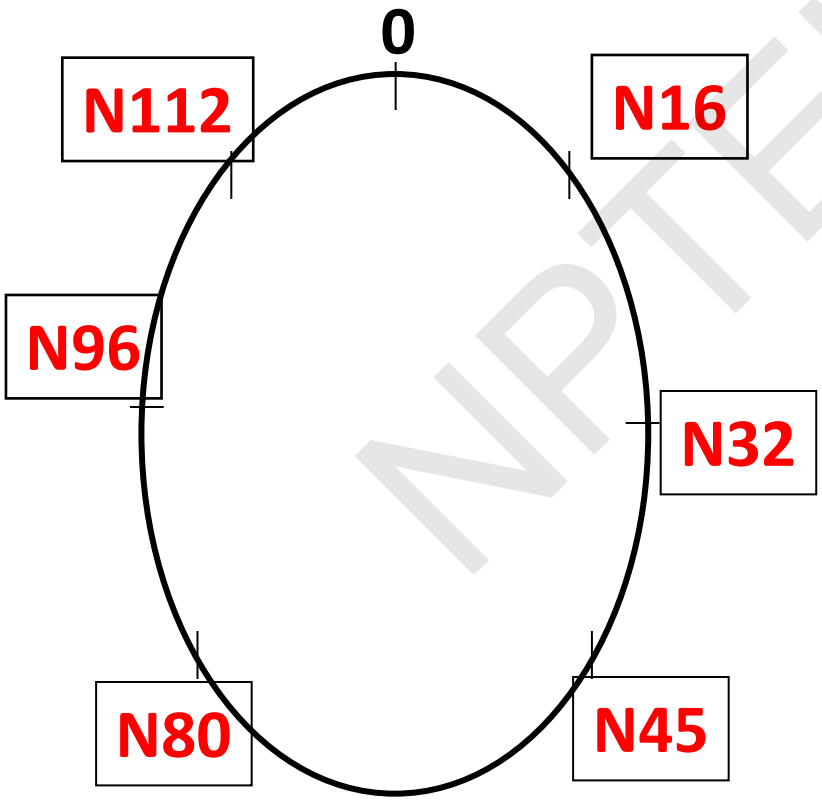
Chord

- **Developers:** I. Stoica, D. Karger, F. Kaashoek, H. Balakrishnan, R. Morris, Berkeley and MIT
- Intelligent choice of neighbors to reduce latency and message cost of routing (lookups/inserts)
- Uses ***Consistent Hashing*** on node's (peer's) address
  - **SHA-1**(ip_address,port) → 160 bit string
  - Truncated to *m* bits
  - Called peer *id* (number between 0 and $2^m - 1$)
  - Not unique but id conflicts very unlikely
  - Can then map peers to one of 2^m logical points on a circle

Ring of peers

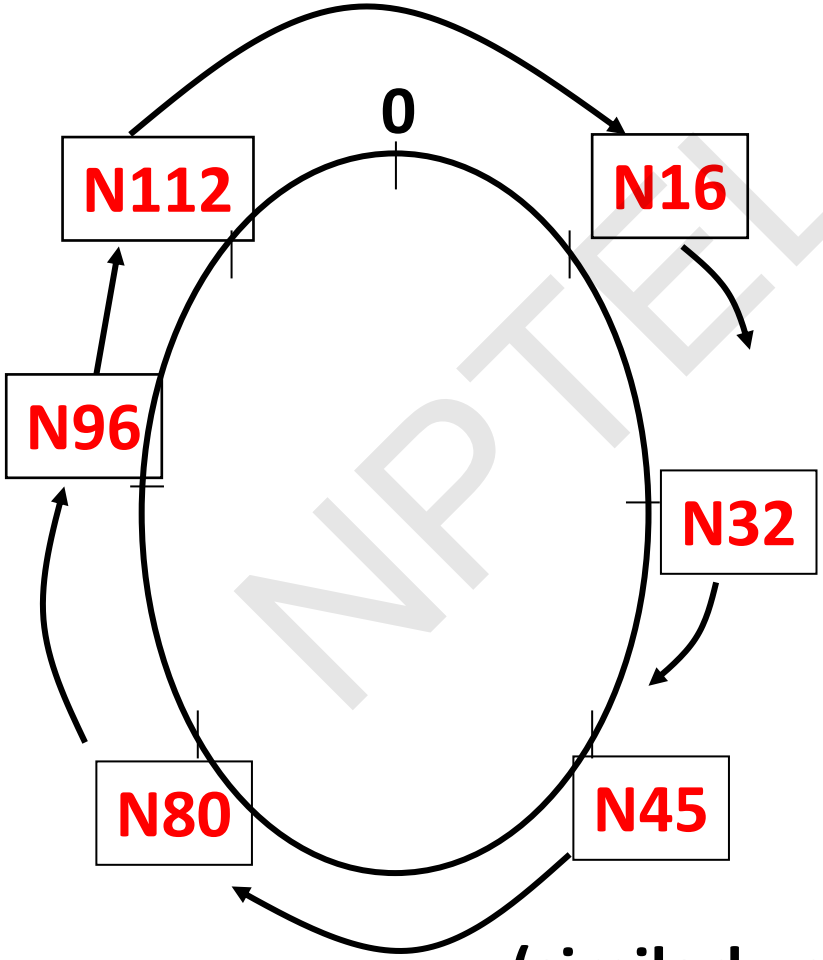
Say *m=7*

6 nodes



Peer Pointers (1): Successors

Say *m=7*



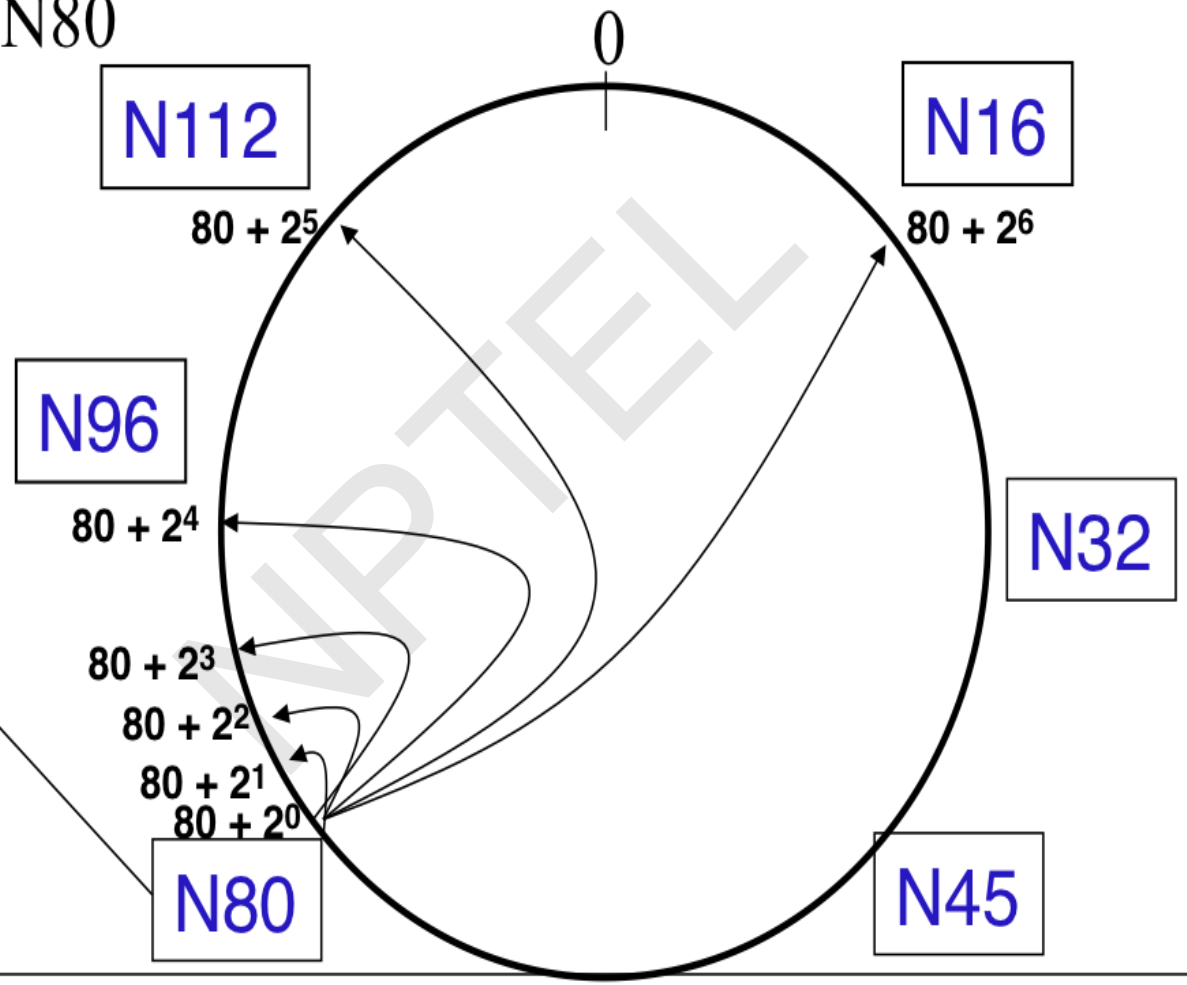
(similarly predecessors)

Peer Pointers (2): finger tables

Say $m=7$

Finger Table at N80

i	$ft[i]$
0	96
1	96
2	96
3	96
4	96
5	112
6	16



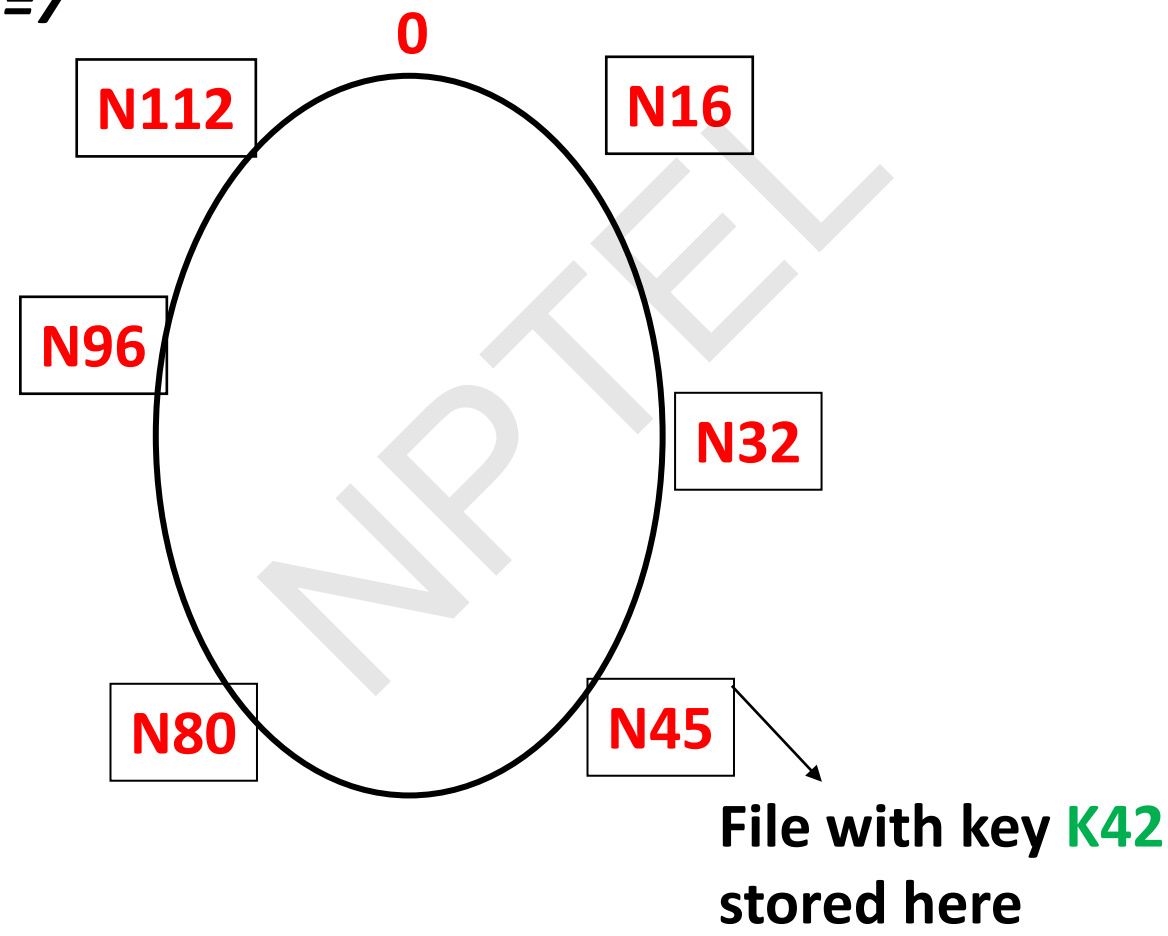
ith entry at peer with id n is first peer with id >= $n + 2^i \pmod{2^m}$

What about the files?

- Filenames also mapped using same consistent hash function
  - SHA-1(filename) →160 bit string (*key*)
  - File is stored at **first peer with id greater than or equal to its key (mod 2^m)**
- File *cnn.com/index.html* that maps to key K42 is stored at first peer with id greater than 42
  - Note that we are considering a different file-sharing application here : *cooperative web caching*
  - The same discussion applies to any other file sharing application, including that of mp3 files.
- Consistent Hashing => with K keys and N peers, each peer stores *O(K/N)* keys. (i.e., < *c.K/N*, for some constant *c*)

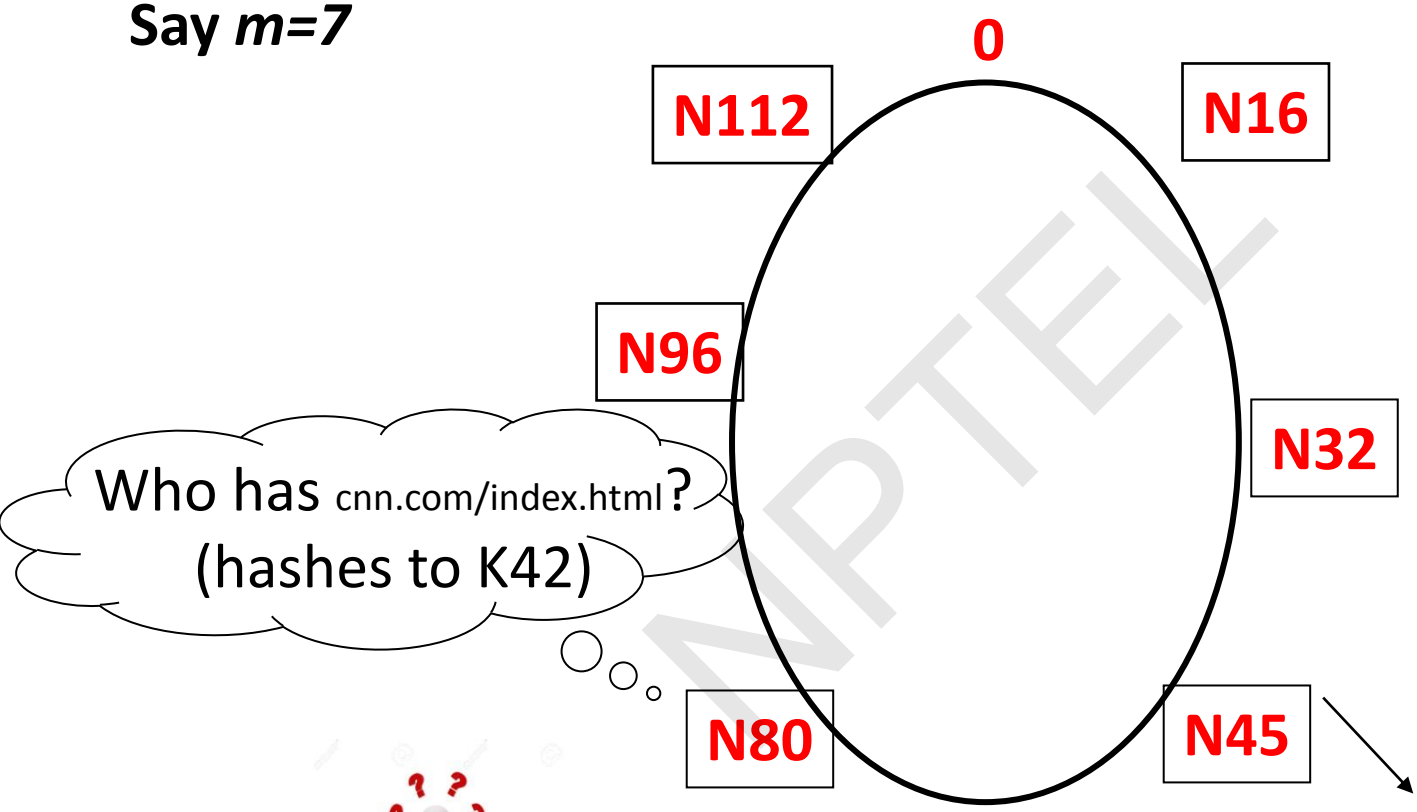
Mapping Files

Say *m=7*



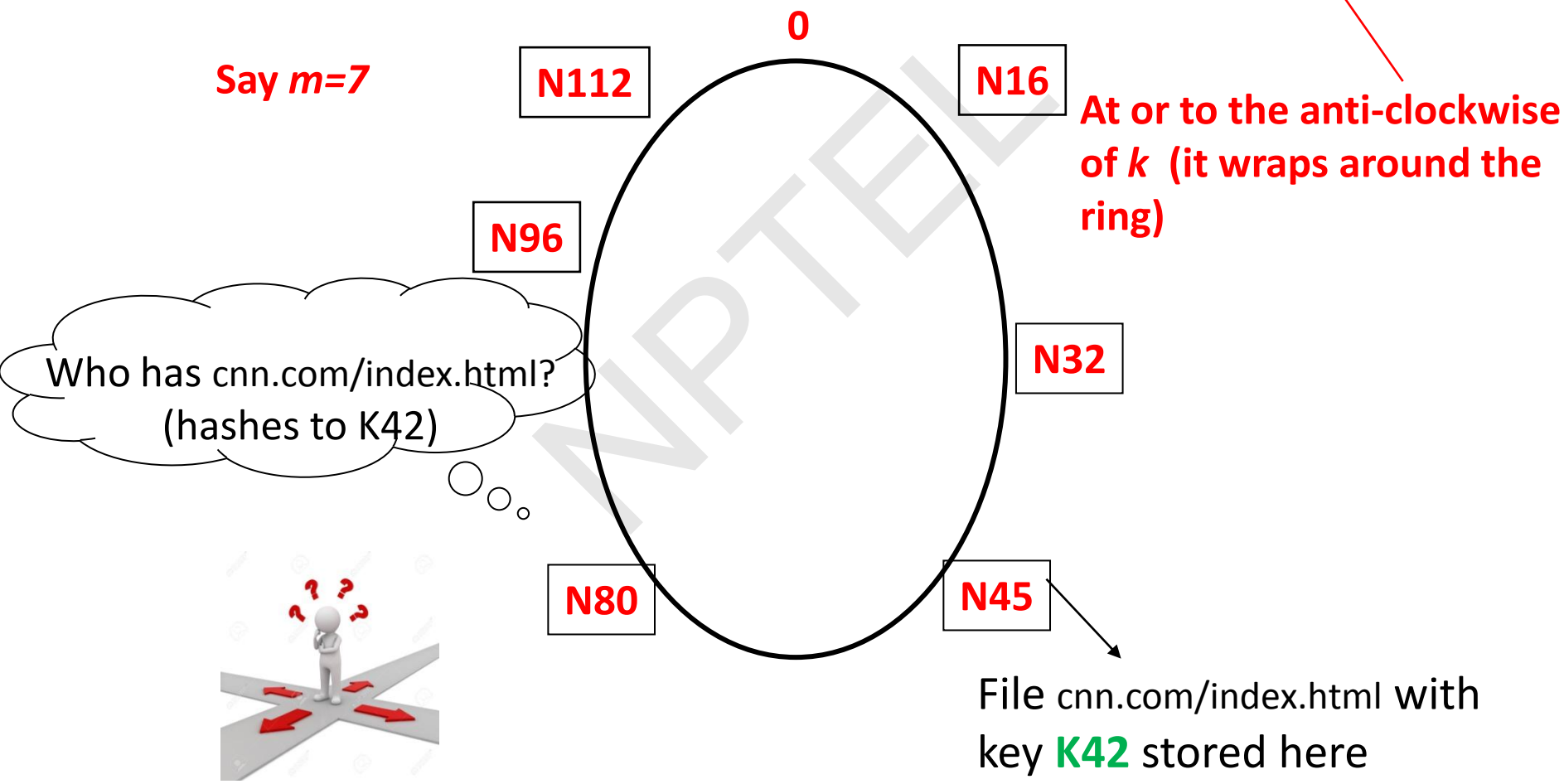
Search

Say *m=7*



Search

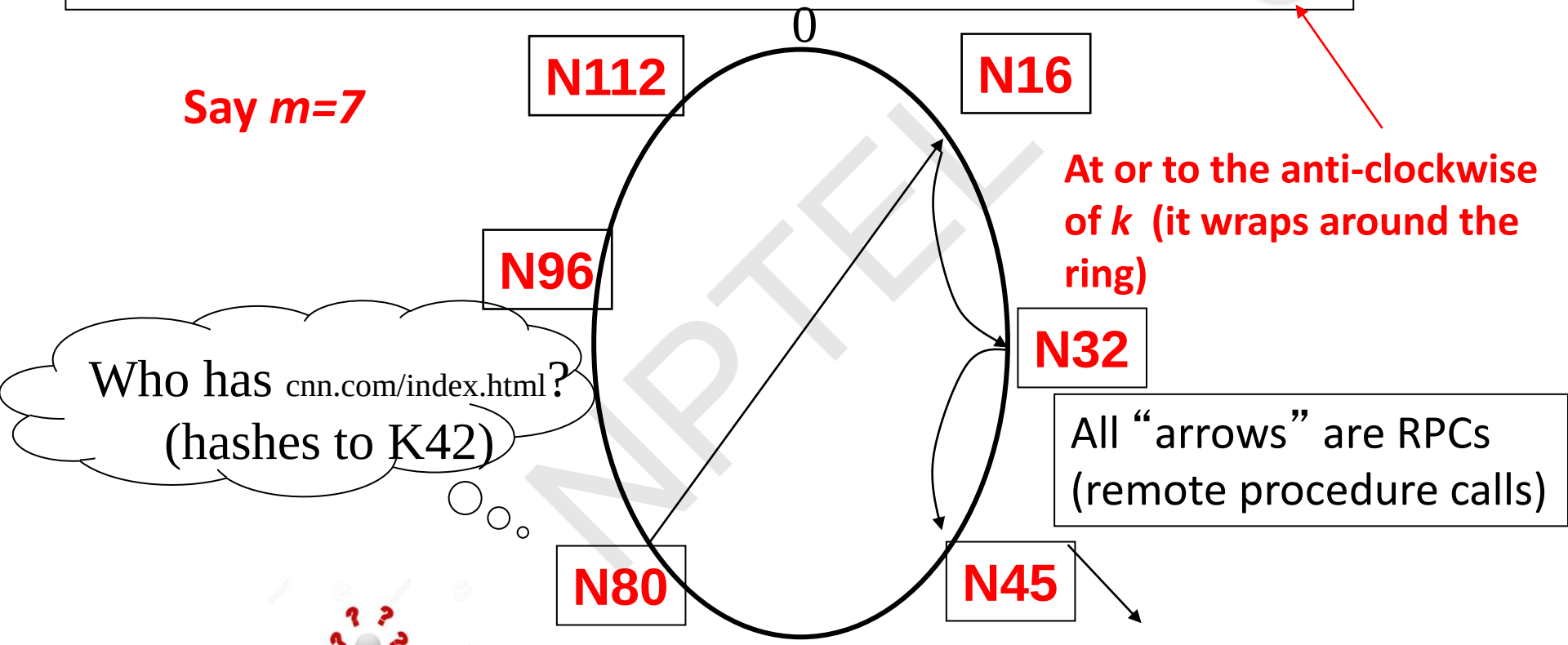
At node *n*, send query for key *k* to largest successor/finger entry <= *k*
if none exist, send query to *successor(n)*



Search

At node *n*, send query for key *k* to largest successor/finger entry <= *k*
if none exist, send query to *successor(n)*

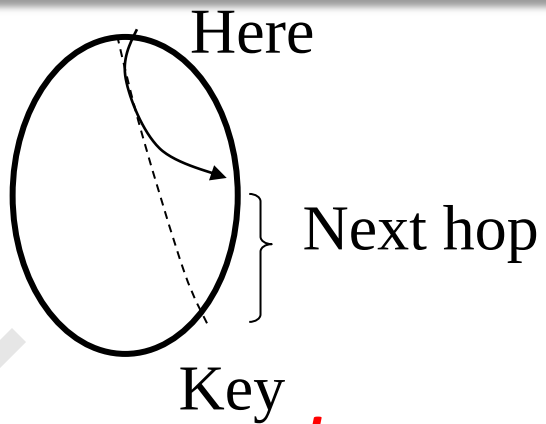
Say *m=7*



Analysis

Search takes *O(log(N))* time

Proof :



- (intuition): ***at each step, distance between query and peer-with-file reduces by a factor of at least 2***
- (intuition): after *log(N)* forwardings, distance to key is at most
- Number of node identifiers in a range of $2^m / 2^{\log(N)} = 2^m / N$ is *O(log(N))* with high probability (why? SHA-1! and "Balls and Bins")

  So using *successors* in that range will be ok, using another *O(log(N))* hops

Analysis (Contd.)

- *O(log(N))* search time holds for file insertions too (in general for ***routing*** *to any key*)
  - "Routing" can thus be used as a **building block** for
    - All operations: insert, lookup, delete

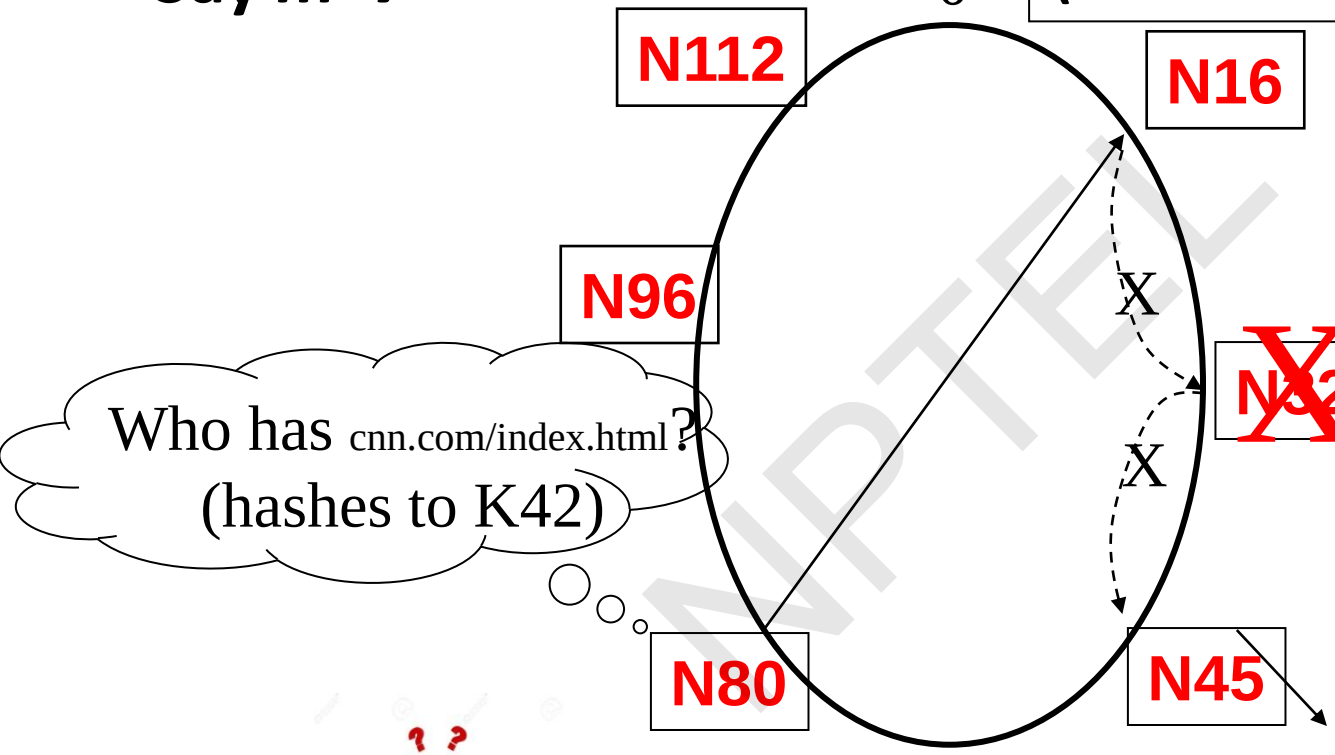
- *O(log(N))* time true only if finger and successor entries correct

- When might these entries be wrong?
  - When you have failures

Search under peer failures

Say *m=7*

Lookup fails
(N16 does not know N45)

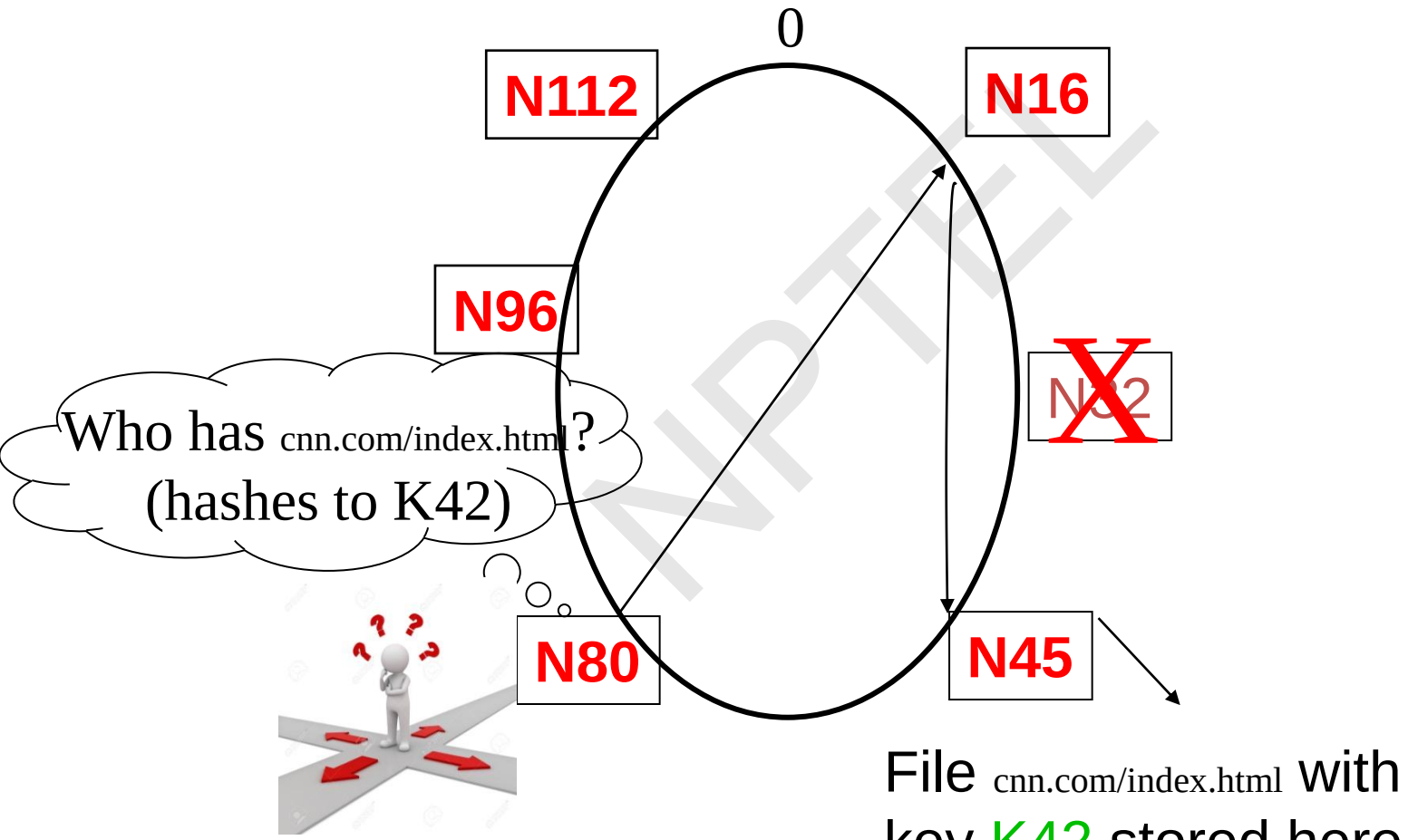


File cnn.com/index.html with key K42 stored here

Search under peer failures

One solution: maintain *r* multiple *successor* entries
In case of failure, use successor entries

Say *m=7*



Search under peer failures

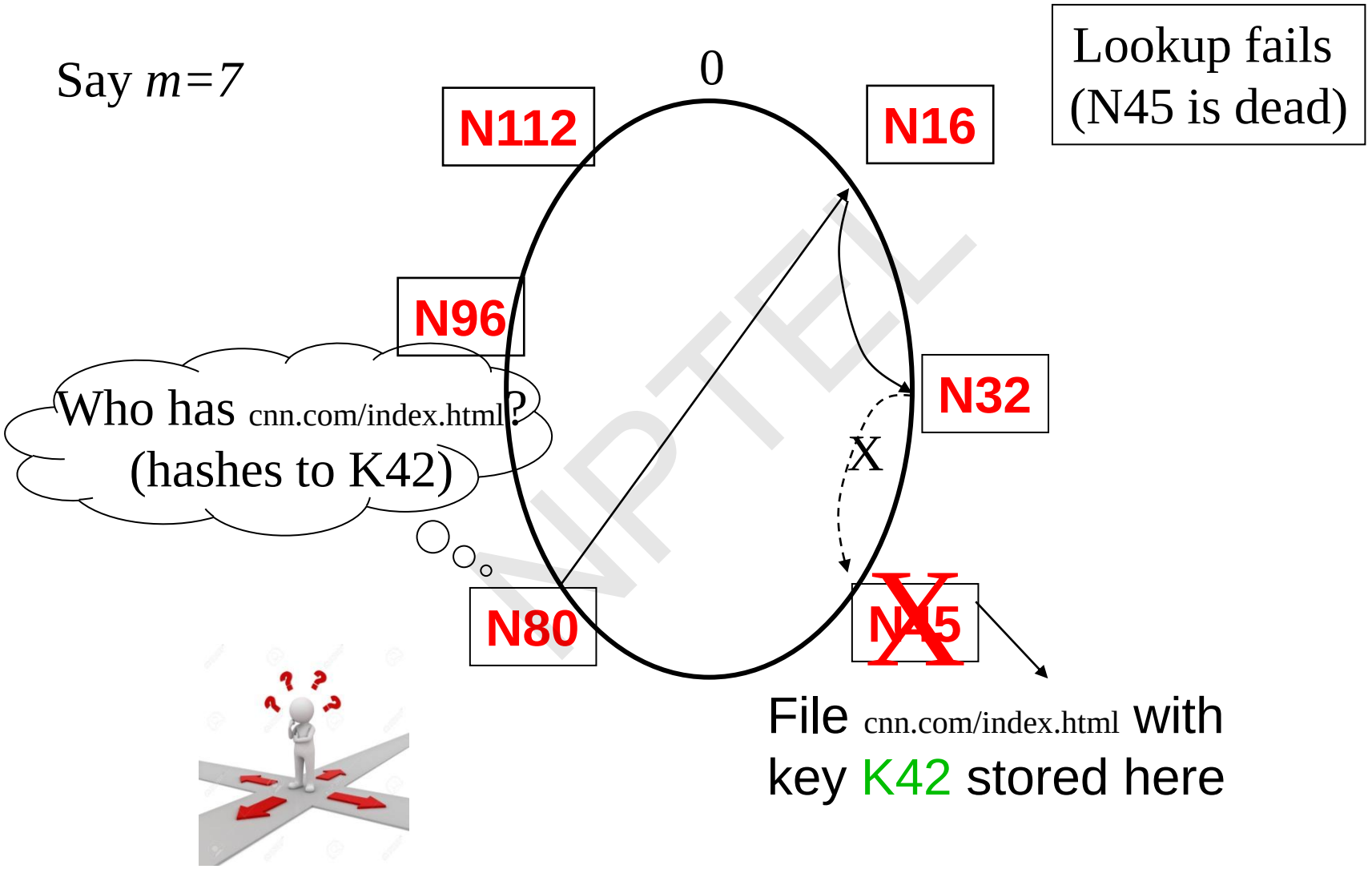
- Choosing *r=2log(N)* suffices to maintain *lookup correctness* with high probability (i.e., ring connected)
  - Say 50% of nodes fail
  - **Pr(at given node, at least one successor alive)=**

$$1-(\frac{1}{2})^{2\log N} = 1-\frac{1}{N^2}$$

  - **Pr(above is true at all alive nodes)=**

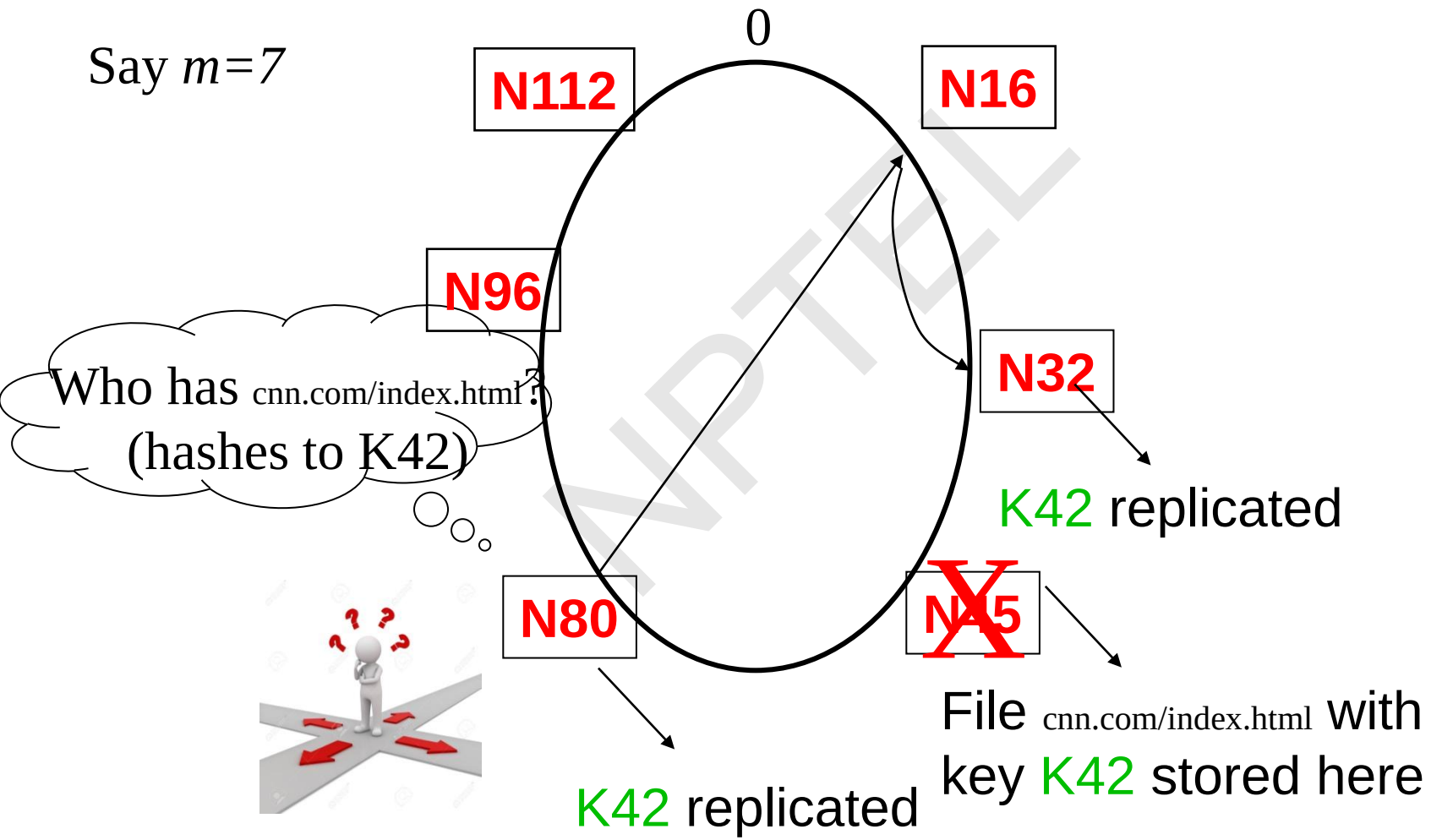
$$(1-\frac{1}{N^2})^{N/2} = e^{-\frac{1}{2N}} \approx 1$$

Search under peer failures (2)



Search under peer failures (2)

One solution: replicate file/key at *r* successors and predecessors



Need to deal with dynamic changes

- ✓ Peers fail
- New peers join
- Peers leave
  - P2P systems have a high rate of ***churn*** (node join, leave and failure)
    - 25% per hour in Overnet (eDonkey)
    - 100% per hour in Gnutella
    - Lower in managed clusters
    - Common feature in all distributed systems, including wide-area (e.g., PlanetLab), clusters (e.g., Emulab), clouds (e.g., AWS), etc.

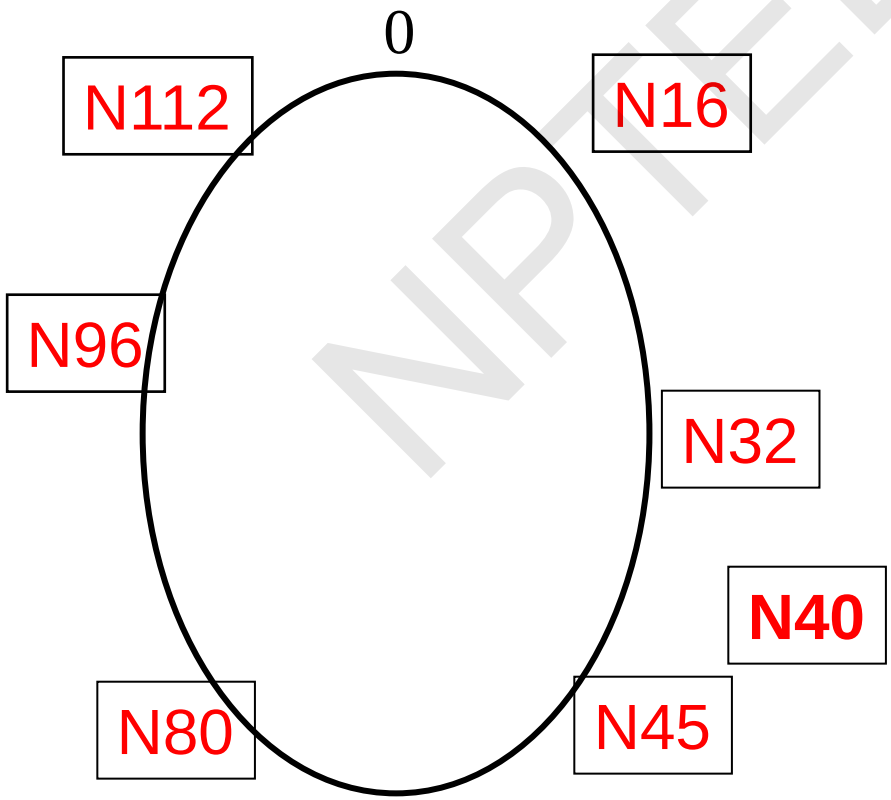
So, all the time, need to:

→ Need to update *successors* and *fingers*, and copy keys

New peers joining

Introducer directs N40 to N45 (and N32)
N32 updates successor to N40
N40 initializes successor to N45, and inits fingers from it
N40 periodically talks to neighbors to update finger table

Say *m=7*

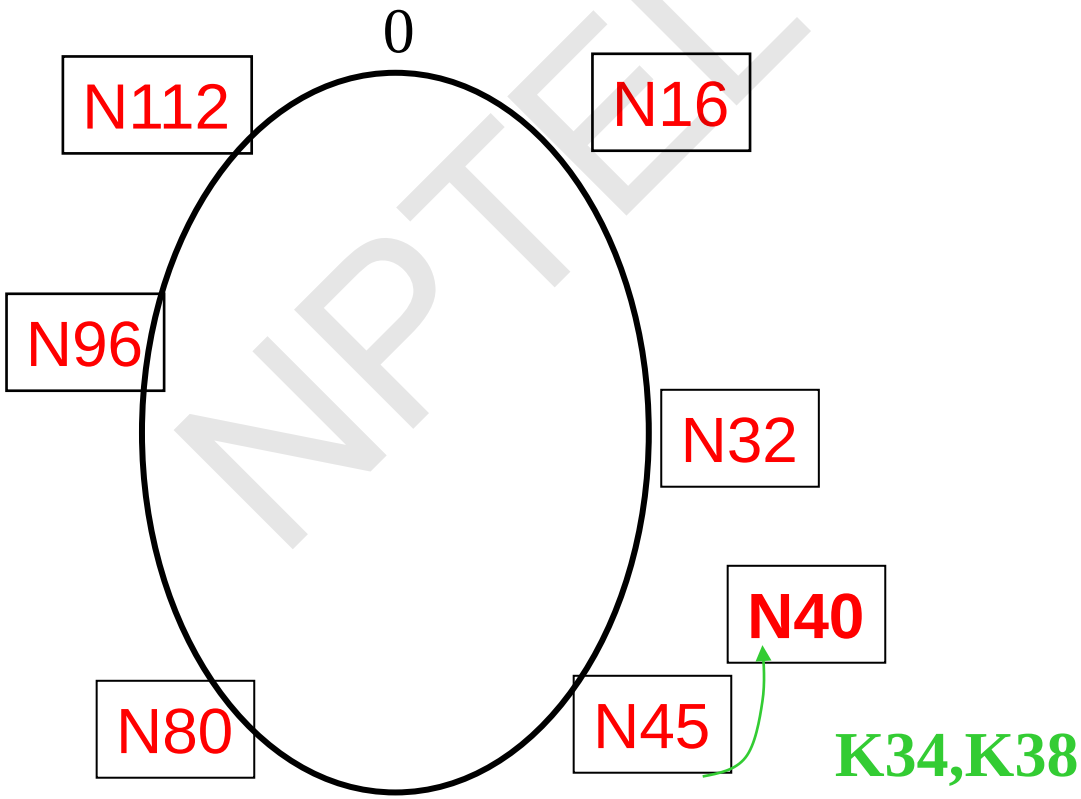


Stabilization Protocol (followed by all nodes)

New peers joining (2)

N40 may need to copy some files/keys from N45 (files with file id between 32 and 40)

Say *m=7*



New peers joining (3)

- A new peer affects *O(log(N))* other finger entries in the system, on average [Why?]
- Number of messages per peer join= *O(log(N)*log(N))*
- Similar set of operations for dealing with peers leaving
  - For dealing with failures, also need ***failure detectors.***

Stabilization Protocol

- Concurrent peer joins, leaves, failures might cause loopiness of pointers, and failure of lookups
  - Chord peers periodically run a *stabilization* algorithm that checks and updates pointers and keys
  - Ensures *non-loopiness* of fingers, eventual success of lookups and *O(log(N))* lookups with high probability
  - Each stabilization round at a peer involves a constant number of messages
  - Strong stability takes $O(N^2)$ stabilization rounds

Churn

- **When nodes are constantly joining, leaving, failing**
  - Significant effect to consider: traces from the Overnet system show *hourly* peer turnover rates (***churn***) could be *25-100%* of total number of nodes in system
  - Leads to excessive (unnecessary) key copying (remember that keys are replicated)
  - Stabilization algorithm may need to consume more bandwidth to keep up
  - Main issue is that files are replicated, while it might be sufficient to replicate only meta information about files
  - **Alternatives**
    - Introduce a level of indirection, i.e., store only pointers to files (any p2p system)
    - Replicate metadata more, e.g., Kelips

Virtual Nodes

- Hash can get non-uniform ➔ Bad load balancing
  - Treat each node as multiple virtual nodes behaving independently
  - Each joins the system
  - Reduces variance of load imbalance

Remarks

- Virtual Ring and Consistent Hashing used in Cassandra, Riak, Voldemort, DynamoDB, and other key-value stores

- **Current status of Chord project:**
  - File systems (CFS,Ivy) built on top of Chord
  - DNS lookup service built on top of Chord
  - Internet Indirection Infrastructure (I3) project at UCB
  - Spawned research on many interesting issues about p2p systems

https://github.com/sit/dht/wiki

Pastry

- **Designed by Anthony Rowstron (Microsoft Research) and Peter Druschel (Rice University)**

- Assigns ids to nodes, just like Chord **(using a virtual ring)**

- **Leaf Set -** Each node knows its successor(s) and predecessor(s)

Pastry Neighbors

- **Routing tables** based on **prefix matching**
  - Think of a hypercube
- Routing is thus based on prefix matching, and is thus log(N)
  - And hops are short (in the underlying network)

Pastry Routing

- Consider a peer with id 01110100101. It maintains a neighbor peer with an id matching each of the following prefixes (* = starting bit differing from this peer's corresponding bit):
  - *
  - 0*
  - 01*
  - 011*
  - … 0111010010*
- When it needs to route to a peer, say 01110111001, it starts by forwarding to a neighbor with the largest matching prefix, i.e., 011101*

Pastry Locality

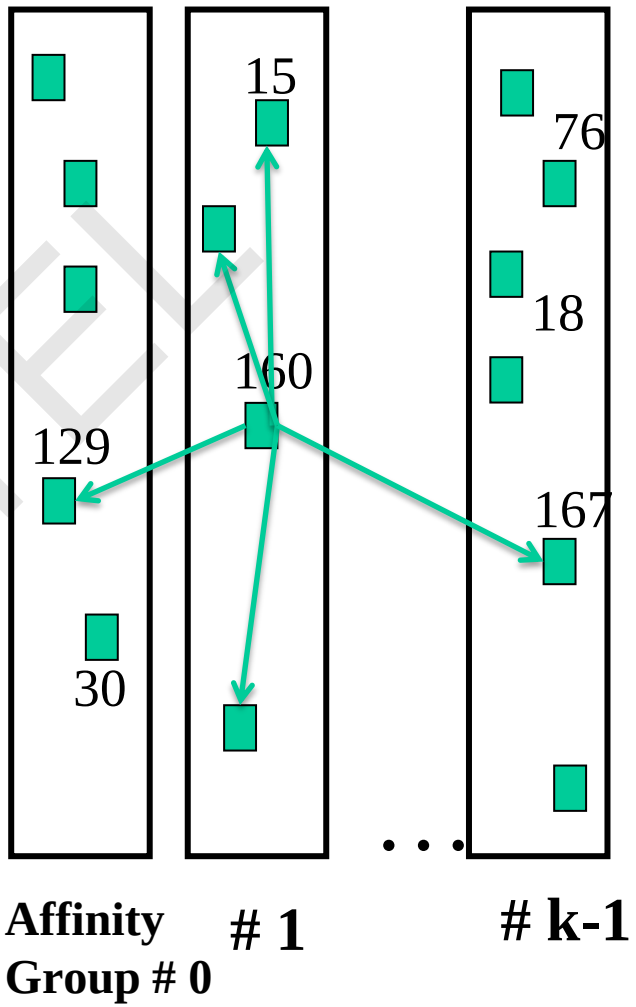
- For each prefix, say 011*, among all potential neighbors with the matching prefix, the neighbor with the shortest round-trip-time is selected
- Since shorter prefixes have many more candidates (spread out throughout the Internet), the neighbors for shorter prefixes are likely to be closer than the neighbors for longer prefixes
- Thus, in the prefix routing, early hops are short and later hops are longer
- Yet overall “stretch”, compared to direct Internet path, stays short

Summary: Chord and Pastry

- **Chord and Pastry protocols:**
  - More structured than Gnutella
  - Black box lookup algorithms
  - Churn handling can get complex
  - ***O(log(N))*** memory and lookup cost
    - ***O(log(N))*** lookup hops may be high
    - Can we reduce the number of hops?

Kelips : A 1 hop Lookup DHT

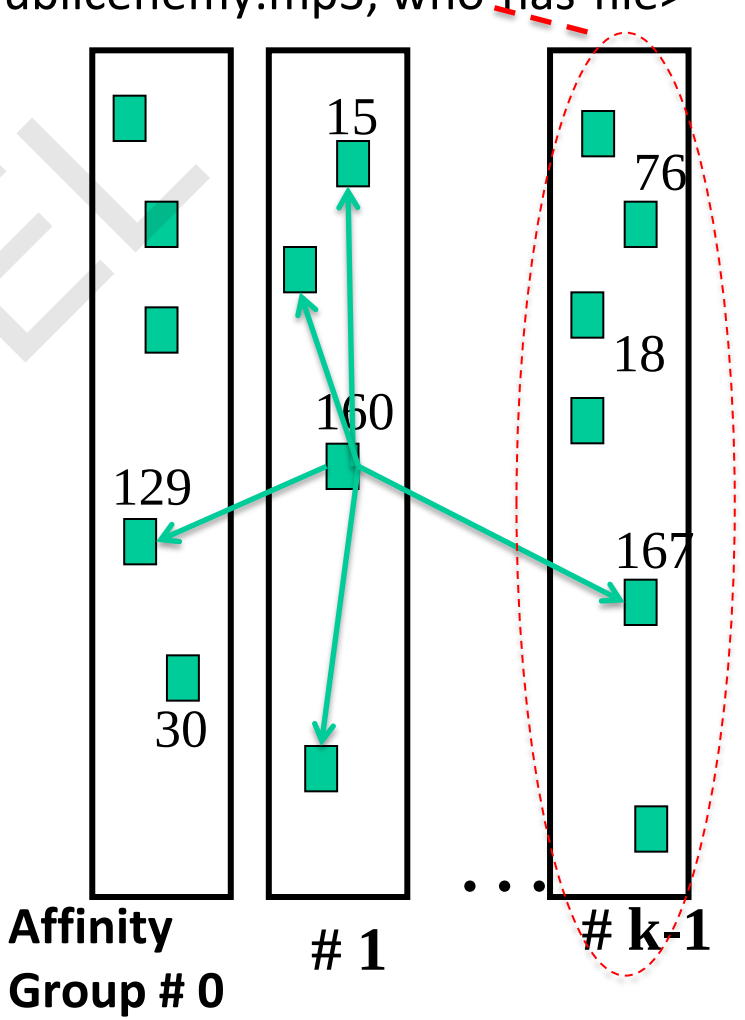
- k "affinity groups"
  - k ~ √ N
- Each node hashed to a group (hash mod k)
- Node's neighbors
  - (Almost) all other nodes in its own affinity group
  - One contact node per foreign affinity group



Kelips Files and Metadata

- File can be stored at any (few) node(s)
- Decouple file replication/location (outside Kelips) from file querying (in Kelips)
- Each filename hashed to a group
  - All nodes in the group replicate pointer information, i.e., <filename, file location>
  - Spread using gossip
  - Affinity group **does not** store files

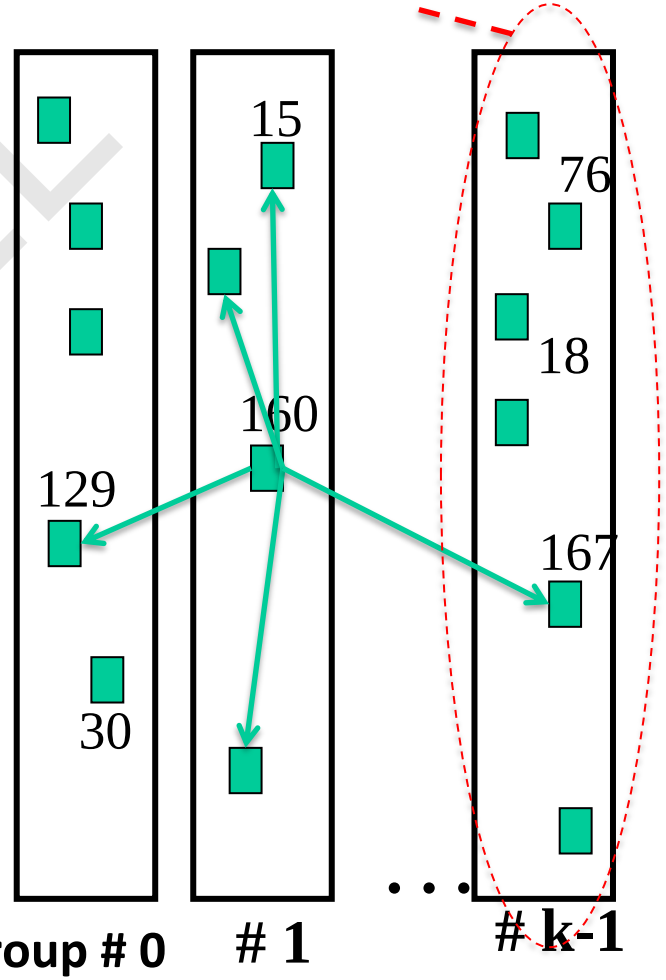
- Publicenemy.mp3 hashes to k-1
- Everyone in this group stores <Publicenemy.mp3, who-has-file>



Kelips Lookup

- **Lookup**
  - Find file affinity group
  - Go to your contact for the file affinity group
  - Failing that try another of your neighbors to find a contact
- **Lookup = 1 hop (or a few)**
  - Memory cost O(√ N)
  - 1.93 MB for 100K nodes, 10M files
  - Fits in RAM of most workstations/laptops today (COTS machines)

- Publicenemy.mp3 hashes to k-1
- Everyone in this group stores <Publicenemy.mp3, who-has-file>

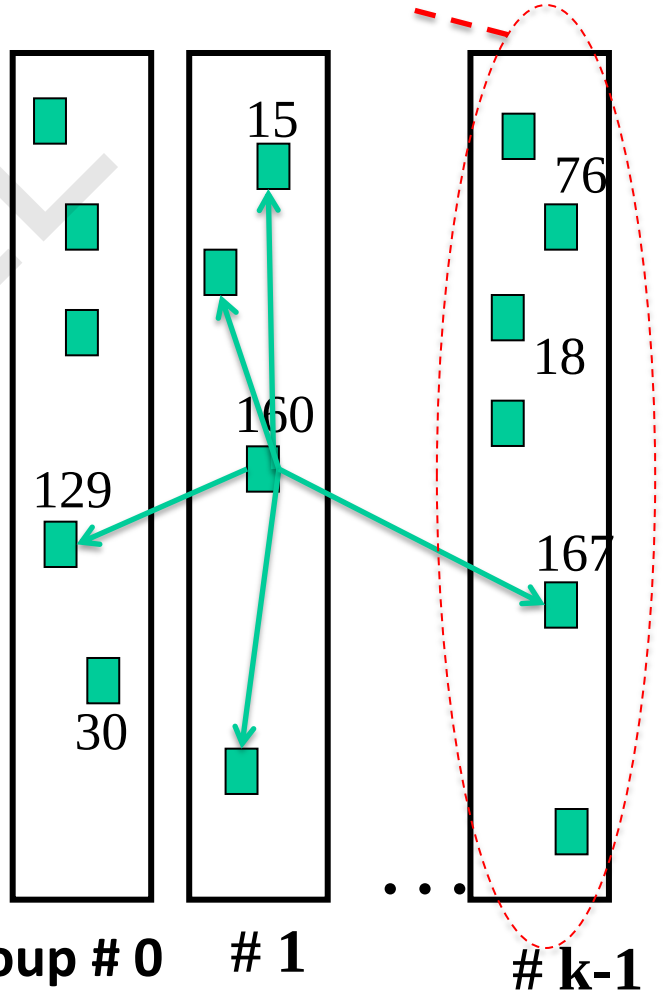


Kelips Soft State

- Publicenemy.mp3 hashes to k-1
- Everyone in this group stores <Publicenemy.mp3, who-has-file>

- **Membership lists**
  - Gossip-based membership
  - Within each affinity group
  - And also across affinity groups
  - *O(log(N))* dissemination time

- **File metadata**
  - Needs to be periodically refreshed from source node
  - Times out



Chord vs. Pastry vs. Kelips

- **Range of tradeoffs available:**
  - Memory vs. lookup cost vs. background bandwidth (to keep neighbors fresh)

Conclusion

- In this lecture, we have studied some of the **widely-deployed P2P Systems such as:**
  1. Napster
  2. Gnutella
  3. Fasttrack
  4. BitTorrent

- We have also discussed some of the **P2P Systems with Provable Properties such as:**
  1. Chord
  2. Pastry
  3. Kelips

MapReduce

Dr. Rajiv Misra
Associate Professor
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 '**MapReduce paradigm**' and its internal working and implementation overview.

- We will also see many examples and different applications of MapReduce being used, and look into how the '**scheduling and fault tolerance**' works inside MapReduce.

Introduction

- **MapReduce** is a programming model and an associated implementation for **processing and generating large data sets.**

- Users specify a ***map*** function that processes a key/value pair to generate a set of intermediate key/value pairs, and a ***reduce*** function that merges all intermediate values associated with the same intermediate key.

- Many real world tasks are expressible in this model.

Contd…

- Programs written in this functional style **are automatically parallelized and executed** on a large cluster of commodity machines.
- The **run-time system** takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication.
- This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.
- A **typical MapReduce computation processes** many terabytes of data on thousands of machines. Hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

Distributed File System

Chunk Servers

- File is split into contiguous chunks
- Typically each chunk is 16-64MB
- Each chunk replicated (usually 2x or 3x)
- Try to keep replicas in different racks

Master node

- Also known as Name Nodes in HDFS
- Stores metadata
- Might be replicated

Client library for file access

- Talks to master to find chunk servers
- Connects directly to chunkservers to access data

Motivation for Map Reduce (Why)

- **Large-Scale Data Processing**
  - Want to use 1000s of CPUs
  - But don't want hassle of managing things

- **MapReduce Architecture provides**
  - Automatic parallelization & distribution
  - Fault tolerance
  - I/O scheduling
  - Monitoring & status updates

MapReduce Paradigm

What is MapReduce?

- Terms are borrowed from Functional Language (e.g., Lisp)

Sum of squares:

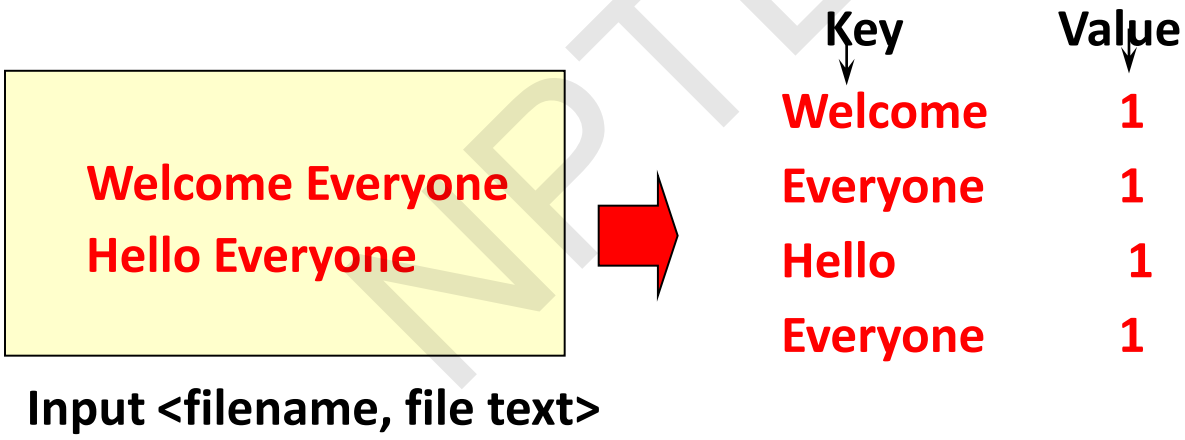
- **(map square '(1 2 3 4))**
  - **Output: (1 4 9 16)**

  [processes each record sequentially and independently]
- **(reduce + '(1 4 9 16))**
  - **(+ 16 (+ 9 (+ 4 1)))**
  - **Output: 30**

  [processes set of all records in batches]
- Let's consider a sample application: **Wordcount**
  - You are given a **huge** dataset (e.g., Wikipedia dump or all of Shakespeare's works) and asked to list the count for each of the words in each of the documents therein

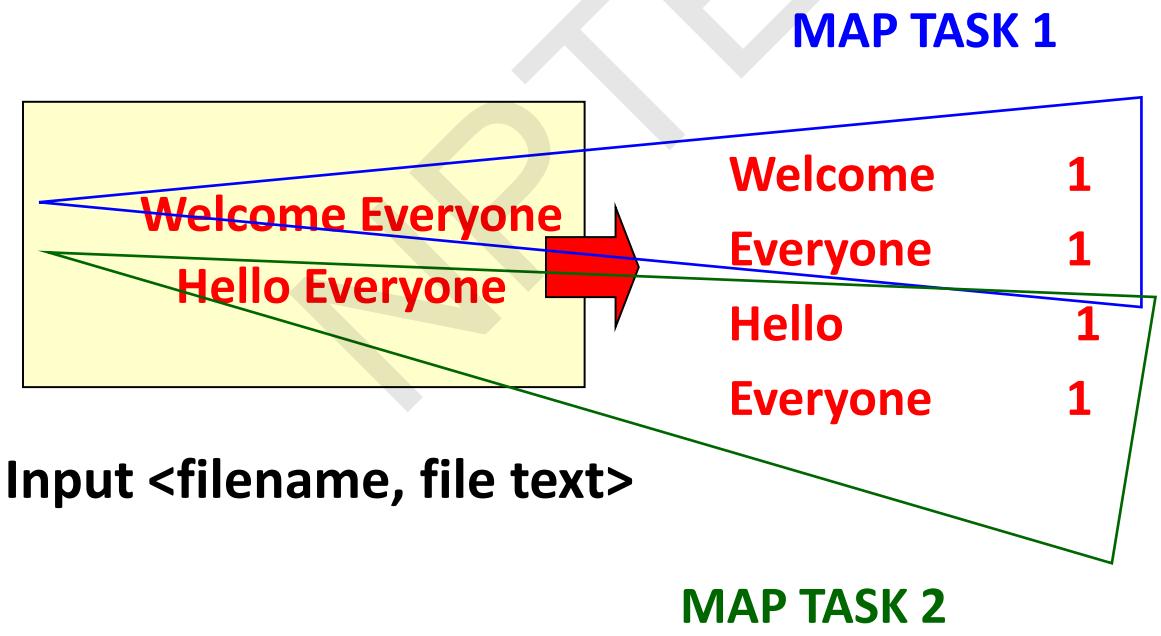
Map

- Process individual records to generate intermediate key/value pairs.



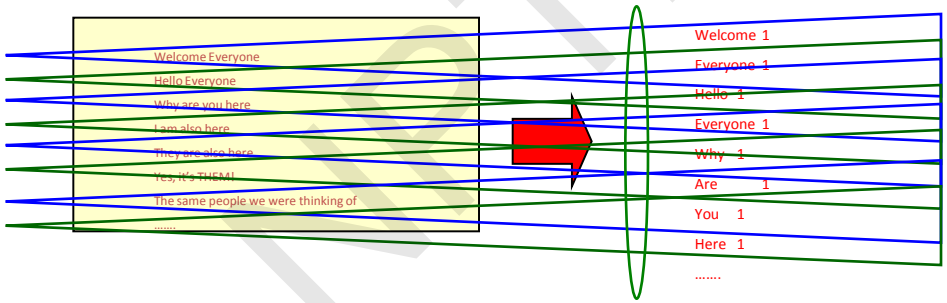
Map

- **Parallelly** Process individual records to generate intermediate key/value pairs.



Map

- **Parallelly** Process **a large number** of individual records to generate intermediate key/value pairs.

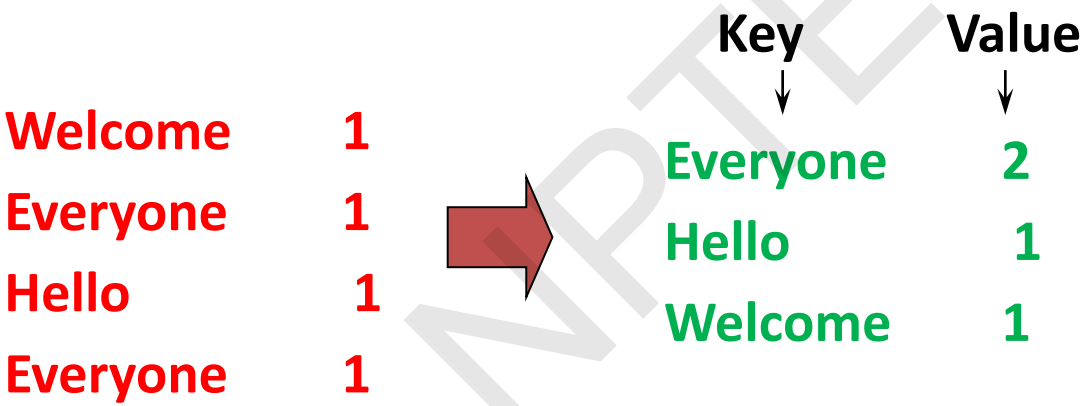


Input <filename, file text>

MAP TASKS

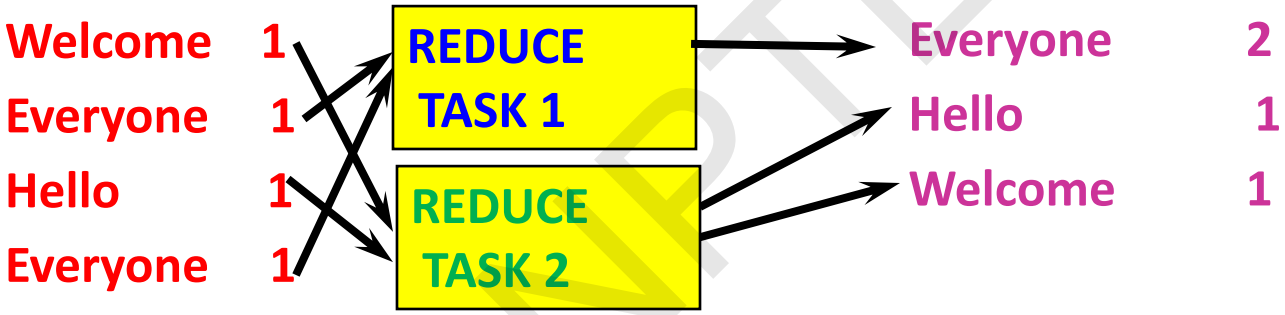
Reduce

- Reduce processes and merges all intermediate values associated per **key**



Reduce

- Each key assigned to one Reduce
- Parallelly Processes and merges all intermediate **values by partitioning keys**



- Popular: ***Hash partitioning***, *i.e.,* key is assigned to
  - reduce # = hash(key)%number of reduce tasks

Programming Model

- The computation takes a set of **input key/value pairs**, and produces a set of **output key/value pairs.**

- The user of the MapReduce library expresses the computation as two functions:

(i) The Map

(ii) The Reduce

(i) Map Abstraction

- Map, written by the user, takes an input pair and produces a set of **intermediate key/value pairs.**

- The MapReduce library groups together all intermediate values associated with the same **intermediate key *'I'*** and passes them to the **Reduce function.**

(ii) Reduce Abstraction

- **The Reduce function,** also written by the user, accepts an **intermediate key *'I'*** and a set of values for that key.

- It merges together these values to form a **possibly smaller set of values.**

- Typically just **zero or one output value** is produced per Reduce invocation. The **intermediate values** are supplied to the user's reduce function via **an iterator.**

- This allows us to **handle lists of values** that are too large to fit in memory.

Map-Reduce Functions for Word Count

map(key, value):

```
// key: document name; value: text of document
  for each word w in value:
        emit(w, 1)
```

reduce(key, values):

```
// key: a word; values: an iterator over counts
        result = 0
        for each count v in values:
              result += v
        emit(key, result)
```

Map-Reduce Functions

- **Input:** a set of key/value pairs
- User supplies two functions:

  map(k,v) → list(k1,v1)

  reduce(k1, list(v1)) → v2
- (k1,v1) is an intermediate key/value pair
- **Output** is the set of (k1,v2) pairs

MapReduce Applications

Applications

- Here are a few simple applications of interesting programs that can be easily expressed as **MapReduce computations.**
- **Distributed Grep:** The map function emits a line if it matches a supplied pattern. The reduce function is an identity function that just copies the supplied intermediate data to the output.
- **Count of URL Access Frequency:** The map function processes logs of web page requests and outputs (URL; 1). The reduce function adds together all values for the same URL and emits a (URL; total count) pair.
- **ReverseWeb-Link Graph:** The map function outputs (target; source) pairs for each link to a target URL found in a page named source. The reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair: (target; list(source))

Contd…

- **Term-Vector per Host:** A term vector summarizes the most important words that occur in a document or a set of documents as a list of (word; frequency) pairs.

- The map function emits a (hostname; term vector) pair for each input document (where the hostname is extracted from the URL of the document).

- The reduce function is passed all per-document term vectors for a given host. It adds these term vectors together, throwing away infrequent terms, and then emits a final (hostname; term vector) pair

Contd…

- **Inverted Index:** The map function parses each document, and emits a sequence of (word; document ID) pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a (word; list(document ID)) pair. The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions.

- **Distributed Sort:** The map function extracts the key from each record, and emits a (key; record) pair. The reduce function emits all pairs unchanged.

Applications of MapReduce

(1) Distributed Grep:

- Input: large set of files
- Output: lines that match pattern

- Map – ***Emits a line if it matches the supplied pattern***

- Reduce – ***Copies the intermediate data to output***

Applications of MapReduce

(2) Reverse Web-Link Graph:

- **Input:** Web graph: tuples (a, b)
  where (page a → page b)
- **Output:** For each page, list of pages that link *to* it
- Map – ***process web log and for each input <source, target>, it outputs <target, source>***
- Reduce - ***emits <target, list(source)>***

Applications of MapReduce

(3) Count of URL access frequency:

- Input: Log of accessed URLs, e.g., from proxy server
- Output: For each URL, % of total accesses for that URL

- Map – ***Process web log and outputs <URL, 1>***
- Multiple Reducers - ***Emits <URL, URL_count>***

(So far, like Wordcount. But still need %)

- Chain another MapReduce job after above one
- Map – ***Processes <URL, URL_count>*** and outputs ***<1, (<URL, URL_count>)>***
- 1 Reducer – Does two passes. In first pass, sums up all ***URL_count's*** to calculate overall_count. In second pass calculates %'s

Emits multiple <URL, URL_count/overall_count>

Applications of MapReduce

(4) Map task's output is sorted (e.g., quicksort)
Reduce task's input is sorted (e.g., mergesort)

Sort

- Input: Series of (key, value) pairs
- Output: Sorted <value>s

- Map – ***<key, value> → <value, _> (identity)***
- Reducer – ***<key, value> → <key, value> (identity)***
- Partitioning function – partition keys across reducers based on **ranges** (can't use hashing!)
  - Take data distribution into account to balance reducer tasks

MapReduce Scheduling

Programming MapReduce

Externally: For **user**

1. Write a Map program (short), write a Reduce program (short)
2. Specify number of Maps and Reduces (parallelism level)
3. Submit job; wait for result
4. Need to know very little about parallel/distributed programming!

Internally: For the Paradigm and Scheduler

1. Parallelize Map
2. Transfer data from Map to Reduce (**shuffle data**)
3. Parallelize Reduce
4. Implement Storage for Map input, Map output, Reduce input, and Reduce output

(Ensure that no Reduce starts before all Maps are finished. That is, ensure **the *barrier*** between the Map phase and Reduce phase)

Inside MapReduce

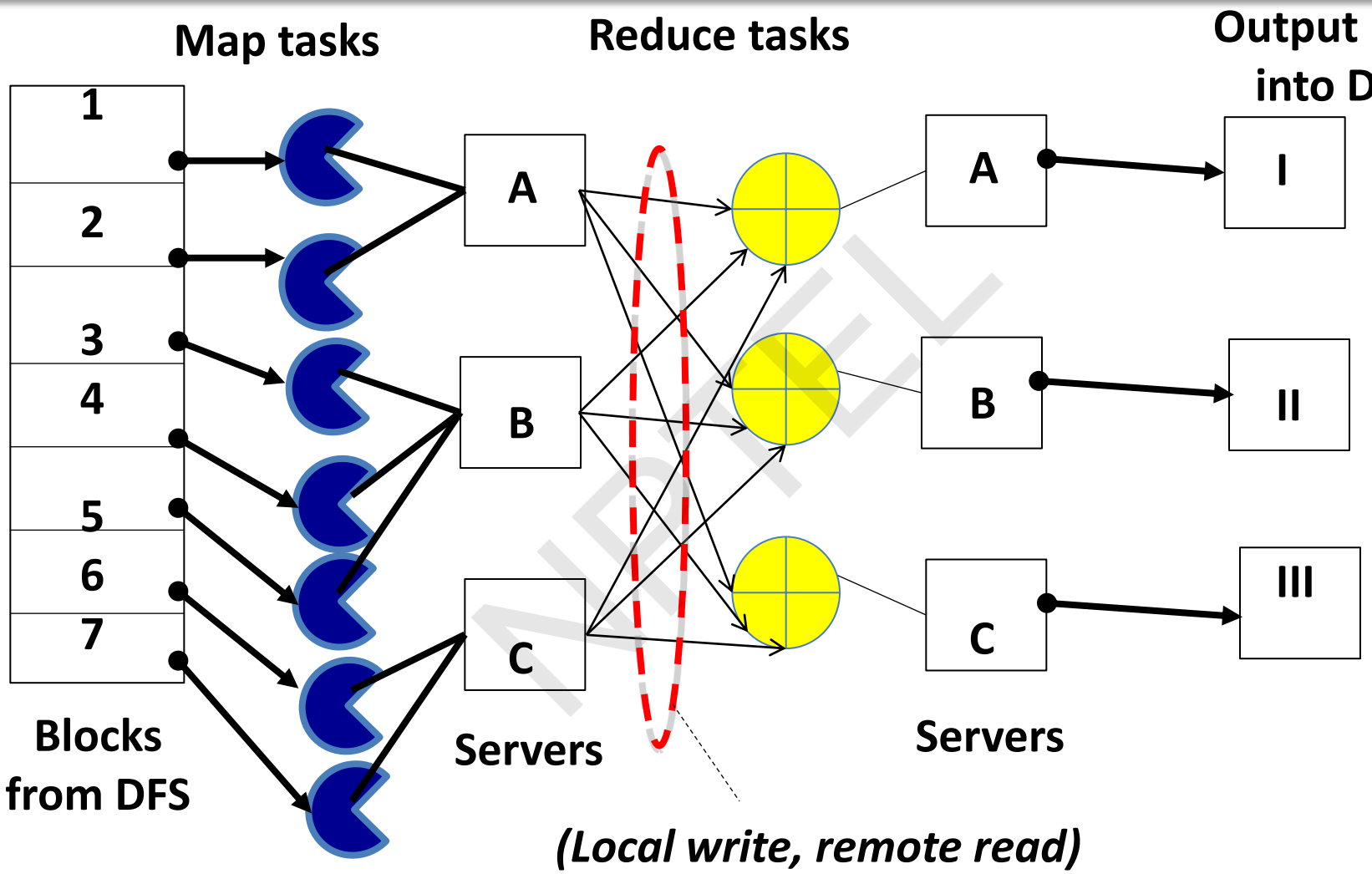
For the cloud:

1. Parallelize Map: **easy!** each map task is independent of the other!
   - All Map output records with same key assigned to same Reduce
2. Transfer data from Map to Reduce:
   - Called Shuffle data
   - All Map output records with same key assigned to same Reduce task
   - use **partitioning function, e.g., hash(key)%number of reducers**
3. Parallelize Reduce: **easy!** each reduce task is independent of the other!
4. Implement Storage for Map input, Map output, Reduce input, and Reduce output
   - Map input: from **distributed file system**
   - Map output: to local disk (at Map node); uses **local file system**
   - Reduce input: from (multiple) remote disks; uses local file systems
   - Reduce output: to distributed file system

local file system = Linux FS, etc.

distributed file system = GFS (Google File System), HDFS (Hadoop Distributed File System)

Internal Workings of MapReduce



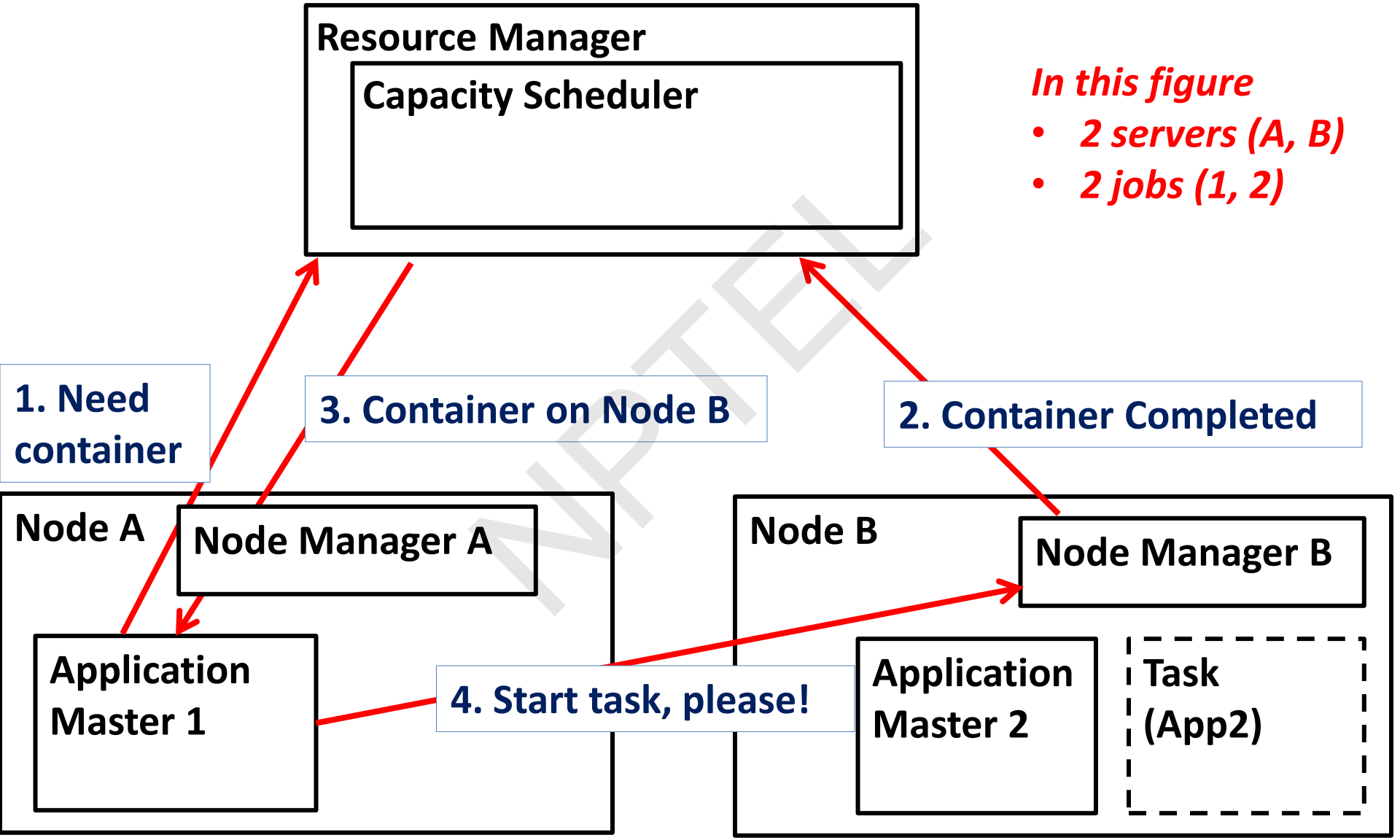
Resource Manager (assigns maps and reduces to servers)

The YARN Scheduler

- Used underneath Hadoop 2.x +
- YARN = Yet Another Resource Negotiator
- Treats each server as a collection of ***containers***
  - Container = fixed CPU + fixed memory

- Has 3 main components
  - **Global *Resource Manager (RM)***
    - Scheduling
  - **Per-server *Node Manager (NM)***
    - Daemon and server-specific functions
  - **Per-application (job) *Application Master (AM)***
    - Container negotiation with RM and NMs
    - Detecting task failures of that job

YARN: How a job gets a container



MapReduce Fault-Tolerance

Fault Tolerance

- **Server Failure**
  - **NM heartbeats to RM**
    - If server fails, RM lets all affected AMs know, and AMs take appropriate action
  - **NM keeps track of each task running at its server**
    - If task fails while in-progress, mark the task as idle and restart it
  - **AM heartbeats to RM**
    - On failure, RM restarts AM, which then syncs up with its running tasks
- **RM Failure**
  - Use old checkpoints and bring up secondary RM
- Heartbeats also used to piggyback container requests
  - Avoids extra messages

Slow Servers

Slow tasks are called **Stragglers**

- The slowest task slows the entire job down (why?)
- Due to Bad Disk, Network Bandwidth, CPU, or Memory
- Keep track of "progress" of each task (% done)
- Perform backup (**replicated)** execution of straggler tasks
  - A task considered done when its first replica complete called **Speculative Execution.**

Locality

- **Locality**
  - Since cloud has hierarchical topology **(e.g., racks)**
  - GFS/HDFS stores 3 replicas of each of chunks (e.g., 64 MB in size)
    - Maybe on different racks, e.g., 2 on a rack, 1 on a different rack
  - **Mapreduce attempts to schedule a map task on**
    1. a machine that contains a replica of corresponding input data, or failing that,
    2. on the same rack as a machine containing the input, or failing that,
    3. Anywhere

Implementation Overview

Implementation Overview

- Many different implementations of the MapReduce interface are possible. The right choice depends on the environment.

- **For example,** one implementation may be suitable for a small shared-memory machine, another for a large NUMA multi-processor, and yet another for an even larger collection of networked machines.

- Here we describes an implementation targeted to the computing environment in wide use at Google: large clusters of commodity PCs connected together with switched Ethernet.

Contd…

(1) Machines are typically dual-processor x86 processor running Linux, with 2-4 GB of memory per machine.

(2) Commodity networking hardware is used . Typically either 100 megabits/second or 1 gigabit/second at the machine level, but averaging considerably less in overall bisection bandwidth.

(3) A cluster consists of hundreds or thousands of machines, and therefore machine failures are common.

(4) Storage is provided by inexpensive IDE disks attached directly to individual machines.

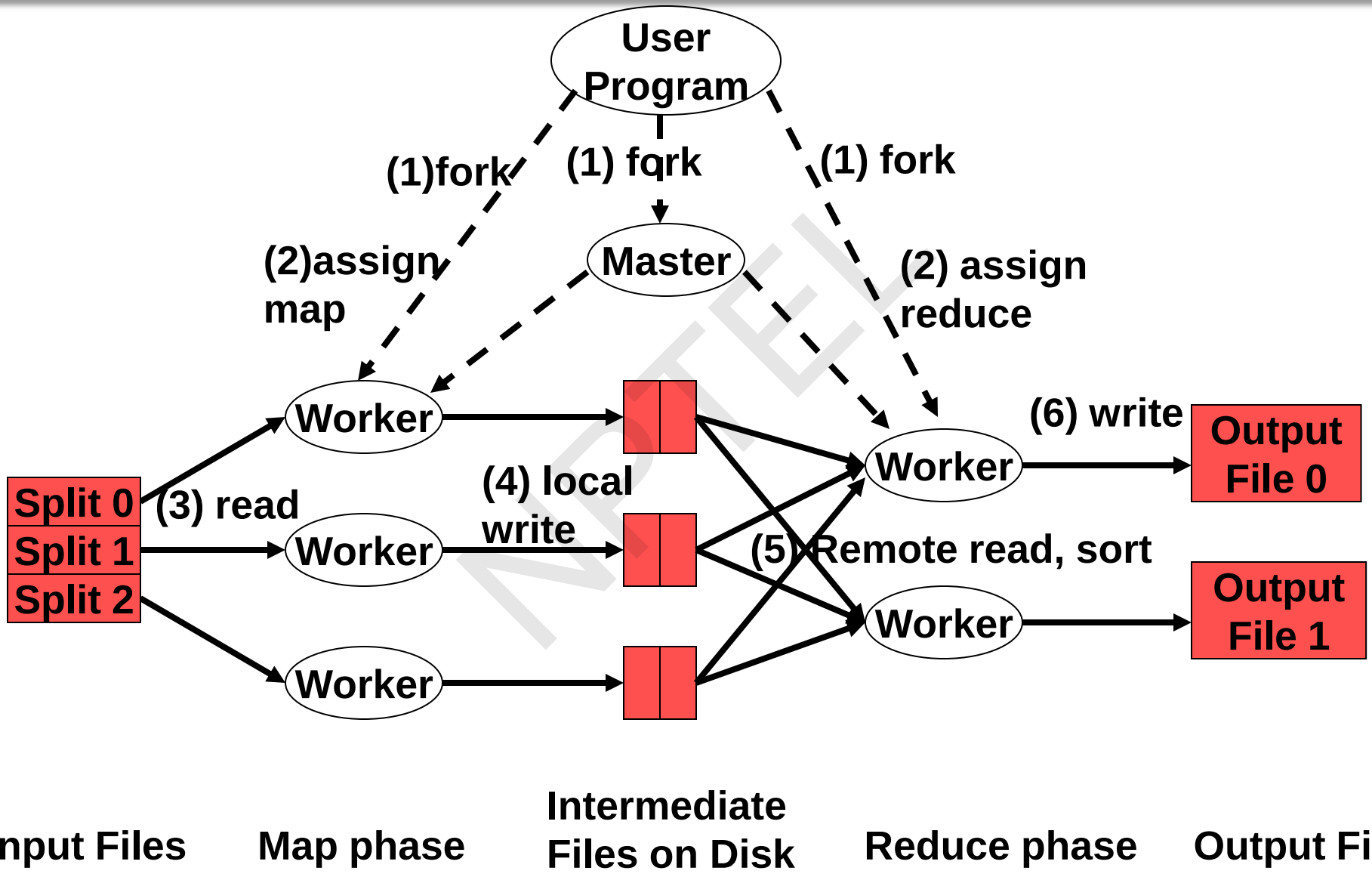
(5) Users submit jobs to a scheduling system. Each job consists of a set of tasks, and is mapped by the scheduler to a set of available machines within a cluster.

Distributed Execution Overview

- The **Map invocations** are distributed across multiple machines by automatically partitioning the input data into a set of M splits.
- The input splits can be processed in parallel by different machines.

- **Reduce invocations** are distributed by partitioning the intermediate key space into R pieces using a partitioning function (e.g., hash(key) mod R).
- The number of partitions (R) and the partitioning function are specified by the user.
- Figure 1 shows the overall flow of a MapReduce operation.

Distributed Execution Overview



Sequence of Actions

When the user **program calls the MapReduce function**, the following **sequence of actions** occurs:

1. The MapReduce library in the user program first splits the input files into M pieces of typically 16 megabytes to 64 megabytes (MB) per piece. It then starts up many copies of the program on a cluster of machines.
2. One of the copies of the program is special- the master. The rest are workers that are assigned work by the master. There are M map tasks and R reduce tasks to assign. The master picks idle workers and assigns each one a map task or a reduce task.
3. A worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined Map function. The intermediate key/value pairs produced by the Map function are buffered in memory.

Contd…

4. Periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function.

- The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers.

5. When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together.

- The sorting is needed because typically many different keys map to the same reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used.

Contd…

6. The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user's Reduce function.

- The output of the Reduce function is appended to a final output file for this reduce partition.

7. When all map tasks and reduce tasks have been completed, the master wakes up the user program.

- At this point, the MapReduce call in the user program returns back to the user code.

Contd…

- After successful completion, the output of the mapreduce execution is available in the R output files (one per reduce task, with file names as specified by the user).

- Typically, users do not need to combine these R output files into one file- they often pass these files as input to another MapReduce call, or use them from another distributed application that is able to deal with input that is partitioned into multiple files.

Master Data Structures

- The master keeps several data structures. For each map task and reduce task, it stores the **state (idle, in-progress, or completed),** and the identity of the worker machine **(for non-idle tasks).**

- The master is the conduit through which the location of intermediate le regions is propagated from map tasks to reduce tasks. Therefore, for each completed map task, the master stores the locations and sizes of the R intermediate file regions produced by the map task.

- Updates to this location and size information are received as map tasks are completed. The information is pushed incrementally to workers that have in-progress reduce tasks.

Fault Tolerance

- Since the MapReduce library is designed to help process very large amounts of data using hundreds or thousands of machines, the library must tolerate machine failures gracefully.
- **Map worker failure**
  - Map tasks completed or in-progress at worker are reset to idle
  - Reduce workers are notified when task is rescheduled on another worker
- **Reduce worker failure**
  - Only in-progress tasks are reset to idle
- **Master failure**
  - MapReduce task is aborted and client is notified

Locality

- Network bandwidth is a relatively scarce resource in the computing environment. We can conserve network bandwidth by taking advantage of the fact that the input data (managed by GFS) is stored on the local disks of the machines that make up our cluster.
- GFS divides each file into 64 MB blocks, and stores several copies of each block (typically 3 copies) on different machines.
- The MapReduce master takes the location information of the input les into account and attempts to schedule a map task on a machine that contains a replica of the corresponding input data. Failing that, it attempts to schedule a map task near a replica of that task's input data (e.g., on a worker machine that is on the same network switch as the machine containing the data).
- When running large MapReduce operations on a significant fraction of the workers in a cluster, most input data is read locally and consumes no network bandwidth.

Task Granularity

- The **Map phase is subdivided into M pieces and the reduce phase into R pieces.**
- Ideally, M and R should be much larger than the number of worker machines.
- Having each worker perform many different tasks improves dynamic load balancing, and also speeds up recovery when a worker fails: the many map tasks it has completed can be spread out across all the other worker machines.
- There are practical bounds on how large M and R can be, since the master must make **O(M + R) scheduling decisions** and keeps **O(M * R) state in memory.**
- Furthermore, R is often constrained by users because the output of each reduce task ends up in a separate output file.

Partition Function

- Inputs to map tasks are created by contiguous splits of input file
- For reduce, we need to ensure that records with the same intermediate key end up at the same worker
- System uses a default partition function e.g., **hash(key) mod R**
- Sometimes useful to override
  - E.g., **hash(hostname(URL)) mod R** ensures URLs from a host end up in the same output file

Ordering Guarantees

- It is guaranteed that within a given partition, the intermediate key/value pairs are processed in increasing key order.

- This ordering guarantee makes it easy to generate a sorted output file per partition, which is useful when the output file format needs to support efficient random access lookups by key, or users of the output and it convenient to have the data sorted.

Combiners Function (1)

- In some cases, there is significant repetition in the intermediate keys produced by each map task, and the user specified Reduce function is commutative and associative.

- **A good example of this is the word counting example. Since word frequencies tend to follow a Zipf distribution, each map task will produce hundreds or thousands of records of the form <the, 1>.**

- All of these counts will be sent over the network to a single reduce task and then added together by the Reduce function to produce one number. We allow the user to specify an optional Combiner function that does partial merging of this data before it is sent over the network.

Combiners Function (2)

- The Combiner function is executed on each machine that performs a map task.
- Typically the same code is used to implement both the combiner and the reduce functions.
- The only difference between a reduce function and a combiner function is how the MapReduce library handles the output of the function.
- The output of a reduce function is written to the final output file. The output of a combiner function is written to an intermediate le that will be sent to a reduce task.
- Partial combining significantly speeds up certain classes of MapReduce operations.

MapReduce Examples

Example: 1 Word Count using MapReduce

```
map(key, value):
// key: document name; value: text of document
    for each word w in value:
        emit(w, 1)
```

```
reduce(key, values):
// key: a word; values: an iterator over counts
        result = 0
        for each count v in values:
            result += v
        emit(key, result)
```

Count Illustrated

map(key=url, val=contents):

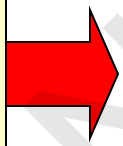
For each word *w* in contents, emit (w, "1")

reduce(key=word, values=uniq_counts):

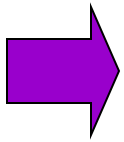
Sum all "1"s in values list

Emit result "(word, sum)"

see bob run
see spot throw



see 1
bob 1
run 1
see 1
spot 1
throw 1



bob 1
run 1
see 2
spot 1
throw 1

Example 2: Counting words of different lengths

- **The map function takes a value and outputs key:value pairs.**

- For instance, if we define a map function that takes a string and outputs the length of the word as the key and the word itself as the value then

- map(steve) would return 5:steve and
- map(savannah) would return 8:savannah.

This allows us to run the map function against values in parallel and provides a huge advantage.

Example 2: Counting words of different lengths

Before we get to the reduce function, the mapreduce framework groups all of the values together by key, so if the map functions output the following **key:value pairs:**

3 : the
3 : and
3 : you
4 : then
4 : what
4 : when
5 : steve
5 : where
8 : savannah
8 : research

They get grouped as:

3 : [the, and, you]
4 : [then, what, when]
5 : [steve, where]
8 : [savannah, research]

Example 2: Counting words of different lengths

- Each of these lines would then be passed as an argument to the reduce function, which accepts a key and a list of values.
- In this instance, we might be trying to figure out how many words of certain lengths exist, so our reduce function will just count the number of items in the list and output the key with the size of the list, like:

3 : 3

4 : 3

5 : 2

8 : 2

Example 2: Counting words of different lengths

- The reductions can also be done in parallel, again providing a huge advantage. We can then look at these final results and see that there were only two words of length 5 in the corpus, etc...

- **The most common example of mapreduce is for counting the number of times words occur in a corpus.**

Example 3: Finding Friends

- Facebook has a list of friends (note that friends are a bi-directional thing on Facebook. If I'm your friend, you're mine).
- They also have lots of disk space and they serve hundreds of millions of requests everyday. They've decided to pre-compute calculations when they can to reduce the processing time of requests. **One common processing request is the "You and Joe have 230 friends in common" feature.**
- When you visit someone's profile, you see a list of friends that you have in common. This list doesn't change frequently so it'd be wasteful to recalculate it every time you visited the profile (sure you could use a decent caching strategy, but then we wouldn't be able to continue writing about mapreduce for this problem).
- We're going to use mapreduce so that we can calculate everyone's common friends once a day and store those results. Later on it's just a quick lookup. We've got lots of disk, it's cheap.

Example 3: Finding Friends

- Assume the friends are stored as **Person->[List of Friends],** our friends list is then:

- A -> B C D
- B -> A C D E
- C -> A B D E
- D -> A B C E
- E -> B C D

Example 3: Finding Friends

For map(A -> B C D) :

(A B) -> B C D

(A C) -> B C D

(A D) -> B C D

For map(B -> A C D E) : (Note that A comes before B in the key)

(A B) -> A C D E

(B C) -> A C D E

(B D) -> A C D E

(B E) -> A C D E

Example 3: Finding Friends

For map(C -> A B D E) :

(A C) -> A B D E

(B C) -> A B D E

(C D) -> A B D E

(C E) -> A B D E

For map(D -> A B C E) :

(A D) -> A B C E

(B D) -> A B C E

(C D) -> A B C E

(D E) -> A B C E

And finally for map(E -> B C D):

(B E) -> B C D

(C E) -> B C D

(D E) -> B C D

Example 3: Finding Friends

- Before we send these key-value pairs to the reducers, we group them by their keys and get:

(A B) -> (A C D E) (B C D)

(A C) -> (A B D E) (B C D)

(A D) -> (A B C E) (B C D)

(B C) -> (A B D E) (A C D E)

(B D) -> (A B C E) (A C D E)

(B E) -> (A C D E) (B C D)

(C D) -> (A B C E) (A B D E)

(C E) -> (A B D E) (B C D)

(D E) -> (A B C E) (B C D)

Example 3: Finding Friends

- Each line will be passed as an argument to a reducer.
- The **reduce function will simply intersect the lists of values** and output the same key with the result of the intersection.
- For example, **reduce((A B) -> (A C D E) (B C D))**

  will **output (A B) : (C D)**
- **and means that friends A and B have C and D as common friends.**

Example 3: Finding Friends

- The result after reduction is:
- (A B) -> (C D)
- (A C) -> (B D)
- (A D) -> (B C)
- (B C) -> (A D E)
- (B D) -> (A C E)
- (B E) -> (C D)
- (C D) -> (A B E)
- (C E) -> (B D)
- (D E) -> (B C)

Now when D visits B's profile, we can quickly look up (B D) and see that they have three friends in common, (A C E).

Reading

Jeffrey Dean and Sanjay Ghemawat,

"MapReduce: Simplified Data Processing on Large Clusters"

http://labs.google.com/papers/mapreduce.html

Conclusion

- **The MapReduce programming model has been successfully used at Google for many different purposes.**

- The model is easy to use, even for programmers without experience with parallel and distributed systems, since it hides the details of parallelization, fault-tolerance, locality optimization, and load balancing.

- A large variety of problems are easily expressible as MapReduce computations.

- **For example, MapReduce is used for the generation of data for Google's production web search service, for sorting, for data mining, for machine learning, and many other systems.**

Conclusion

- Mapreduce uses **parallelization + aggregation** to schedule applications across clusters
- **Need to deal with failure**
- Plenty of ongoing research work in **scheduling and fault-tolerance for Mapreduce and Hadoop.**