

CS 5450

Peer-to-Peer Systems

Vitaly Shmatikov

Overlay Networks

- ◆ Overlay network = one network (or a network-like data structure) is superimposed upon an underlying network
- ◆ Why?
 - Superimpose some form of routed behavior on a set of nodes
 - The underlying network gives the nodes a way to talk to each other, e.g. over TCP or with IP packets
 - May want a behavior that goes beyond just being able to send packets and reflects some kind of end-user “behavior” that we want to implement


Examples of Overlay Networks

◆ VPNs

◆ Tor

◆ Skype

◆ Bitcoin

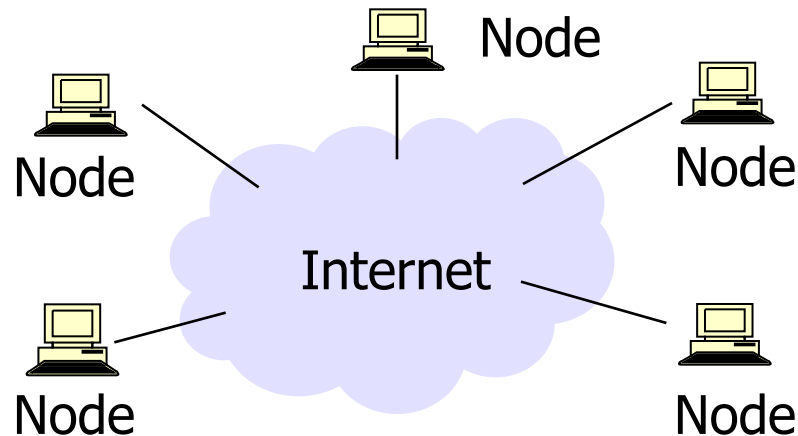
◆ File sharing services (e.g., BitTorrent) 

- A way to create a list of places that have the file you want
- A way to connect to one of those places to pull the file from that machine to yours
 - Once you have the file, your system becomes a possible source for other users to download from
 - In practice, some users tend to run servers with better resources and others tend to be mostly downloaders

Technical Issues

- ◆ What's the very best way for a massive collection of computers in the wide-area Internet (the WAN) to implement these two aspects
 - Best way to do search?
 - Best way to implement peer-to-peer downloads?
- ◆ Cloud computing solutions often have a search requirement
 - Useful even within a single data center

Peer-to-Peer (P2P) System



- ◆ No centralized control
- ◆ Nodes are roughly symmetric in function
- ◆ Large number of unreliable nodes

P2P Environment

◆ Nodes have symmetric functionalities

- Anybody can join and leave
 - “Churn”: nodes come and go at will, possibly quite frequently (a few minutes)
- Everybody gives and takes

◆ Nodes have different capacities

- Bandwidth, processing, storage

◆ Nodes may behave badly

- Promise to do something (store a file) and not do it (free-loaders)
- Attack the system

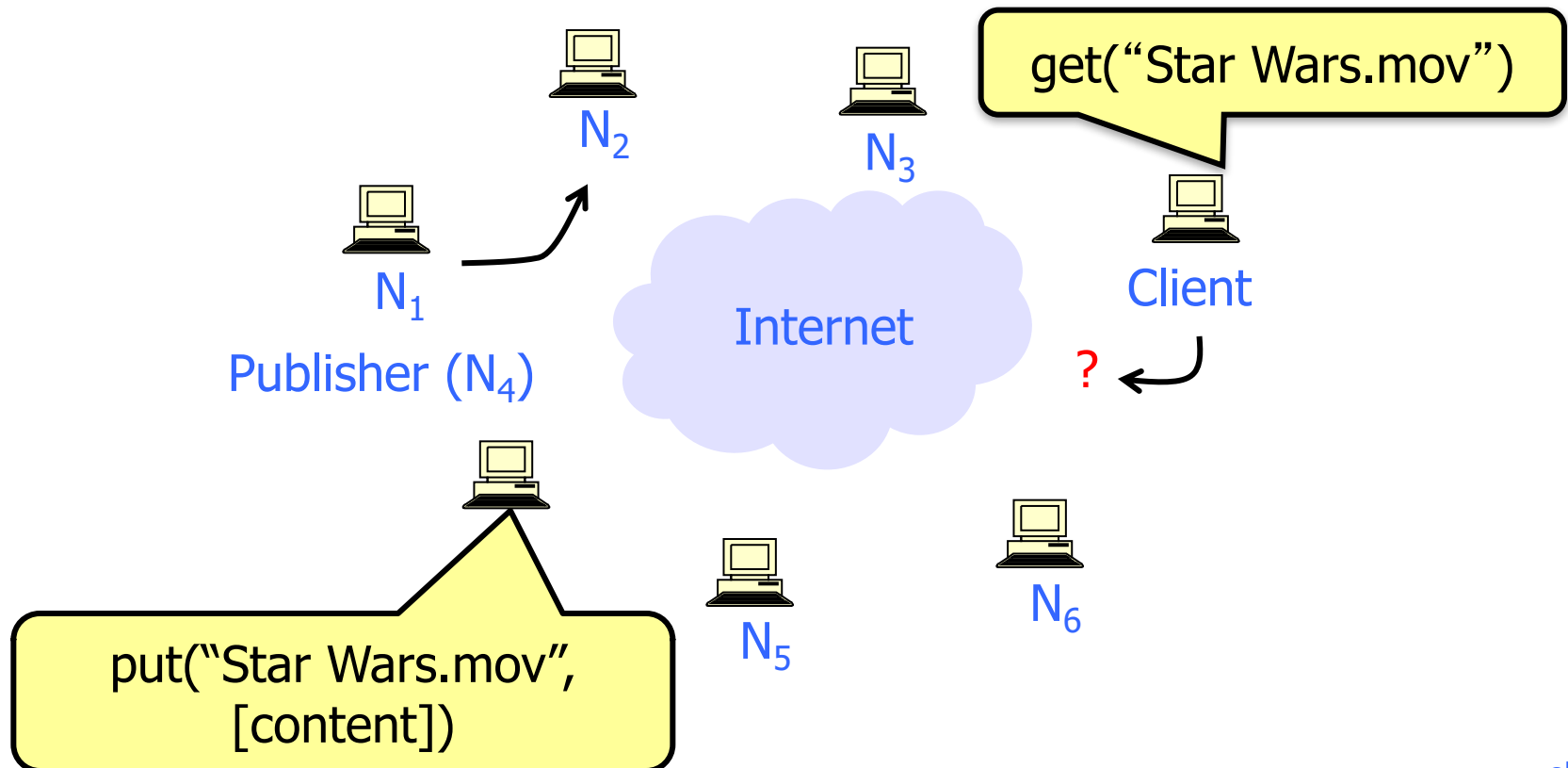
Why P2P?

- ◆ High capacity through parallelism (potentially)
 - Many disks
 - Many network connections
 - Many CPUs
- ◆ Absence of a centralized server or servers
 - Less chance of service overload as load increases
 - Easier deployment
 - A single failure won't wreck the whole system
 - System as a whole is harder to attack

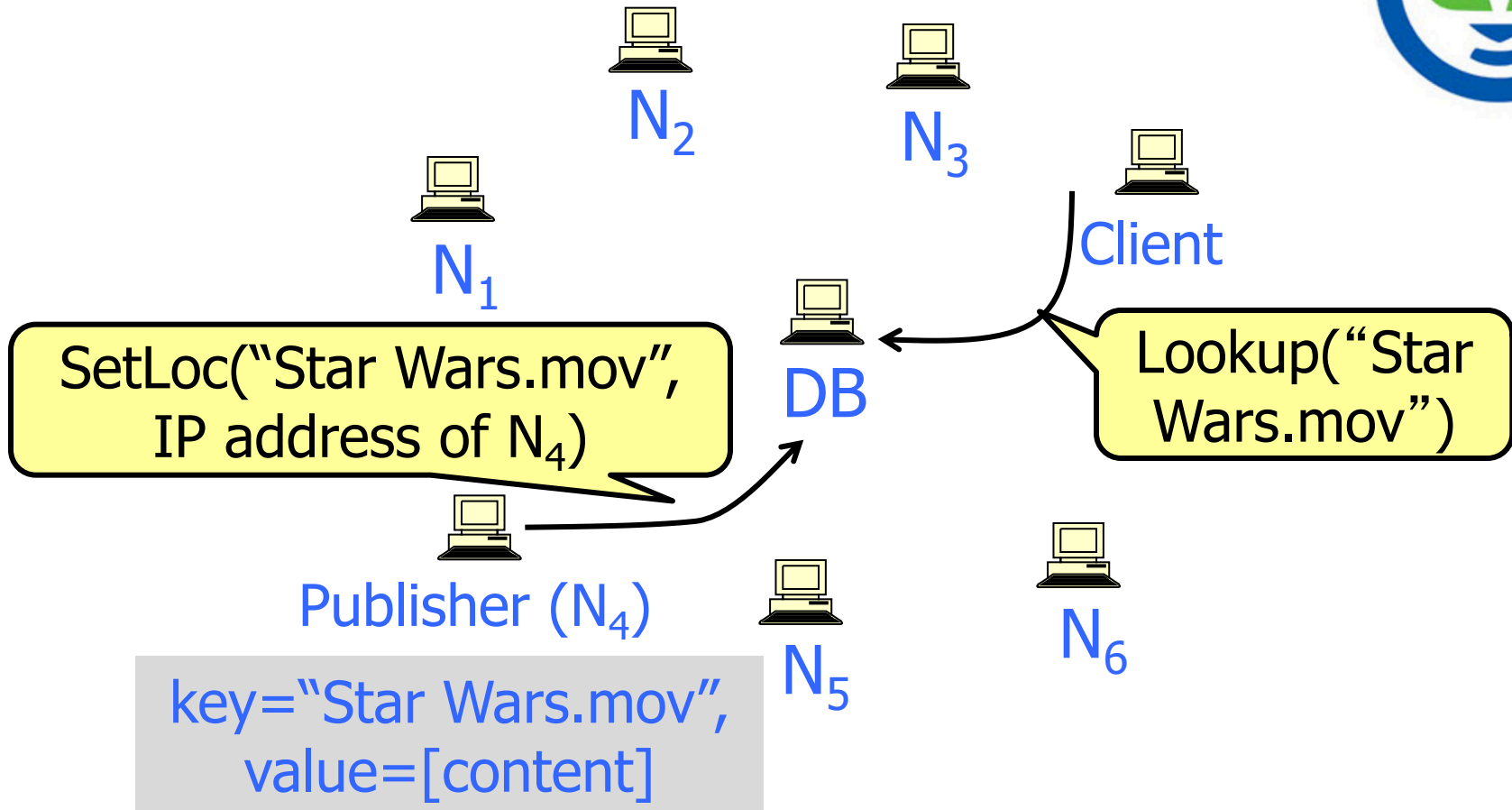
Basic Problem: Lookup

How to locate something given its name?

- Examples: file sharing, VoIP, CDNs...



Centralized Lookup (Napster)



Centralized Database

- ◆ **Join:** on startup, client contacts central server
- ◆ **Publish:** reports list of files to central server
- ◆ **Search:** query the server => return someone that stores the requested file
- ◆ **Fetch:** get the file directly from peer

Centralized DB: Pros and Cons

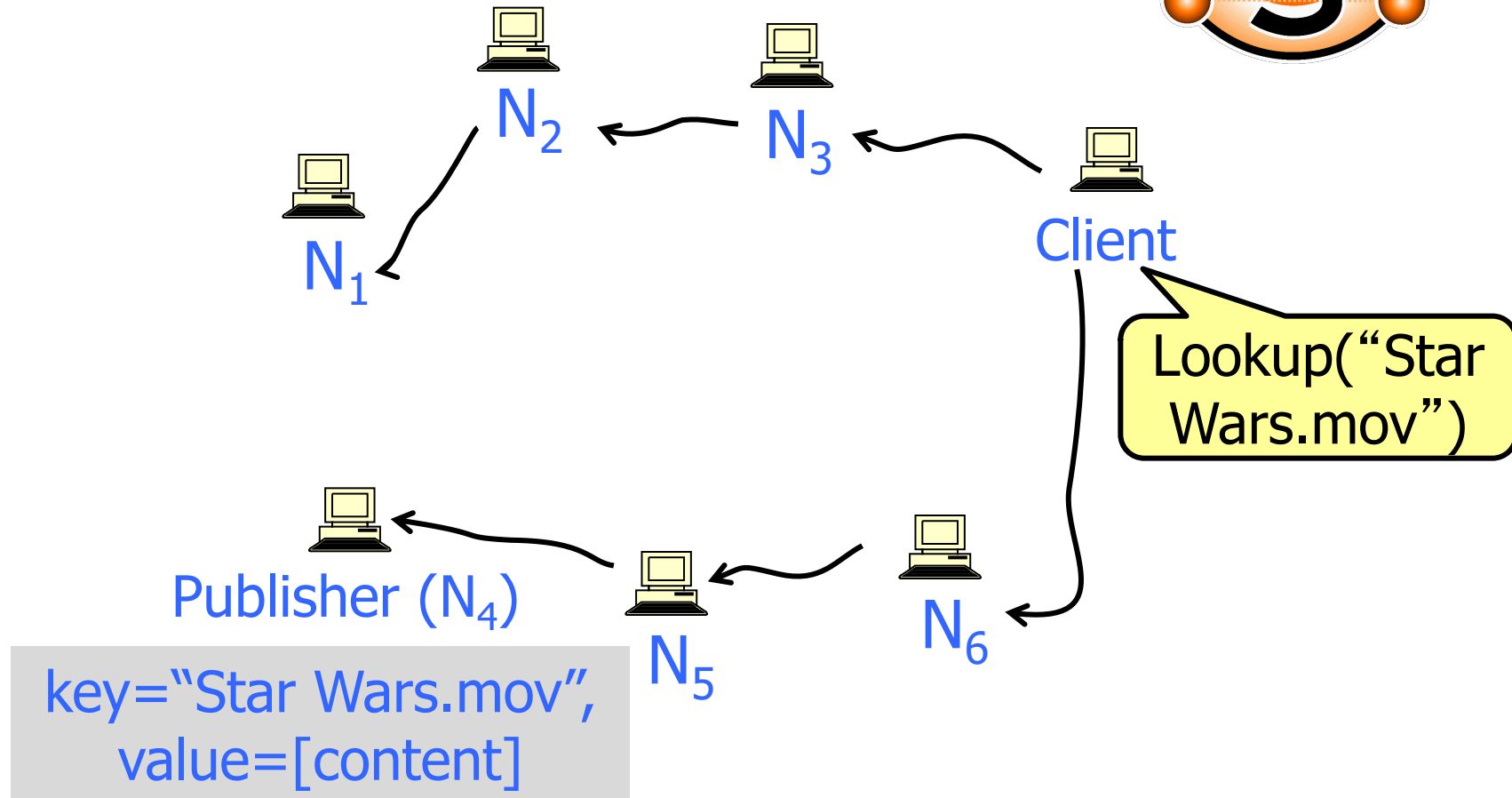
◆ Pros

- Simple
- Search scope is $O(1)$
- Controllable (pro or con?)

◆ Cons

- Server maintains $O(N)$ state
- Server does all processing
- Single point of failure

Flooding (“Old” Gnutella)



Query Flooding

- ◆ **Join:** on startup, client contacts a few other nodes; these become its “neighbors”
- ◆ **Publish:** no need
- ◆ **Search:** ask neighbors, who ask their neighbors, and so on... when/if found, reply to sender.
 - TTL limits propagation
- ◆ **Fetch:** get the file directly from peer

Query Flooding: Pros and Cons

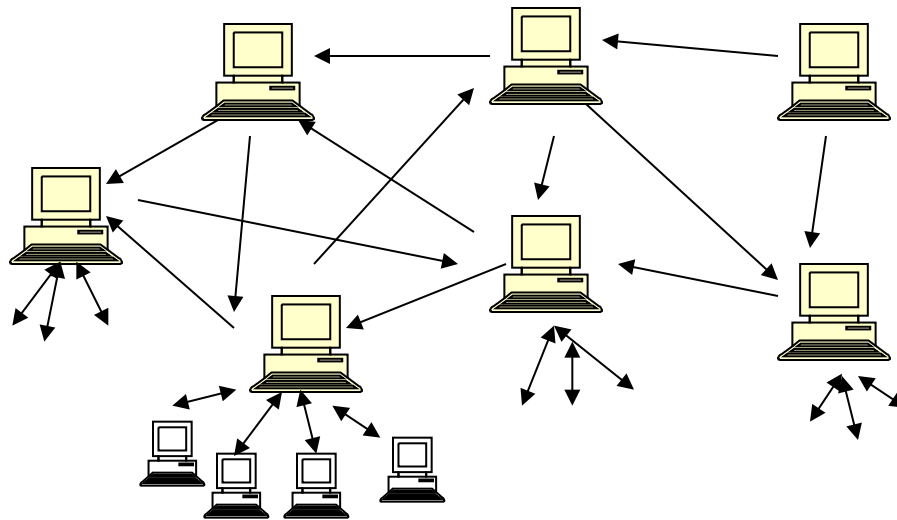
◆ Pros

- Fully de-centralized
- Search cost distributed
- Processing at each node permits powerful search semantics

◆ Cons

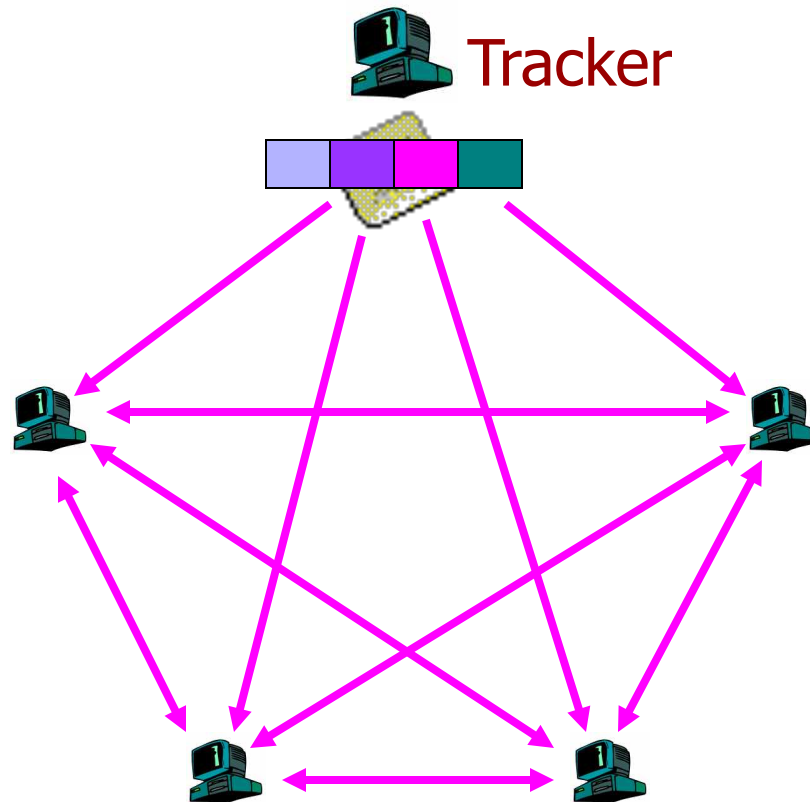
- Search scope is $O(N)$, a lot of traffic
- Search time is $O(???)$, no guarantee of results
- Nodes leave often, network unstable

Improvement: "Super Peers"



- ◆ Examples: Kazaa, Skype
- ◆ Lookup only floods super peers
- ◆ Still no guarantees for lookup results

BitTorrent: Publish/Join



© 2006 The Authors
Journal compilation © 2006 Blackwell Publishing Ltd



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
- ◆ Big differences from Napster
 - Chunk based downloading
 - “Few large files” focus
 - Anti-freeloading mechanisms

BitTorrent: Sharing Strategy

- ◆ Employ “tit-for-tat” sharing strategy
 - 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!
 - Also allows you to discover better peers to download from when they reciprocate
- ◆ Goal: Pareto Efficiency
 - Game Theory: “No change can make anyone better off without making others worse off”
 - Does it work? (not perfectly, but perhaps good enough?)

BitTorrent: Pros and Cons

◆ Pros

- Works reasonably well in practice
- Gives peers incentive to share resources; avoids freeloaders

◆ Cons

- Pareto Efficiency relative weak condition
- Central tracker server needed to bootstrap swarm
 - Alternate tracker designs exist (e.g., DHT-based)

DHT

- ◆ Goal: make sure that an item (file) identified is always found in a reasonable number of steps
- ◆ Abstraction: a **distributed hash table (DHT)** data structure
 - `insert(id, item);`
 - `item = query(id);`
 - Item can be anything: a data object, document, file, pointer to a file...
- ◆ Implementation: nodes in system form a distributed data structure
 - Can be ring, tree, hypercube, skip list, butterfly network, ...

Hash Tables

◆ Local hash table

- `key = Hash(name)`
- `put(key, value)`
- `get(key) → value`

◆ Constant-time insertion and lookup

◆ How to do (roughly) this across millions of hosts on the Internet?

Distributed Hash Tables

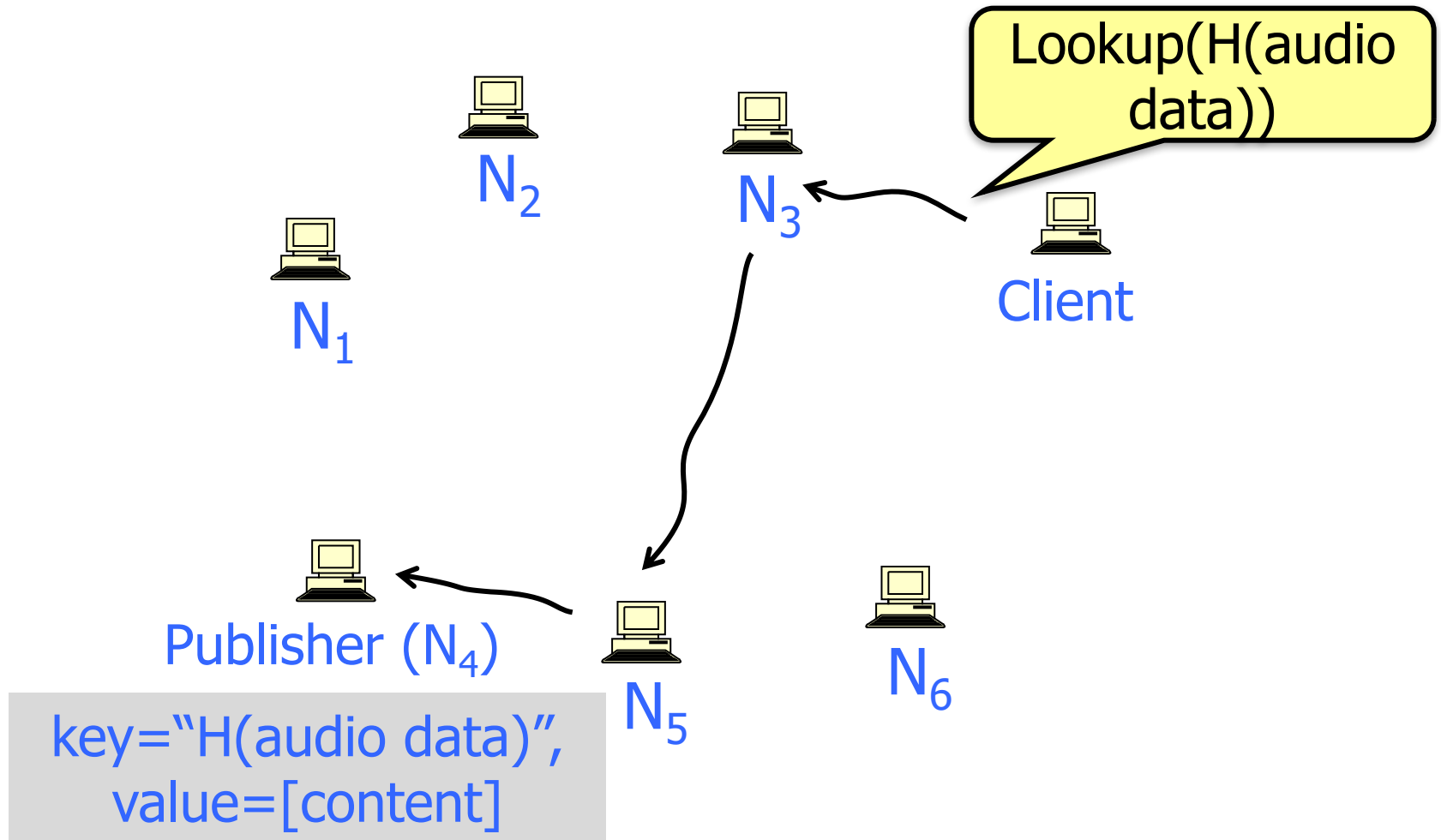
◆ Distributed hash table

- $\text{key} = \text{hash}(\text{data})$
- $\text{lookup}(\text{key}) \rightarrow \text{IP addr}$
- $\text{send-RPC}(\text{IP address}, \text{put}, \text{key}, \text{data})$
- $\text{send-RPC}(\text{IP address}, \text{get}, \text{key}) \rightarrow \text{data}$

◆ Partitioning data in truly large-scale distributed systems

- Tuples in a global database engine
- Data blocks in a global file system
- Files in a P2P file-sharing system

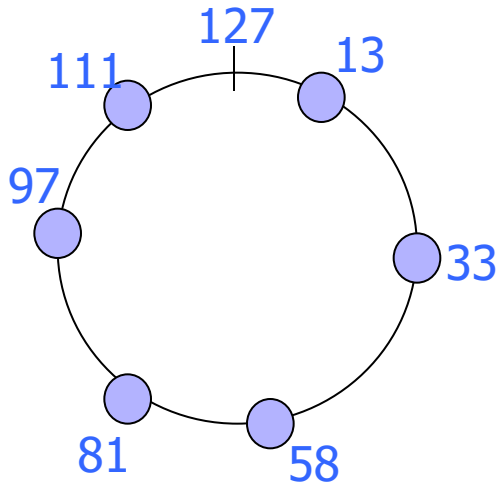
Routed DHT Queries



Structured Overlay Routing

- ◆ **Join:** Contact a “bootstrap” node, integrate self into the distributed data structure, get a node id
- ◆ **Publish:** Route publication for the file id toward a close node id along the data structure
- ◆ **Search:** Route a query for a file id toward a close node id
 - Data structure guarantees query will meet publication
- ◆ **Fetch**
 - Publication contains actual file => fetch from where query stops
 - Publication says “I have file X” => query tells you 128.2.1.3 has X, use IP routing to get X from 128.2.1.3

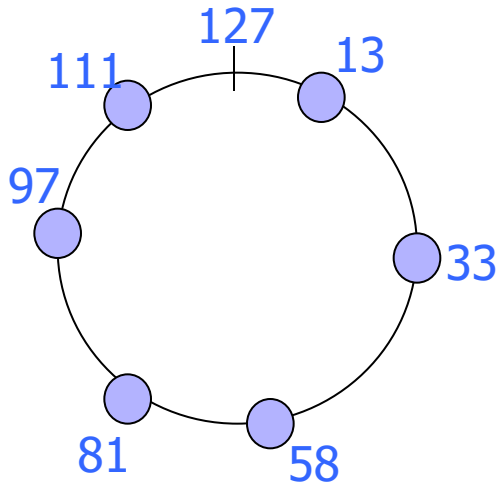
Basics of All DHTs




◆ Goal: build some “structured” overlay network with the following characteristics:

- Node IDs can be mapped to the space of hash key
- Given a hash key as a “destination address”, can route through the network to the right node
- Always route to the same node no matter where you start from

Simple Example (Doesn't Scale)



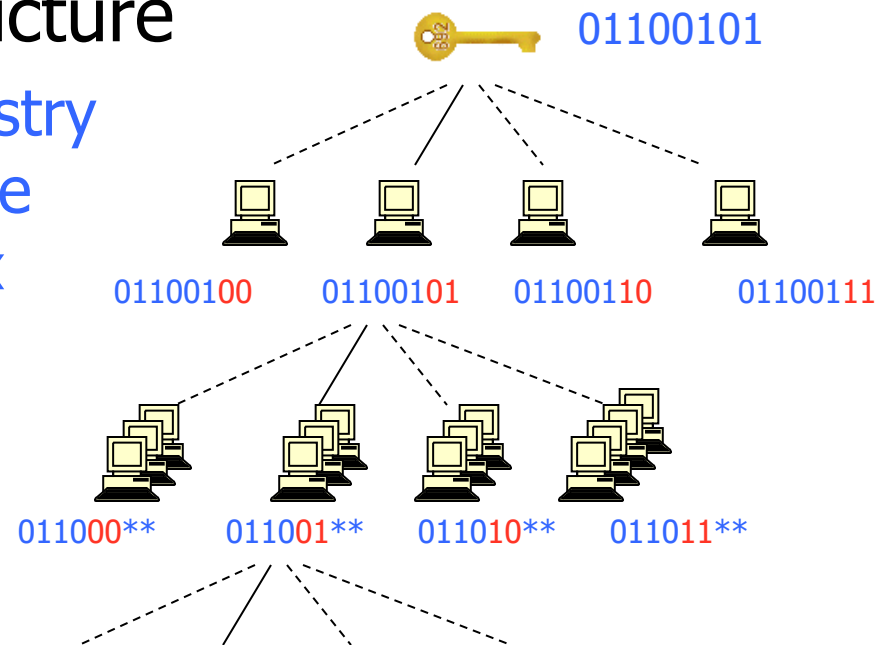
- ◆ Circular number space 0 to 127
- ◆ Routing rule is to move clockwise until current node ID  key, and last hop node ID < key
- ◆ Example: key = 42
- ◆ Obviously you will route to node 58 from no matter where you start

Scalable Routing

- ◆ Given document XYZ, choose node to use
- ◆ Need some notion of “closeness” between nodes

- ◆ Example: tree-like structure

- Pastry, Kademlia, Tapestry
- Distance = length of the longest matching prefix with the lookup key



Mapping Hashes to Nodes

Suppose we use modulo as a simple hash function

- ◆ Number nodes from 1...n
- ◆ Place document XYZ on node (XYZ mod n)
- ◆ What happens when a node fails? Or if different people have different measures of n?
 - $n \rightarrow n-1$
- ◆ Why might this be bad?

Consistent Hashing

[Karger '97]

“view” = subset of all visible hash buckets (nodes)

◆ Smoothness

- Little impact on hash bucket contents when buckets are added or removed

◆ Spread

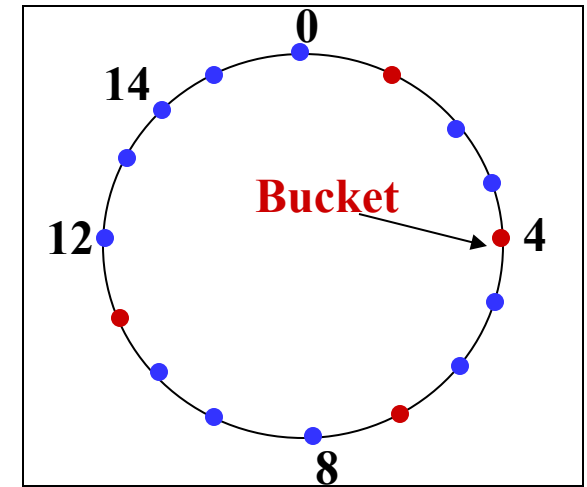
- Small set of hash buckets that may hold an object regardless of views

◆ Load

- Across all views the number of objects assigned to hash bucket is small

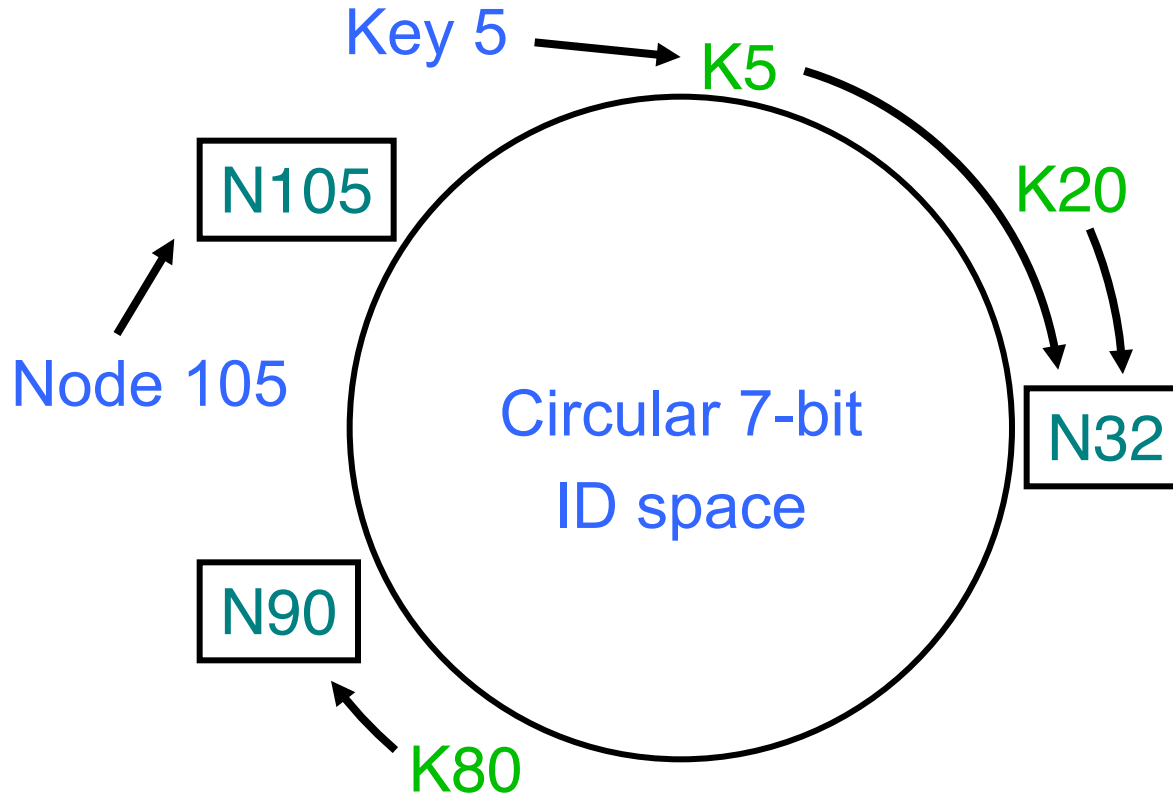
Consistent Hashing: Example

- ◆ Assign each of C hash buckets to random points on mod $2n$ circle, where hash key size = n
- ◆ Map object to random position on unit interval
- ◆ Hash of object = closest bucket



- Monotone → addition of bucket does not cause much movement between existing buckets
- Spread load → small set of buckets that lie near object
- Balance → no bucket is responsible for large number of objects

Consistent Hashing in DHT



Key is stored at the node with next-higher ID

DHTs: Pros and Cons

◆ Pros

- Guaranteed Lookup
- $O(\log N)$ per node state and search scope

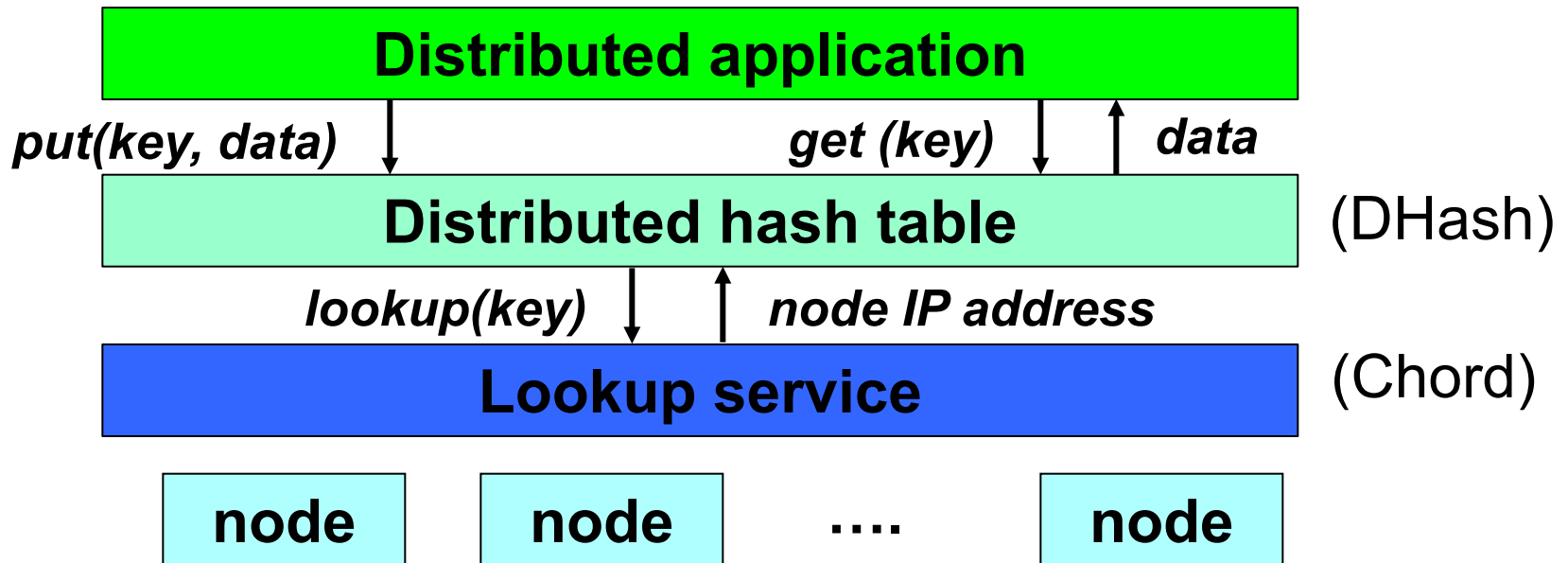
◆ Cons

- No one uses them? (only one file sharing app)
- Supporting non-exact match search is hard

DHT as Universal Service

- ◆ API supports a wide range of applications
 - DHT imposes no structure/meaning on keys
- ◆ Key/value pairs are persistent and global
 - Can store keys in other DHT values
 - ... and thus build complex data structures

Cooperative Storage with a DHT



App may be distributed over many nodes

DHT distributes data storage over many nodes

DHT History

- ◆ Original DHTs (CAN, Chord, Kademlia, Pastry, Tapestry) proposed in 2001-02
- ◆ Following 5-6 years saw proliferation of DHT-based applications:
 - Filesystems (eCFS, Ivy, OceanStore, Pond, PAST)
 - Naming systems (SFR, Beehive)
 - Application-layer multicast (Scribe, Bayeux, Splitstream)
 - Content distribution systems (Coral)
 - Distributed databases (PIER)

Why Didn't DHTs Succeed?

- ◆ High latency and limited bandwidth between peers
 - Compare: between server cluster in datacenter
- ◆ User computers are less reliable than managed servers
- ◆ Lack of trust in peers' correct behavior
- ◆ Churn
- ◆ Securing DHT routing hard, unsolved in practice

Why DHTs Got Right

◆ Consistent hashing

- Elegant way to divide a workload across machines
- Very useful in clusters: actively used today in Amazon Dynamo and other systems

◆ Replication for high availability, efficient recovery after node failure

◆ Incremental scalability: “add nodes, capacity increases”

◆ Self-management: minimal configuration

◆ Unique trait: no single server to shut down/monitor