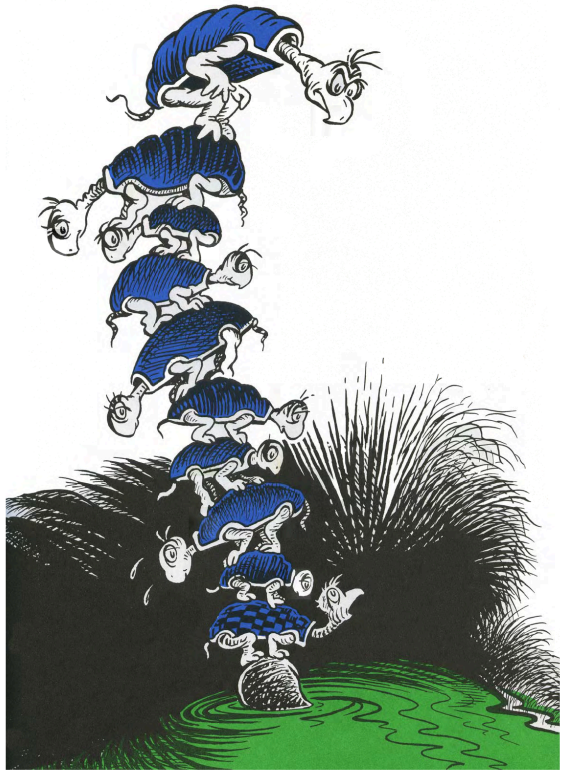
Overlay networks

To do …

- Today
- Overlay networks
- P2P evolution
- DHTs in general, Chord and Kademlia

Turtles all the way down



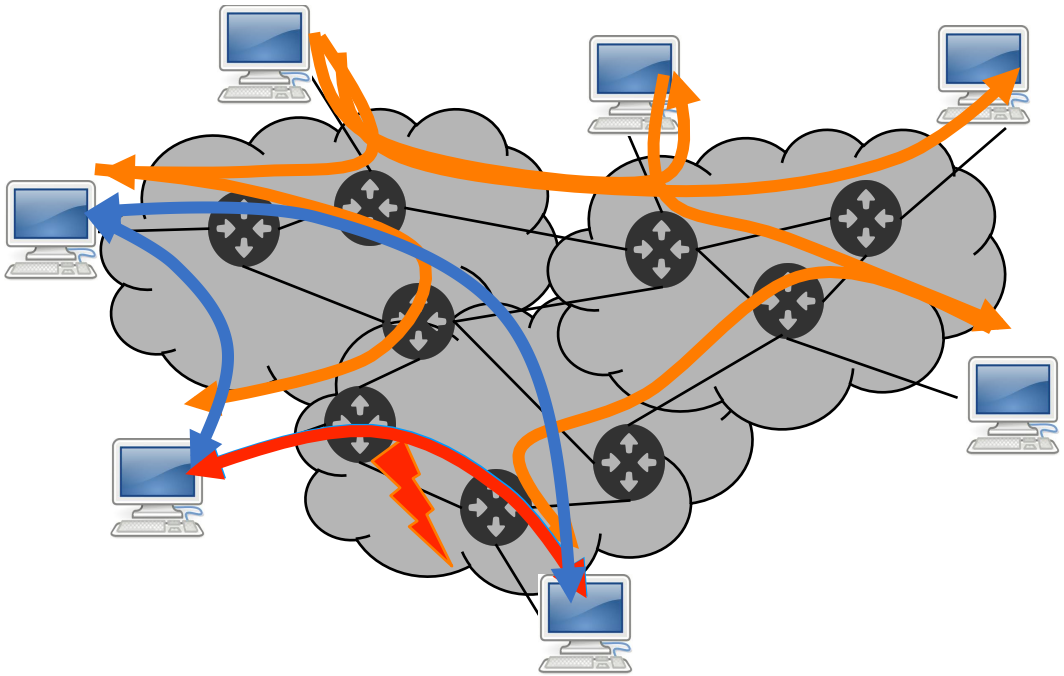
Overlay networks – virtual networks

- Different applications with a wide range of needs …
- Provide a service tailored to a class of applications
  - P2P file sharing, content distribution (CDNs)
- Support efficient operation in a given network environment
  - Wireless ad-hoc networks, delay tolerant networking (DNT)
- Add extra features such as multicast or secure communication
  - IPv6, (overlay) multicast, resilience (RON), mobility, security (VPN)

Overlay networks

- A logical network built on top of a physical network
  - Overlay links are tunnels through the underlying network
- Nodes are often end hosts
  - Intermediate nodes that forward traffic
  - Provide a service (e.g., storage, cpu)

Resilient Overlay Network (RON) as an example



Overlay networks

- Multiple overlays at once
  - Over the same underlying net, each providing its own particular service
- Who controls the nodes providing service?
  - The one who providing the service (e.g., Akamai)
  - A distributed collection of end users (e.g., P2P)
- The price to pay
  - Additional level of indirection
  - Opacity of the underlying network
  - Complexity of the network services

Peer-to-peer – The most common overlays

- Enabled by technology improvements in computing and networking
- A distributed architecture
  - No centralized control
  - Nodes are symmetric in function
  - Larger number of *unreliable* nodes
- Promise
  - *Reliability* – no central point of failure
    - Multiple replicas, geographic distribution
  - High capacity through parallelism
  - Automatic configuration
  - …

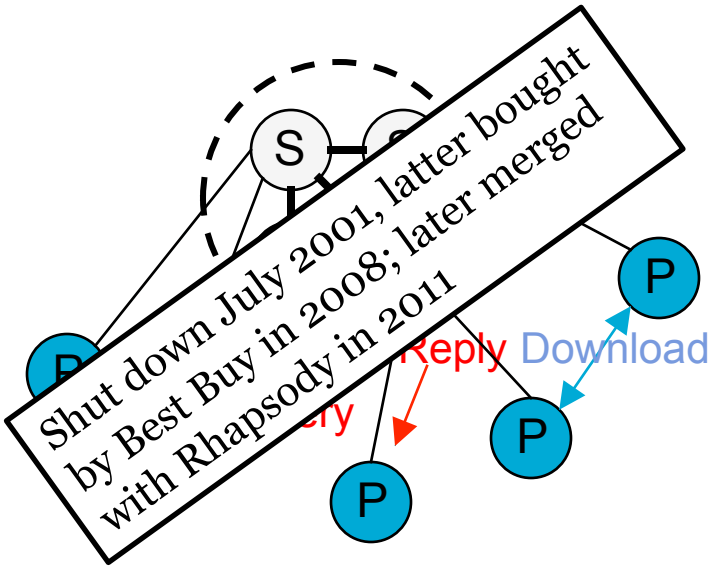
Three generations of P2P

- Many predecessors – DNS, Usenet, Grapevine, …
- Unstructured and centralized
  - Napster
- Unstructured and decentralized
  - Gnutella, Kazaa, …
  - Peers connect with other random peers
  - Semi-structured models (superpeers) for scalability
- Structured and decentralized
  - E.g. DHTs like Chord, Tapestry, Pastry, Kademlia and CAN

- *Key common service – placing and finding resources on an overlay*

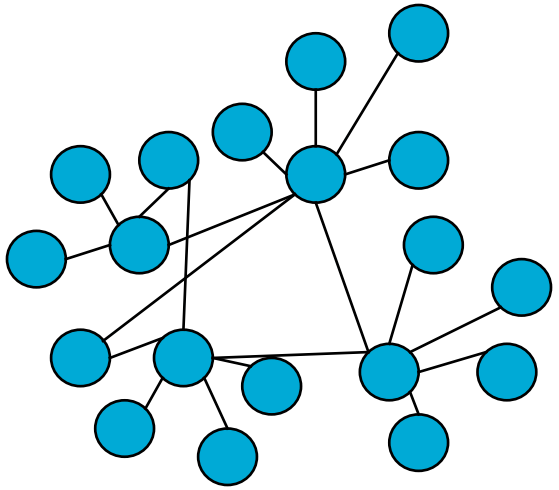
napster's legacy

- Grassroots service for sharing digital music files
- To find a file – a unified index of files being shared
- Cluster of central servers
  - Maintain index of files (replicated, weak consistency)
  - Monitor state of each peer in the system and maintains metadata (e.g. connection bandwidth)
  - Cooperate to process queries, returning a list of matching files and locations
- Peers
  - Maintains a connection to one of the central servers
  - Issue query to servers, and server request from peers



Gnutella – Unstructured and decentralized

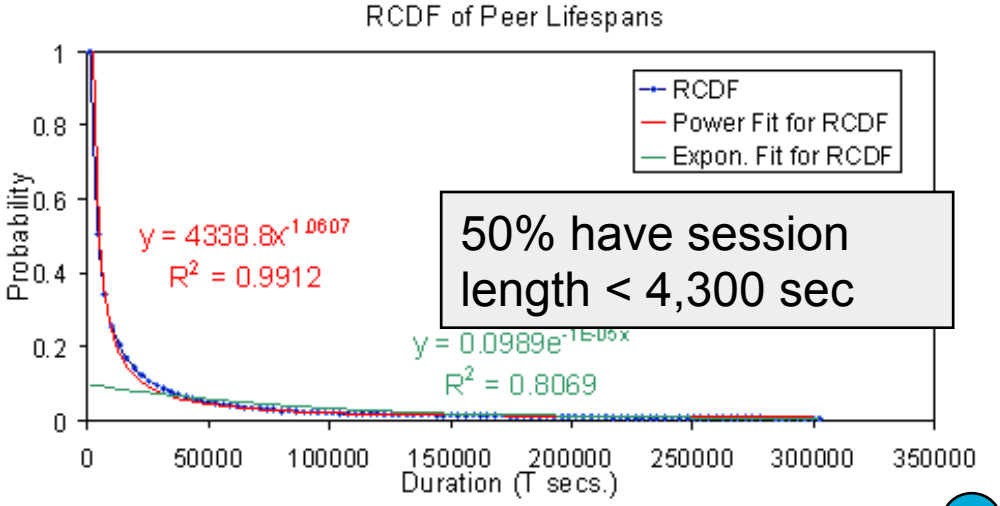
- Peers are all equal and can connect to anyone (V0.4)
  - Need a peer to get started
  - Connect and send pings; pings are forwarded among peers
  - Reply with pongs; new peers can then setup other connections

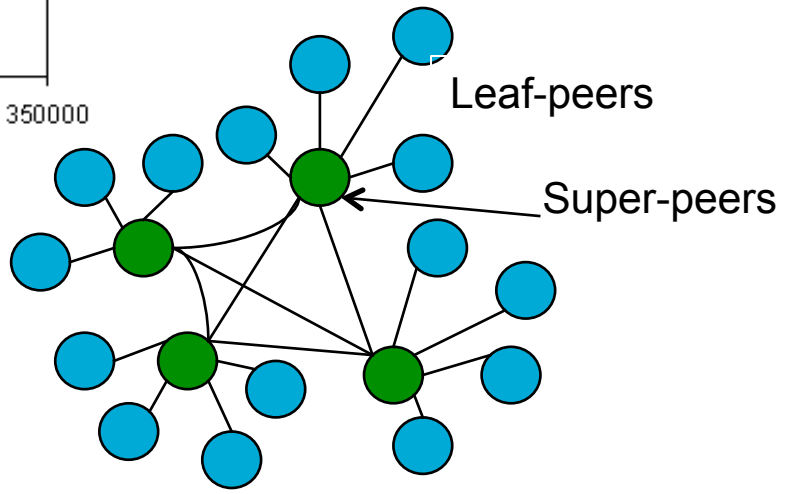


- No constraints on data object placement
- To find object - flooding or random walk
  - Query sent over TCP connections; peers forward it
  - QueryHit sent over the reverse path

Gnutella – Unstructured and decentralized

- A key issue – high overhead and churn
- To scale search and handle churn – super-peers (V0.6)
  - Super-peers form a core
  - Leaf-peers can only connect to super-peers





*Bustamante and Qiao. **Friendships that last: Peer lifespan and its role in P2P protocols**, In Proc of WCW, 2003.

Gnutella pros and cons

- Other issues …
  - Large search scope
  - Long search times
- Advantages
  - Fully decentralized – avoiding the issue with Napster
  - *Limewire was order to shut down October 2010, by November 2010 – Limewire Pirate edition*
  - Search cost distributed
  - Processing per node permits powerful search semantics

Skype – an example overlay

- Peer-to-peer VoIP
  - Developed by Kazaa in 2003, acquired by Microsoft in 2011 for US$ 8.5 billions
  - 34% of the call market share, 50 million concurrent online users in January 2013

- Notes on design*
  - Super-peer structure (super-peer selected based on availability, reachability, bandwidth, etc)
  - Users login through a well-known server, but connect to the network and others through super-peers
  - TCP for control, TCP or UDP for voice

*Baset & Schulzrinne's studies

Distributed Hash Tables (DHTs)

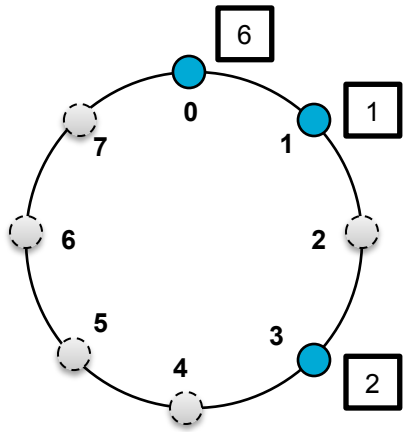
- Goal – quick retrieval, storage of <key, value> pairs
- General approach
  - Map node IDs to a (large) circular space
  - Map keys to same circular space
  - <key, value> pairs are stored in nodes with IDs that are close for some notion of closeness
- A simple interface
  - *put(key, value)*
  - *get(key)* ➔ *value*
- Two examples
  - Chord – one of the original DHTs; I. Stoica et al., SIGCOMM 2001
  - Kademlia – A popular second system; P. Maymounkov and D. Mazières, IPTPS 2002

Chord

- Basics
  - IDs space, m-bit long – 128-160 bits such as SHA-1
  - Identifiers are ordered in an identifier circle modulo 2^m
  - Key *k* "belongs" to nearest node – node with the smallest id ≥ *k; successor* of *k*

An id circle with $m = 3$
Three nodes: 0, 1 and 3

Key 2 is in node 3
pred(3) < 2 ≤ 3



- To resolve *k* to address of *succ(k)*
  - Simplest approach – go around the circle until it finds a node that succeed *k*
    - If *pred(p) < k ≤ p,* then *p* should return *p*'s address

Chord

- Short-cuts to speeds things up (not for correctness)
- Nodes keep a finger table of at most *m* entries
  - If FT_p denotes the finger table of *p*, $FT_p[i] = succ(p+2^{i-1})$
    - i.e., the *i-th* entry points to the first node succeeding *p* by at least 2^{i-1}
    - First entry, the immediate successor
  - FT entry contains Chord ID, IP and port
  - The first entry is *p* immediate *successor* on the circle
  - Shortcuts' distance increases exponentially with index

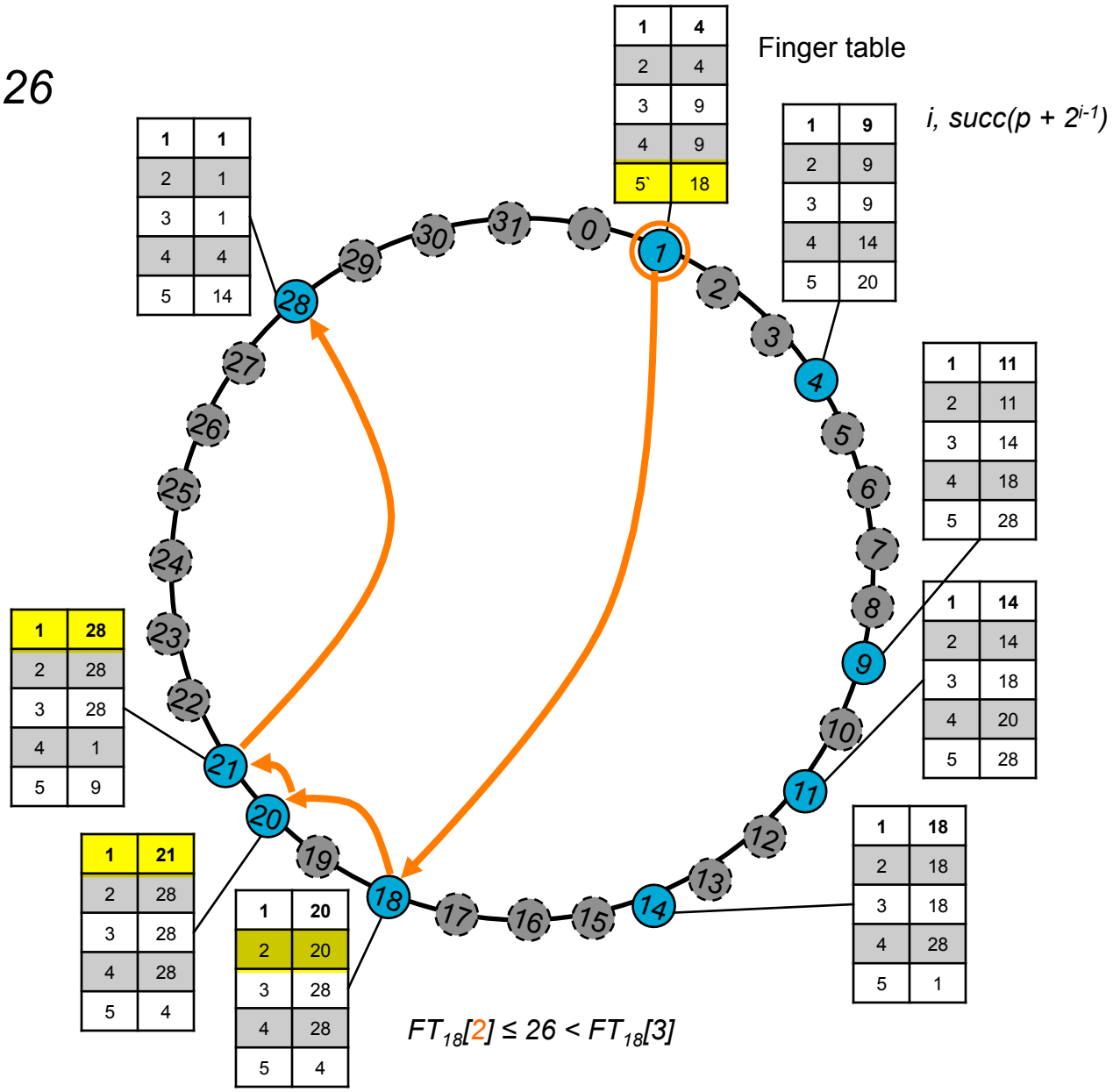
- Too look up key *k,* node *p* will forward request to node *q* with index *j* in *p's* FT where

$$q = FT_p[j] \leq k < FT_p[j+1]$$

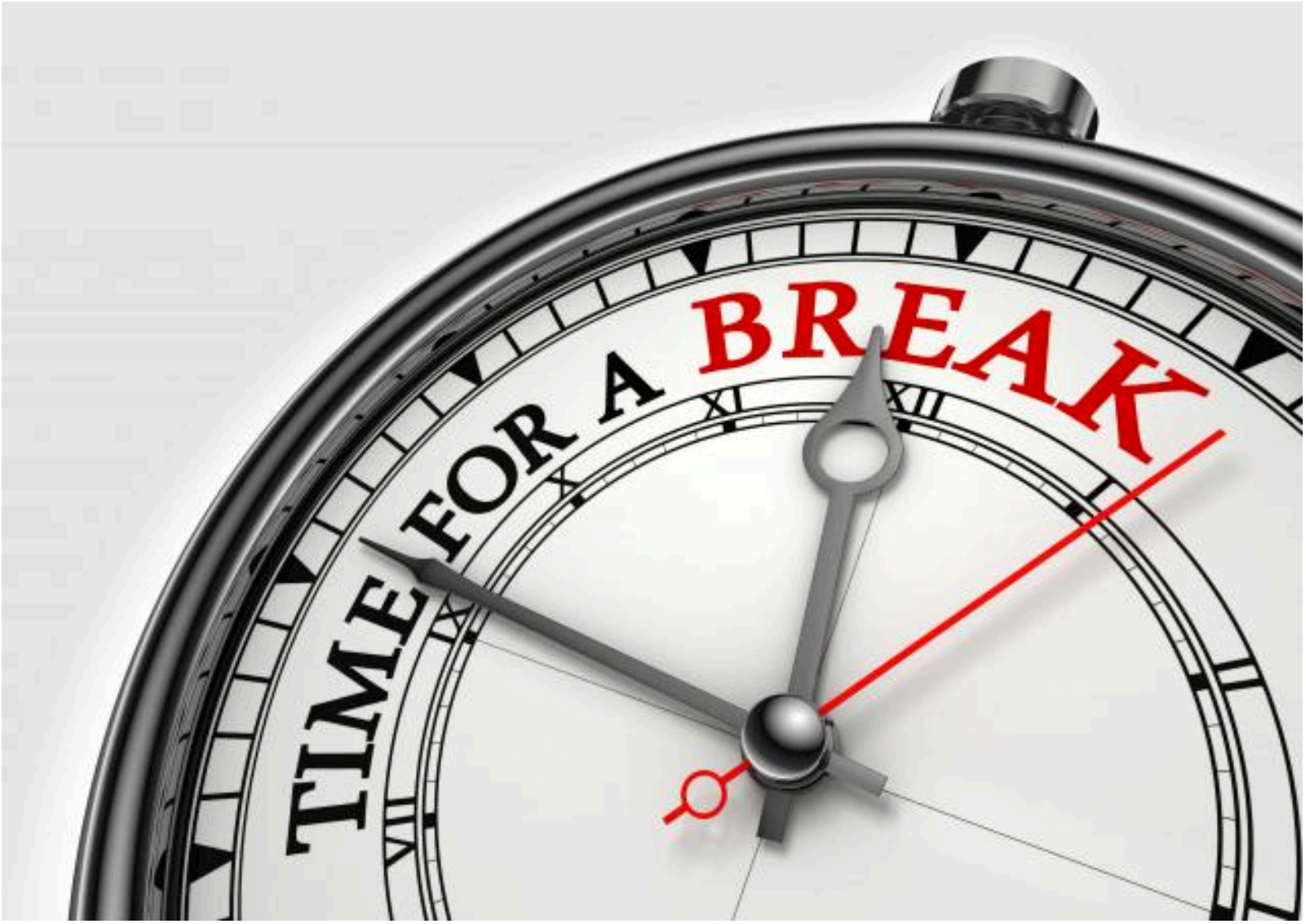
Resolving in Chord – example

Resolving *k = 26* from node 1

Done, now return address of node 28 to node 1

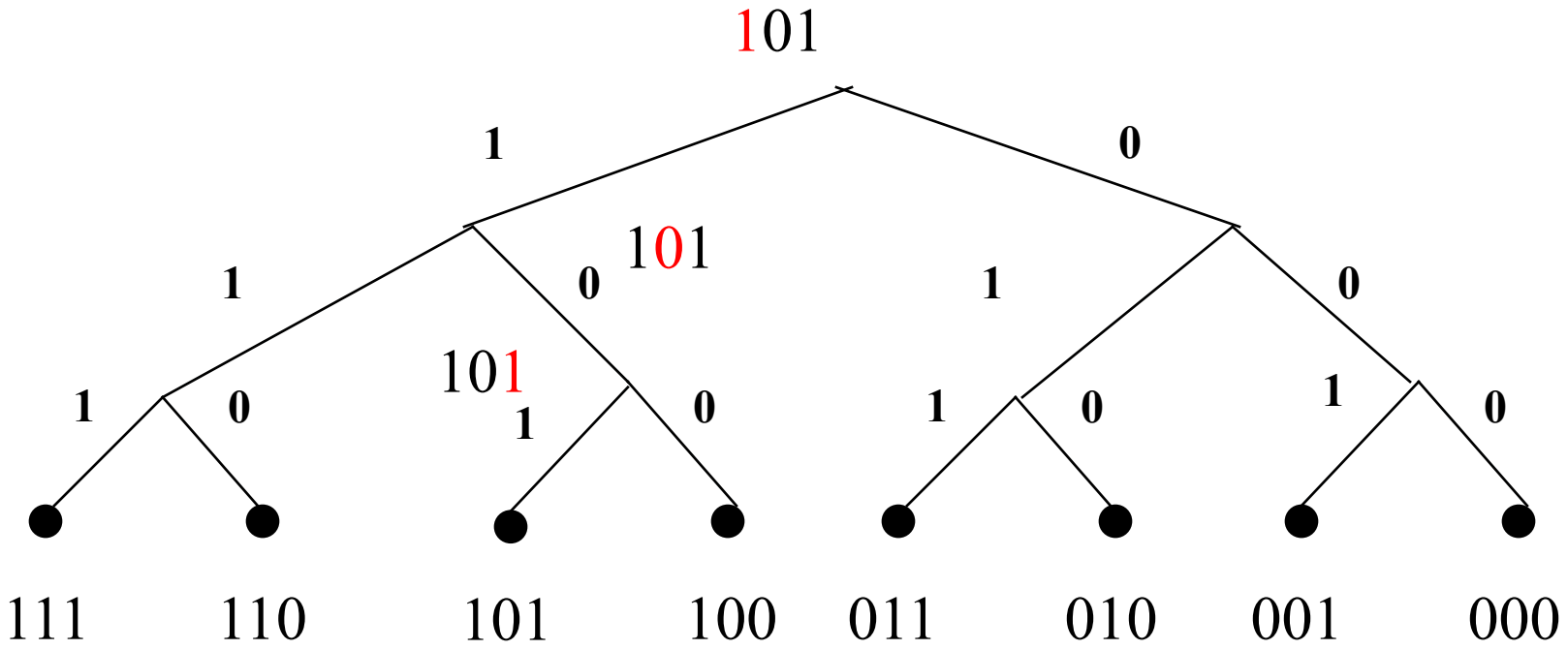


Back in 5’



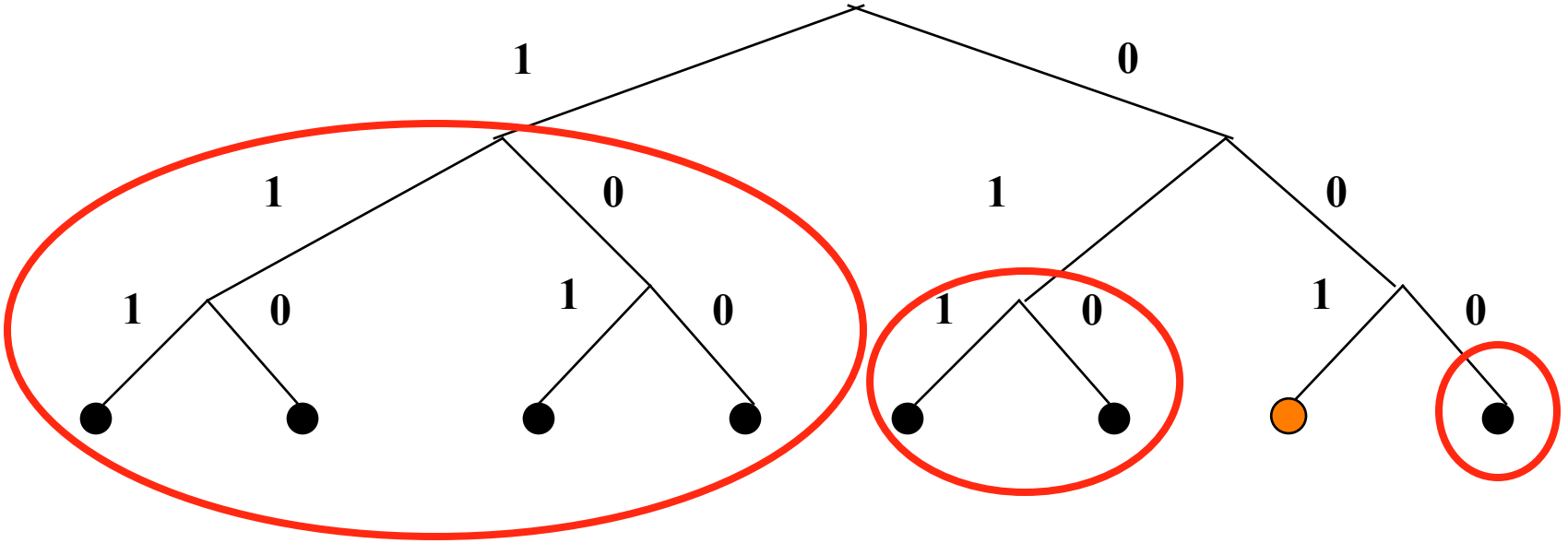
Kademlia DHT – A different model

- Treat nodes as leaves of a binary tree
- Node position determined by shortest unique prefix of its ID



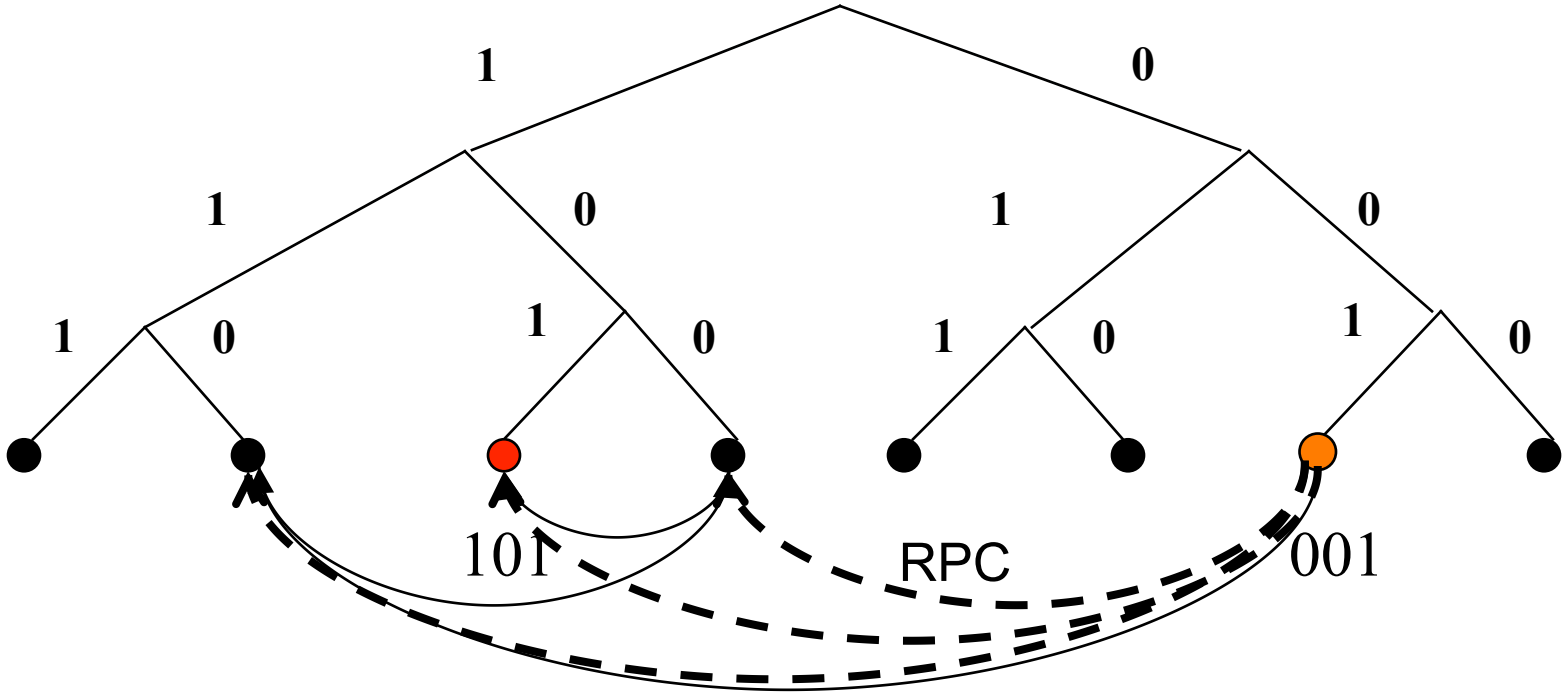
Kademlia

- Overarching idea …
  - For any given node, divide the tree into a series of successively lower sub-trees that don't contain the node
  - Ensures every node knows at least of one node in each of its sub-trees, if there is a node there



Kademlia

- Overarching idea …
  - To find a node of interest, successively query the best node you knows of to find a contact in lower and lower sub-trees
  - Every hop brings you into a smaller sub-tree around the target



The XOR metric

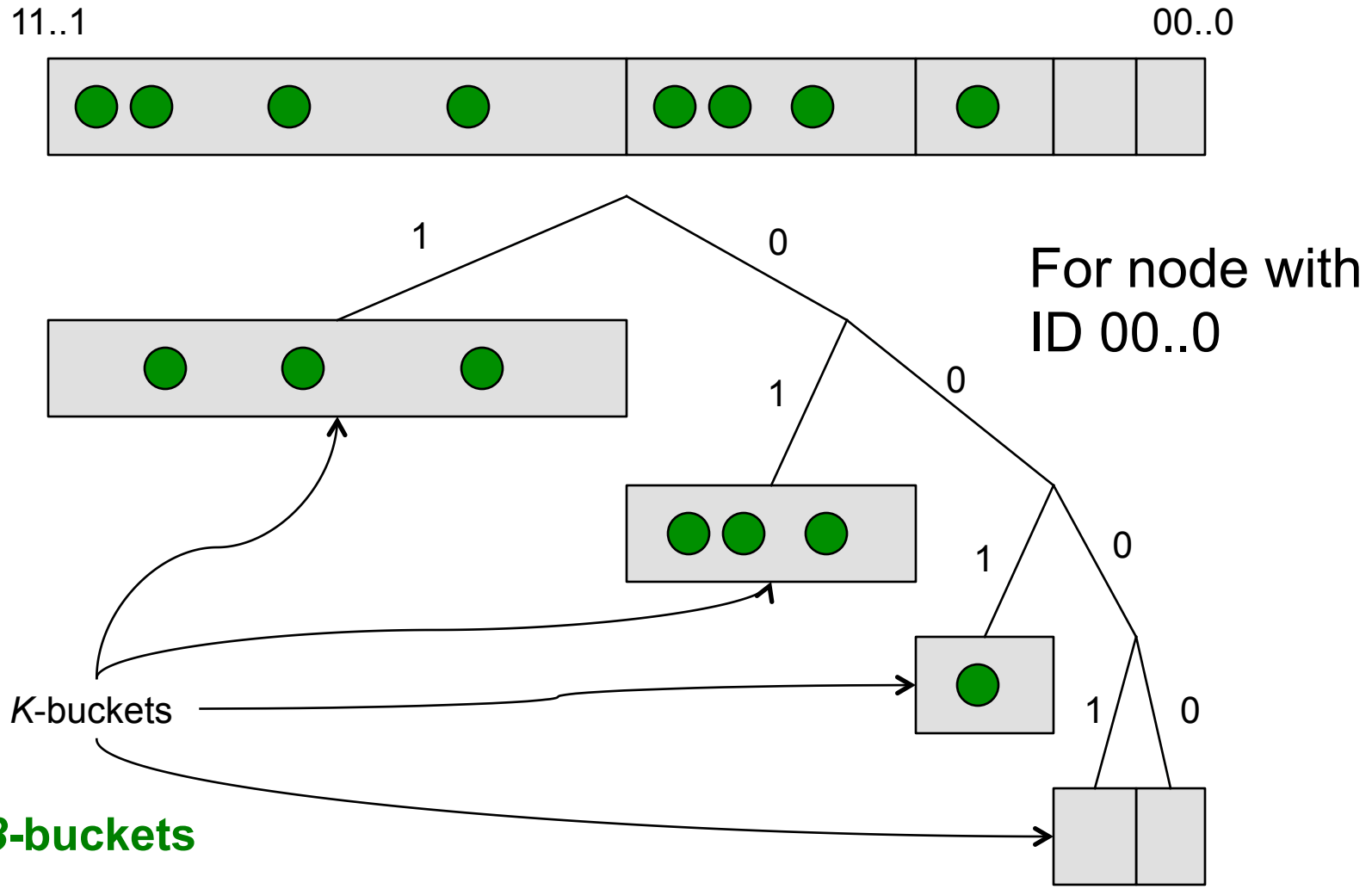
- Distance defined as the bitwise exclusive or
  - *d(x,y) = x XOR y*
- A valid metric
  - *d(x,y) > 0 if x ≠ y*
  - *d(x,y) = 0 iff x= y – Distance bet/ a node and itself*
  - *d(x,y) = d(y,x) - Symmetric*
  - *d(x,y) + d(y,z) ≥ d(x, z) – Triangle inequality property*
- Also
  - XOR is unidirectional – for each *x* and distance *t*, there is exactly one node y for which *d(x,y) = t*
    - So all lookups for the same key converge along the path …
    - Caching along the path can alleviate hot spots

x	y	xor
0	0	0
0	1	1
1	0	1
1	1	0

Node state

- For each i (0 ≤ i <160) every node keeps a list of `<IP-address, Port, Node-id>` triples for nodes of distance between 2^i and $2^{(i+1)}$ from itself
  - With 160 bits, 160 lists
  - Lists are called *k*-buckets
- Each k-list is sorted by time last seen, most recently seen peer at the tail (least at the head)
  - For high bit lists, there are a lot of options so they are easy to populate; low bit lists may be empty
- A list can grow up to size *k*, a system-wide replication parameter
  - Chosen so that any give set of *k* nodes is unlikely to fail within an hour (e.g., 20)

Routing table data structure



Maintaining k-buckets

- Due to XOR topology's symmetry
  - The distribution of nodes that call us is the same as that of contacts we need for our routing tables
- Whenever a node receives a message, the appropriate k-bucket (for the sender's id) is updated

```
If sender is already in recipient's k-bucket
    Move it to the tail // Most recently seen
Else
    If k-bucket is not full, insert at the tail
    Else // If k-bucket is full …
        Ping the least-recently seen node
        If it does not respond, evict it and insert
        sender at the tail
        Else discard sender
```

- A least-recently seen eviction policy, except that live nodes are never removed
  - Based on distribution of peers' lifespan

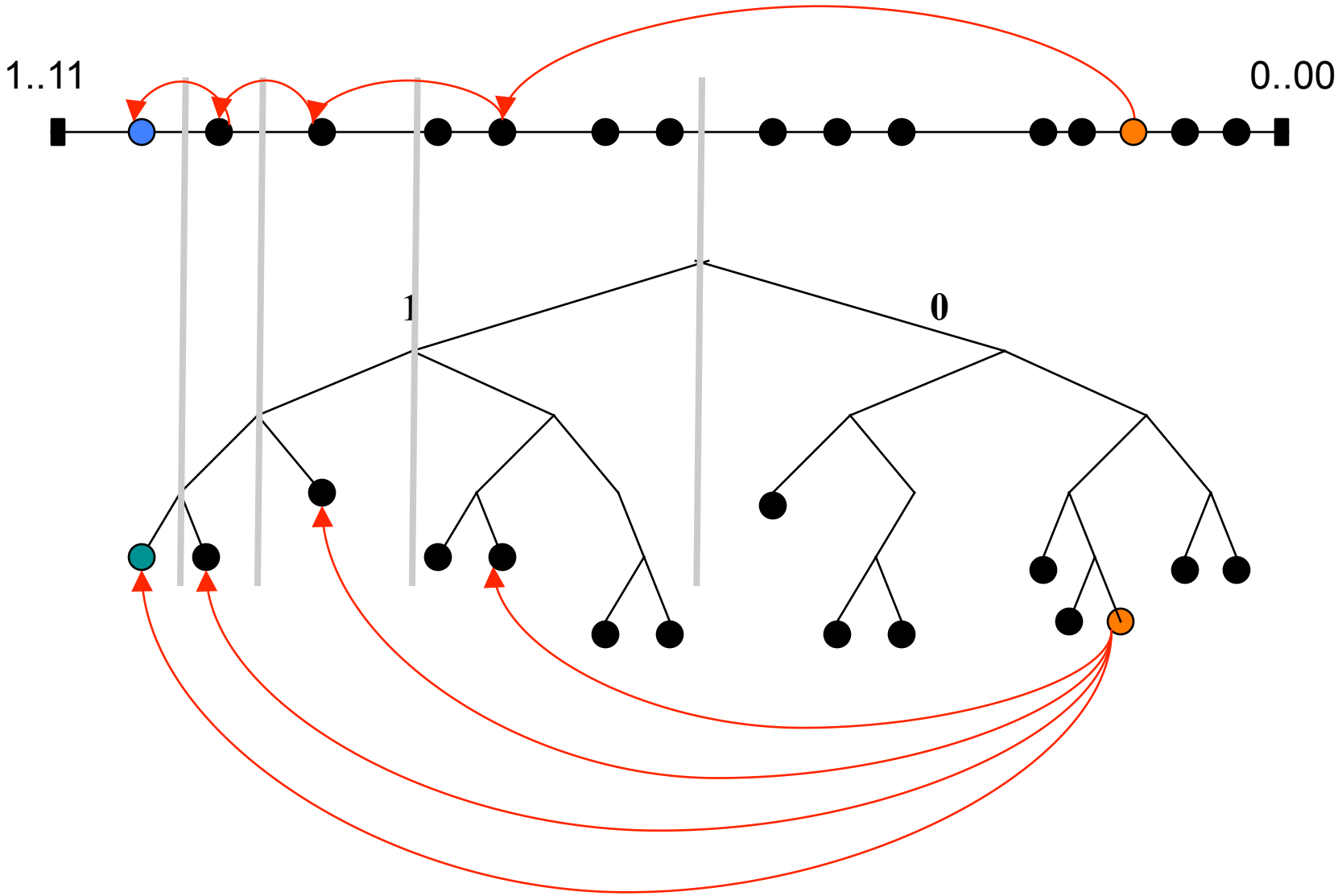
Joining, leaving and refreshes

- Node join
  - Borrow some contact from an already online node
  - Lookup itself
  - Cost of join is O(log n) messages
- Node leave – nothing to do
  - Good considering nodes that may disconnect multiple times during a long online session
- Hourly k-bucket refreshes (if necessary)

Node lookup algorithm

- FIND_NODE$_n$: takes an ID as an argument, a recipient *n* returns `<IP address, UDP port, node-id>` of the *k* nodes that it knows about closest to target ID
  - From same or multiple k-buckets (if the closest one is not full)
- Most important procedure – lookup
  - Find the *k* closest nodes to a given target *T*
  - n_0 – ourselves (the node that is performing the lookup)
  - N_1 – find_node$_{n0}$(T)
  - N_2 – find_node$_{n1}$(T)
  - …
  - This ends when N_l contains no contact that have not been called already
- n_i is any contact in N_i
- Describe as recursive but really iterative

Simple lookup



Concurrent lookup

- Trade bandwidth for lower latency lookups
- Goal
  - Route through closer/faster machines
  - Avoid delays due to timeout
- Idea – perform $\alpha > 1$ (e.g., 3) calls to $\text{find\_node}_n(T)$ in parallel
- When $\alpha = 1$, lookup resembles that in Chord in terms of message cost, and the latency of detecting the failed nodes
- Unlike Chord, Kademlia has the flexibility of choosing any one of the *k* nodes in a bucket, so it can forward with lower latency

Kademlia protocol

- PING: to test whether a node is online
- FIND_NODE: takes an ID as an argument, a recipient returns <IP address, UDP port, node id> of the *k* nodes that it knows from the set of nodes closest to ID
- FIND_VALUE: behaves like FIND_NODE, unless the recipient received a STORE for that key, it just returns the stored value
- STORE: instruct a node to store a <key, value> pair for later retrieval

Find a <key, value>

- To find a <key,value> pair, perform a lookup of k closest nodes to the key
- Value lookup use FIND_VALUE, rather than FIND_NODE
- Procedure halts when any node returns the value
- Once a lookup succeed, requesting node stores <key, value> pair at the closest node observed that did not return the value
- Given unidirectionality of topology, future search will likely hit the cache

Store <key, value>

- Most operations implemented in terms of the lookup procedure
- To store a <key,value> pair, a participant locates the k closest nodes to the key and sends them STORE RPCs
- Each node re-publishes <key,value> pairs as necessary to keep them alive
  - For file sharing application, the original publisher is required to republish it every 24 hours (otherwise it expires)

Kademlia today

- One of many DHTs: Chord, Pastry, Tapestry, CAN …
- Current de facto standard algorithm for P2P search
- Used by LimeWire, Gnutella, Overnet, EDonkey2000, eMule, BitTorrent, …
- The TDL4 botnets is based on Kad, which is based on Kademlia

Summary

- New applications with new demands on the underlying network
- Architectural changes are, at best, difficult
- Overlays both as a path to deployment and an experimental testbed
  - Deploying narrow fixes?
  - No demands on underlying network (to ensure deployment)
- From grassroots efforts and research labs to products
  - But much research to be done
- Future Internet and overlays?