Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# Chapter 5 addendum
# Dynamic Host Configuration Protocol


Computer Networks

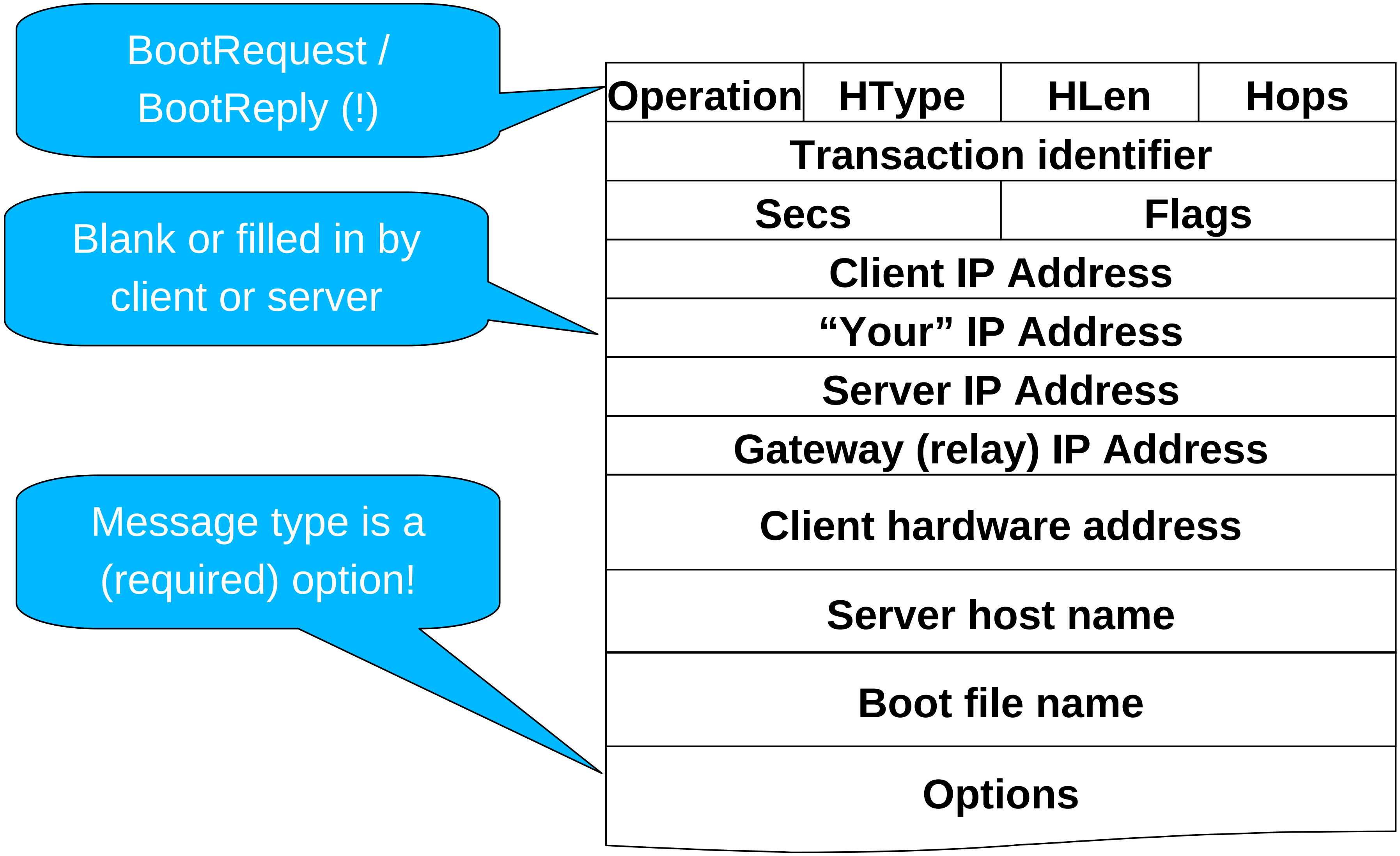
Timothy Roscoe

Summer 2007

# *How do I find my IP Address?*

- Dynamic Host Configuration Protocol (DHCP)
  - RFC 2131
  - Surprisingly complex (45 pages!)
  - Surprisingly flexible (lots of configuration & reconfiguration)
- Required client can still be very small
  - No TCP required (and not much UDP/IP)
- Basic idea:
  - One *or more* configuration servers
  - Client broadcasts MAC address, Servers all reply
  - Client chooses a configuration (*lease)*
  - Periodically renewed (soft state)

# *DHCP message format (one size fits all!)*



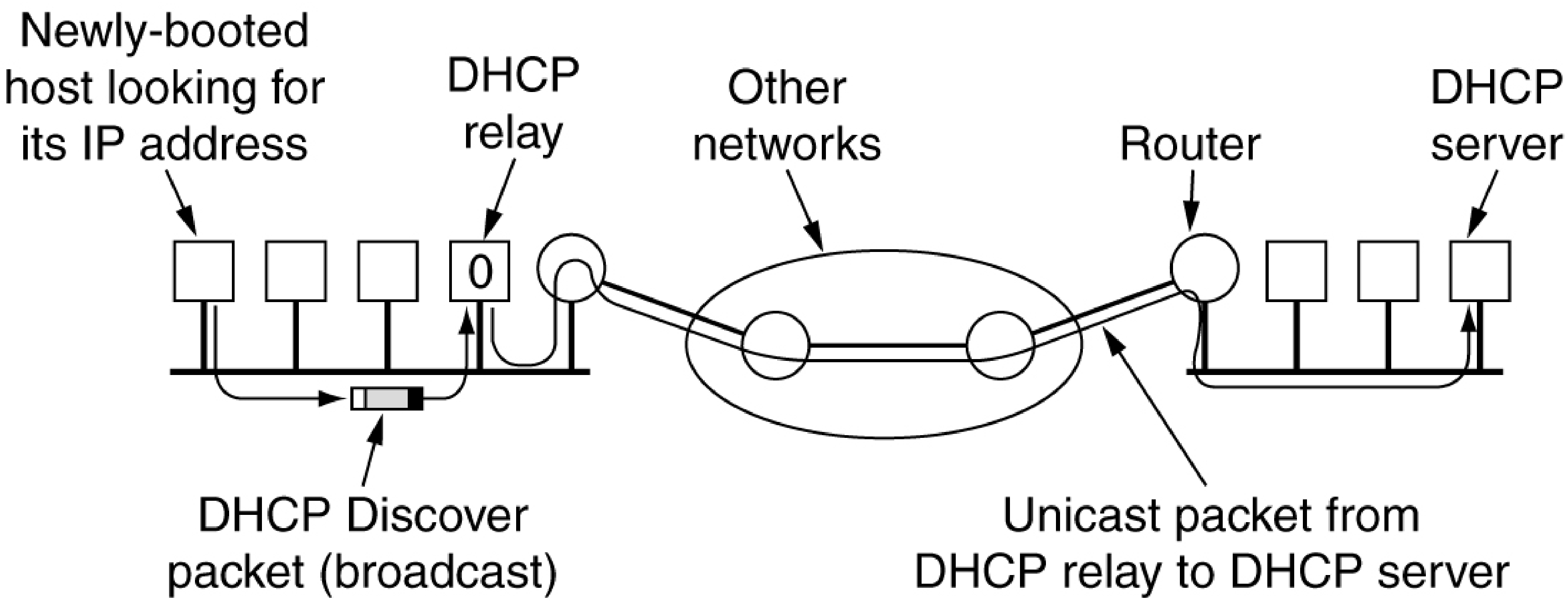
# *DHCP in operation: getting an IP address*

1. Client broadcasts DHCPDISCOVER on physical network (using UDP!)
2. Server unicasts DHCPOFFER to client with possible address (and other configuration info)
3. Client sends DHCPREQUEST asking for configuration
4. Server sends DHCPACK granting the address

- Extra options can specify e.g.:
  - DNS resolver, DNS name
  - Routing gateway/subnet

Best illustrated by example…

# *DHCP Relaying (really remote booting)*




•

# *Wider class of "bootstrapping protocols"*

- Things to ponder:
  - Why two phases (DISCOVER *and* REQUEST)?
  - Is this a link, networking, or application level protocol?

- General problem: how do I find out what I need to communicate?
  - C.f. ARP
  - Such protocols often break layering models

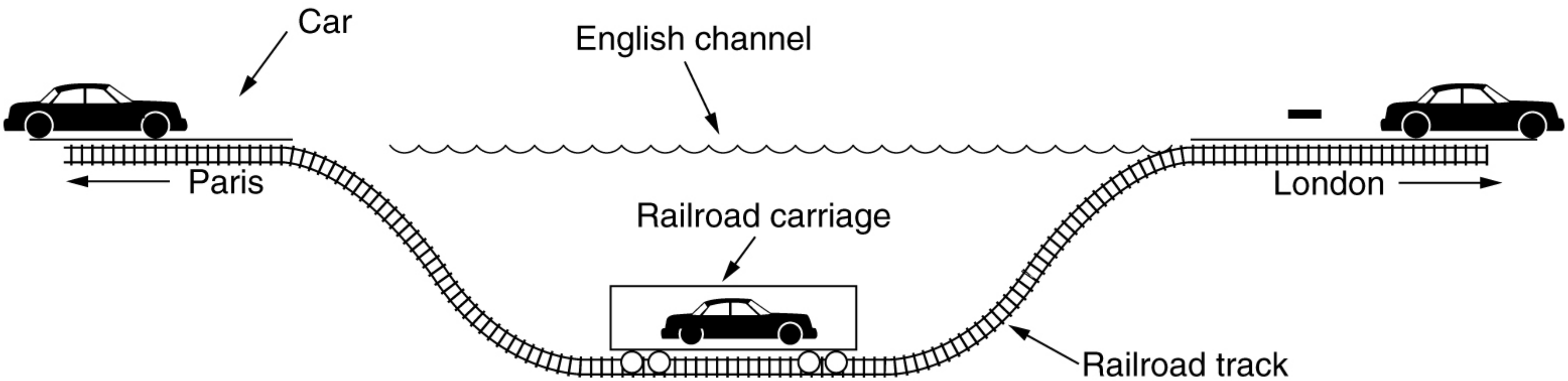
# *Chapter 7*
# *TUNNELS, OVERLAYS and P2P*

Computer Networks

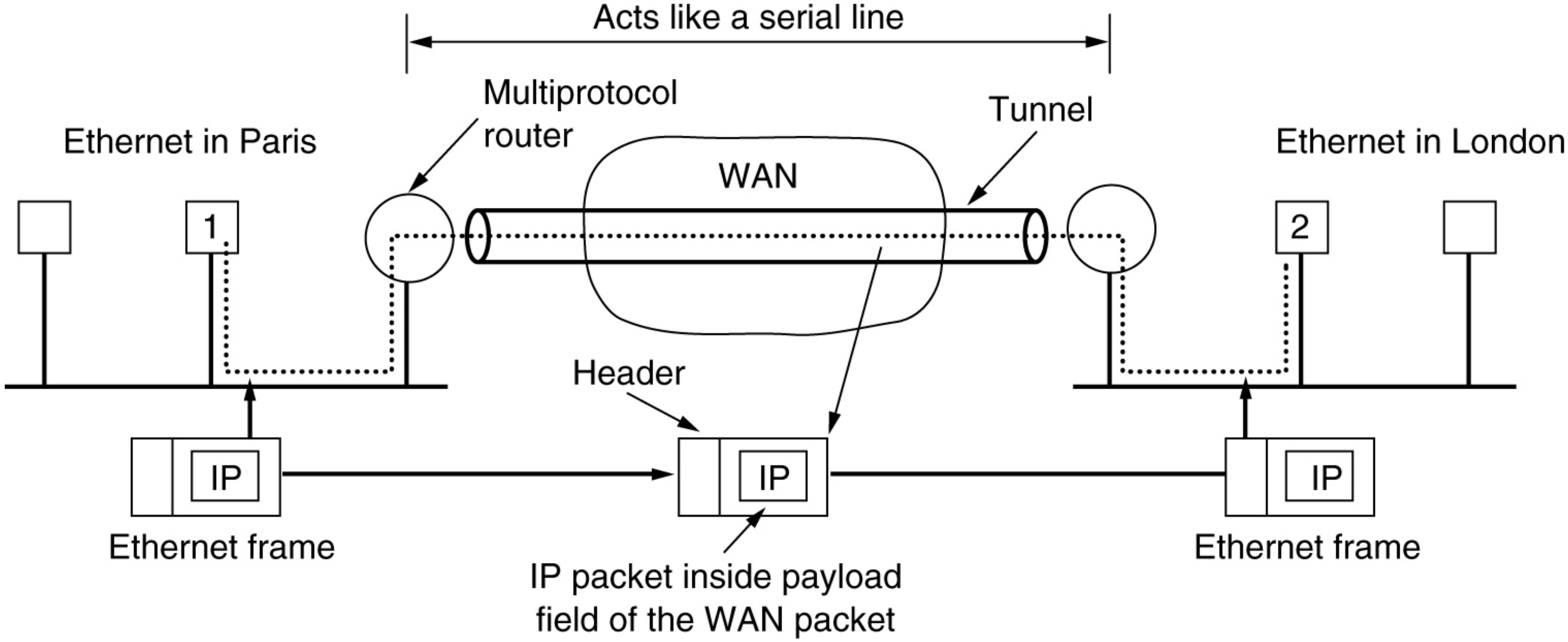
Timothy Roscoe

Summer 2007

# *Tunnelling: the basic idea (from Tanenbaum)*



# *Tunnelling: the basic idea*




- *Tunnelling protocol:*
  - Encapsulates packets (link, network, transport)
  - Runs over another protocol (network, transport, application)
- Seen examples already…

# *Tunnelling: a few examples*

Protocol below

| Protocol above | Network | Transport | Application |
|---|---|---|---|
| Application | | SOCKS<br>SSL/TLS | |
| Transport | IPsec Transport | SSH tunnels | HTTP CONNECT |
| Network | MPLS<br>GRE<br>IPIP<br>IPsec Tunnel | L2TP<br>PPTP | IP over DNS |
| Link | LANE | | |

Also: layers are approximate!

# *Tunnelling: Why?*

- Many uses!  A few examples:
  - Personal routing / NAT avoidance
  - Mitigating DDoS attacks
  - Traffic aggregation and management
  - Security
  - Mobility

# *ssh tunnels (port forwarding)*

```
ssh –L8080:netos-int.inf.ethz.ch:80 \
        cixous.inf.ethz.ch
```

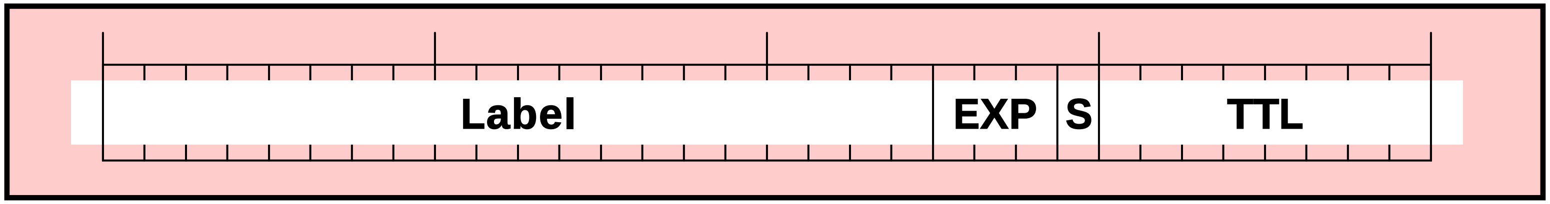
- Accept connections to local host port 8080 and forward them through **`cixous.inf.ethz.ch`** to **`netos-int.inf.ethz.ch`** port 80

```
ssh –R4321:slimserver.home-net.org:9000 \
        cixous.inf.ethz.ch
```

- Accept connections to port 4321 on **`cixous.inf.ethz.ch`** and forward them back through local host to port 9000 on **`slimserver.home-net.org`**

# *MPLS: Multiprotocol Label Switching*

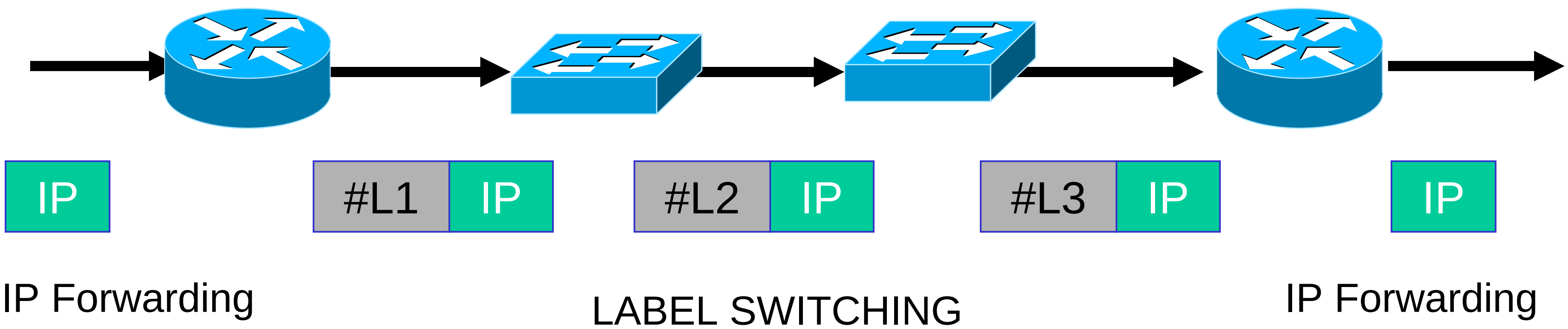
- 32-bit header (over IP at least):



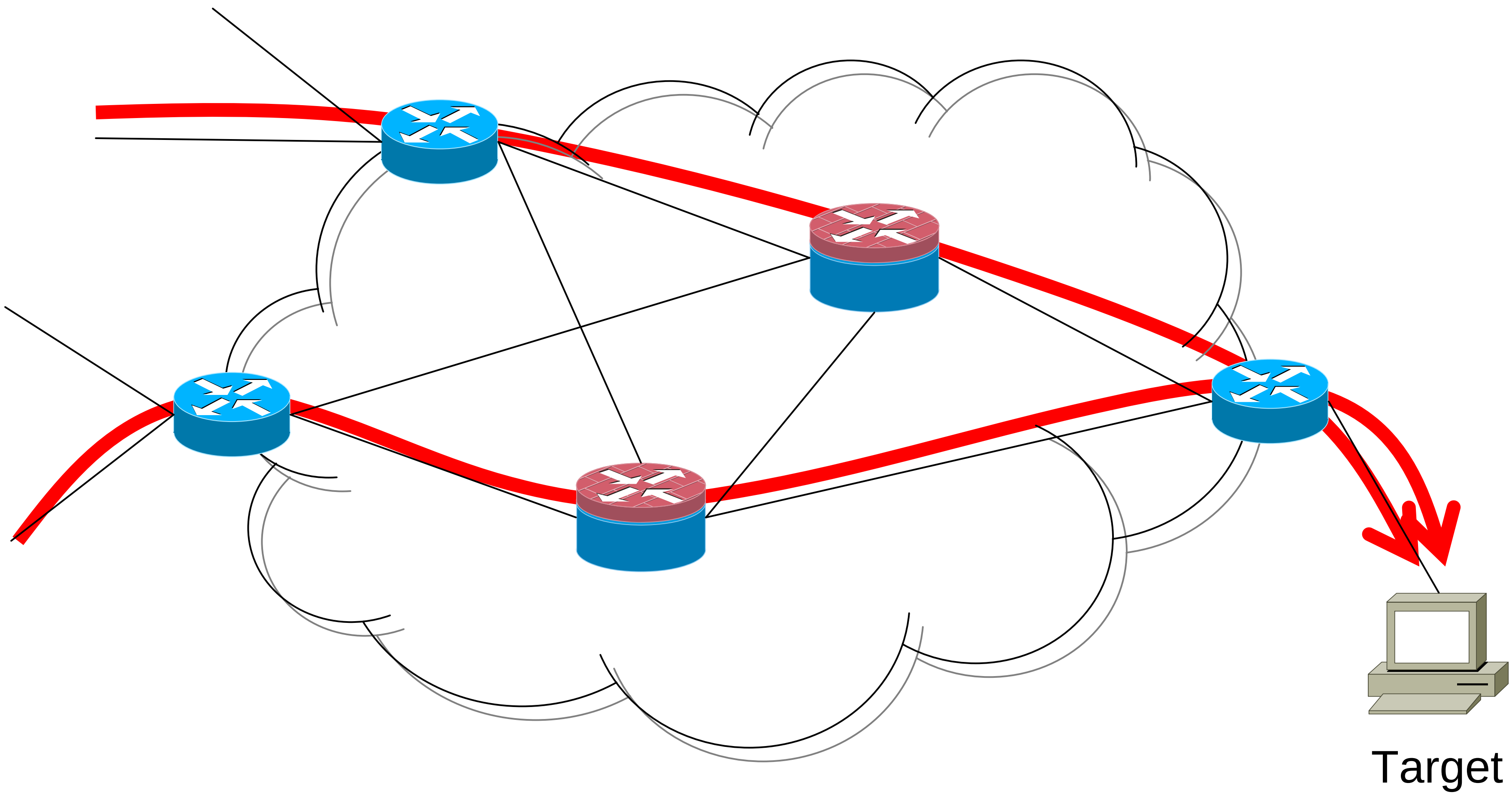

- label
  – used to match packet to LSP
- experimental bits
  – carries packet queuing priority (CoS)
- stacking bit: can build “stacks” of labels
  – goal: nested tunnels!
- time to live
  – copied from IP TTL

# *MPLS: Label switching*

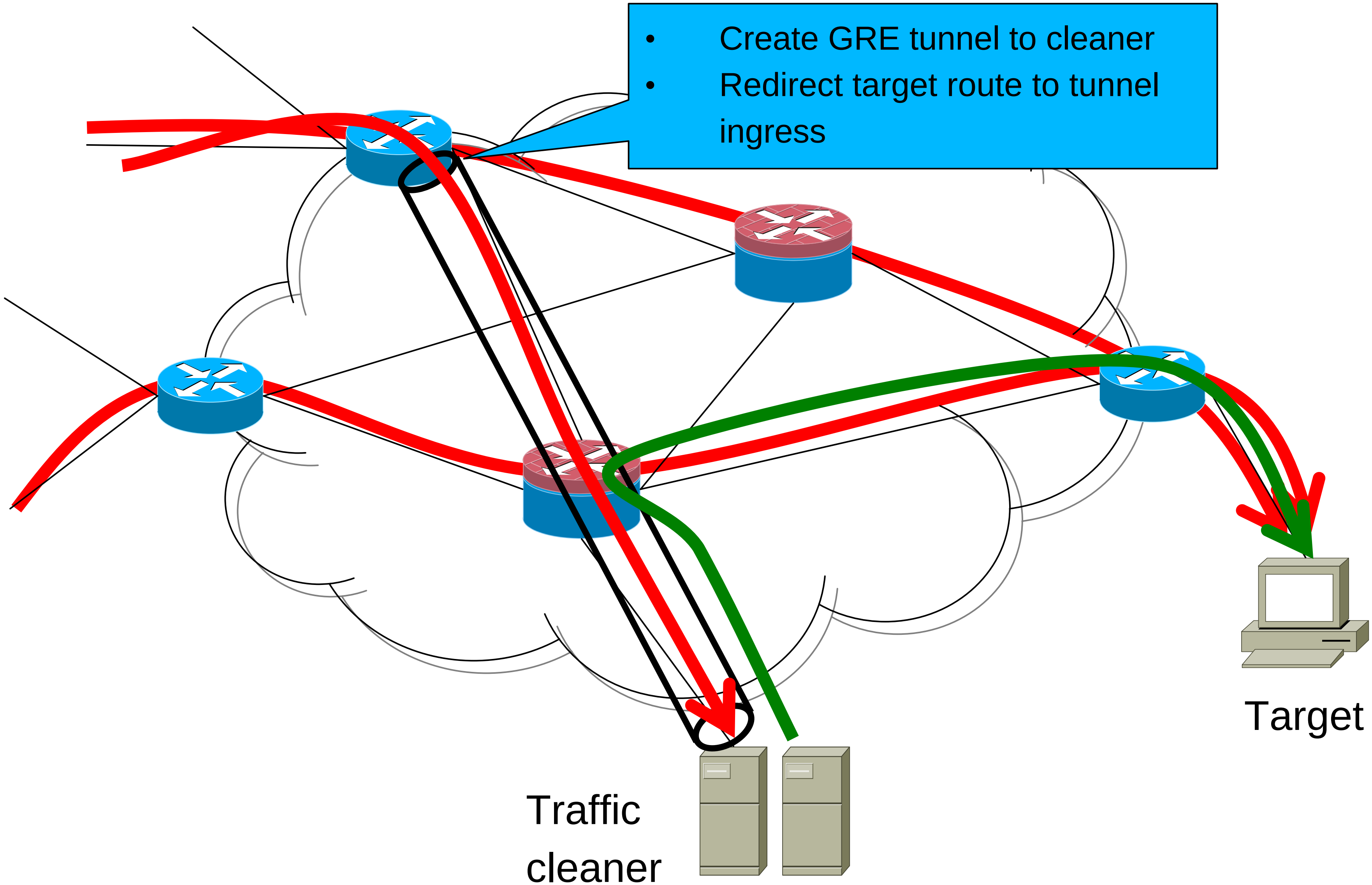
- Labels are local to a link, remapped at LSRs
  - “Label switching routers”
  - Looks a lot like virtual circuits
- Labels distributed via IP routing protocols, and LDPs.
- MPLS is about:
  - Aggregation
  - Traffic engineering
  - Virtual networks (see later)



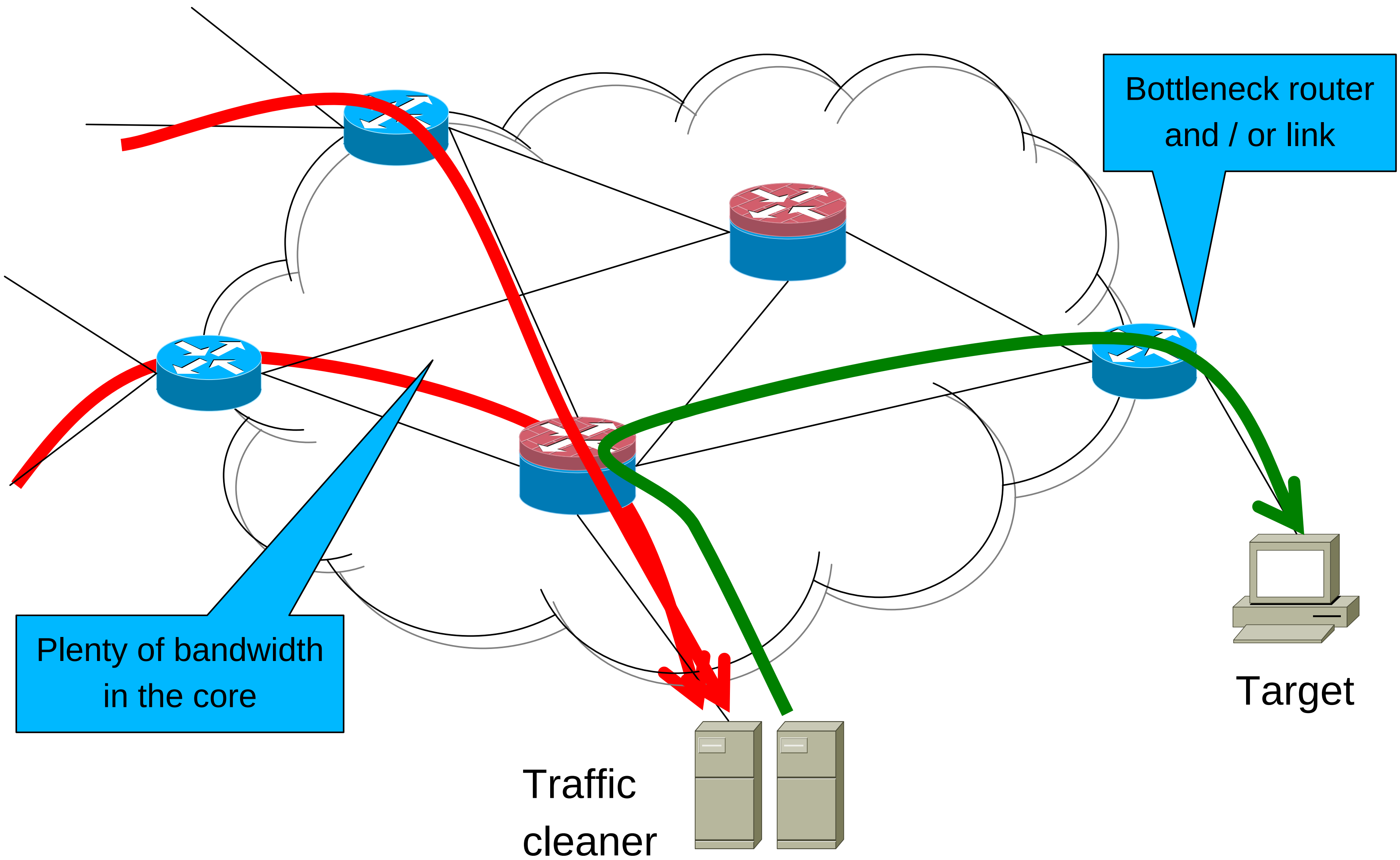
# DDoS mitigation



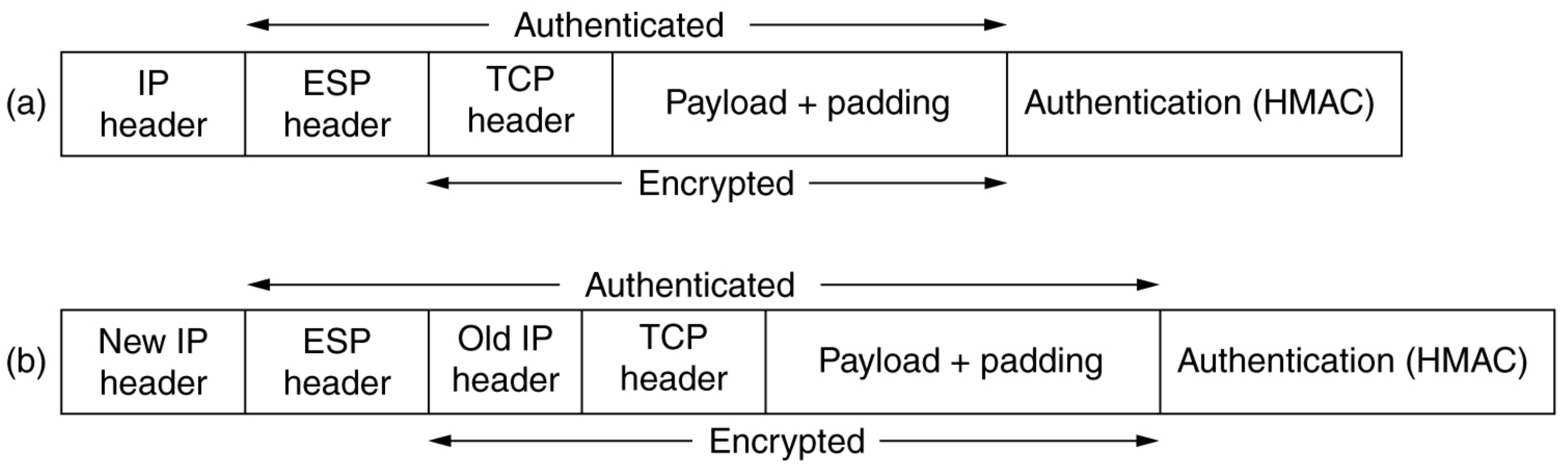
# *DDoS mitigation*



# *DDoS mitigation*



# IPsec: ESP (transport and tunnel modes)




- Security algorithm-independent
  - Header is just two words:
    - "security parameters index" and sequence number
- HMAC is at the end of the packet (why?)
  - And why the padding?

# *Mobility*

- Problem:
  - When computer moves, address has to change (why?)
- Solution: computer has two addresses!
  - Locally acquired one (e.g. WiFi in coffee shop)
  - Semi-permanent (acquired from “home” or provider)
- IP traffic on semi-permanent address tunnelled to provider over local interface (using PPTP)
- MobileIP doesn’t use tunnelling (much); but this method is becoming more popular (why?)

# *Tunnel with care…*

- Complicates routing
  - Adding additional “links” to a network
  - Statically routed ⇒ suboptimal (ignores routing protocol)
  - Dynamically routed ⇒ routing protocol doesn’t know it’s a tunnel
  - Encapsulation can lead to routing pathologies
- Complicates management / provisioning
  - Unexpected traffic patterns (loops?)
  - Traffic is now “opaque” to the carrier
- Complicates forwarding (for IP)
  - Packets require “shim” header for encapsulation
    ⇒ reduced MTU, or fragmentation, or loss of framing…

# *Virtual private networks*

- Idea: use tunnels as link layers
- ⇒ Can build private IP network over tunnels over public IP network.
- ISPs sell VPN services to large customers
  - Router support makes this easy (though complex)
  - Probably pays for the Internet…
- Typically IP over IP tunnels
  - GRE, IPIP, PPTP, AYIYA…

- VPNs are the “respectable” (to the carriers) face of a more general calls of *Overlay Networks*.

# *Overlay Networks*

- Observation:
  - Can use IP connections as tunnels for other protocols
    - Including IP
  - If you can establish enough “points of presence”, you can run your own network!
    - Routing protocols, addressing, etc.
    - Best effort, of course…
- Examples:
  - Content distribution networks
  - Application-layer multicast
  - Anonymizers (Tor)
  - RON (Resilient Overlay Networks)
    - Better than IP, over IP!

# *Overlay Networks*

- To deploy and exploit overlay networks effectively, need lots of Points of Presence
- If you're not a large company, this is a problem
  - See PlanetLab discussion next week…
- Time for a detour: Peer-to-Peer Networks and Applications

# *Peer-to-Peer Networks*

- What is Peer-to-Peer?
- Unstructured P2P systems
- Structured P2P overlays (DHTs)
  - Linear hashing
  - Consistent hashing
  - Distributed tree lookups
  - An early DHT in practice: Chord

# *"Peer-to-Peer" is…*

- Software: Gnutella, Tor, Freenet, BitTorrent, Skype…
- Very large overlay networks (millions of nodes)
- 80% of traffic on the Internet by volume.
- File "sharing"
  - Legal issues, RIAA
- Direct data exchange between clients

…a socio-cultural phenomenon!

# *“Peer-to-Peer” is also…*

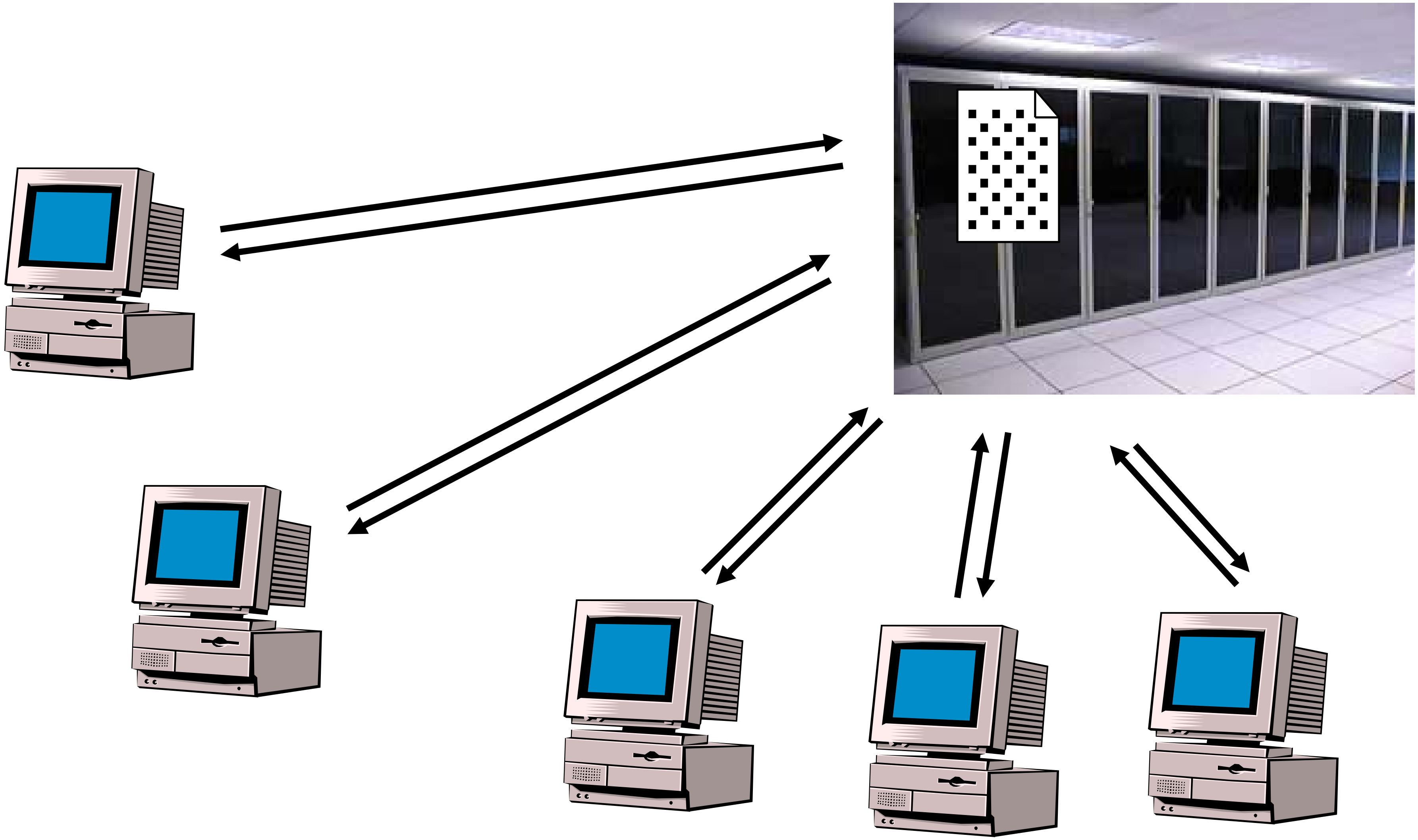
- A hot research area: Chord, Pastry, …
- A paradigm beyond Client/Server
- Dynamics (frequent joins and leaves)
- Fault tolerance
- Scalability
- Dictionary lookup, flat addressing, … and more!
- A fusion of distributed systems and networking

… a new networking philosophy/technology!

# *Why did it happen?*

- Music sharing
  - Music (etc.) suddenly became “non-physical”
  - Bandwidth became adequate
- Decentralized systems
  - Fault-tolerance
  - Self-organization
- Avoiding ISP control
  - Akamai and other CDNs
- Getting past the Internet service model
  - Anonymization services,
  - Overlays
  - Addressing schemes
- …and probably many more factors

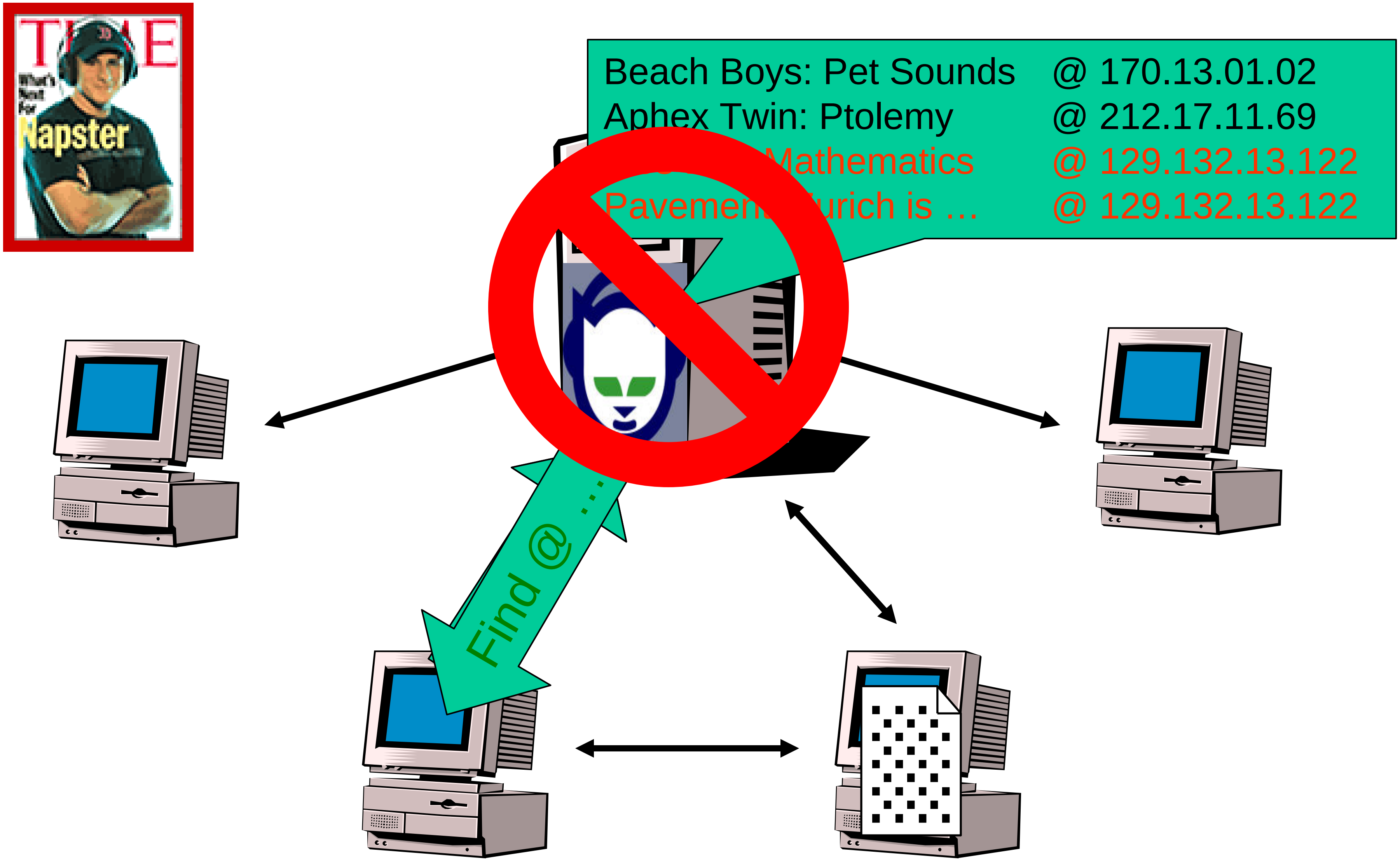
# Client/Server



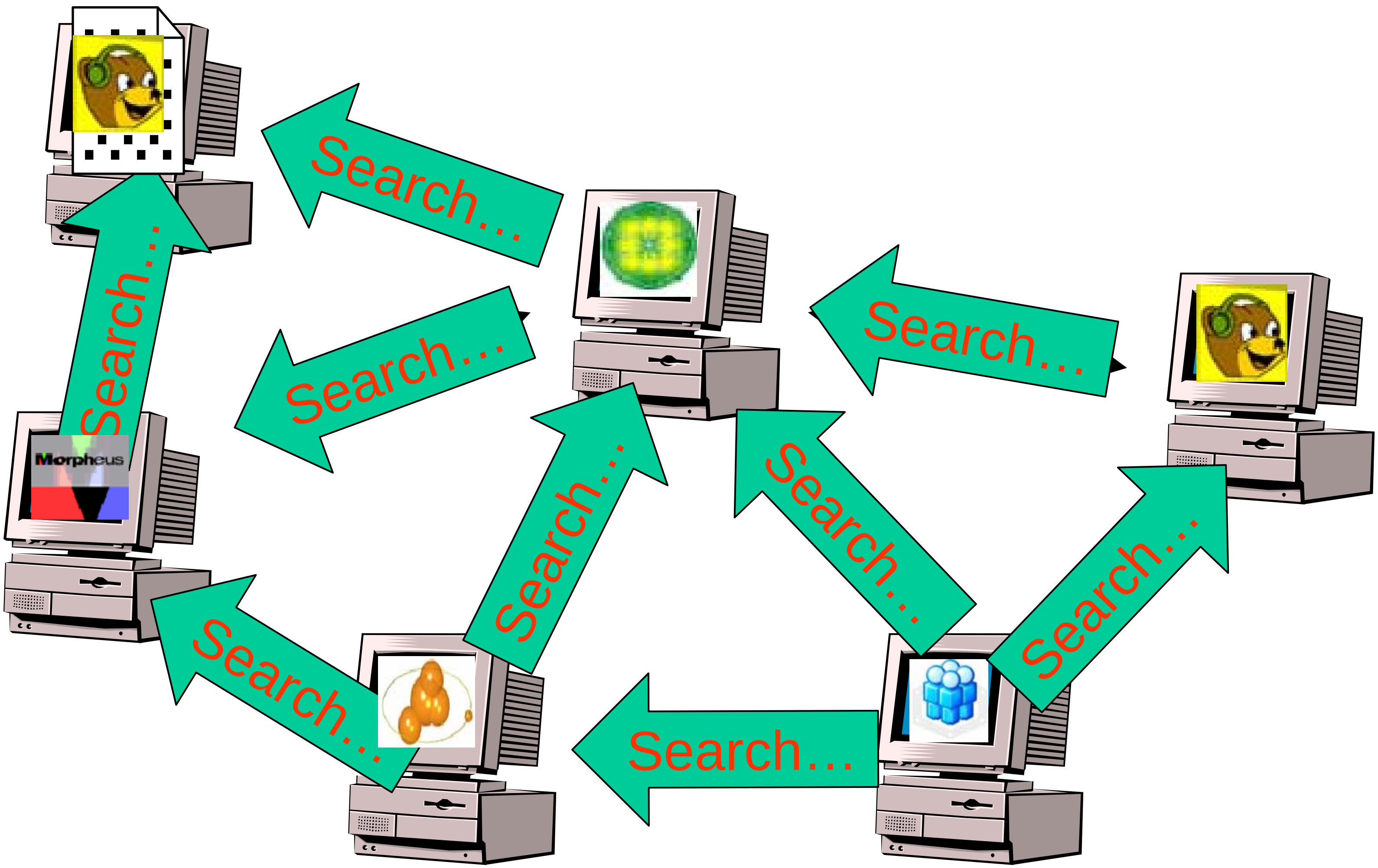
# *Client/Server Problems*

- Scalability
  - Can server serve 100, 1’000, 10’000 clients?
  - What’s the cost?
- Security / Denial-of-Service
  - Servers attract hackers
- Replication
  - Replicating for security
  - Replicating close to clients (“caching”)

# *Case Study: Napster*



# Case Study: Gnutella



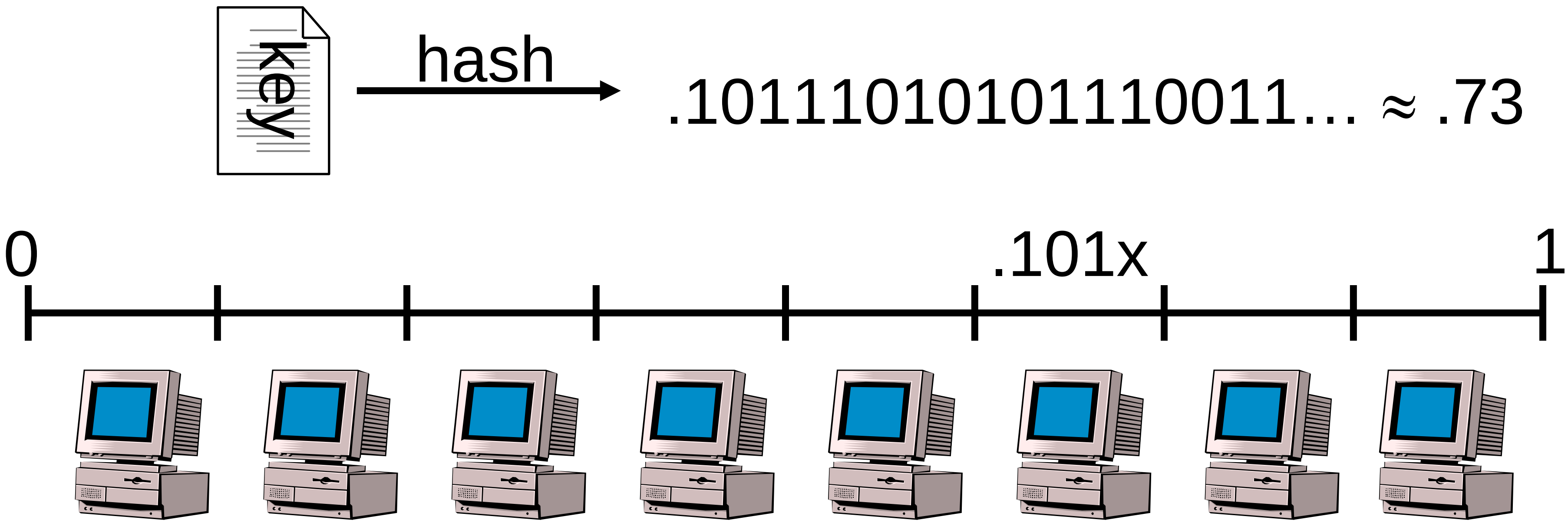
# *Pros/Cons Gnutella*

- totally decentralized
- totally inefficient
  - “flooding” = directionless searching
- Gnutella often does not find searched item
  - TTL
  - Gnutella “not correct”

# *Structured P2P Networks*

- Gnutella is an example of an “Unstructured P2P Network”
  - Viewed as a network, links have no structure or meaning
- Alternative: build overlay links to make searching more efficient
  - “Content-based routing”: route a message to any object
  - Avoid any central coordination
  - Nodes can join and leave at any time
- Term: “Distributed Hash Table” or DHT
  - Implements hash function: $h(object) \rightarrow machine$

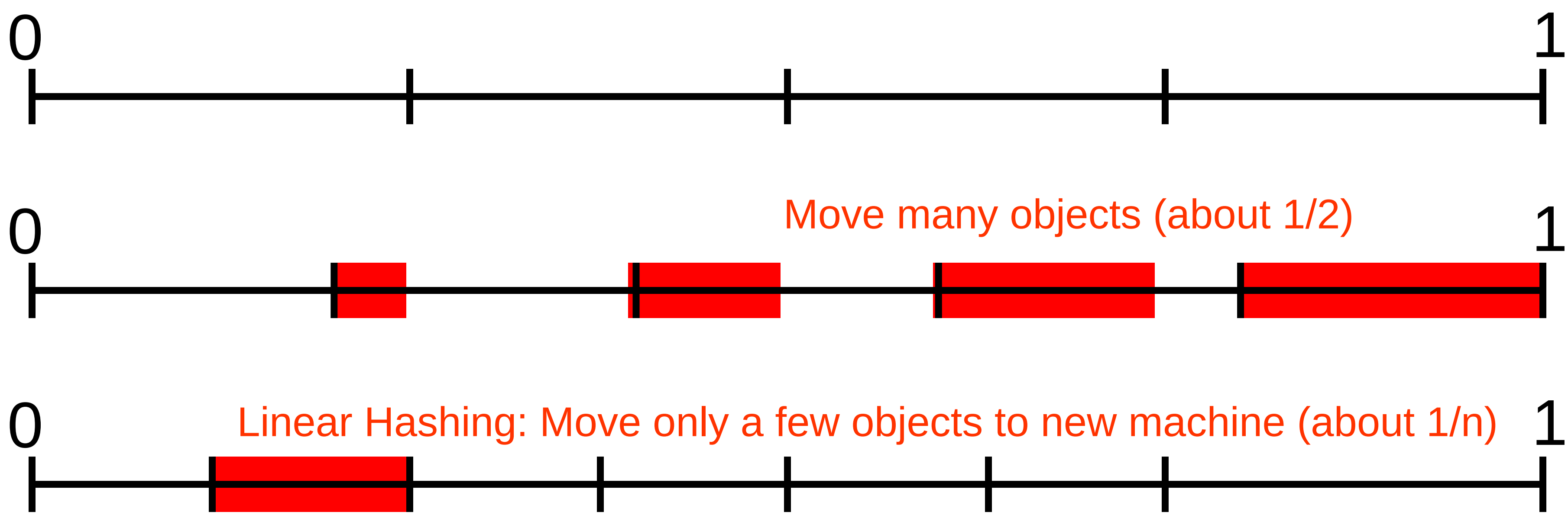
# *Distributed Hashing*




- Remark: Instead of storing a document at the right peer, just store a forward-pointer

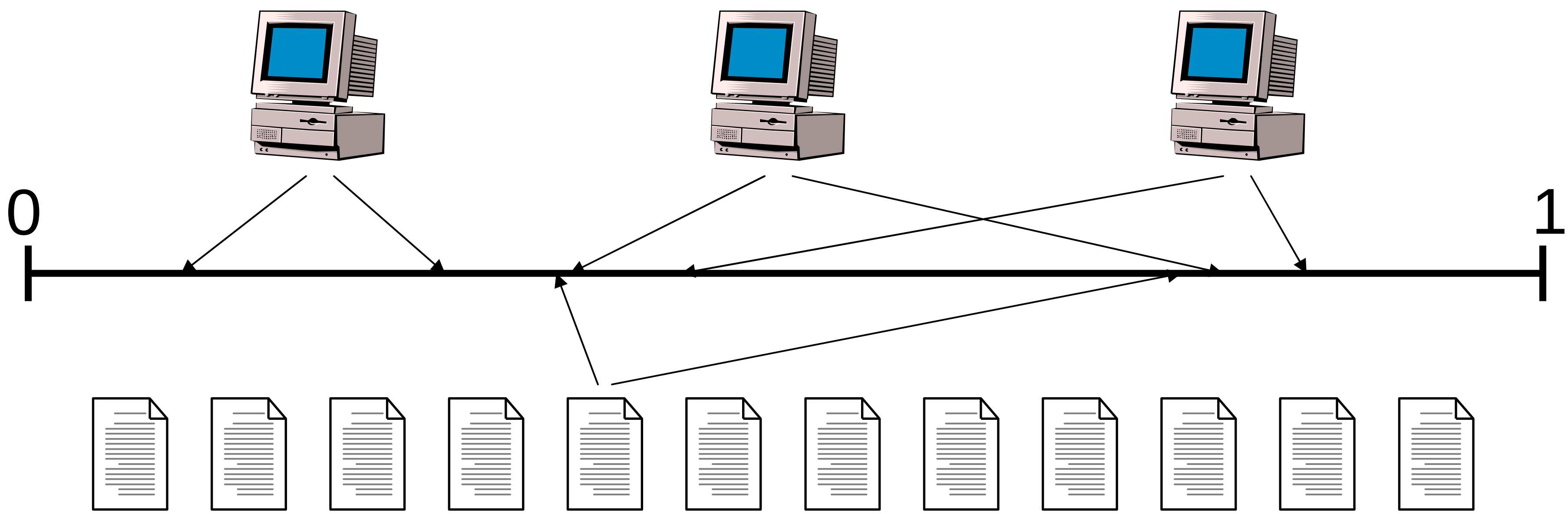
# *Linear Hashing*

- Problem: More and more objects should be stored; need to buy new machines!
- Example: From 4 to 5 machines



# *Consistent Hashing*

- Linear hashing needs central dispatcher
- Idea: Also the machines get hashed! Each machine is responsible for the files closest to it. Use multiple hash funct. for reliability.



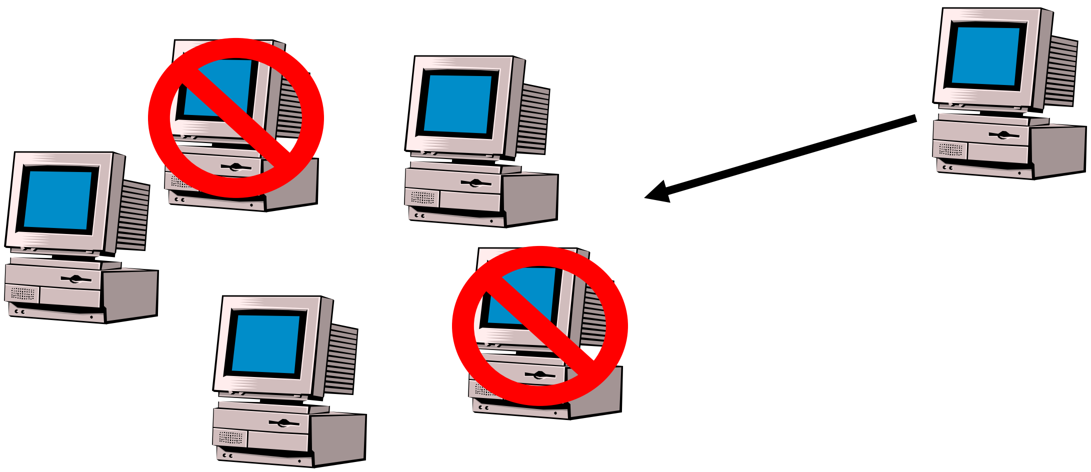
# *Not quite happy yet…*

- Problem with both linear and consistent hashing is that all the participants of the system must know all peers…

- Number one challenge: Dynamics!
  - Peers join and leave
  - Population is never up to date
  - As network scales, state at each node scales
  - As network scales, maintenance traffic explodes

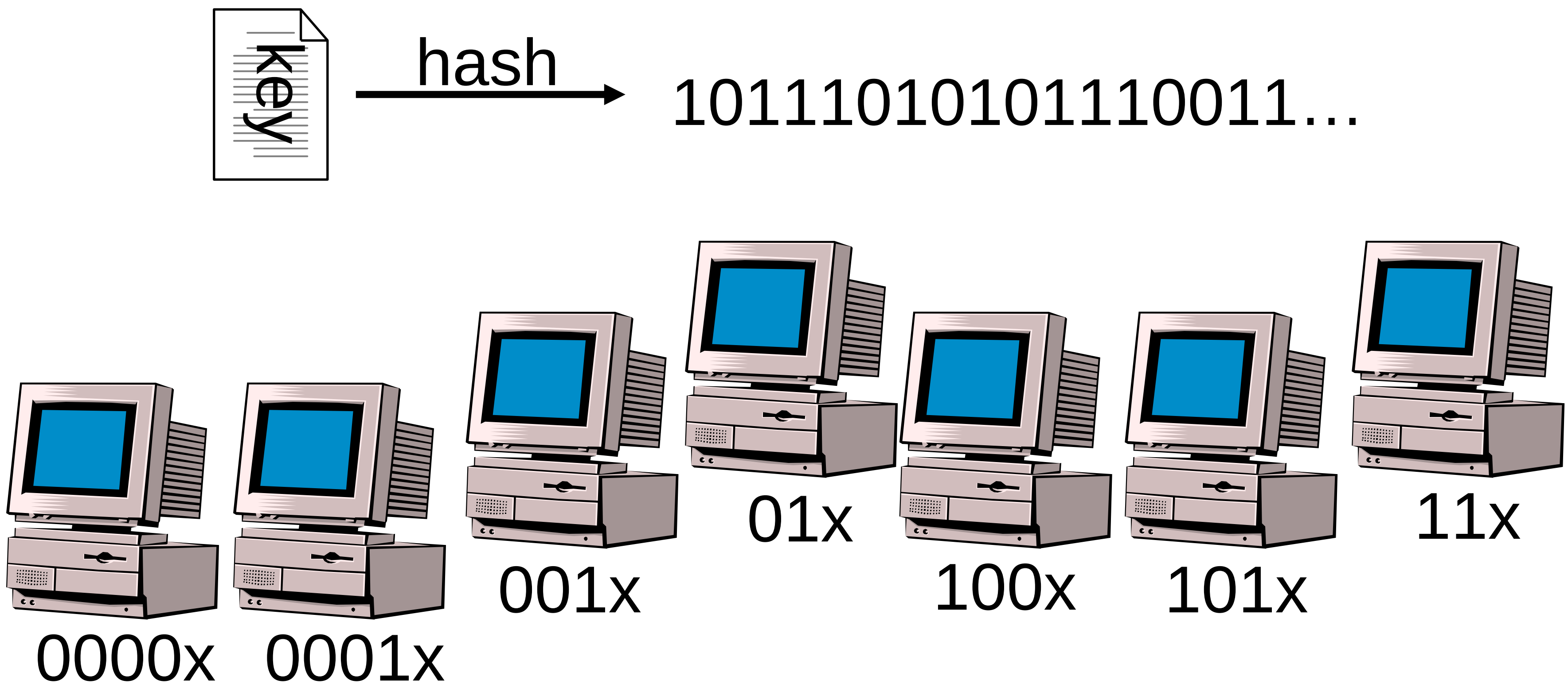
# *Dynamics*

- Machines (peers) are unreliable
  - Joins; worse: spontaneous leaves!



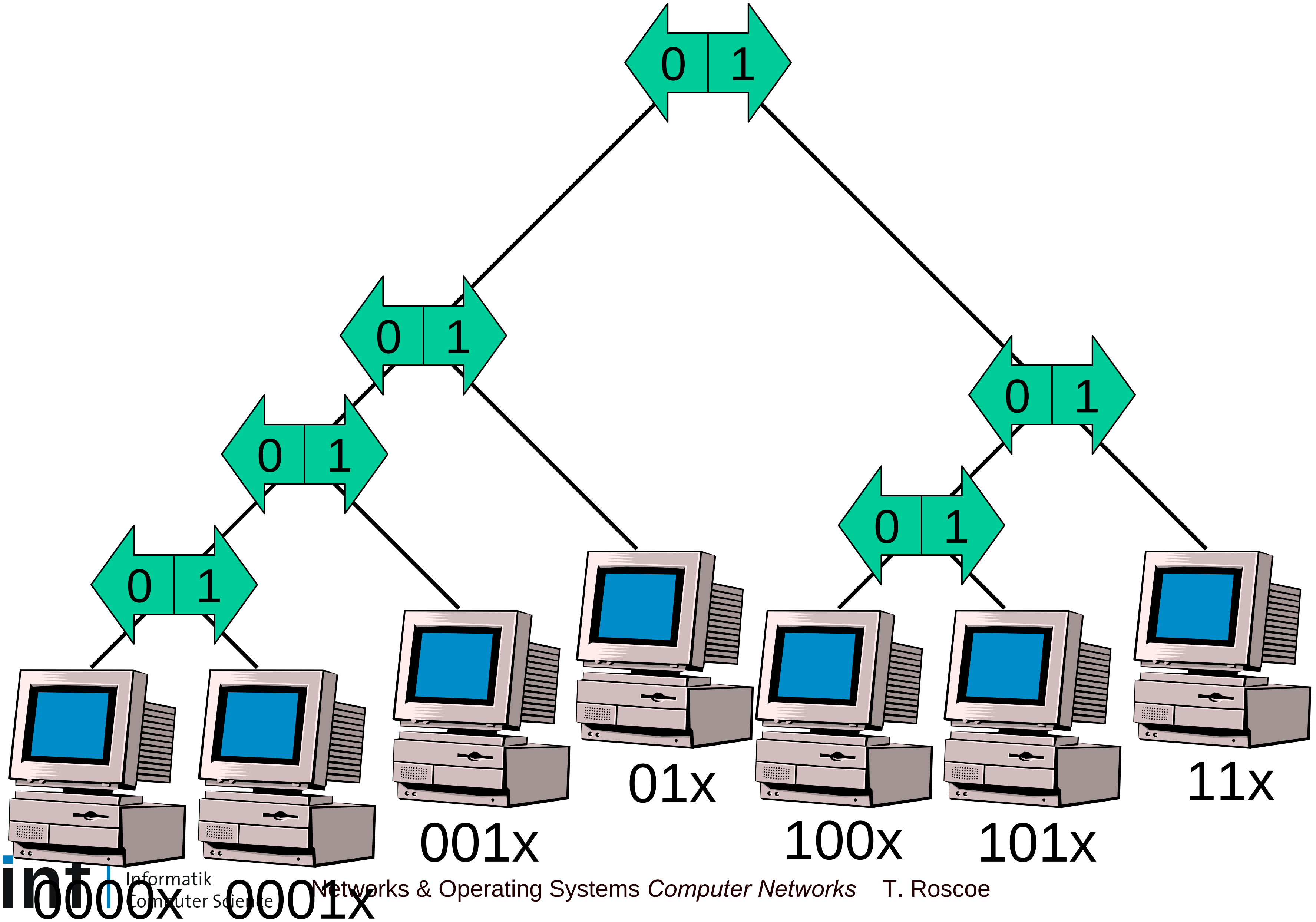
- Decentralized ("symmetric") System
  - scalable, fault tolerant, dynamic

# *P2P Dictionary = Hashing*




- Remark: Instead of storing a document at the right peer, just store a forward-pointer

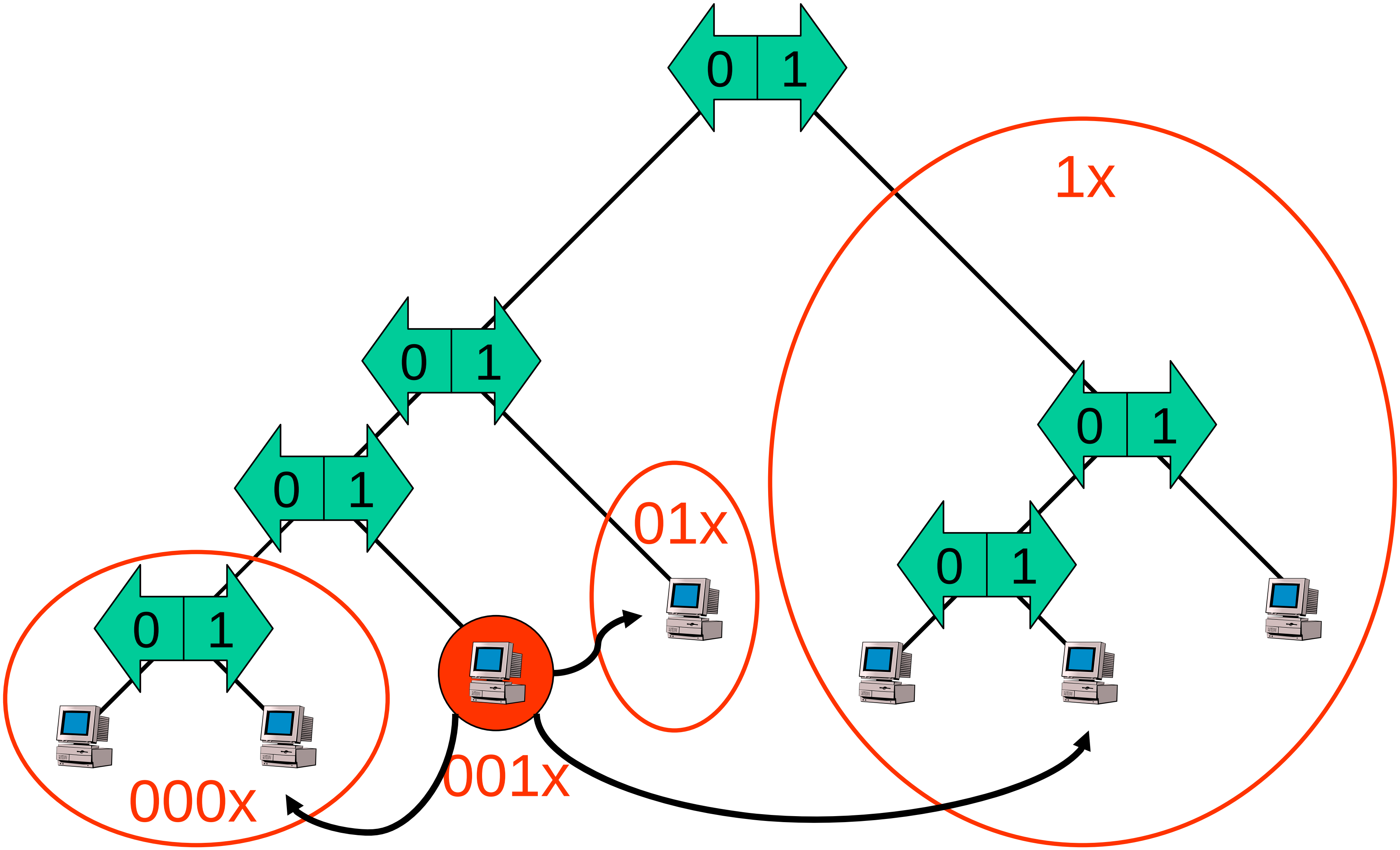
# *DHT as a search tree*



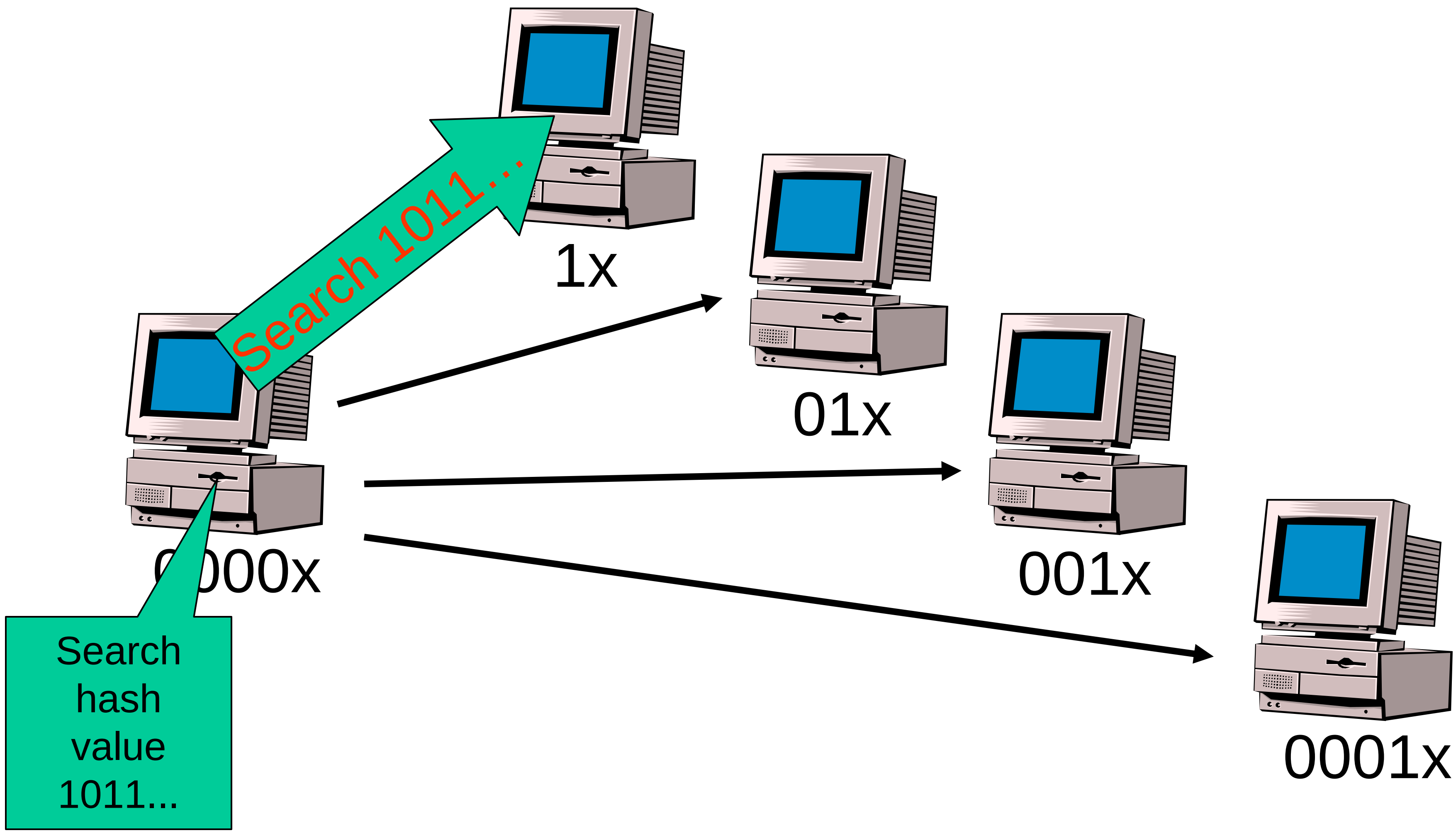
# *But who stores search tree?*

- In particular, where is the root stored?
  - Root is scalability & fault tolerance problem
  - There is no root…!
- If a peer wants to store/search, how does it know where to go?
  - Does every peer know all others?
  - Dynamics! If a peer leaves, all peers must be notified. Too much overhead
  - Idea: Every peer only knows subset of others

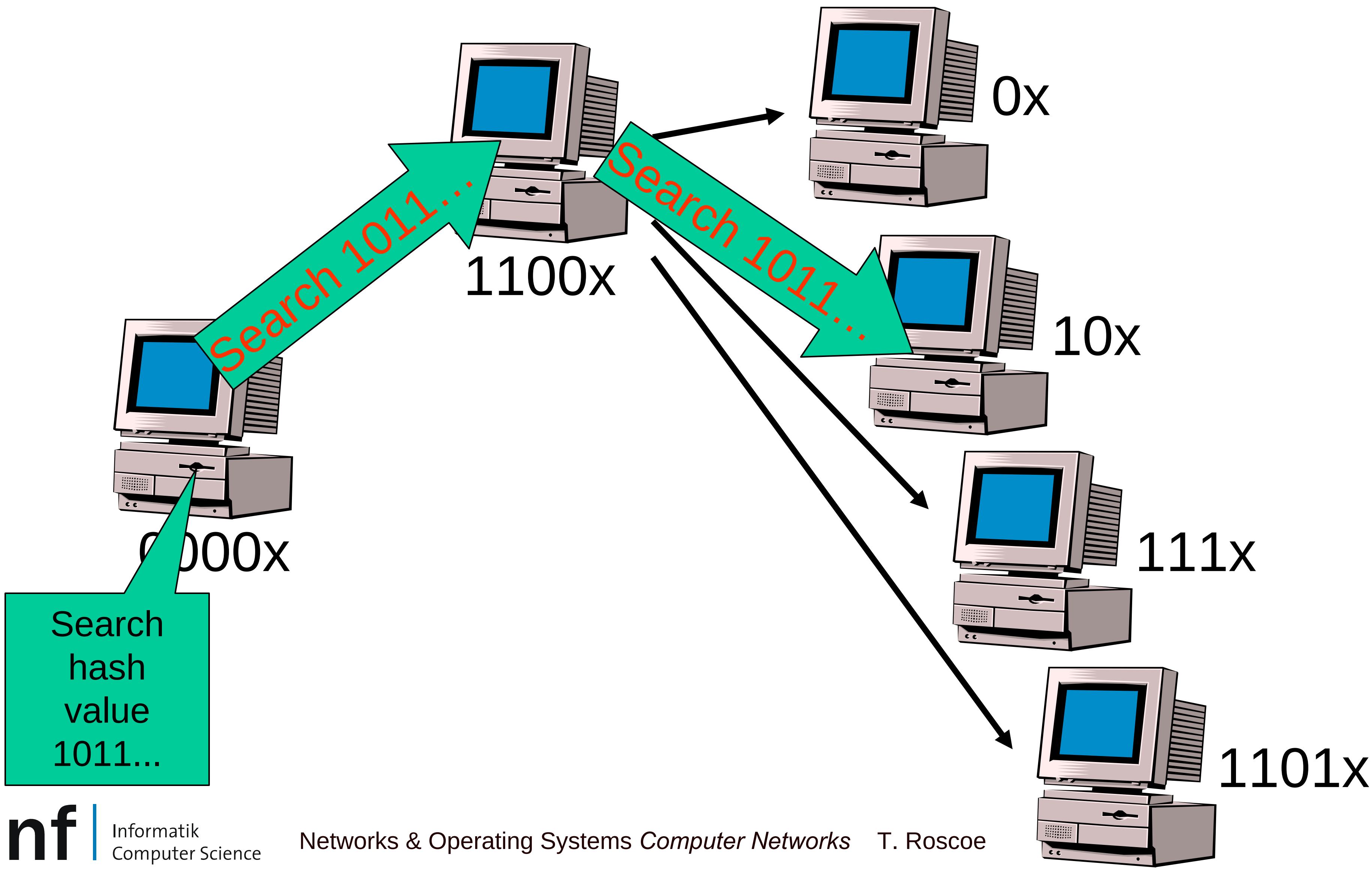
# The Neighbors of Peer 001x



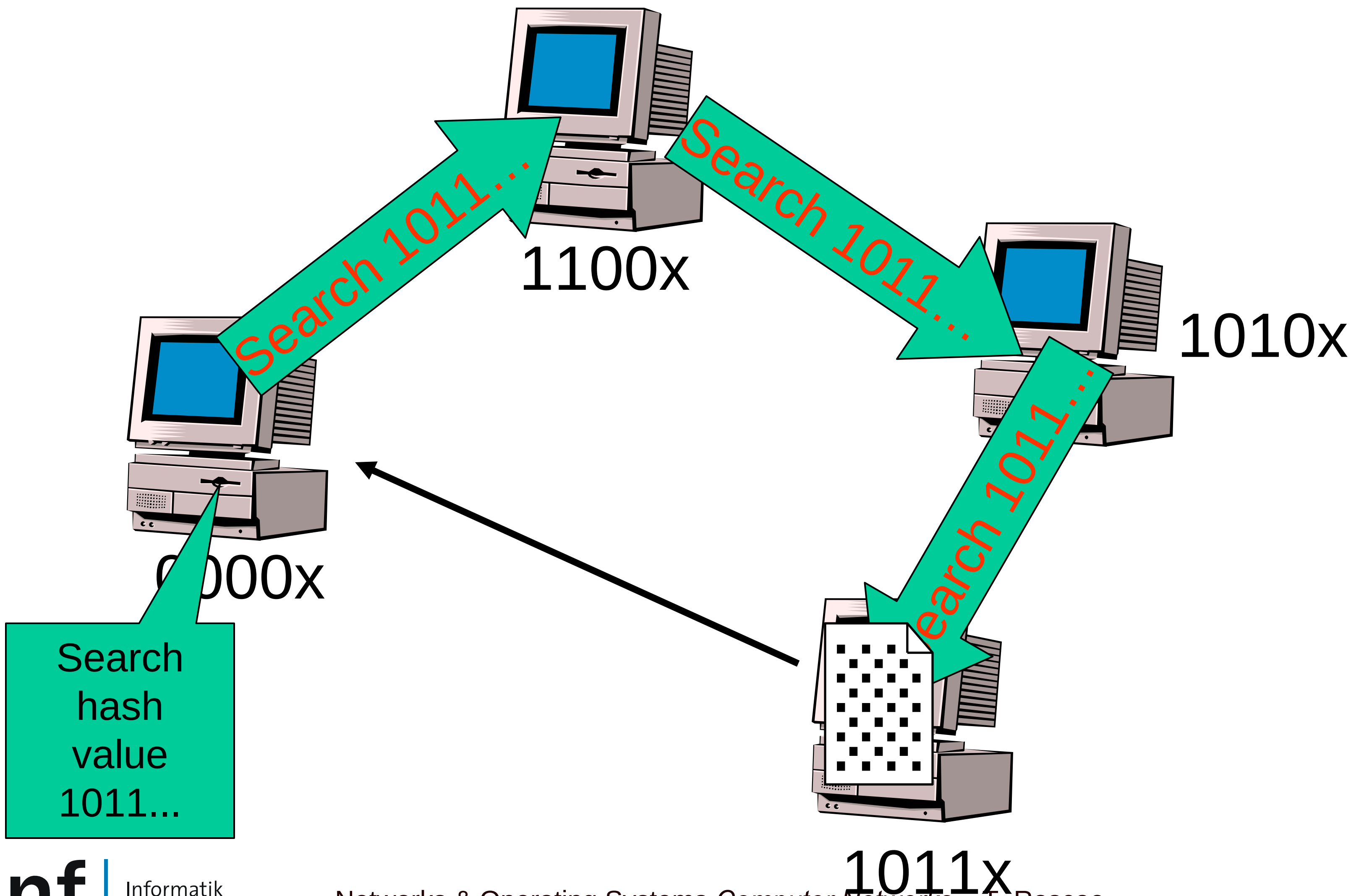
# *P2P Dictionary: Search*



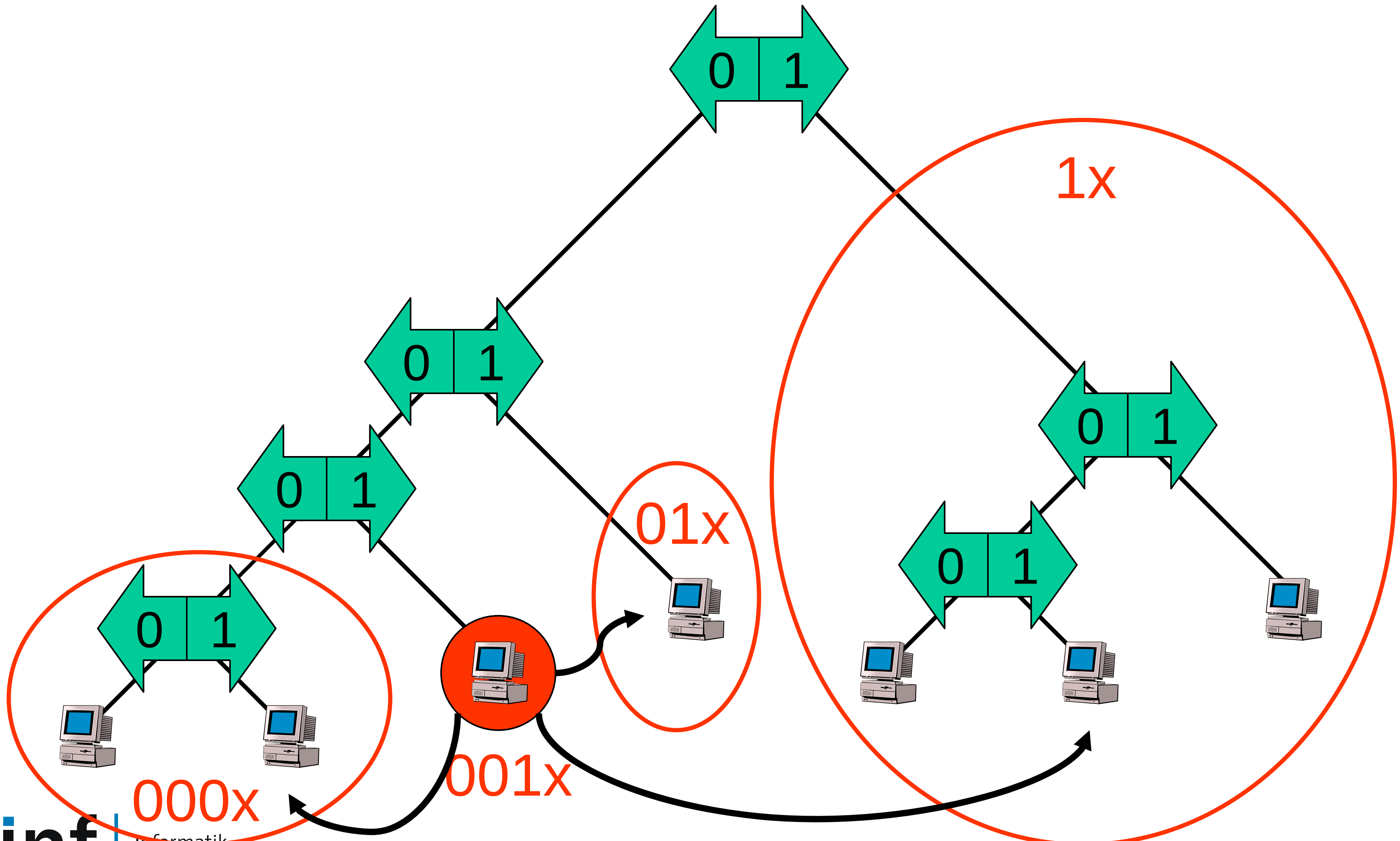
# *P2P Dictionary: Search*



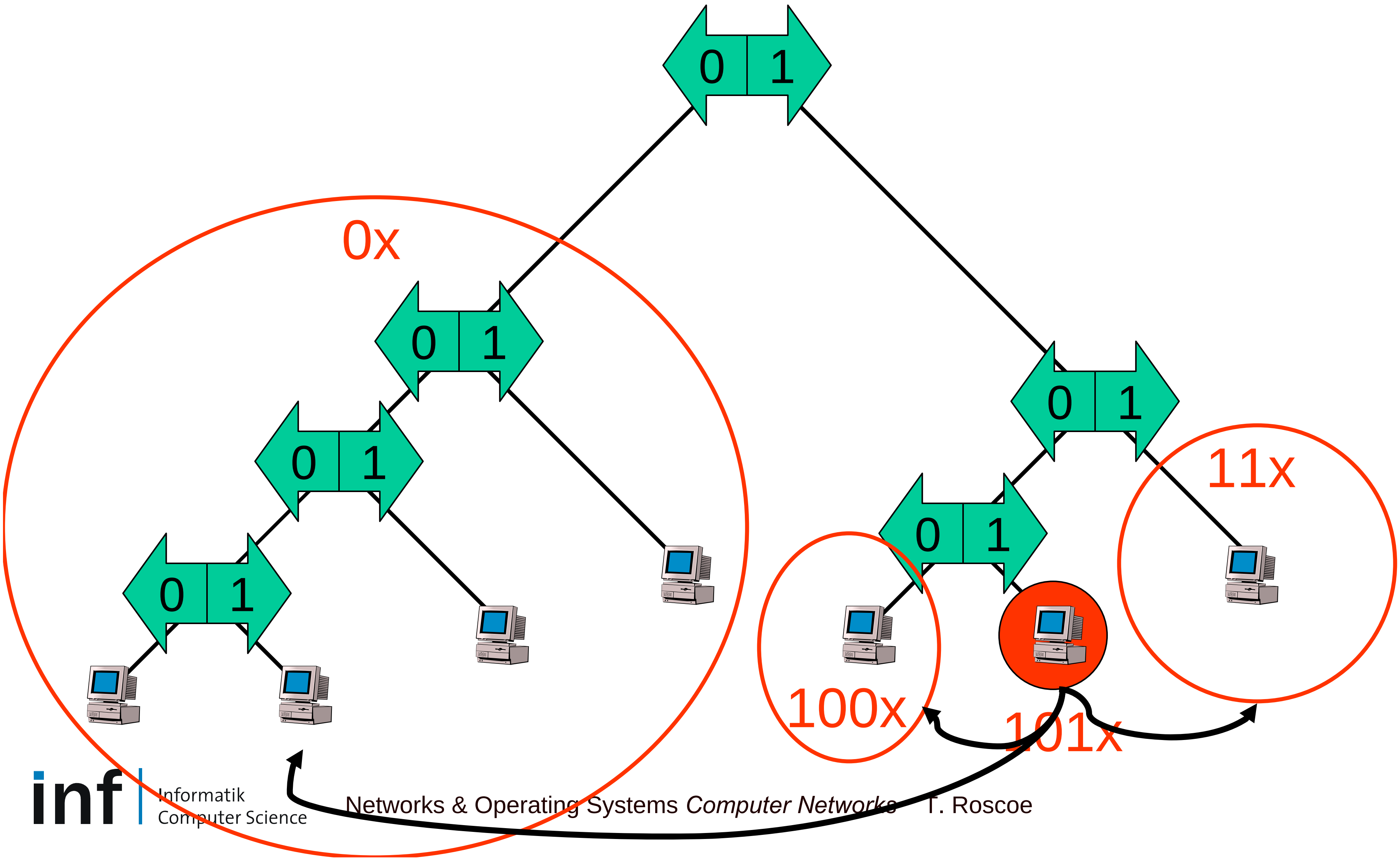
# P2P Dictionary: Search



# *Again: 001 searches 100*



# 001 searches 100 (continued)



# *Search Analysis*

- We have n peers in system
- Assume that "tree" is roughly balanced
  - Leaves (peers) on level $\log_2 n \pm$ constant

- Search has O(log n) steps
  - After k'th step, you are in subtree on level k
  - A "step" is a UDP (or TCP) message
  - Latency is dependent on P2P size (world!)

# *Peer Join*

- Part 1: Bootstrap

- In order to join a P2P system, a joiner must already know a peer already in system. Typical solutions are
  - Ask a central authority for a list of IP addresses that have been in the P2P regularly; look up a listing on a web site
  - Try some of those you met last time
  - Just ping randomly (in the LAN)

- Part 2: Find your place in P2P system

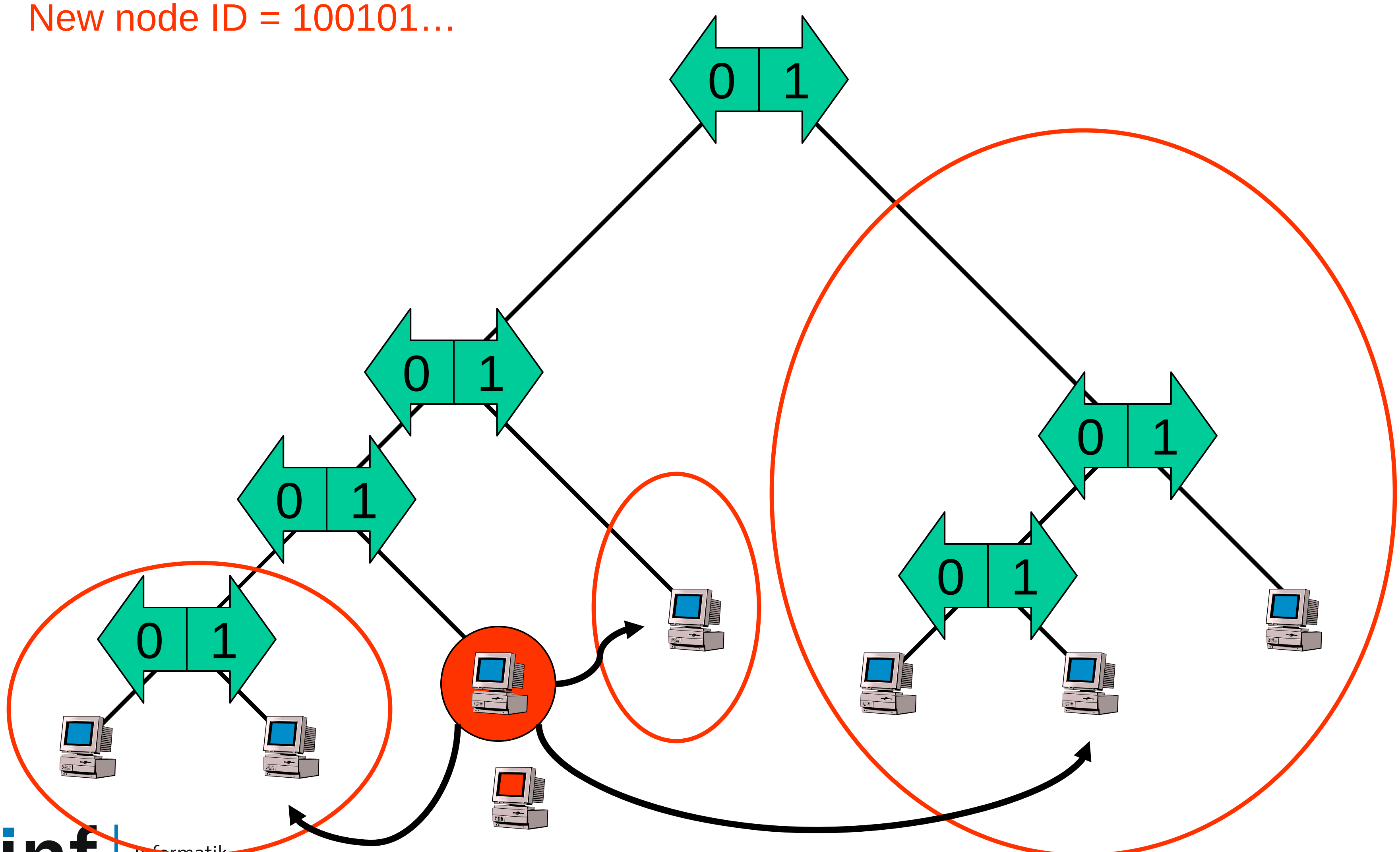
# *2. Find your place*

- The random method: Choose a random bit string (which determines the place)
- Search* for the bit string
- Split with the current leaf responsible for the bit string
- Search* for your neighbors

* These are standard searches

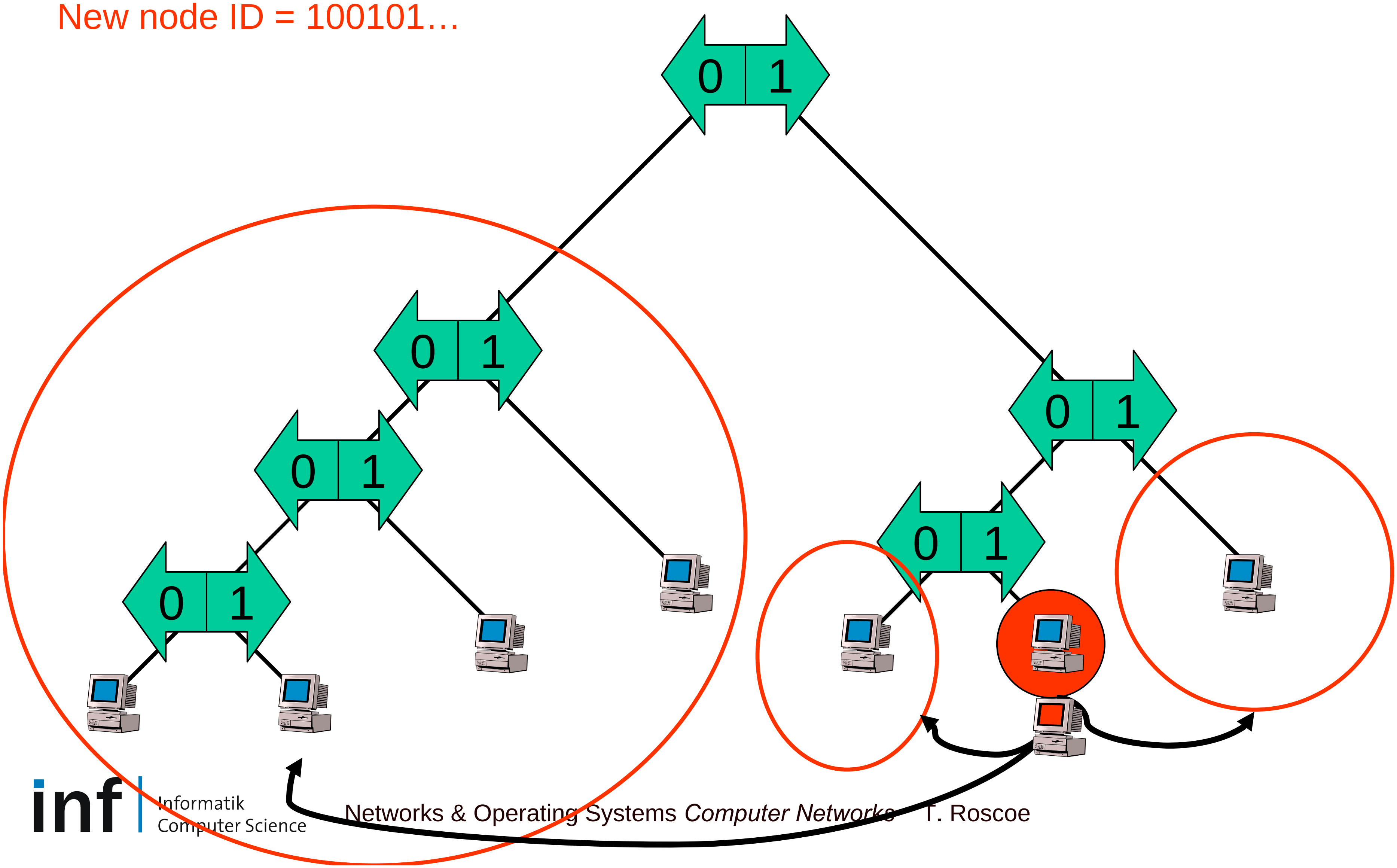
# Example: Bootstrap with 001 peer

New node ID = 100101…

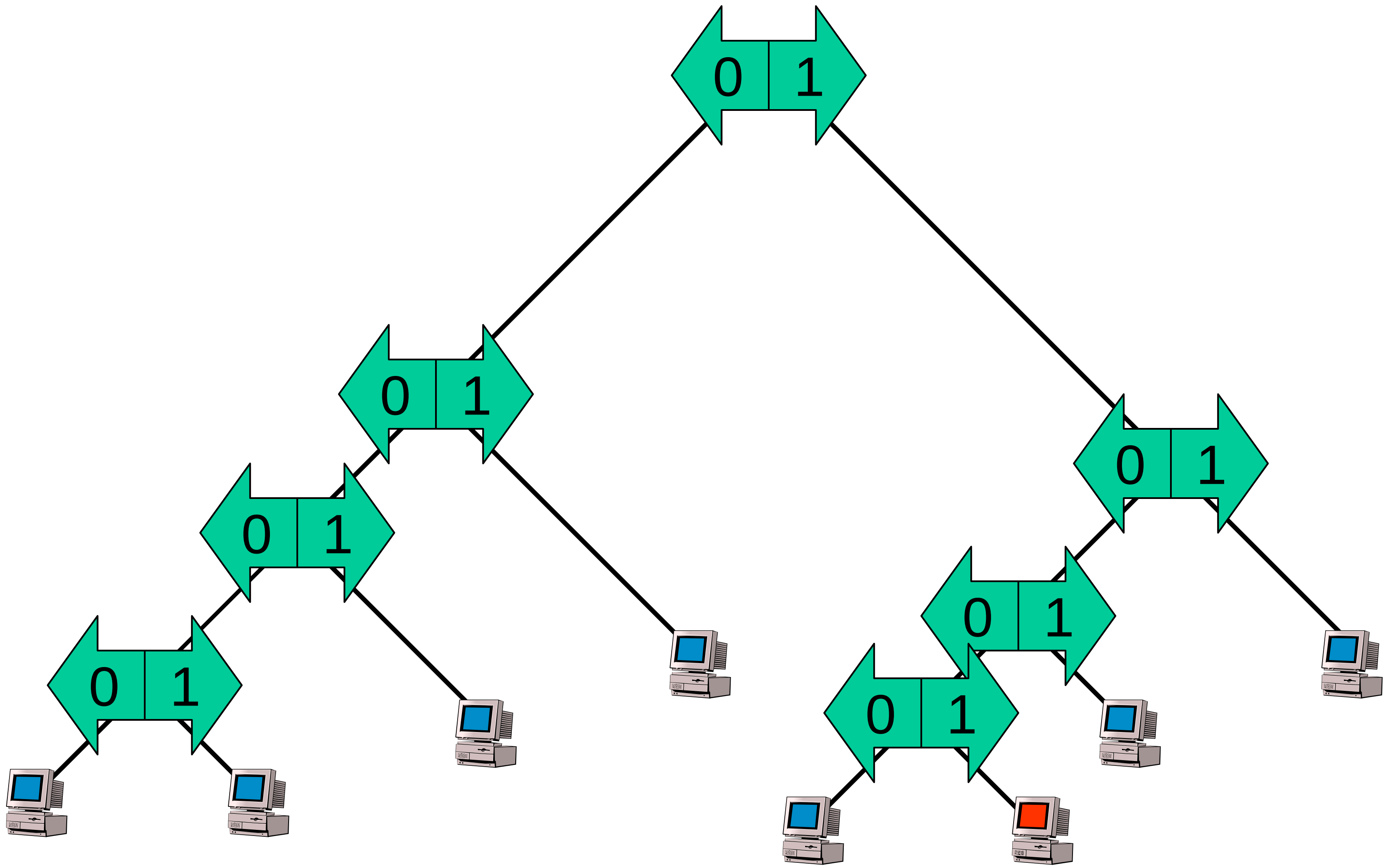


# *Joiner searches 100101…*

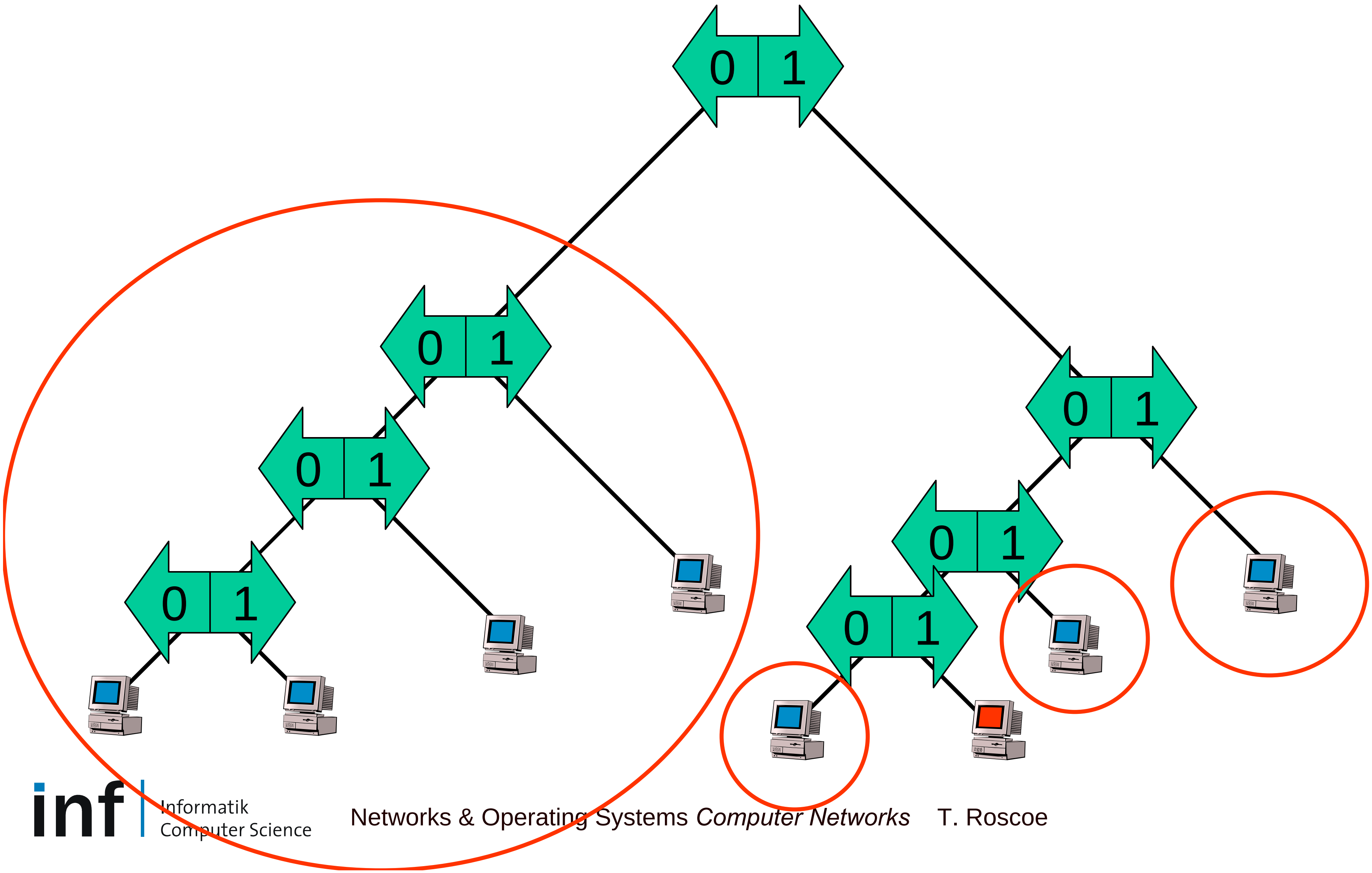
New node ID = 100101…



# *Joiner found 100 leave → split*



# Find neighbors



# *Random Join Discussion*

- If tree is balanced, the time to join is
  - $O(\log n)$ for the first part
  - $O(\log n) \leq O(\log n) = O(\log^2 n)$ for the second part

- It is believed that since all the peers are chosen their position randomly, the tree will more or less be balanced.
  - However, theory and simulations show that this is widely believed but not really true.

# *Leave*

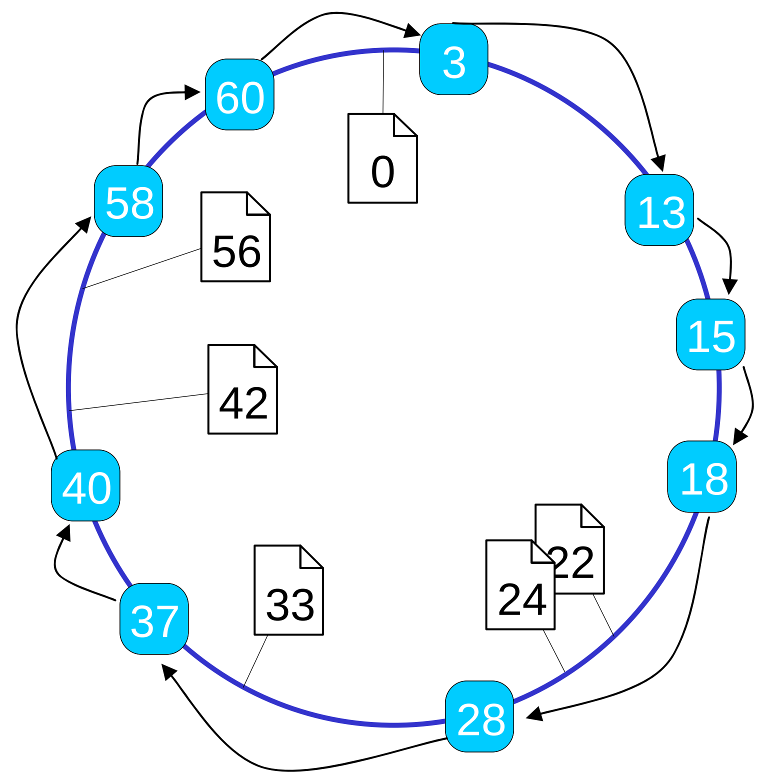
- Since a leave might be spontaneous, it must be detected first. Naturally this is done by the neighbors in the P2P system (all peers periodically ping neighbors).
- If a peer that left was detected, it must be replaced. If peer had sibling leaf, the sibling might just do a "reverse split."
- If not, search recursively… example!

# *Chord*

- The most cited "real" system by Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan, MIT, presented at ACM SIGCOMM 2001.
- Most discussed system in distributed systems and networking books, for example in Edition 4 of Tanenbaum's Computer Networks.
- Among other advantages, it's the easiest to draw.
- There are extensions on top of it, such as CFS, Ivy

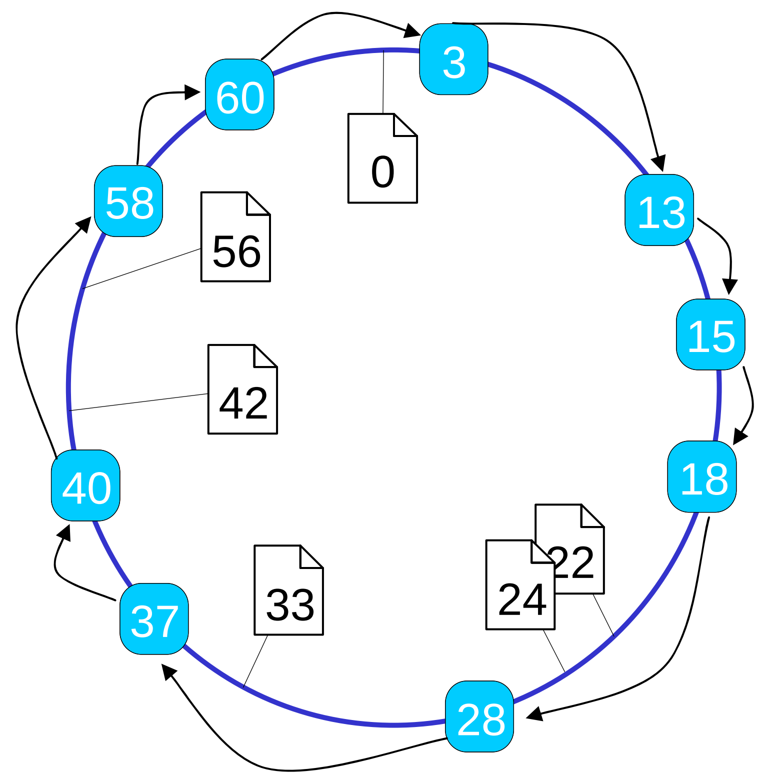
# Building up Chord in stages

- Each node has an ID
  - 160-bit SHA of IP address
- Nodes arranged in a ring
  - Each node has successor pointer
- "Documents" stored at "successor" node
  - Hash each object, store at node with next highest ID



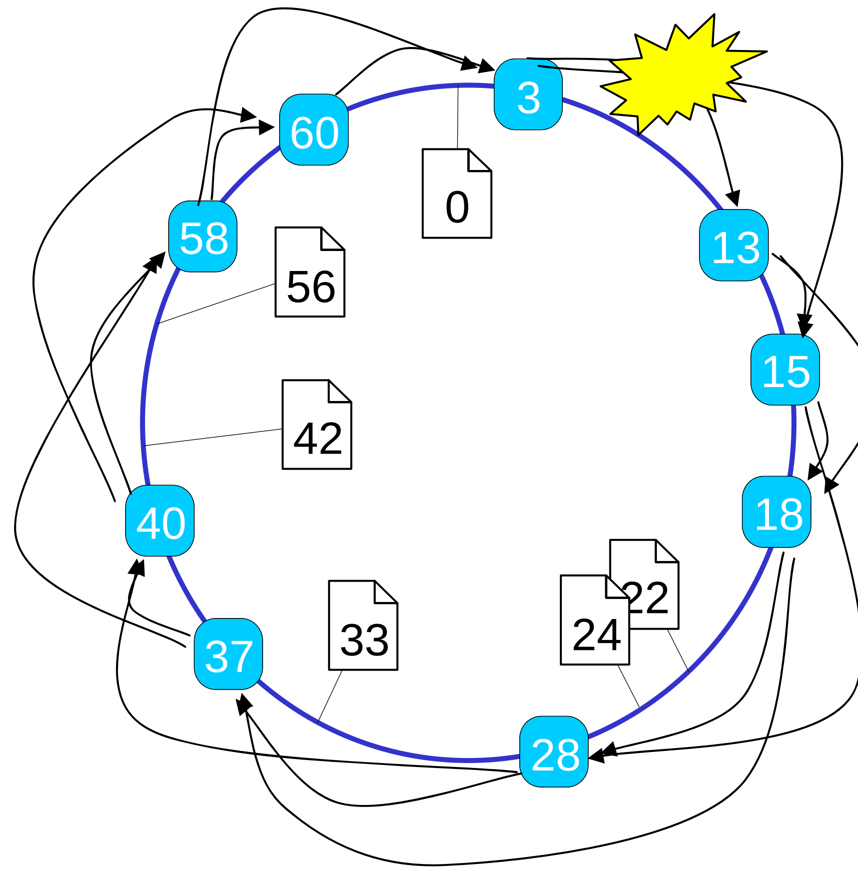
# *Lookup in a simple ring*

- Suppose a node (e.g 13) wants a document whose hash is 24
- Successively walk around the ring until you reach node whose *successor* is >24
  - In this case 18
  - Note: arithmetic must be module 2^160!
- This is where the document should be.
- This sucks: performance is linear in the number of nodes
  - We'll fix this in a few slides



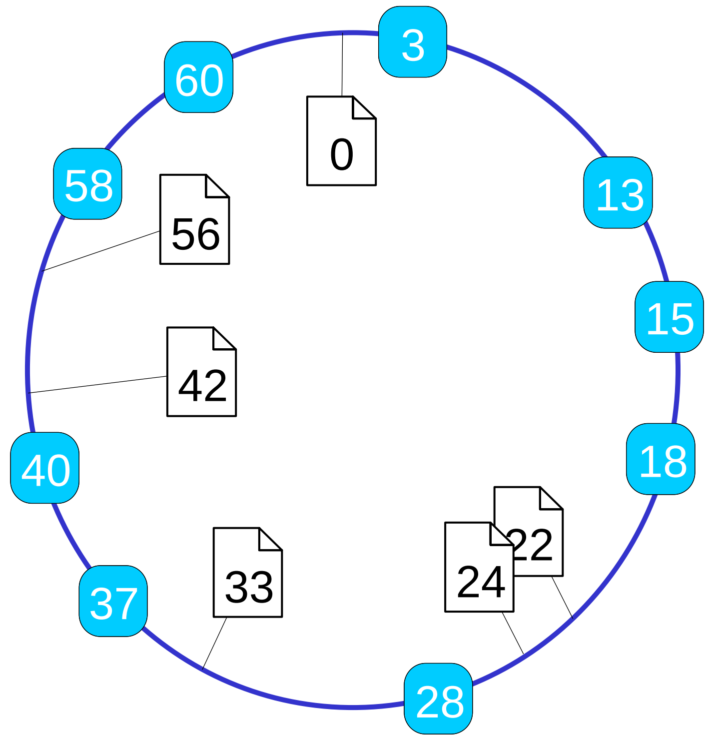
# *Problem: the ring is fragile*

- Failure of a node breaks the ring
- Solution: each node maintains a *successor set* of size *n*
  - *n* is ~log(#nodes)
- Also: predecessor pointer
  - (not shown)



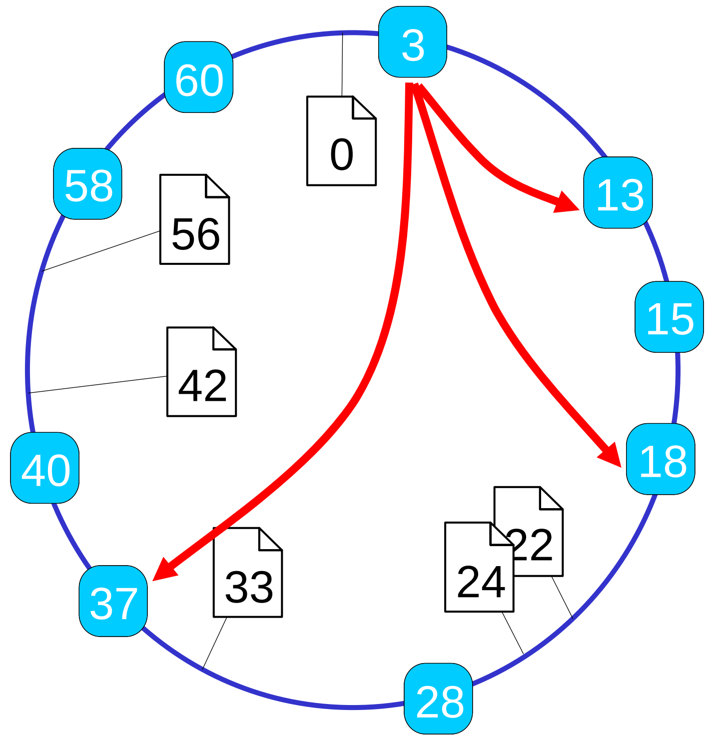
# *Problem: Joins and leaves*

- Joins as before:
  - Look up your own ID's successor
  - Contact for successors and predecessors
- Leaves:
  - Ping successors regularly
  - Always ensure *n* live nodes in successor set
- Note: treat failures as normal!



# *Problem: that performance problem*

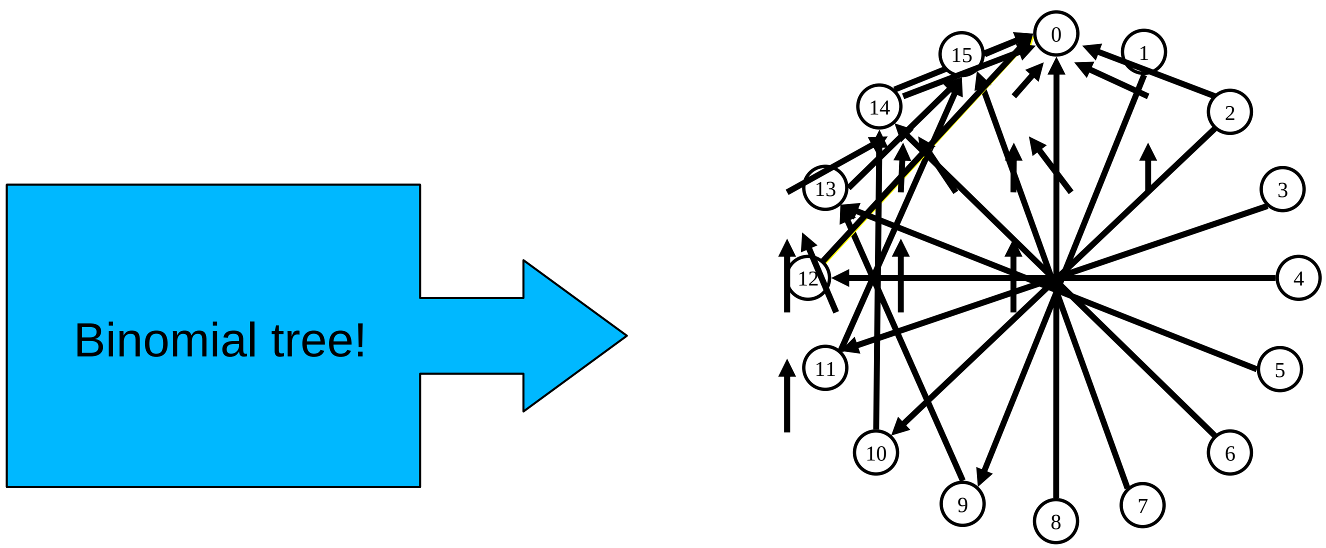
- Nodes maintain a *finger table*
  - Log(*n)* entries
  - Each 2^k away in ID space
- Obtained by lookup
- Continous maintenance process (again)
- New routing/lookup algorithm:
  - If succ(key) ∈ successor set
    return succ(key)
    else
    fwd to highest finger < key



# *Why these are (in some sense) the same*

- Finger tables are the key part of Chord
  - Provides O(log n) hop routing

- If you’re very clever, you’ve realized by now that this is very similar to the previous discussion
  - A “perfect” Chord network is a forest of binomial trees rooted at each key



## *Notice what else we've done:*

- Not just about finding that obscure "Godspeed!" bootleg…
- We have a routing protocol that works on "flat" 160-bit (or any) identifiers!
  - Recall that IP routing is hierarchical
- We can efficiently construct trees rooted at any node
  - Multicast
  - Aggregation
- Two players can "rendezvous" in the network by hashing a shared value
  - Many applications beyond content

# *Many many others...*

- Original theory work by Plaxton, Rajaraman, and Richa on finding replicas in a multiprocessor inspired most early DHTs (e.g. Pastry, CAN, Tapestry, or Kademlia).
- Some proposals improve the design;
  – Viceroy (butterfly graph) and Koorde (DeBruijna graph) use constant number of neighbours per peer
  – Symphony allows random links and addresses the “ID distribution” issue
  – Accordion has a dynamically varying routing table size
  – SkipNet uses skiplists and allows organizational boundaries to be represented
  – Bamboo adapts join and leave algorithms for high churn resilience
  – OpenDHT (over Bamboo) is a public DHT service for shared use

# *Structured vs. Unstructured*

- Unstructured:
  - Out there!  Heavily deployed…
  - Slow, but very resilient
  - For music sharing, users don't care…
  - Some structure appears (e.g. SuperPeers)
  - Hard to subvert (random structure)
- Structured:
  - Efficient for lookups – O(log n) hops
  - Fast
  - Well-defined routing semantics
  - Challenge in handling *churn*
  - Adaptivity to slow links/nodes
  - Overengineered academic plaything?

# *Reality: recent systems are hybrid*

- Recent “small world” graph theory results suggest you can relax the structure a lot and still obtain good performance
- Deployed systems use unstructured links plus a DHT for indexing
  - eMule / eDonkey (based on Kademlia DHT)
  - Trackerless BitTorrent
- Increasingly proposed for “real” applications…

# ***Next week (if all goes well):***

Roger and Timothy present:

"some cool stuff we're working on"