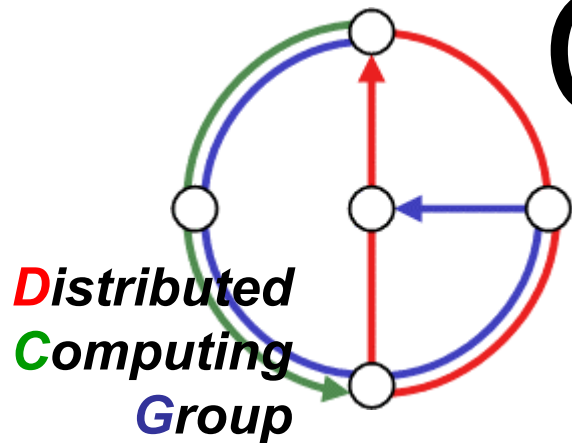


# Chapter 6

# PEER-TO-PEER

# COMPUTING

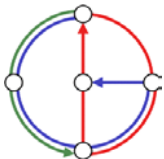


Computer Networks  
Winter 2003 / 2004

# Overview



- What is Peer-to-Peer?
- Dictionary
  - Distributed Hashing
  - Search
  - Join & Leave
- Other systems
- Case study: Spam Filtering
- Conclusion

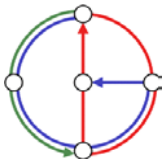


## “Peer-to-Peer” is...



- Software: Napster, Gnutella, Kazaa, ...
- File “sharing”
- Legal issues, RIAA
- Direct data exchange between clients
- Best effort, no guarantees
- 80% of Web Traffic “P2P”

...a socio-cultural phenomenon!

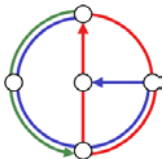


“Peer-to-Peer” is also...

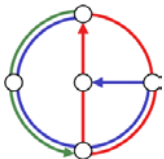
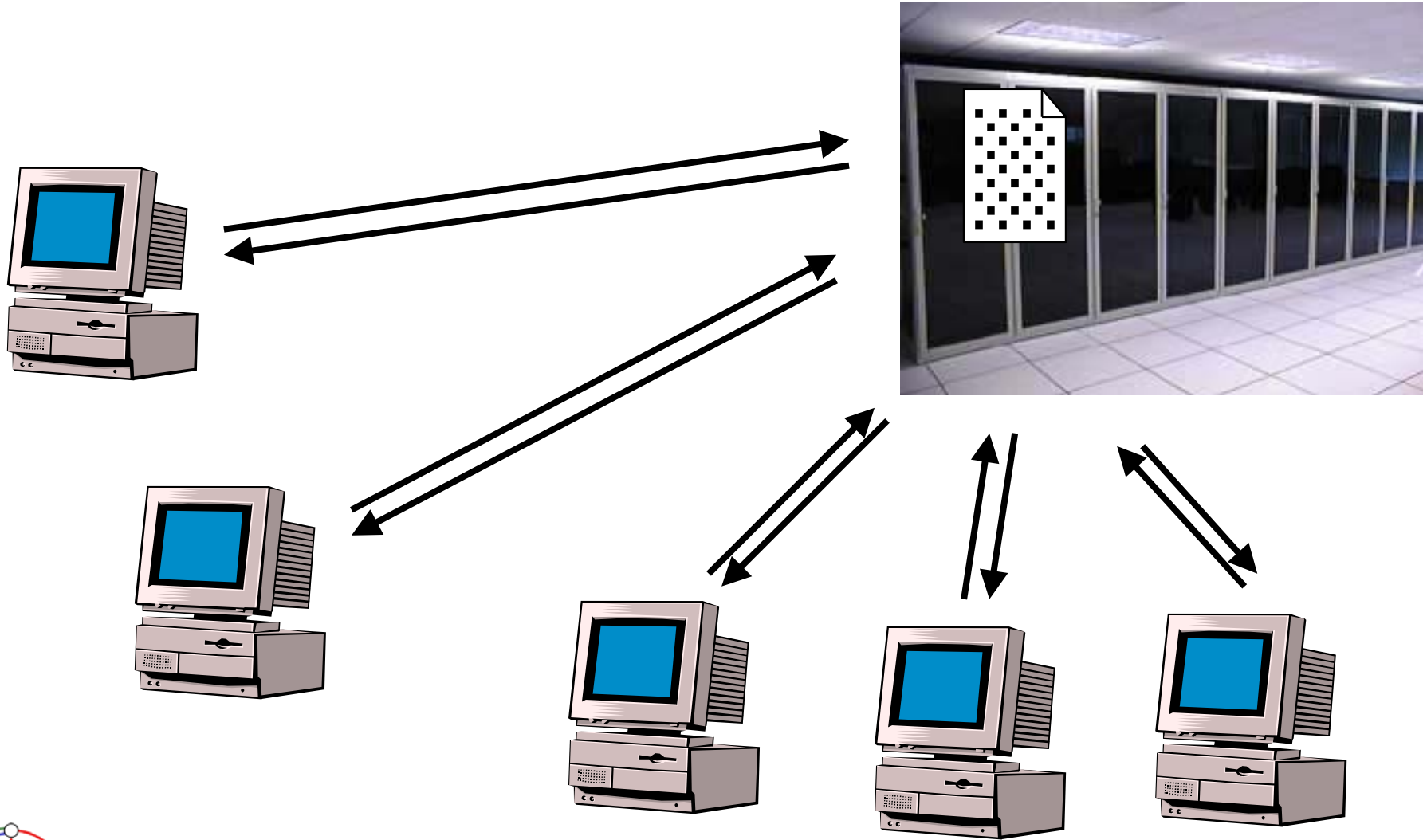


- A hot **research** area: Chord, Pastry, 4S, ...
- A paradigm beyond Client/Server
- **Dynamics** (frequent joins and leaves)
- **Fault tolerance**
- **Scalability**
- **Dictionary**... and more!

... a new **networking philosophy/technology!**



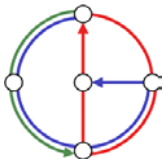
# Client/Server



# Client/Server Problems



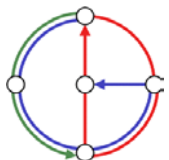
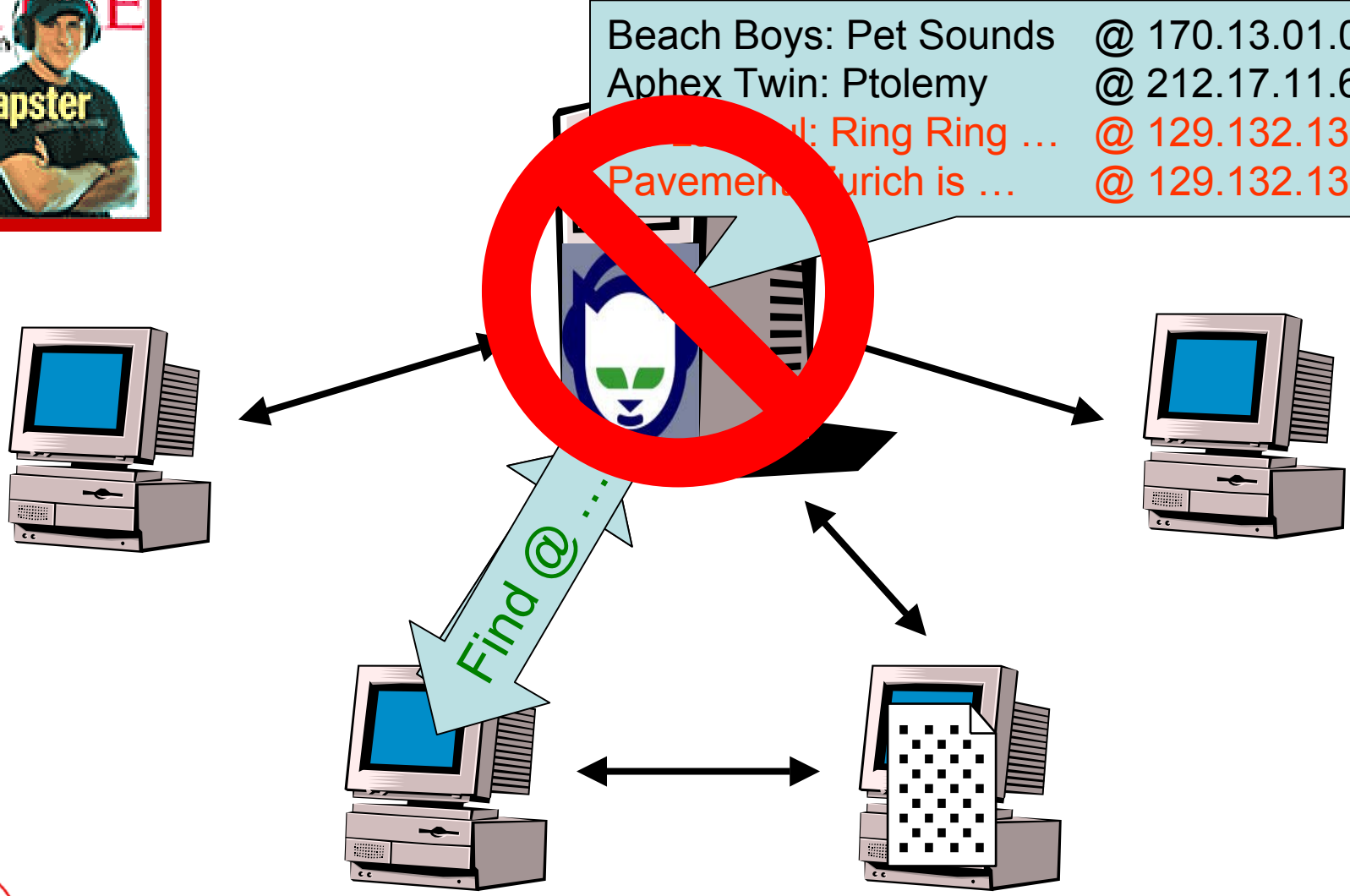
- **Scalability**
  - Can server serve 100, 1'000, 10'000 clients?
  - What's the cost?
- **Security / Denial-of-Service**
  - Servers attract hackers
- **Replication**
  - Replicating for security
  - Replicating close to clients (“caching”)



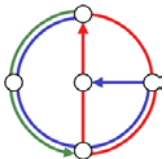
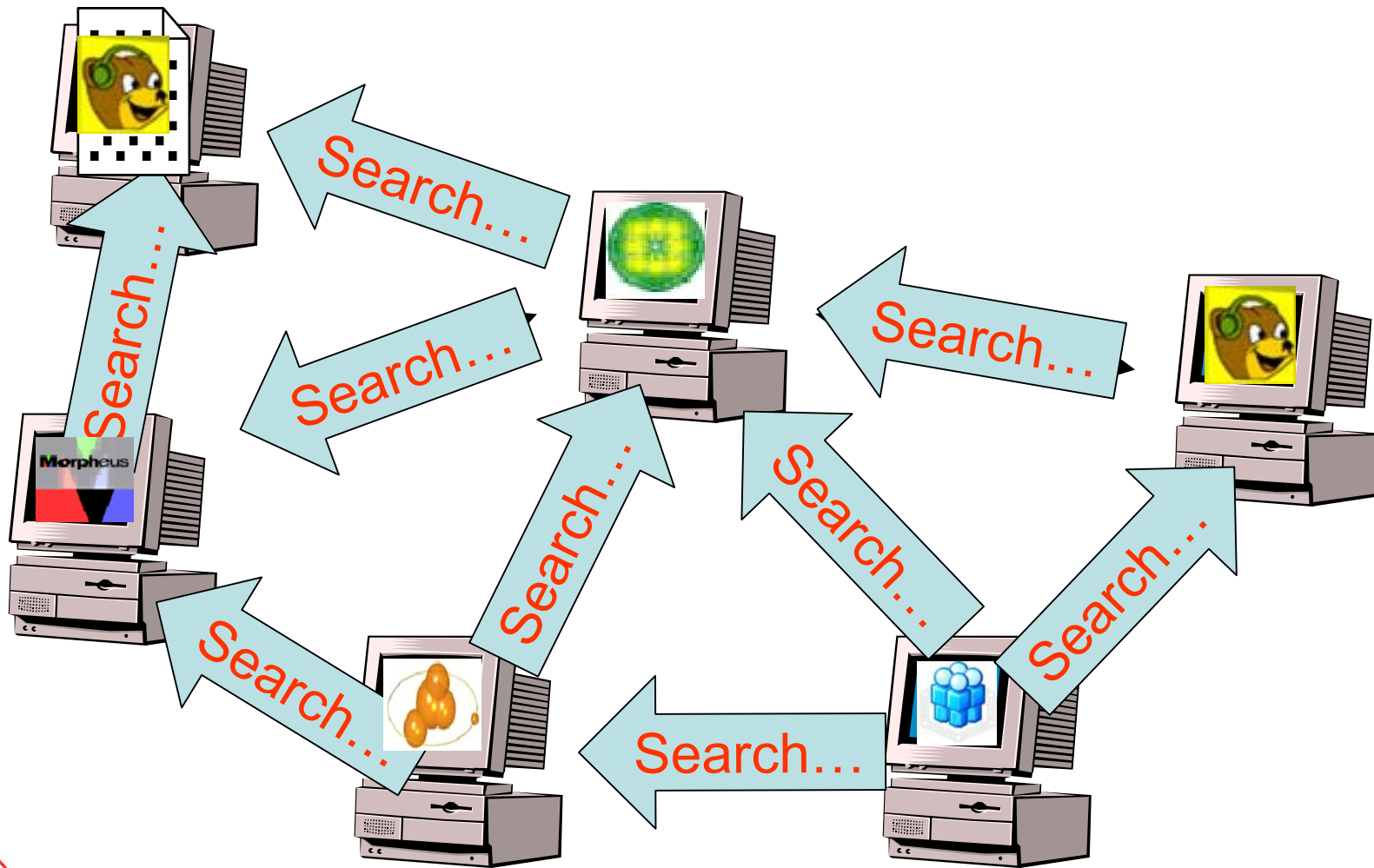
# Case Study: Napster



|                         |                  |
|-------------------------|------------------|
| Beach Boys: Pet Sounds  | @ 170.13.01.02   |
| Aphex Twin: Ptolemy     | @ 212.17.11.69   |
| ...: Ring Ring ...      | @ 129.132.13.122 |
| Pavement: Turich is ... | @ 129.132.13.122 |



# Case Study: Gnutella

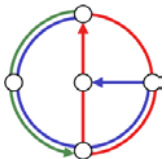




## Pros/Cons Gnutella



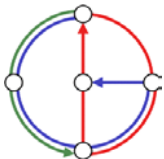
- totally **decentralized**
- totally **inefficient**
  - “flooding” = directionless searching
- Gnutella often does not find searched item
  - TTL
  - Gnutella “**not correct**”



# Dictionary ADT



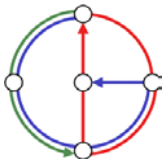
- A collection of objects
  - Each object uniquely identified by key
- Supports these operations:
  - **Search**(key) → object(key)
  - **Insert**(key, object) → OK?
  - **Delete**(key) → OK?



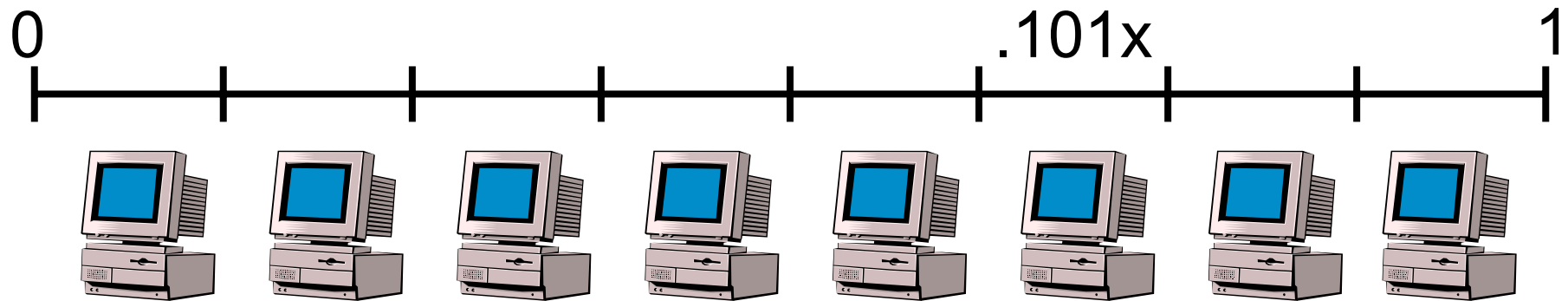
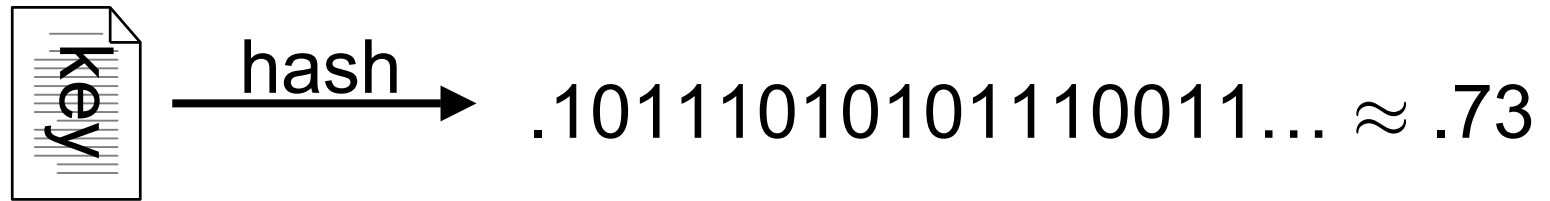
# Dictionary Implementations



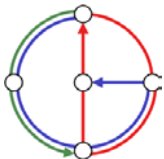
- **Classic** Implementations
  - Search Tree (balanced, B-Tree)
  - Hashing (various forms)
- **“Distributed”** Implementations
  - Linear Hashing
  - Consistent Hashing



# Distributed Hashing



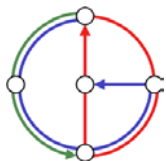
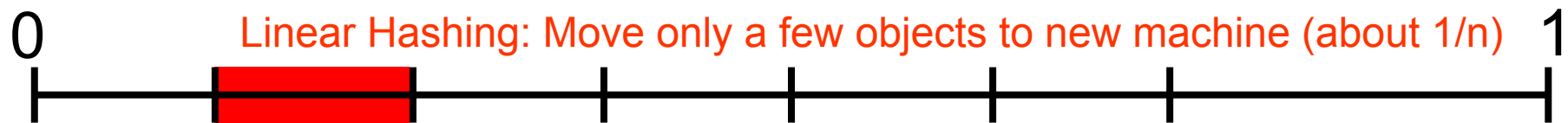
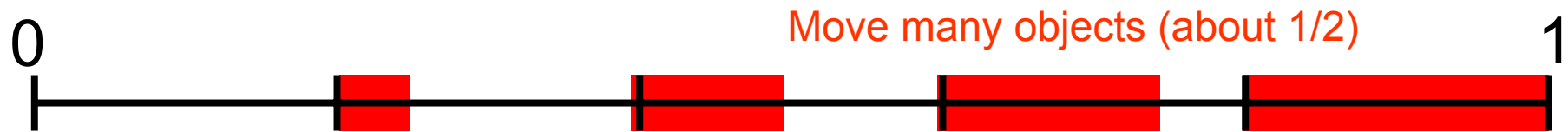
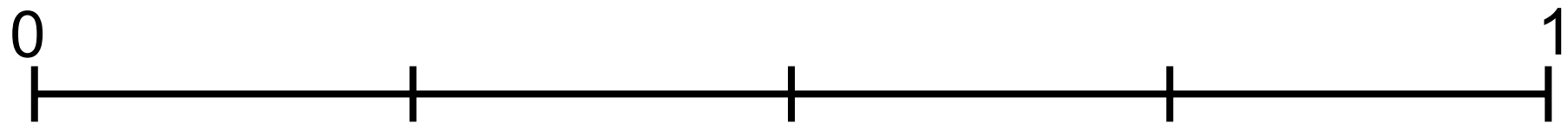
- Remark: Instead of storing a document at the right peer, just store a forward-pointer



# Linear Hashing



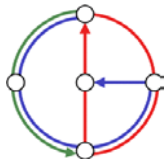
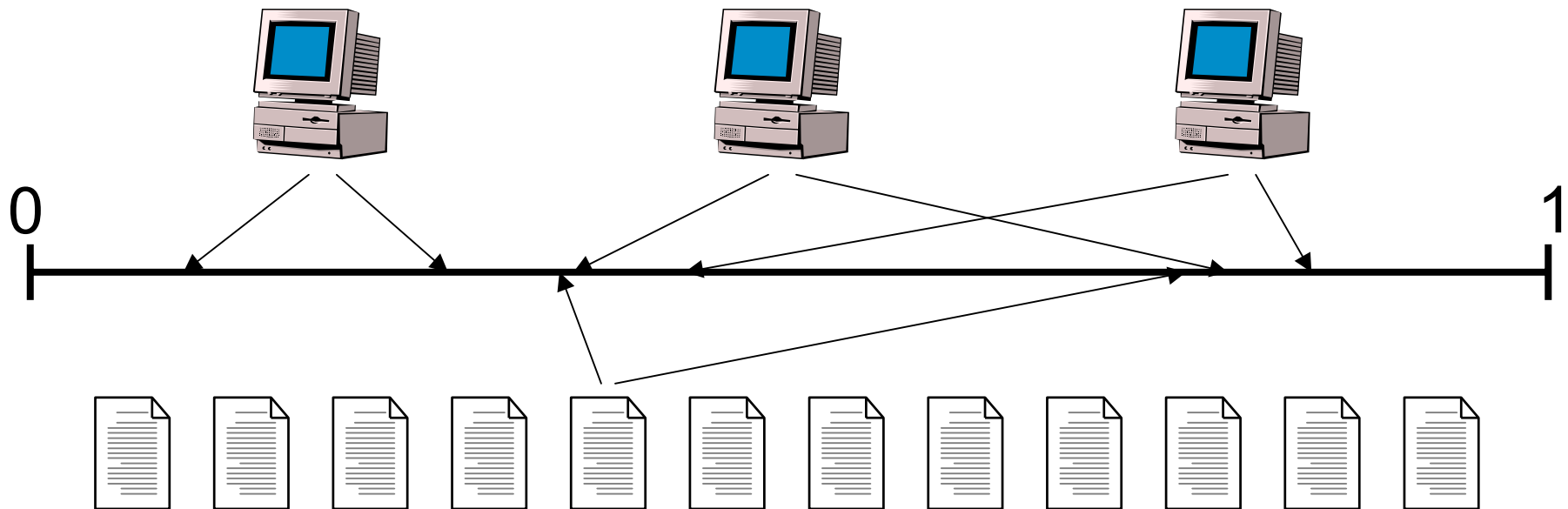
- Problem: More and more objects should be stored; need to buy new machines!
- Example: From 4 to 5 machines



# Consistent Hashing



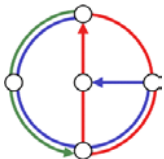
- Needs central dispatcher
- Idea: Also the machines get hashed! Each machine is responsible for the files closest to it. Use multiple hash funct. for reliability.



# Not quite happy yet...



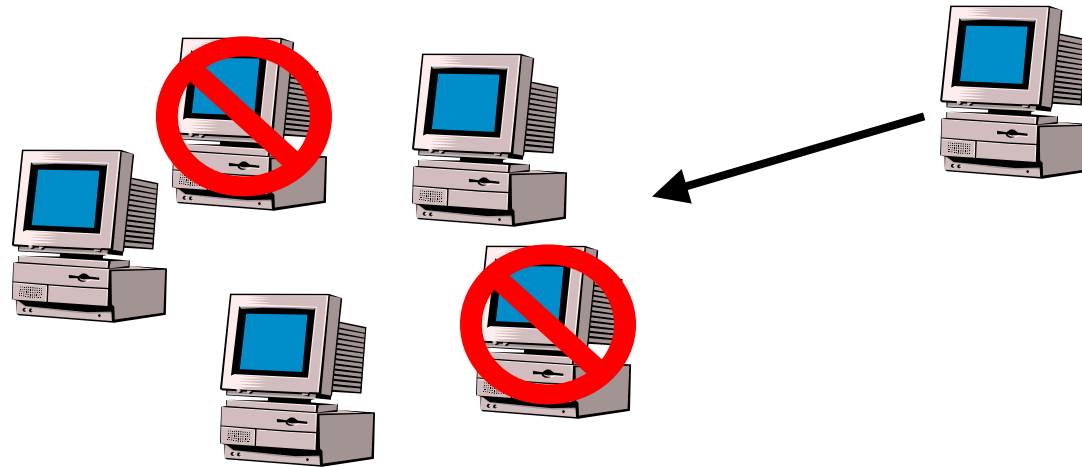
- Problem with both linear and consistent hashing is that all the participants of the system must know all peers...
- Number one challenge: **Dynamics!**
  - Peers join and leave



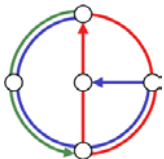
# Dynamics



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

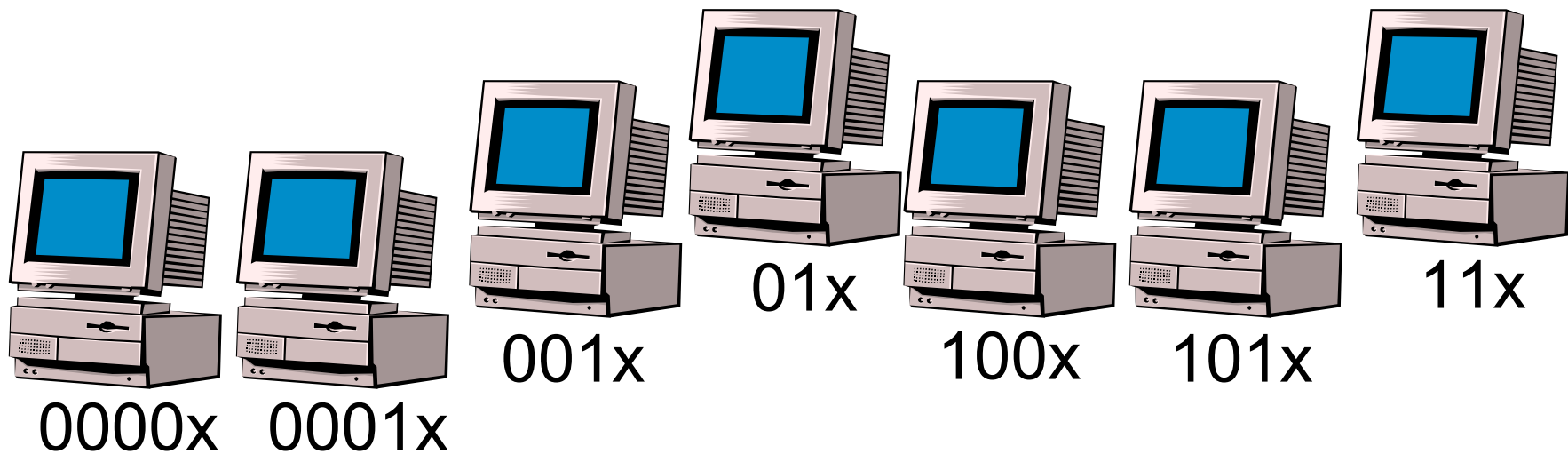
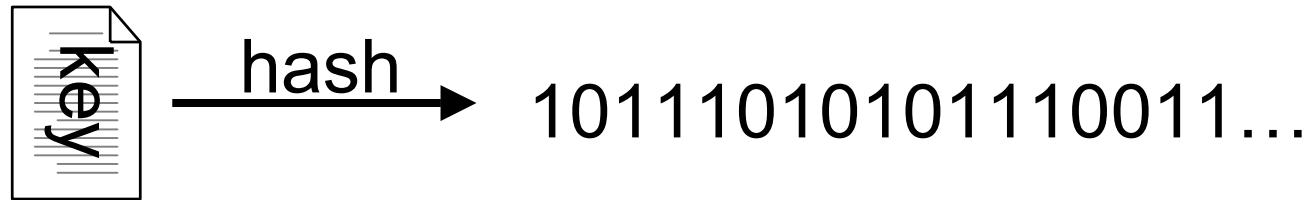


- Decentralized (“symmetric”) System
  - scalable, fault tolerant, dynamic

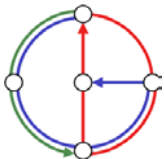




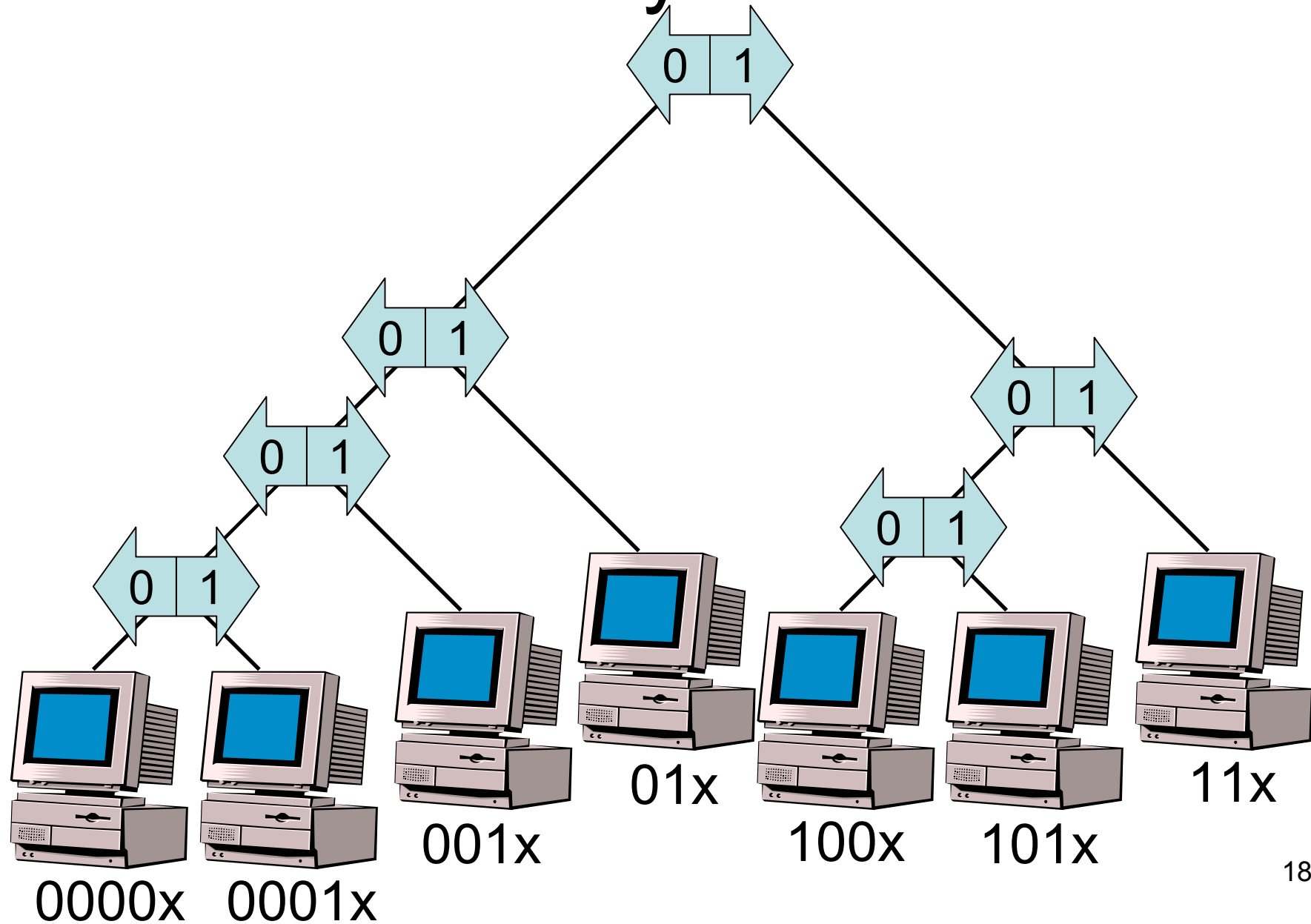
# P2P Dictionary = Hashing



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



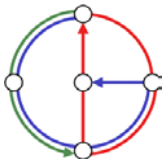
# P2P Dictionary = Search Tree



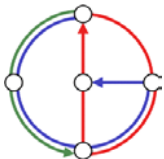
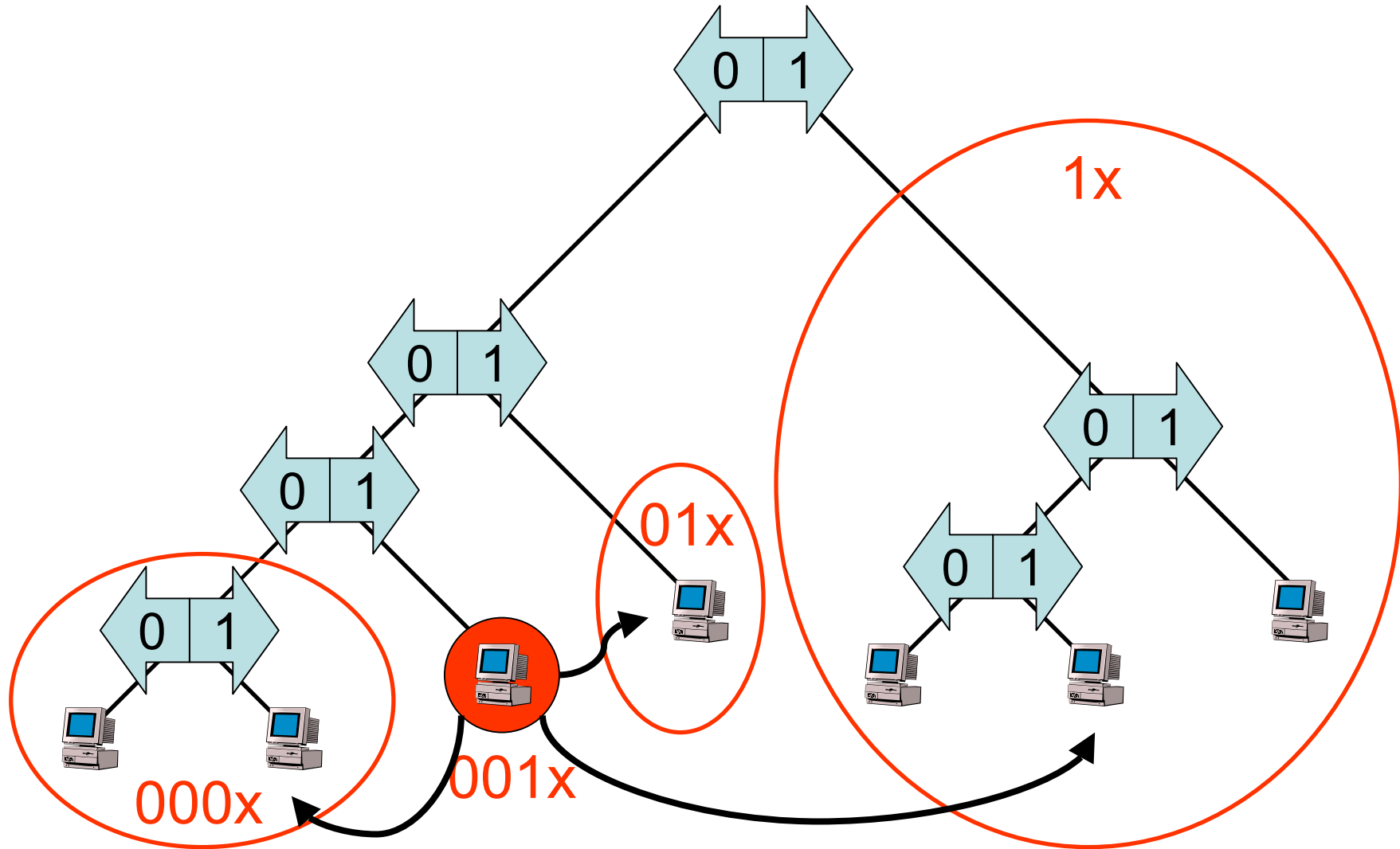
## But who stores search tree?



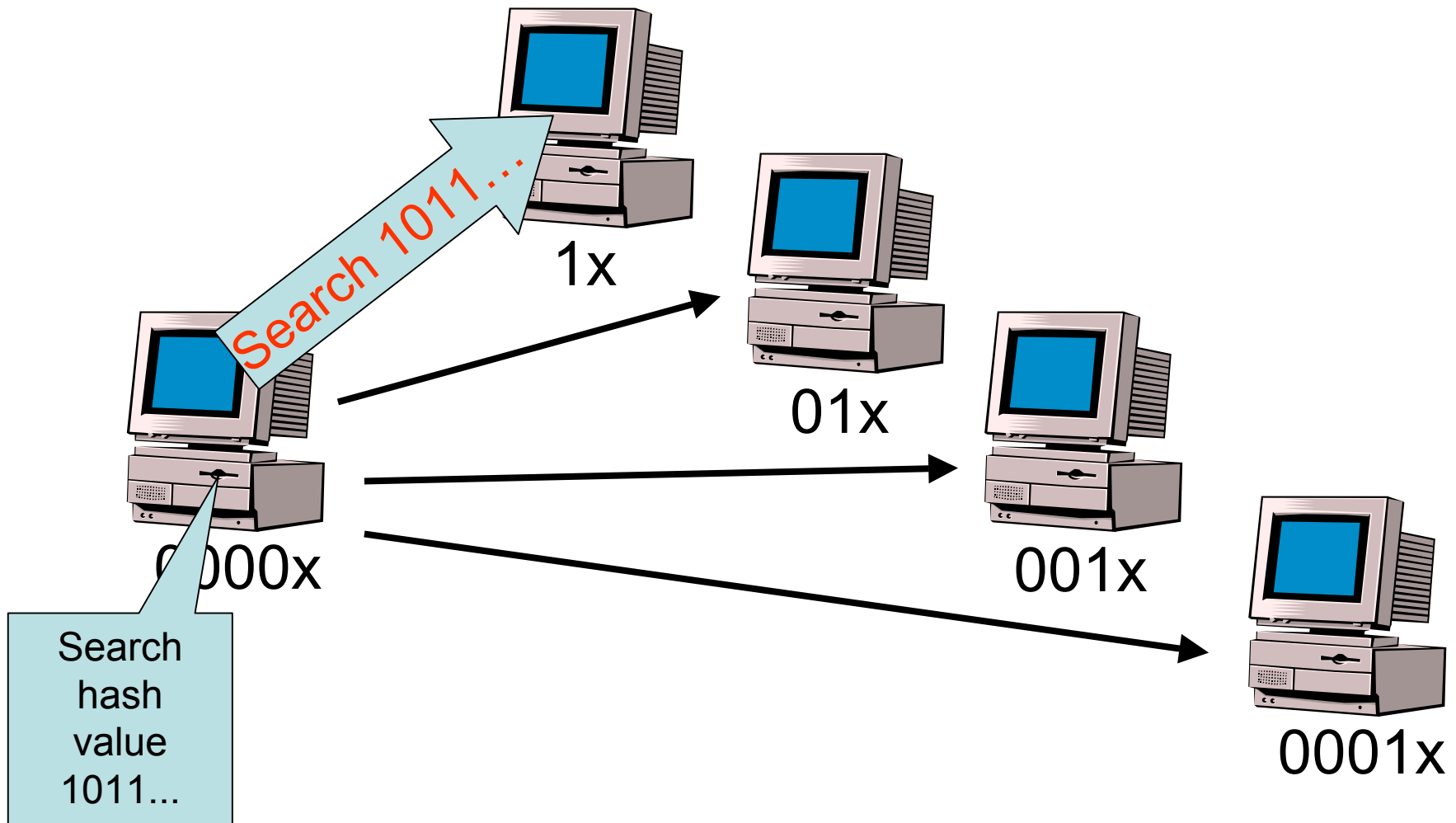
- In particular, where is the **root** stored?
  - Root is scalability & fault tolerance problem
  - There is **no root**...!
- If a peer wants to store/search, how does it know where to go?
  - Does every peer know all others?
  - Dynamics! If a peer leaves, all peers must be notified. Too much overhead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



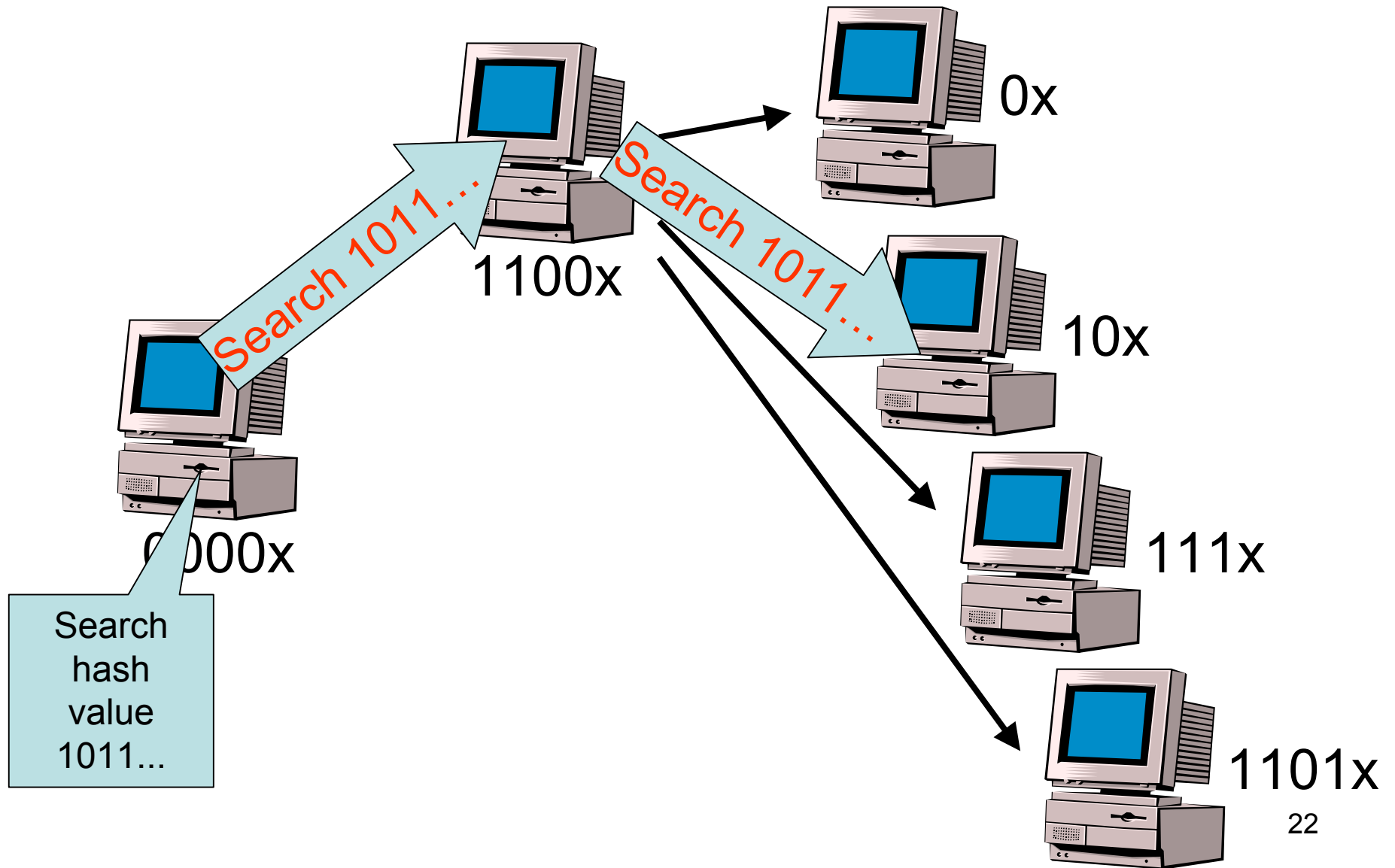
# The Neighbors of Peer 001x



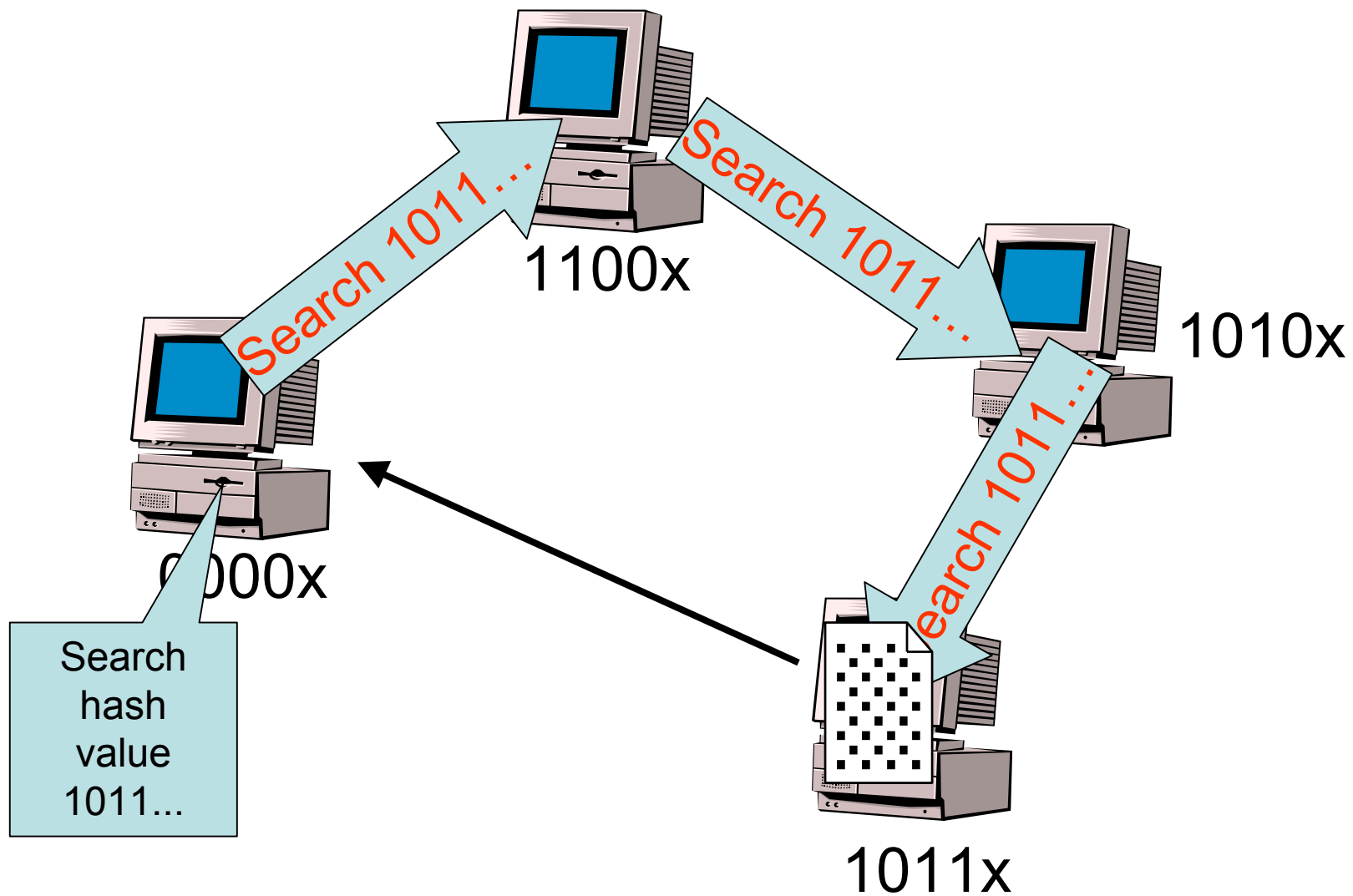
# P2P Dictionary: Search



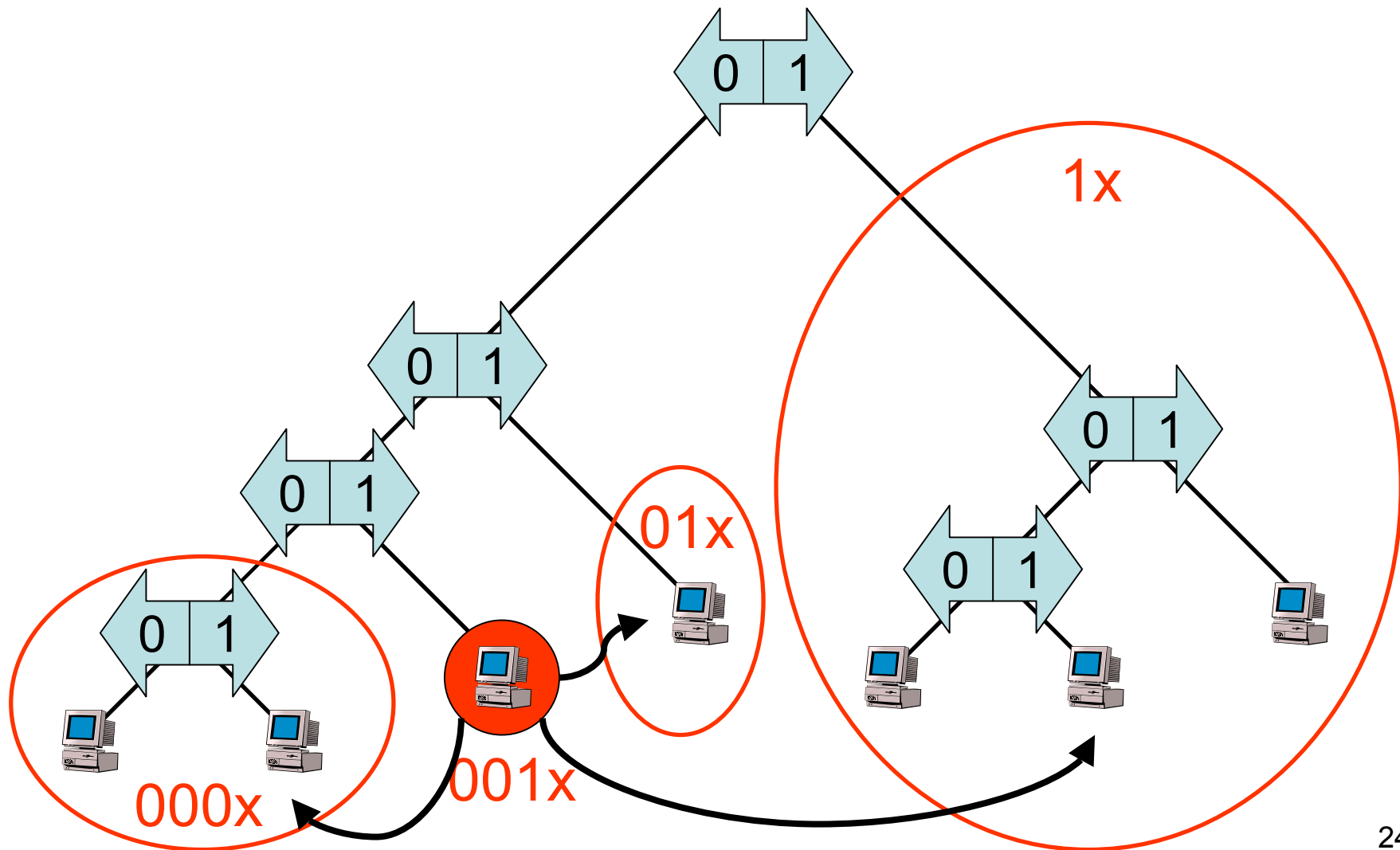
# P2P Dictionary: Search



# P2P Dictionary: Search

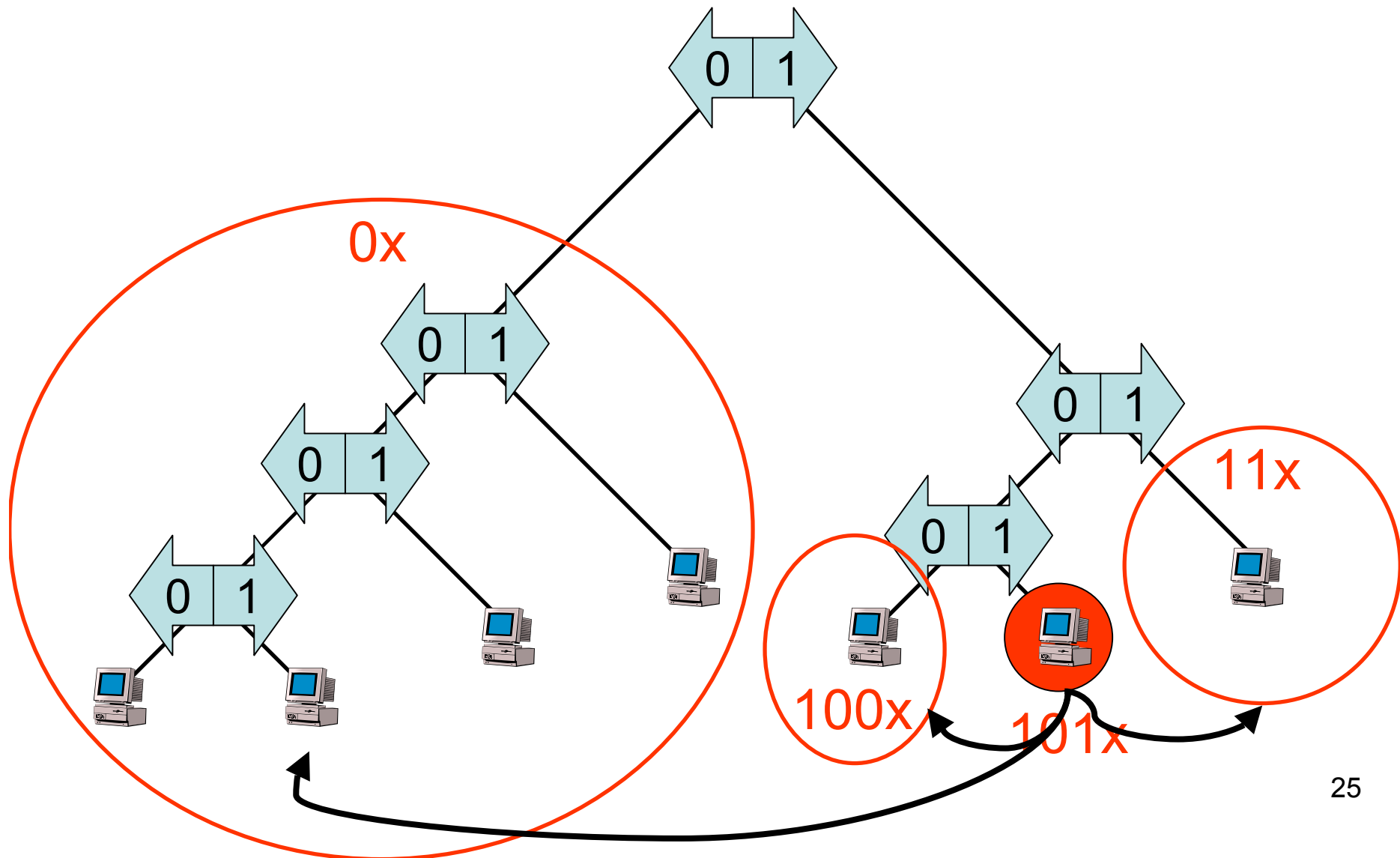


# Again: 001 searches 100





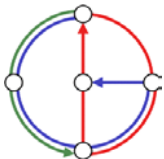
# 001 searches 100 (continued)



# Search Analysis



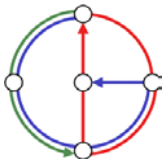
- We have  $n$  peers in system
- Assume that “tree” is roughly **balanced**
  - Leaves (peers) on level  $\log_2 n \pm \text{constant}$
- **Search has  $O(\log n)$  steps**
  - After  $k$ 'th step, you are in subtree on level  $k$
  - A “step” is a UDP (or TCP) message
  - Latency is dependent on P2P 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

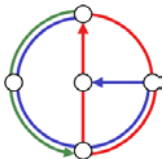


## 2. Find your place

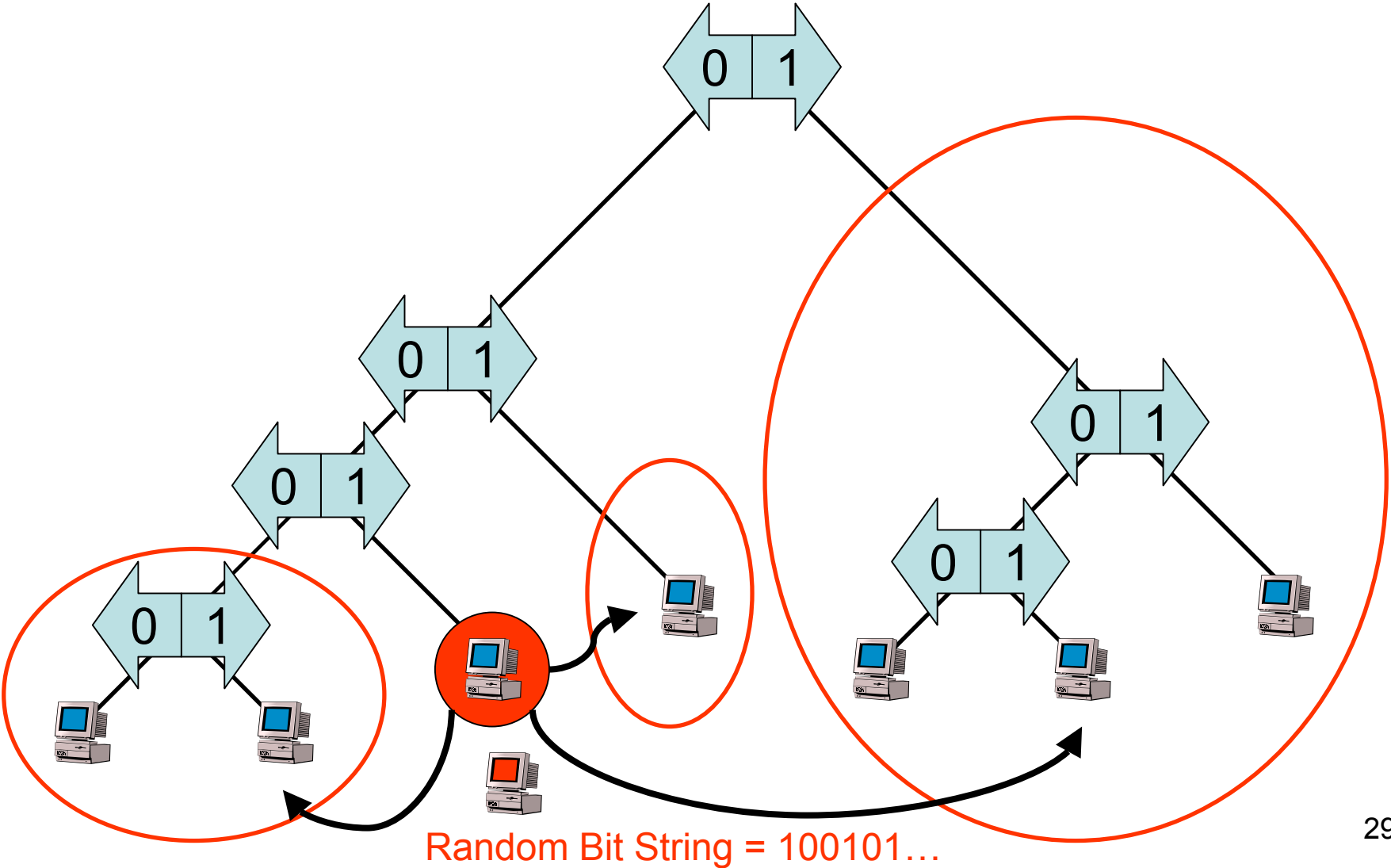


- The random method: Choose a **random bit string** (which determines the place)
- **Search**\* for the bit string
- **Split** with the current leave responsible for the bit string
- **Search**\* for your neighbors

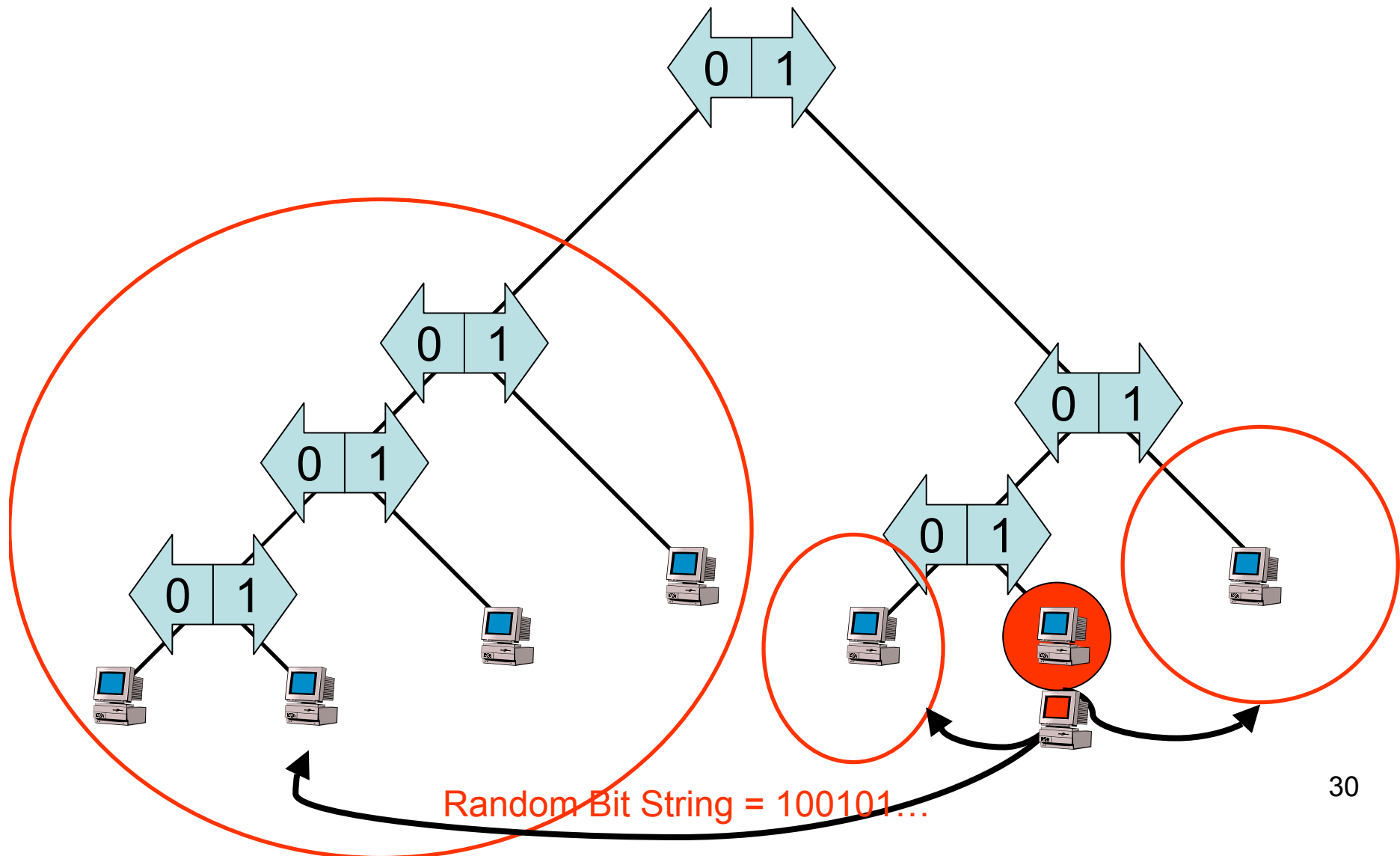
\* These are standard searches



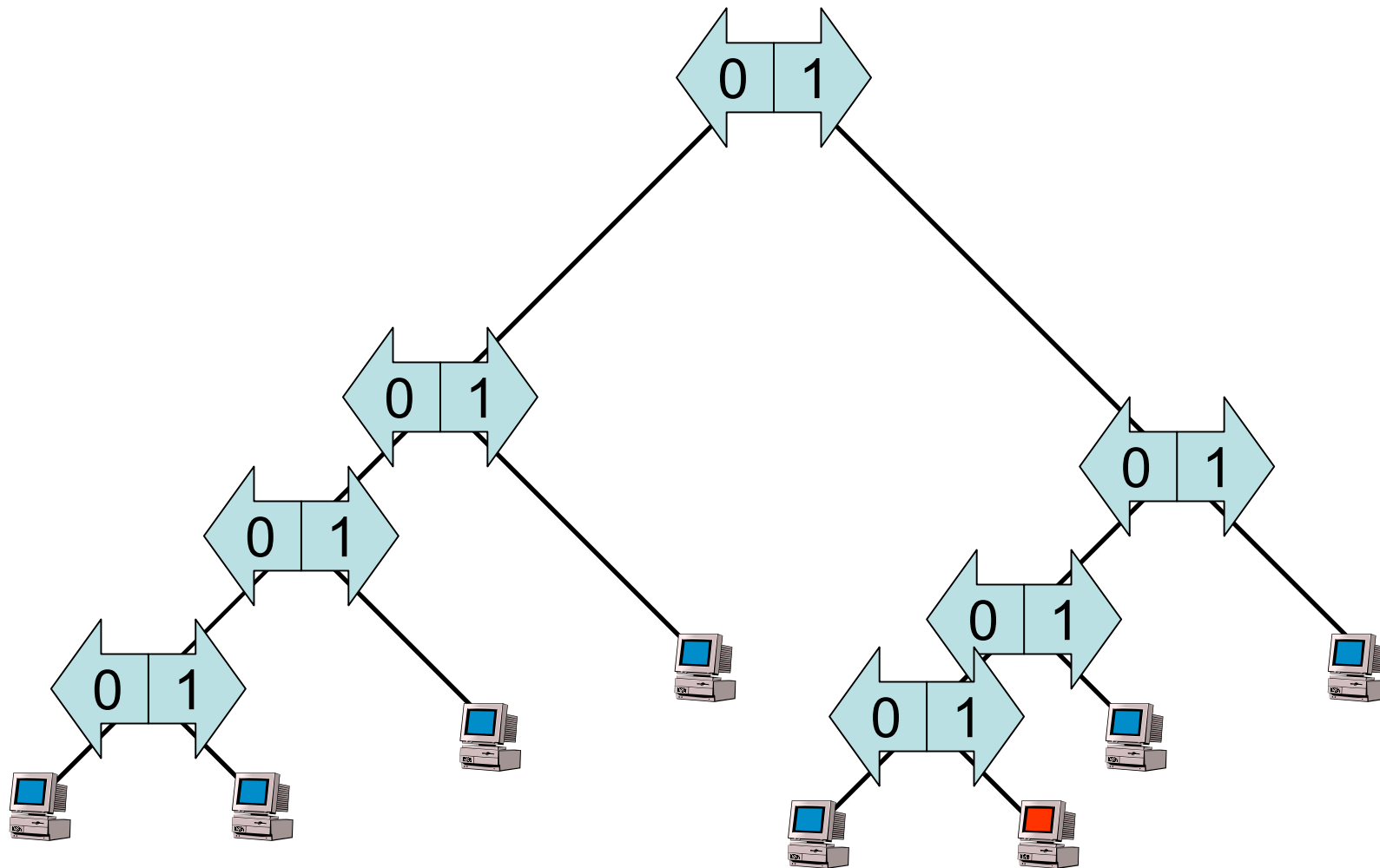
# Example: Bootstrap with 001 peer



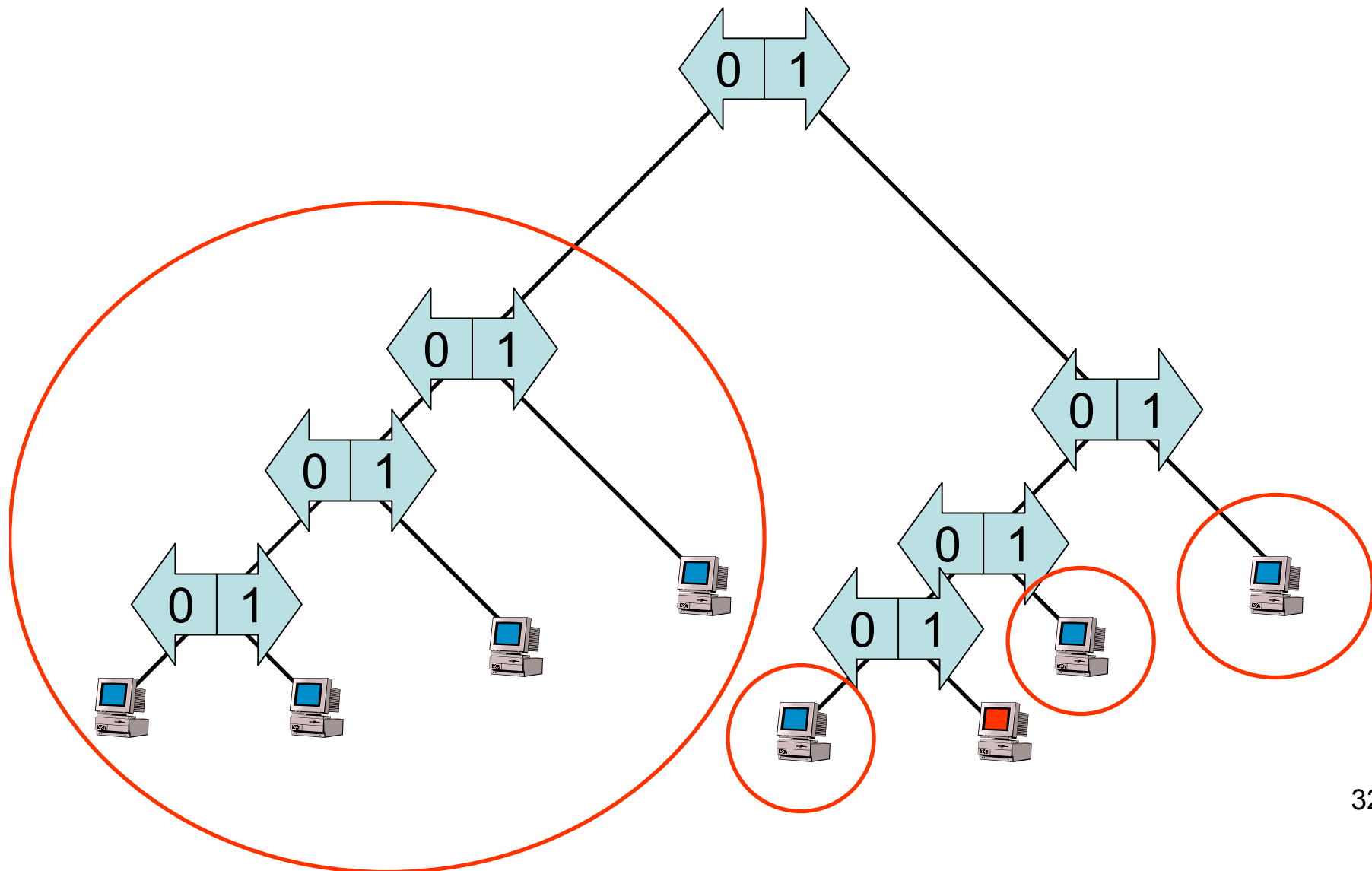
# Joiner searches 100101...



# Joiner found 100 leave → split



# Find neighbors

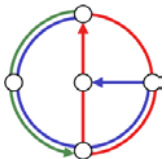




# Random Join Discussion



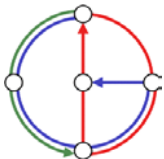
- If tree is balanced, the time to join is
  - $O(\log n)$  for the first part
  - $O(\log n) \cdot O(\log n) = O(\log^2 n)$  for the second part
- It is believed that since all the peers are chosen their position randomly, the tree will more or less be **balanced**.
  - However, theory and simulations show that this is widely believed but **not really true**.



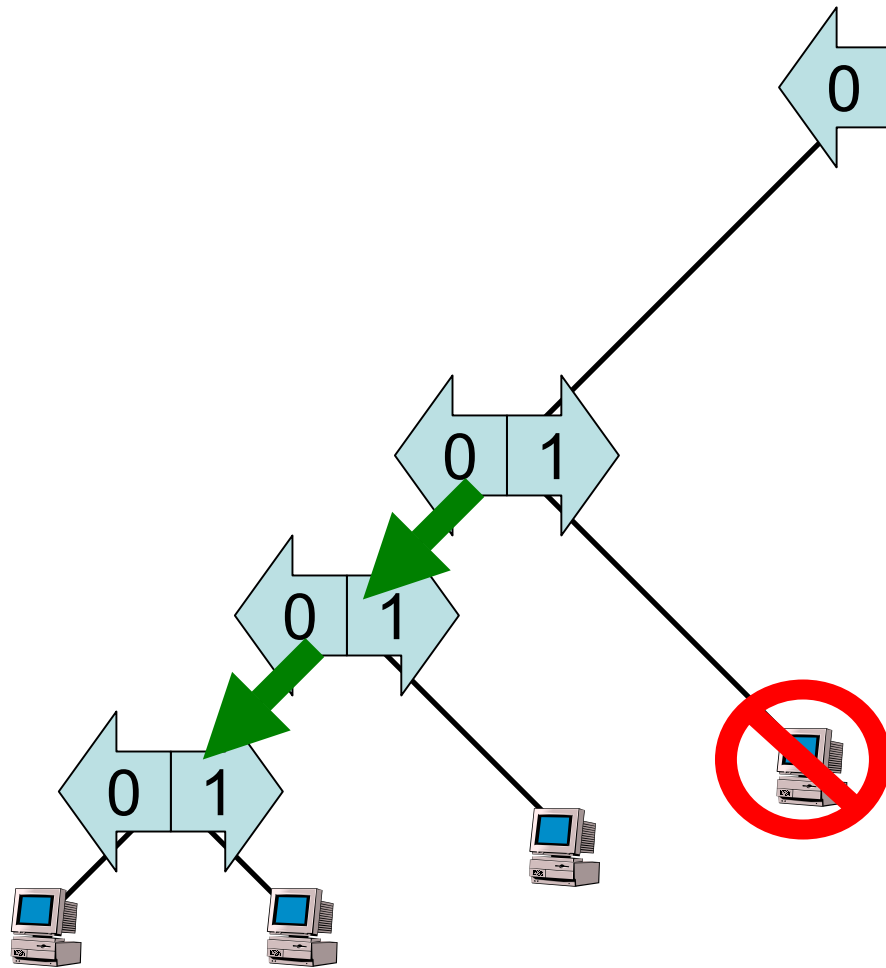
# Leave



- 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!



# Peer 01 leaves spontaneously

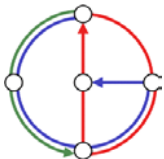


1. Go down sibling tree, until you hit sibling leaves.
2. Make the left sibling the new common node.
3. Move the free right sibling to the empty spot.

## Was that all?



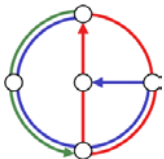
- Yes, you now mastered all the P2P basics...  
**Congratulations!**
- But there are some nasty “technicalities” 😊
- Most importantly we would like to know **what happened to the data that was stored at the peer that left** (important question if we want to use the P2P network as a storage/file system). We study that soon...
- First some other comments...



## Questions of experts...



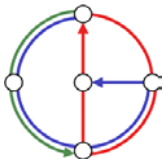
- Q: I know so many other peer-to-peer systems; they are completely different from the one you showed us!
- A: They *look* different, but in fact the difference comes mostly from the way they are presented. (I give a few examples on the next slides)



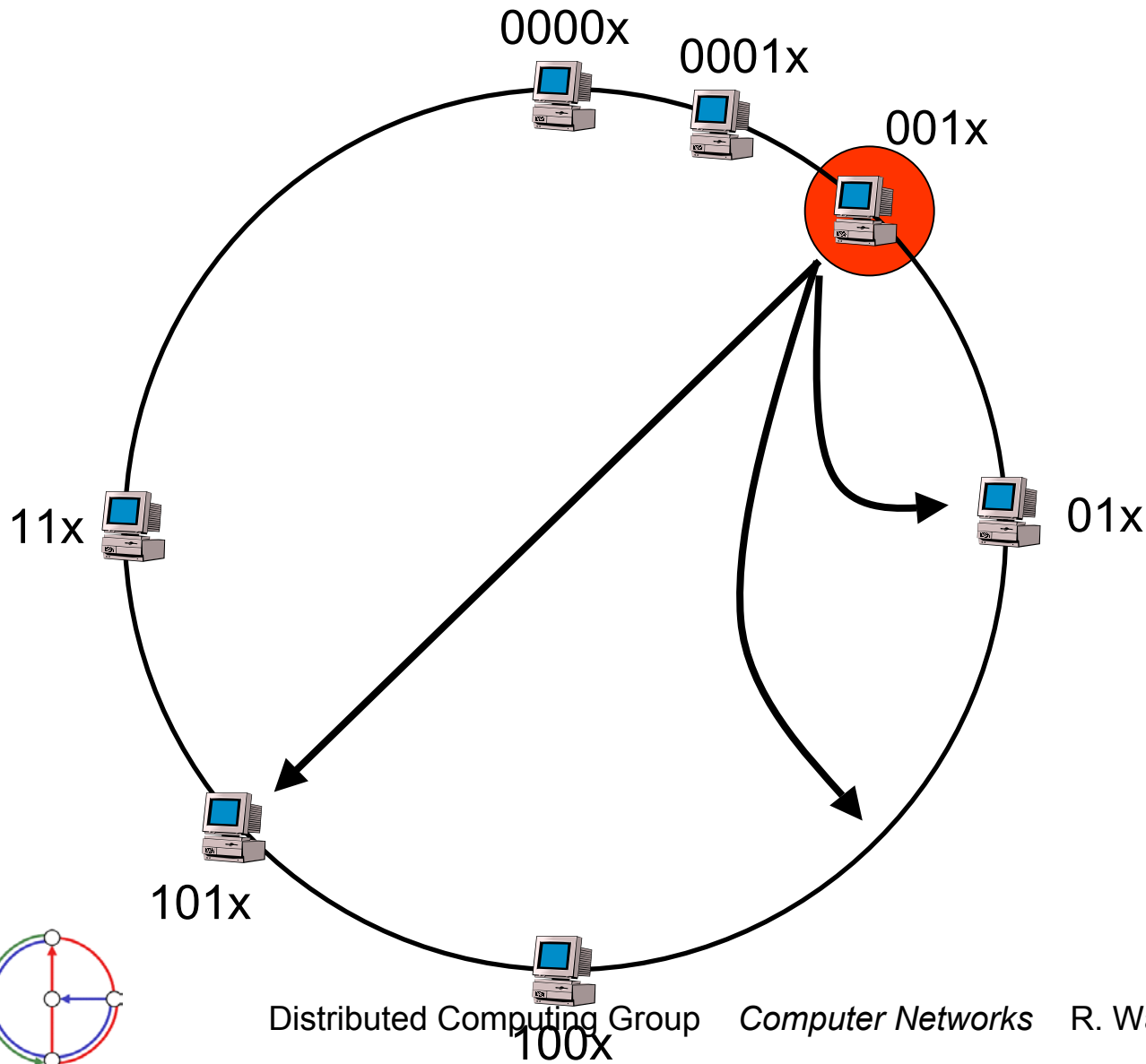
# Chord



- The most cited 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.
- CFS, Ivy



# Chord

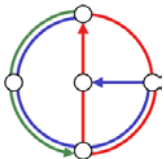


- Every peer has  $\log n$  many neighbors; one in about distance  $2^{-k}$ ,  $k=1, 2, \dots, \log n$

# Skip List

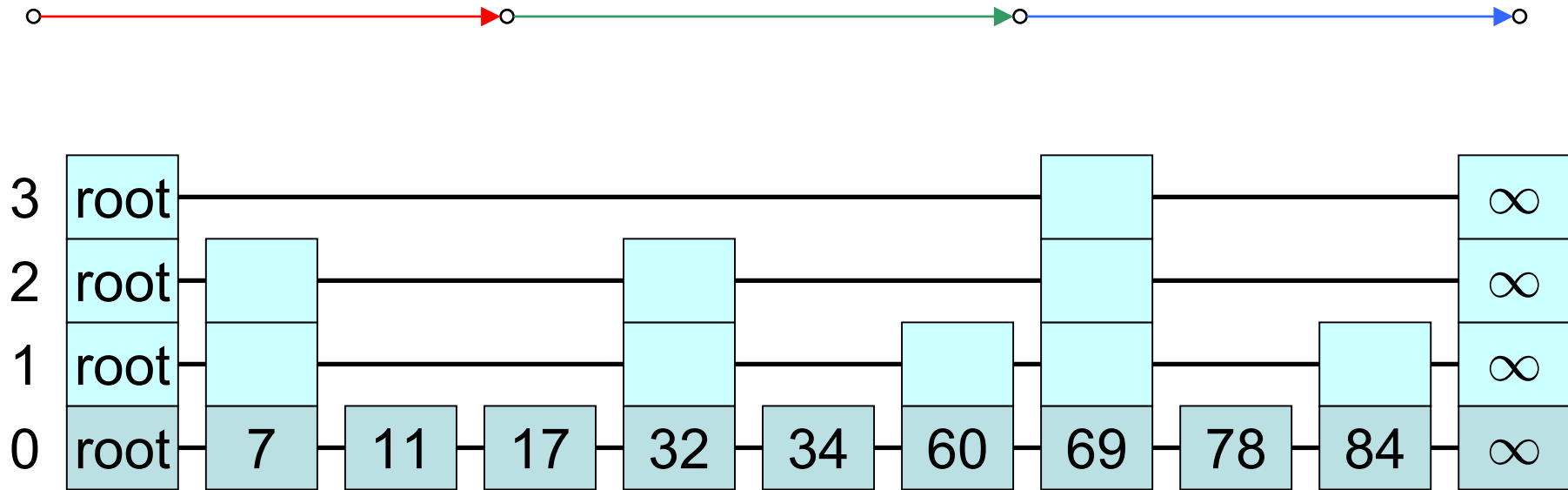


- Are you afraid of programming balanced search trees (e.g. **AVL** or **red-black** tree)?!?
- Then the skip list is a data structure for you!
- Idea: Ordered linked list with extra pointers

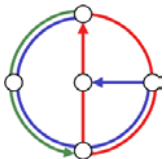




# Skip List



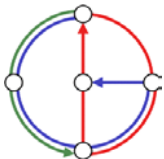
- (Doubly) linked list, with sorted items
- All items have additional pointers on levels 1, ..., k, with probability  $2^{-k}$
- Search, insert, delete: Start with root, search for the right interval on highest level, then continue with lower levels.



# Skip List



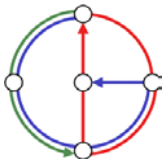
- It can easily be shown that search, insert, and delete terminate in  $O(\log n)$  expected time, if there are  $n$  items in the skip list
- Also, on expectation, the number of pointers is only **twice** as many as with a regular linked list, thus the memory overhead is edible
- As a plus, the items are always ordered...



# Skip Net



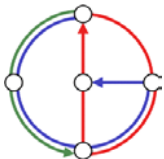
- Use the skip list as a peer-to-peer architecture: Again each peer gets a random value between 0 and 1, and is then responsible for storing that interval.
- Instead of a root and a sentinel node (“ $\infty$ ”), the list is short-wired as a **ring**
- There exist several proposals towards this end...



## Many many others...



- Original work by **Plaxton, Rajaraman, and Richa**; “unfortunately” theory paper, so it include many other things... similar proposals are Pastry/Tapestry, or **Kademlia**.
- Some proposals improve the design; e.g. The **Viceroy** proposal which is Butterfly-based and therefore only needs a constant number of neighbors per peer.
- Some proposals are more complicated than needed; e.g. **CAN**
- Closest/best design in reality is **Freenet**. However, Freenet has some questionable design properties



## Why should I care?



- Q: I **don't** want to program a worldwide music stealing application, so why should I care?
- A: Many future networking applications will have a form of decentralized control, for **scalability, fault-tolerance, and security**.
- Example: P2P Spam-Filtering (e.g. SpamNet).

