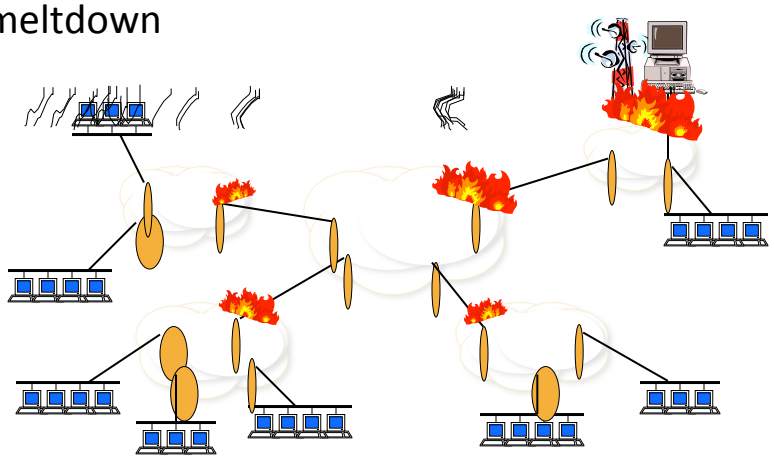
Peer-to-Peer

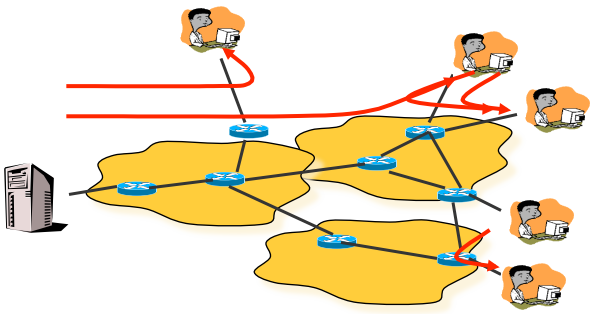
Based on materials by
Ken Birman and Srinivasan Seshan

Scaling Problem

- Millions of clients ⇒server and network meltdown



P2P System



- Leverage the resources of client machines (peers)
  - Computation, storage, bandwidth

3

Why P2P?

- Harness lots of spare capacity
  - 1 Big Fast Server: 1Gbit/s, $10k/month++
  - 2,000 cable modems: 1Gbit/s, $??
  - 1M end-hosts: Uh, wow.
- Build self-managing systems / Deal with huge scale
  - Same techniques attractive for both companies / servers / p2p
    - E.g., Akamai's 14,000 nodes
    - Google's 100,000+ nodes

4

Outline

- p2p file sharing techniques
  - Downloading: Whole-file vs. chunks
  - Searching
    - Centralized index (Napster, etc.)
    - Flooding (Gnutella, etc.)
    - Smarter flooding (KaZaA, …)
    - Routing (Freenet, etc.)
- Uses of p2p - what works well, what doesn't?
  - servers vs. arbitrary nodes
  - Hard state (backups!) vs soft-state (caches)
- Challenges
  - Fairness, freeloading, security, …

5

P2P File-Sharing

- Quickly grown in popularity
  - Dozens or hundreds of file sharing applications
  - 35 million American adults use P2P networks -- 29% of all Internet users in US!
  - Audio/Video transfer now dominates traffic on the Internet

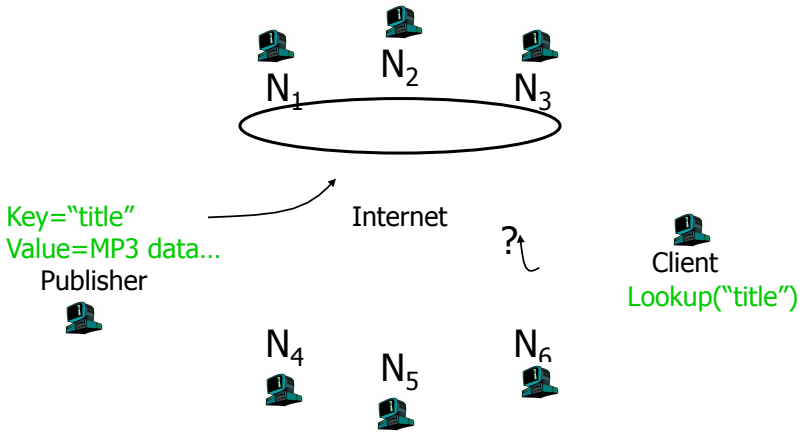
6

What's Out There?

	Central	Flood	Super-node flood	Route
Whole File	Napster	Gnutella		Freenet
Chunk Based	BitTorrent		KaZaA (bytes, not chunks)	DHTs

7

Searching (1)



8

Searching (2)

- Needles vs. Haystacks
  - Searching for top 40, or an obscure punk track from 1981 that nobody's heard of?
- Search expressiveness
  - Whole word? Regular expressions? File names? Attributes? Whole-text search?
    - (e.g., p2p gnutella or p2p google?)

9

Framework

- Common Primitives:
  - **Join**: how to I begin participating?
  - **Publish**: how do I advertise my file?
  - **Search**: how to I find a file?
  - **Fetch**: how do I retrieve a file?

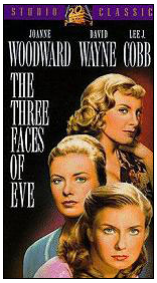
10

What Problems are "Fundamental?"

- Assume clients serve up what people download
  - For less popular items, fewer sources
  - Generally inaccessible
    - If churn is constant
- But clients fall into two categories:
  - Well-connected clients that hang around
  - Poorly-connected clients that also churn

What Problems are "Fundamental?"

- One can have, some claim, as many electronic personas as one has the time and energy to create. – *Judith S. Donath.*
- So-called "Sybil attack…."
  - Attacker buys a high performance computer cluster
  - It registers *many times* with Napster
  - Real clients get poor service or even get snared
  - No p2p system can easily defend against Sybil attacks!



Napster

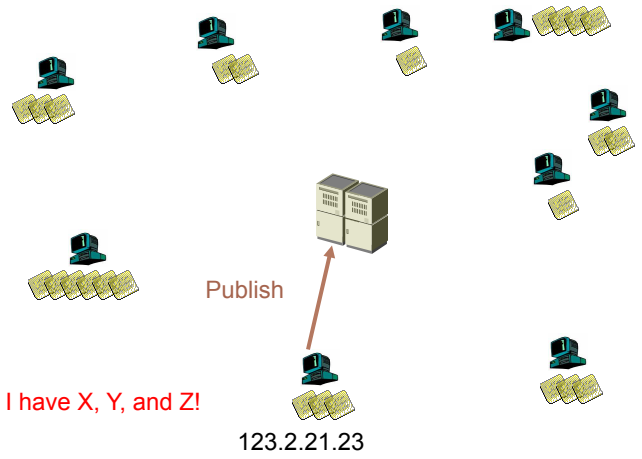
P2P: CENTRALIZED DATABASE

Napster

- History
  - 1999: Sean Fanning launches Napster
  - Peaked at 1.5 million simultaneous users
  - Jul 2001: Napster shuts down
- 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

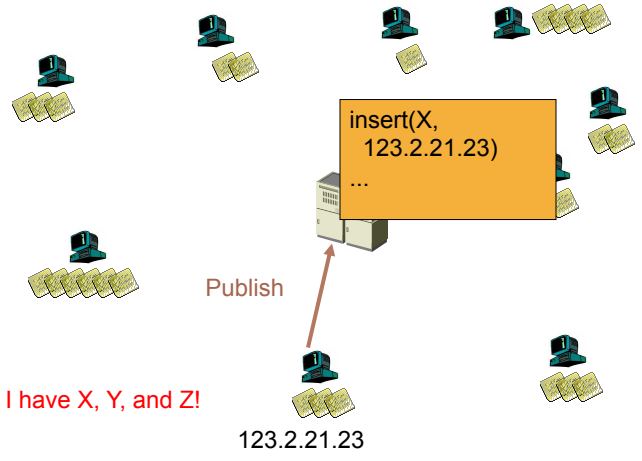
14

Napster: Publish



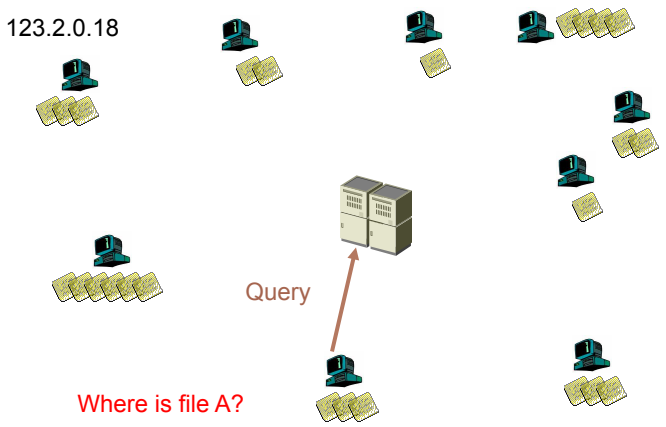
15

Napster: Publish



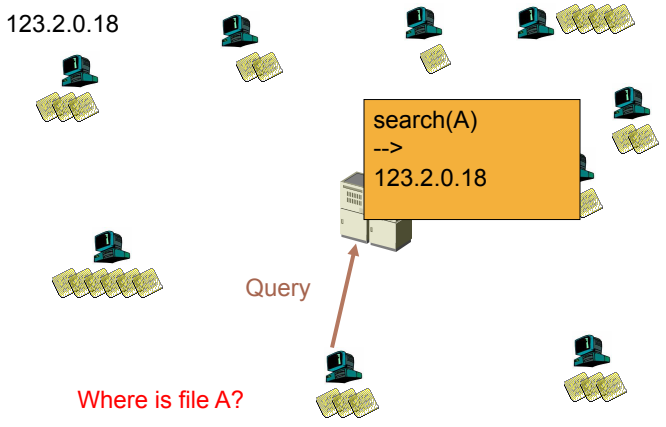
16

Napster: Search



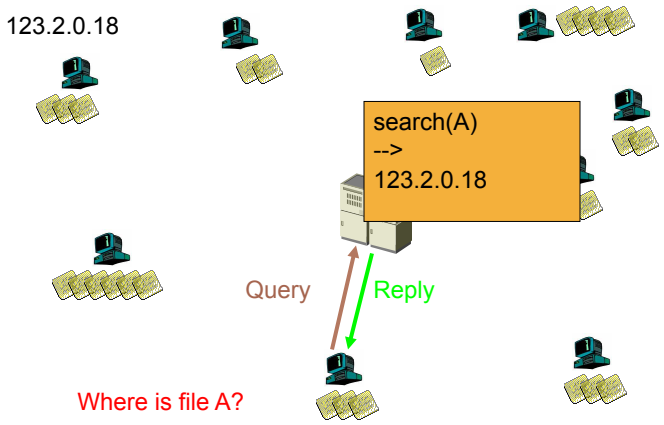
17

Napster: Search



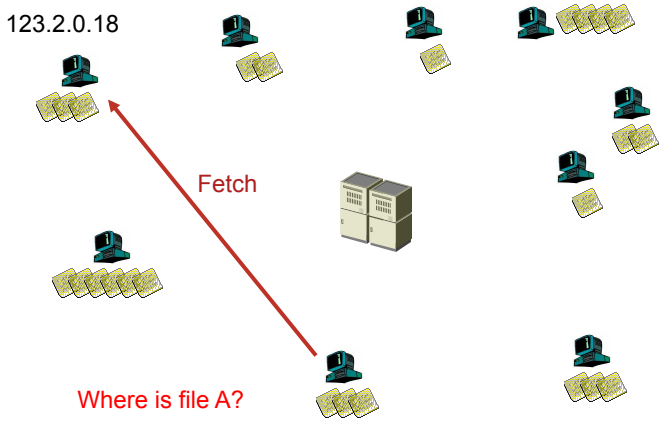
18

Napster: Search



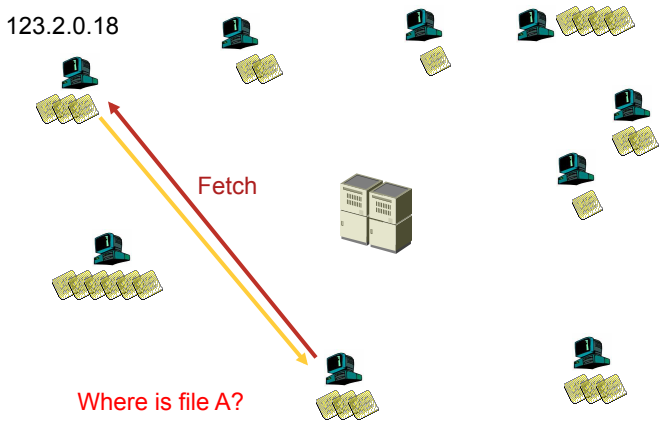
19

Napster: Search



20

Napster: Search



21

Napster: Discussion

- 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

22

Gnutella, Kazaa

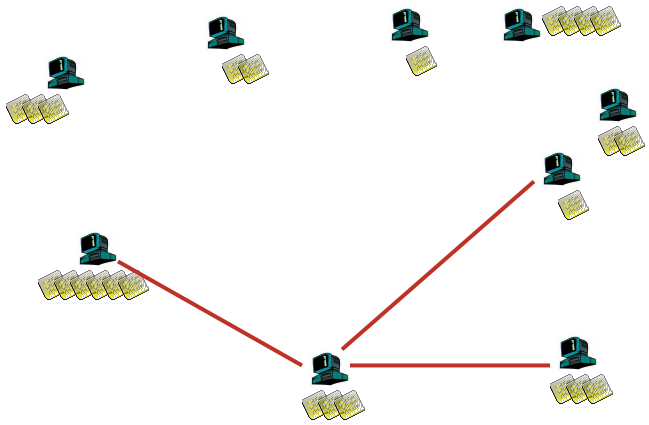
P2P: QUERY FLOODING

Gnutella

- History
  - In 2000, J. Frankel and T. Pepper from Nullsoft released Gnutella
  - In 2001, many protocol enhancements including "ultrapeers"
- 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

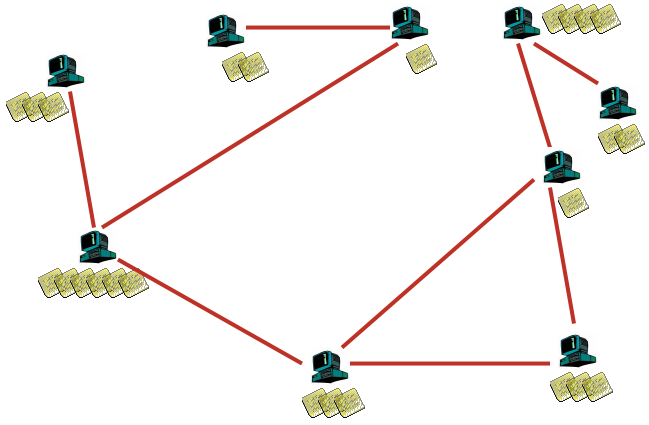
24

Gnutella: Search



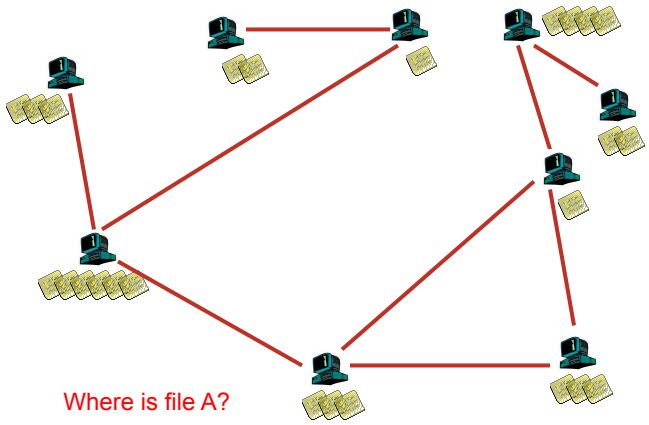
25

Gnutella: Search



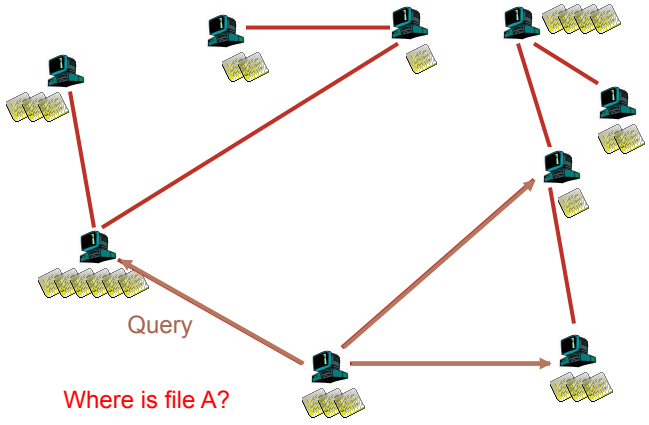
26

Gnutella: Search



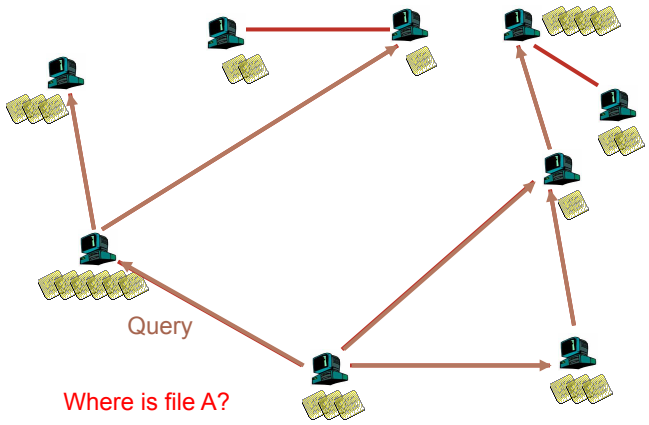
27

Gnutella: Search



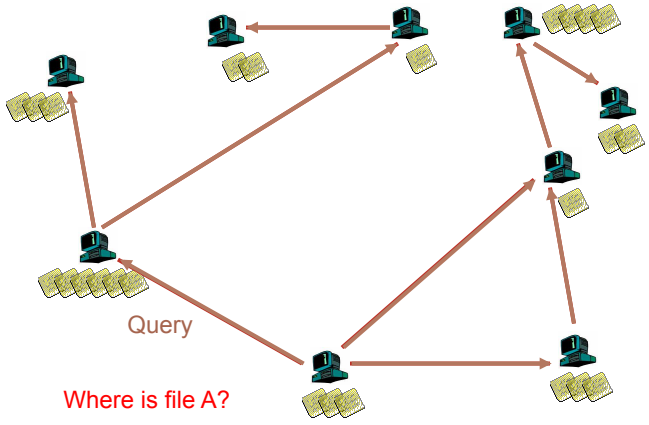
28

Gnutella: Search



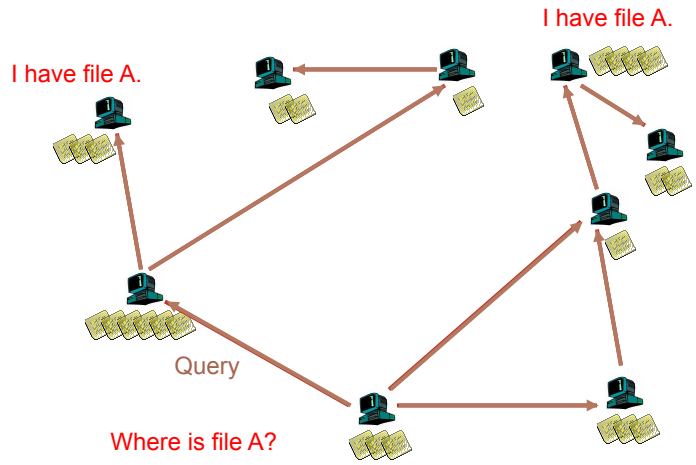
29

Gnutella: Search



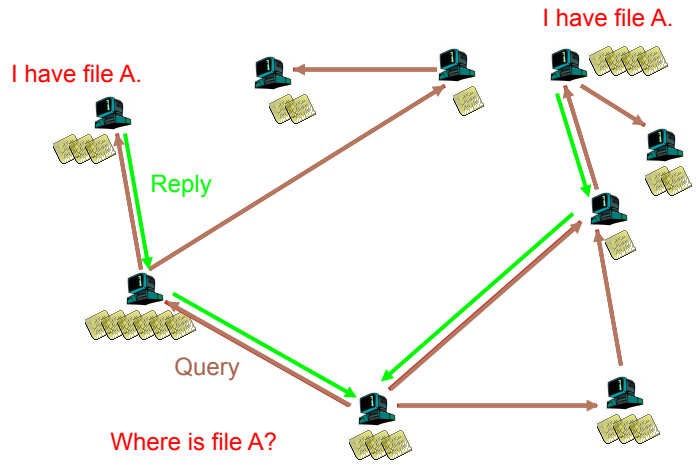
30

Gnutella: Search



31

Gnutella: Search



32

Gnutella: Discussion

- Pros:
  - Fully de-centralized
  - Search cost distributed
  - Processing at each node permits powerful search semantics
- Cons:
  - Search scope is O(N)
  - Search time is O(???)
  - Nodes leave often, network unstable
- TTL-limited search works well for haystacks.
  - For scalability, does NOT search every node. May have to re-issue query later

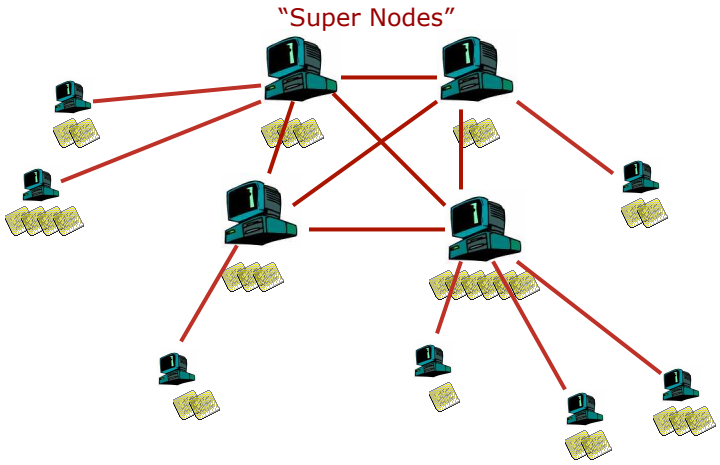
33

Kazaa

- "Supernode" Query Flooding:
  - Join: on startup, client contacts a "supernode" ... may at some point become one itself
  - Publish: send list of files to supernode
  - Search: send query to supernode, supernodes flood query amongst themselves.
  - Fetch: get the file directly from peer(s); can fetch simultaneously from multiple peers

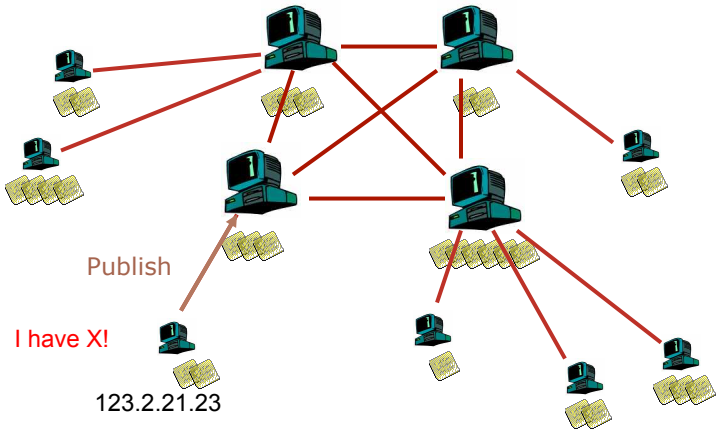
34

Kazaa: Network Design



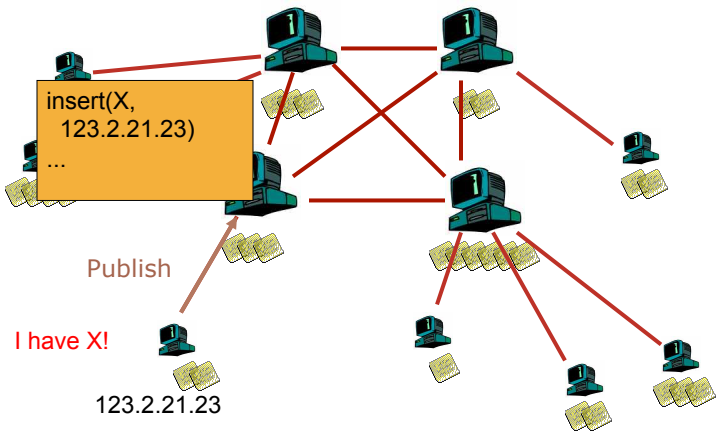
35

Kazaa: File Insert



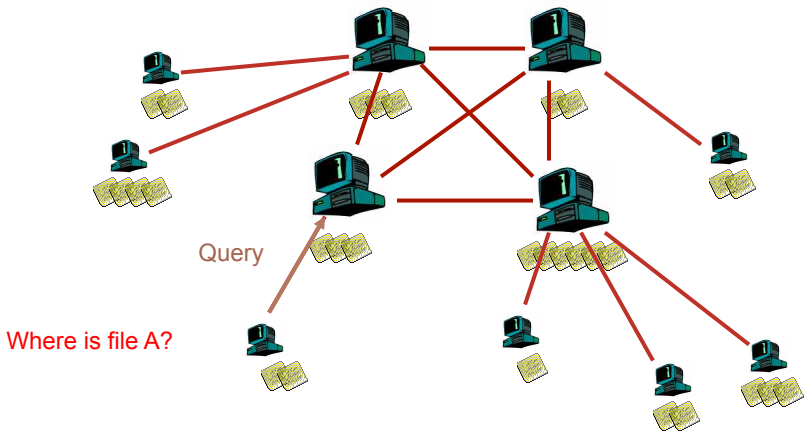
36

Kazaa: File Insert



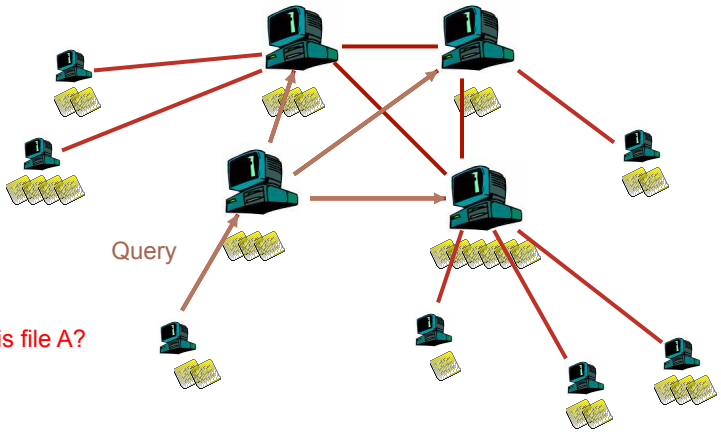
37

Kazaa: File Search



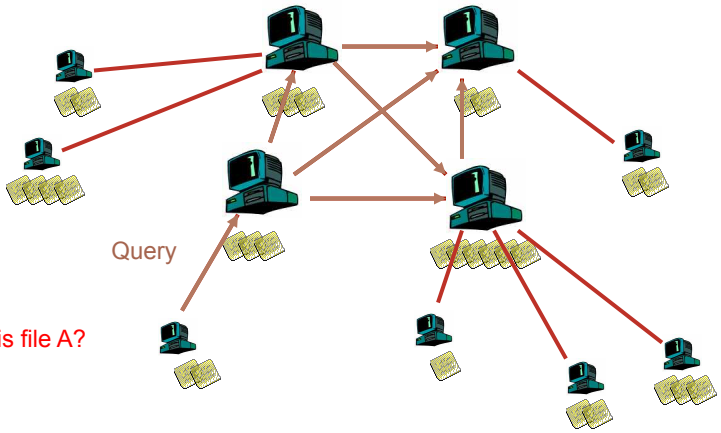
38

Kazaa: File Search



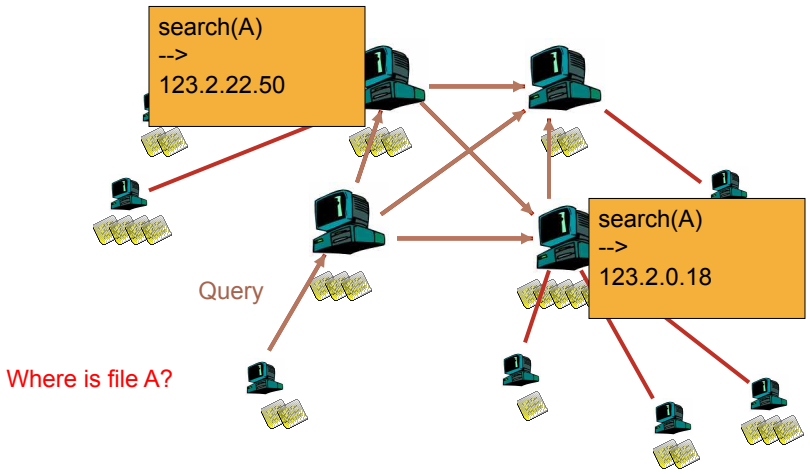
39

Kazaa: File Search



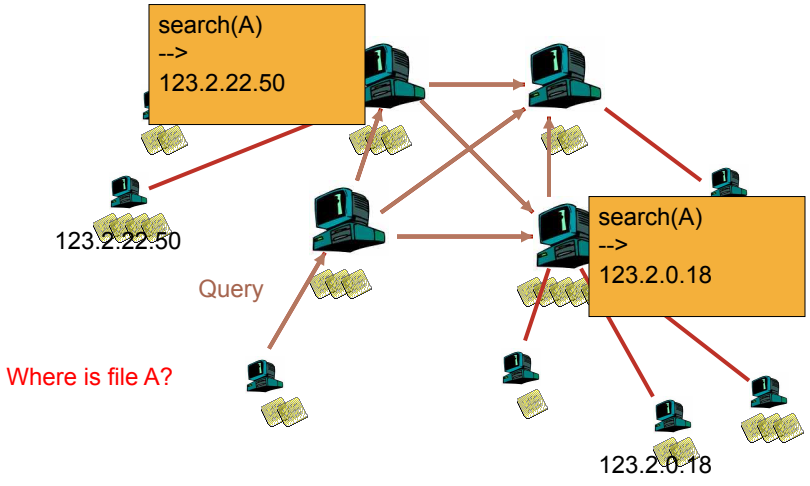
40

Kazaa: File Search



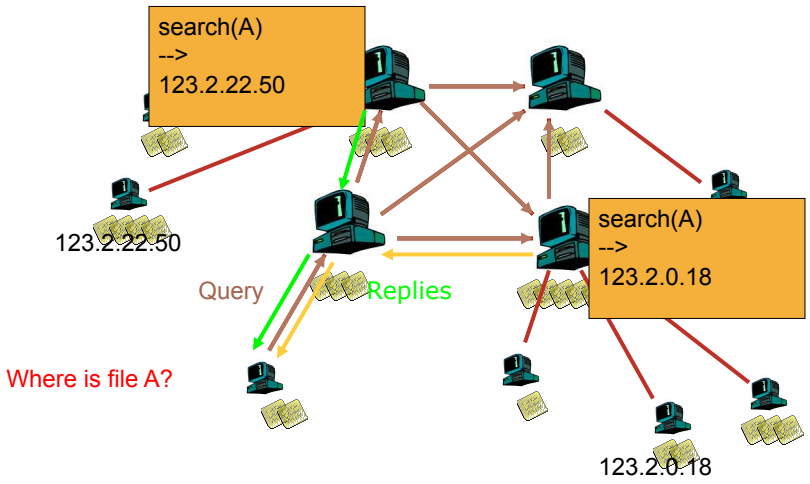
41

Kazaa: File Search



42

Kazaa: File Search



43

Kazaa: File Fetching

- How to tell?
  - Must be able to distinguish identical files
  - Not necessarily same filename
  - Same filename not necessarily same file...
- Use Hash of file
  - KaZaA uses UUHash: fast, but not secure
  - Alternatives: MD5, SHA-1
    - No longer secure…SHA-512? SHA-3?
- How to fetch?
  - Get bytes [0..1000] from A, [1001...2000] from B
    - Striping file across nodes

44

Kazaa: Discussion

- Pros:
  - Tries to take into account node heterogeneity:
    - Bandwidth
    - Host Computational Resources
    - Host Availability (?)
  - Rumored to take into account network locality
- Cons:
  - Mechanisms easy to circumvent
  - Still no real guarantees on search scope or search time
- Similar behavior to Gnutella, but better.

45

Stability and Superpeers

- Why superpeers?
  - Query consolidation
    - Many connected nodes may have only a few files
    - Propagating a query to a sub-node would take more b/w than answering it yourself
  - Caching effect
    - Requires network stability
- Superpeer selection is time-based
  - How long you've been on is a good predictor of how long you'll be around.

46

P2P: Swarming

BitTorrent

BitTorrent: History

- In 2002, B. Cohen debuted BitTorrent
- Key Motivation:
  - Popularity exhibits temporal locality (Flash Crowds)
  - E.g., Slashdot effect, CNN on 9/11, new movie/game release
- Focused on Efficient Fetching, not Searching:
  - Distribute the same file to all peers
  - Single publisher, multiple downloaders
- Has some "real" publishers:
  - Software, game distribution

48

BitTorrent

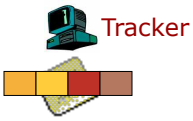
- Bittorrent now accounts for 50% of internet traffic
- ISPs have interrupted torrents
- Now limiting monthly bandwidth usage
- DMCA violation notices
  - Direct vs indirect observation

BitTorrent: Overview

- Swarming:
  - Join: contact centralized "tracker" server, get a list of peers.
  - Publish: Run a tracker server.
  - Search: Out-of-band. E.g., use Google
  - 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

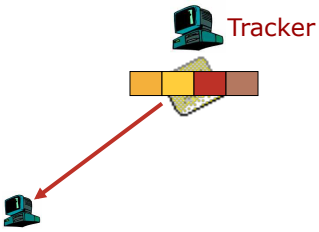
50

BitTorrent: Publish/Join



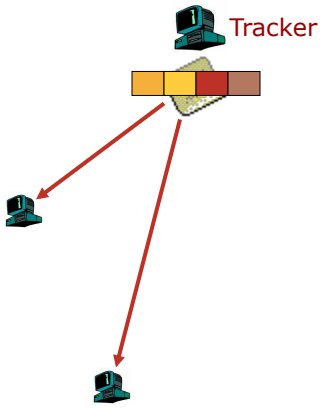
51

BitTorrent: Publish/Join



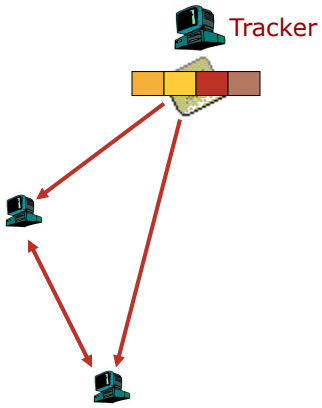
52

BitTorrent: Publish/Join



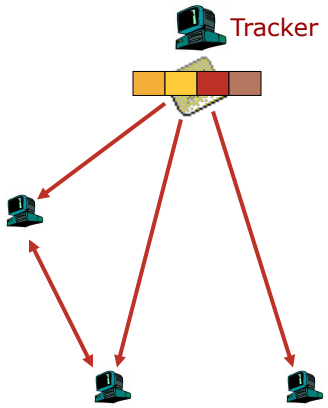
53

BitTorrent: Publish/Join



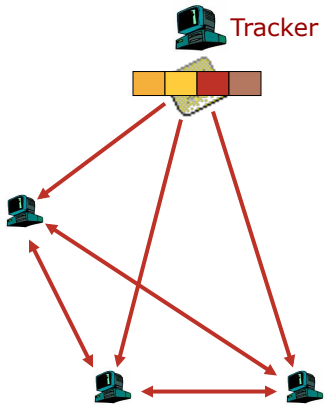
54

BitTorrent: Publish/Join



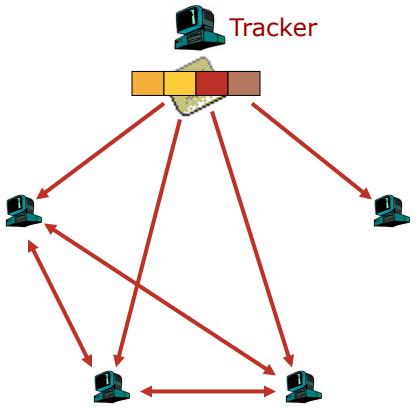
55

BitTorrent: Publish/Join



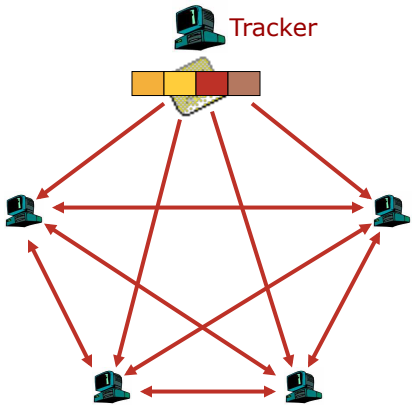
56

BitTorrent: Publish/Join



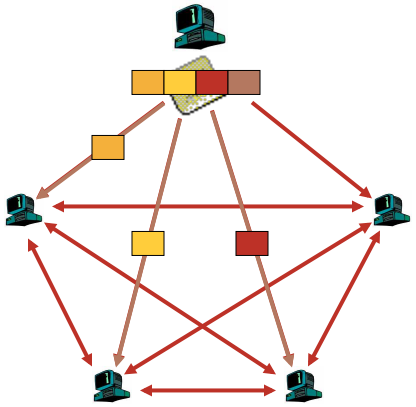
57

BitTorrent: Publish/Join



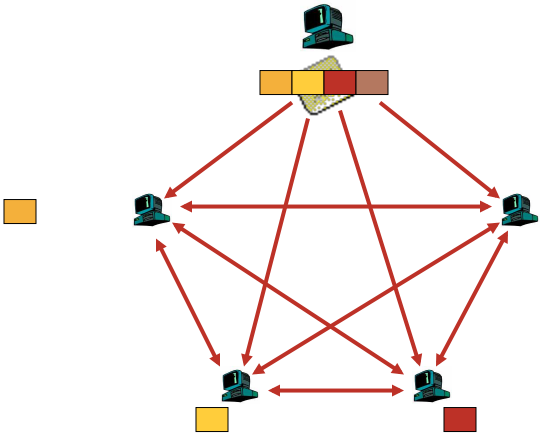
58

BitTorrent: Fetch



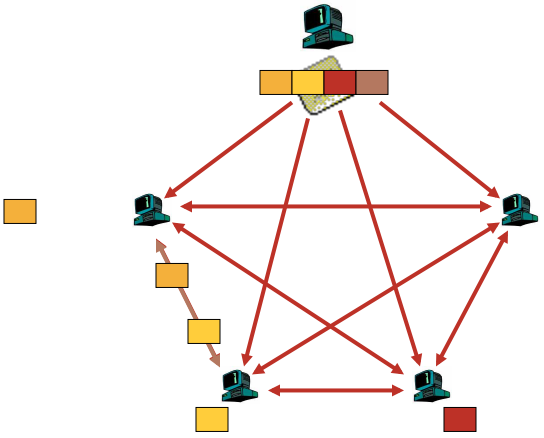
59

BitTorrent: Fetch



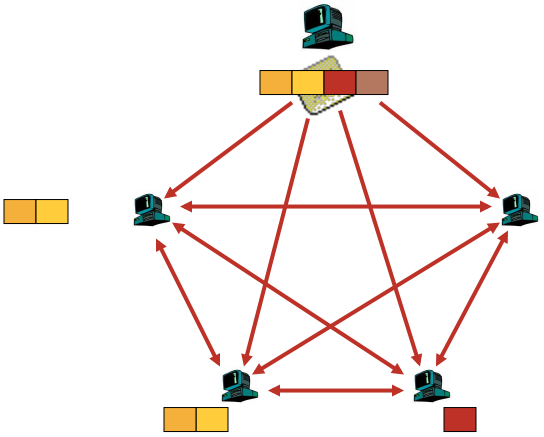
60

BitTorrent: Fetch



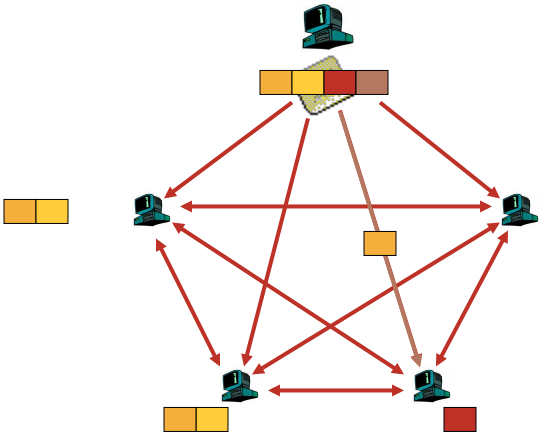
61

BitTorrent: Fetch



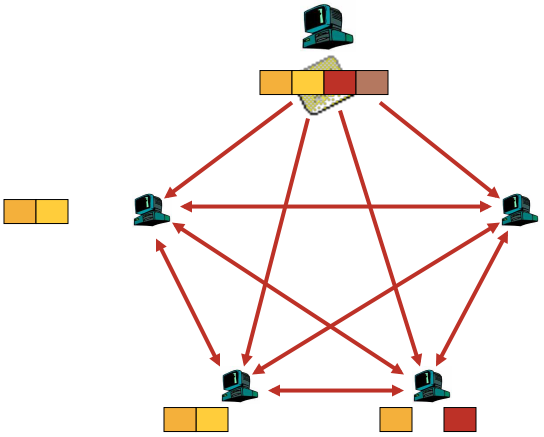
62

BitTorrent: Fetch



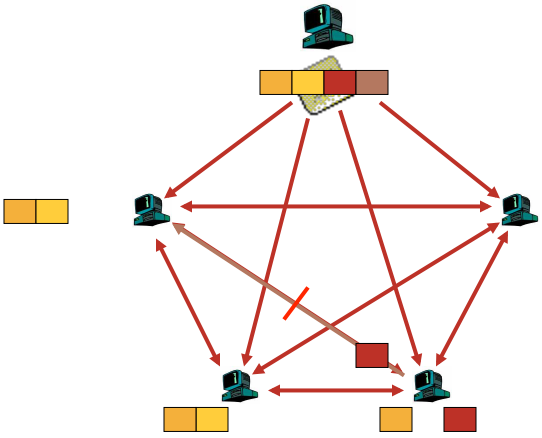
63

BitTorrent: Fetch



64

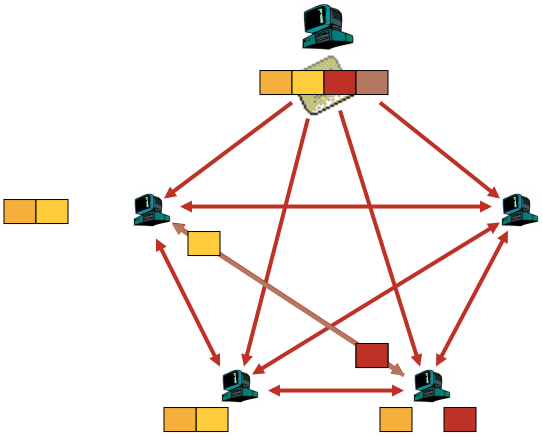
BitTorrent: Fetch



Choking

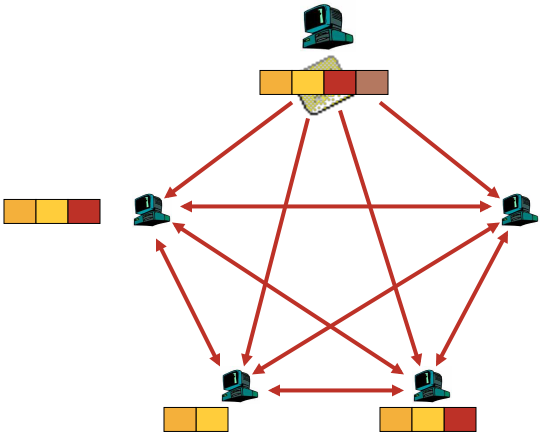
65

BitTorrent: Fetch



66

BitTorrent: Fetch



67

BitTorrent: Fetching 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
    - Bootstrapping
    - Discover better peers to download from
- Goal: Pareto Efficiency
  - Game Theory: "No change can make anyone better off without making others worse off"
  - Does it work?

68

BitTorrent: Summary

- Pros:
  - Works reasonably well in practice
  - Avoids freeloaders
- Cons:
  - Pareto Efficiency relatively weak
  - Central tracker server needed to bootstrap swarm

69

Distributed Hash Tables

STRUCTURED OVERLAY ROUTING

Distributed Hash Tables: History

- Academic answer to p2p
- Goals
  - Guaranteed lookup success
  - Provable bounds on search time
  - Provable scalability
- Makes some things harder
  - Fuzzy queries / full-text search / etc.
- Read-write, not read-only

71

DHT: Overview (1)

- Abstraction: a distributed "hash-table" (DHT) data structure:
  - put(id, item);
  - item = get(id);
- Implementation: nodes in system form a distributed data structure
  - Can be Ring, Tree, Hypercube, Skip List, Butterfly Network, ...

72

DHT: Overview (2)

- Structured Overlay Routing:
  - Join: On startup, contact a "bootstrap" node; get a node id
  - Publish: Route publication for file id toward a close node id
  - Search: Route a query for file id toward a close node id
  - Fetch: Two options:
    - 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

73

Chord: Consistent Hashing

- Associate to each node and file a unique id in an uni-dimensional space (a Ring)
  - E.g., pick from the range $[0...2^m]$
  - Usually the hash of the file or IP address
- Properties:
  - Routing table size is O(log N)
  - Guarantees that a file is found in O(log N) hops

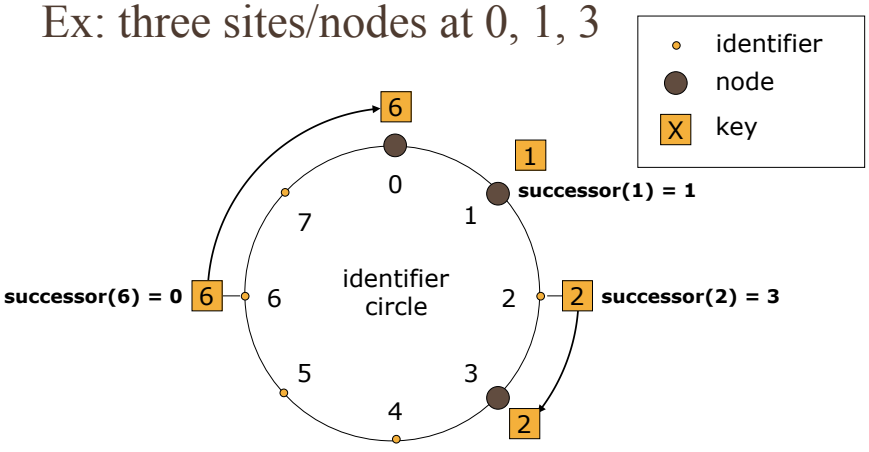
from MIT in 2001

74

Consistent Hashing

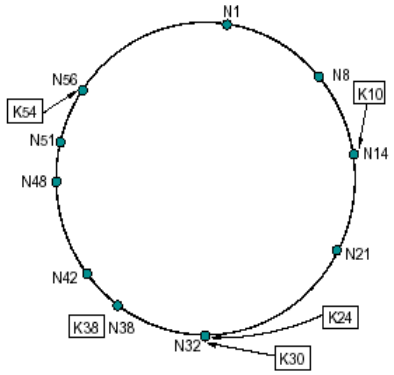
- Key k is assigned to the first node whose identifier is equal to or follows (the identifier of) k in the identifier space
- This node is the successor node of key k, denoted by *successor(k)*

Ex: three sites/nodes at 0, 1, 3



Consistent Hashing

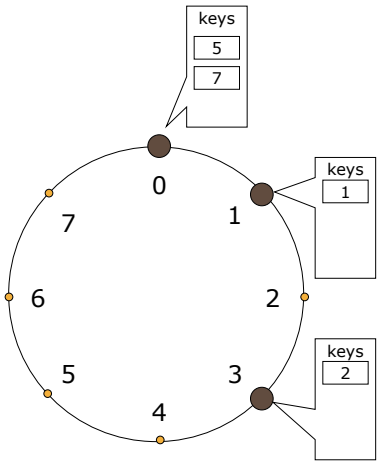
- For m = 6, # of identifiers is 64.
- Following ring has 10 nodes and stores 5 keys.
- The successor of key 10 is node 14.



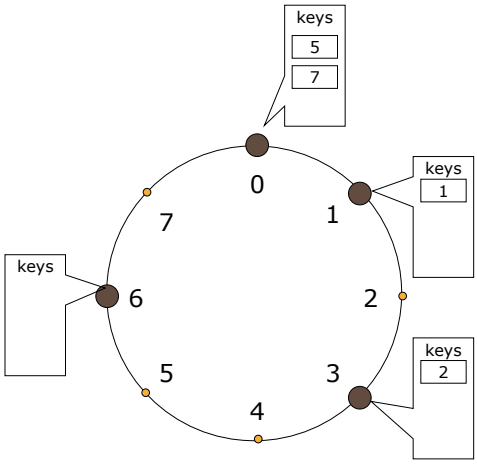
Join and Departure

- When a node n joins the network, certain keys previously assigned to n's successor now become assigned to n.
- When node n leaves the network, all of its assigned keys are reassigned to n's successor.

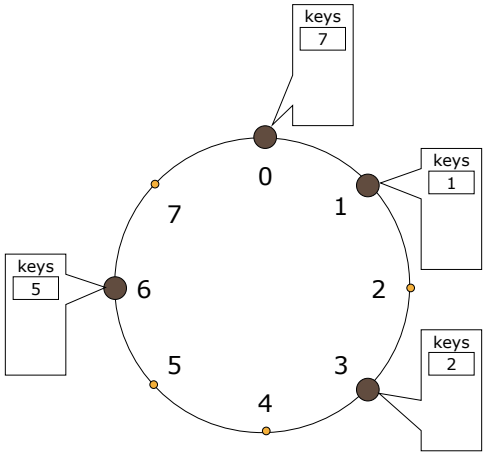
Initial Ring



Node Join

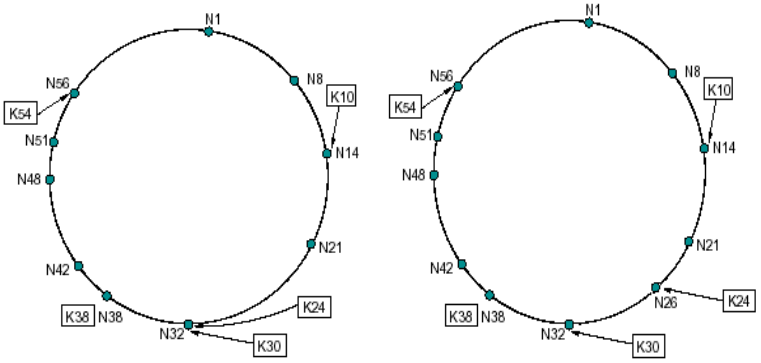


Redistribute Values

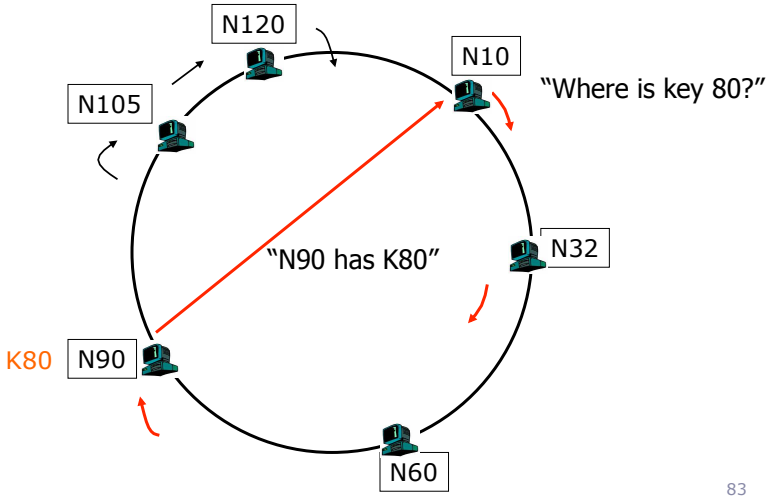


Consistent Hashing

- When node 26 joins the network:



DHT: Basic Chord Lookup



83

A Simple Key Lookup

- Pseudo code for finding successor:

```
// ask node n to find the successor of id
n.find_successor(id)
  if (id ∈ (n, n.successor])
     return n.successor;
  else
     // forward the query around the circle
     return n.successor.find_successor(id);
```

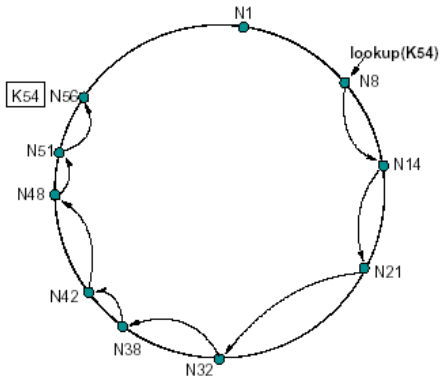
A Simple Key Lookup

- Pseudo code for finding successor:

```
// ask node n to find the successor of id
n.find_predecessor(id):
  n' = n
  while (id ∉ (n', n'.successor])
      n' = n'.successor
  return n'

n.find_successor(id):
  n' =  find_predecessor(id)
  return n'.successor
```

A Simple Key Lookup

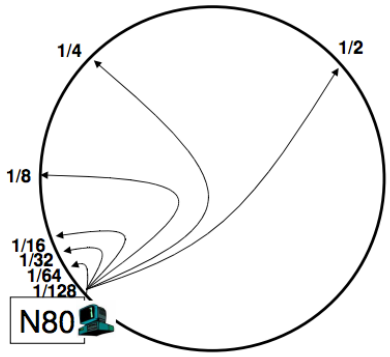
- The path taken by a query from node 8 for key 54:



Finger Tables

- Each node n' maintains a routing table with up to m entries
- The i^{th} entry in the table at node n contains the identity of the *first* node s that succeeds n by at least 2^{i-1} on the identifier circle.
- s = successor(n+2^{i-1}).
- s is called the *i^{th} finger* of node n, denoted by *n.finger(i)*

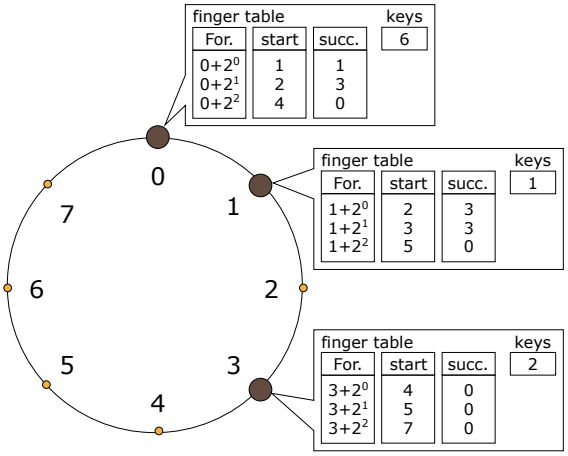
Finger Tables



- Entry *i* in the finger table of node *n* is the first node that succeeds or equals $n + 2^i$ (mod N)
- In other words, the i^{th} finger points $1/2^{n-i}$ way around the ring

88

Finger Tables

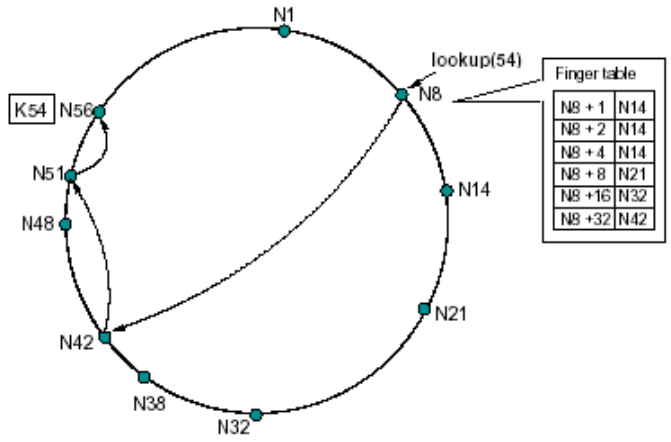


Finger Tables

- A finger table entry includes both the Chord identifier and the IP address (and port number) of the relevant node.
- The first finger of n is the immediate successor of n on the circle.

Example query

- The path of a query for key 54 starting at node 8:



Node Joins and Stabilizations

- "Stabilization" protocol contains 6 functions:
  - `create()`
  - `join()`
  - `stabilize()`
  - `notify()`
  - `fix_fingers()`
  - `check_predecessor()`

Node Joins – join()

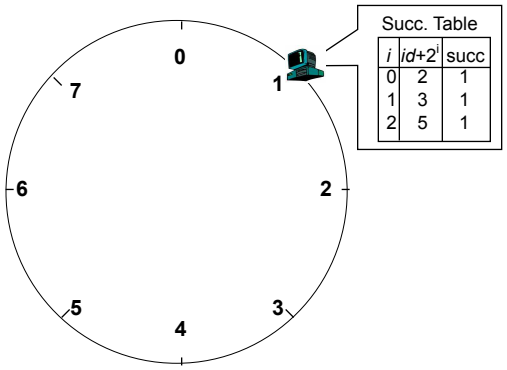
- When node n first starts, it calls n.*join*(n'), where n' is any known Chord node.
- The *join*() function asks n' to find the immediate successor of n.
- *join*() does not make the rest of the network aware of n.

Node Joins – join()

```
// create a new Chord ring.
n.create()
  predecessor = nil;
  successor = n;

// join a Chord ring containing node n'.
n.join(n')
  predecessor = nil;
  successor = n'.find_successor(n);
```

DHT: Chord Join

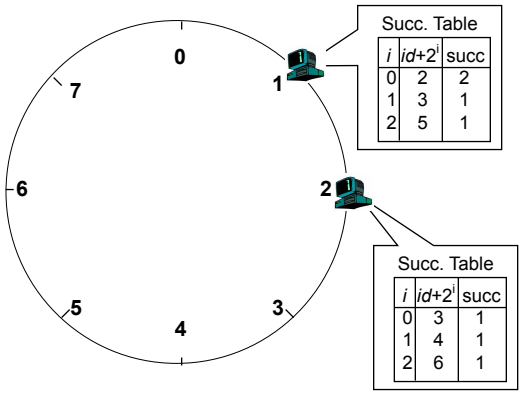
- Assume an identifier space [0..8]
- Node n1 joins



95

DHT: Chord Join

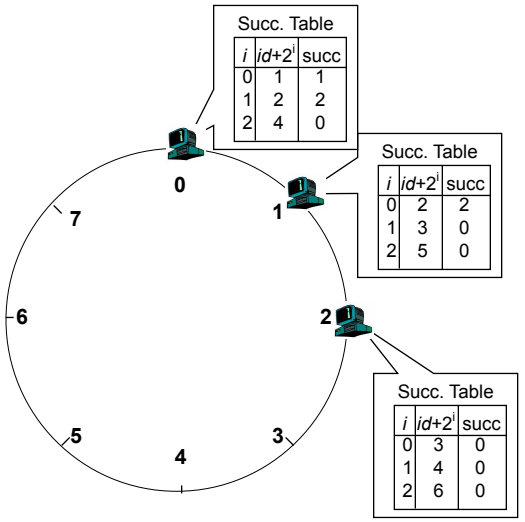
- Node n2 joins



96

DHT: Chord Join

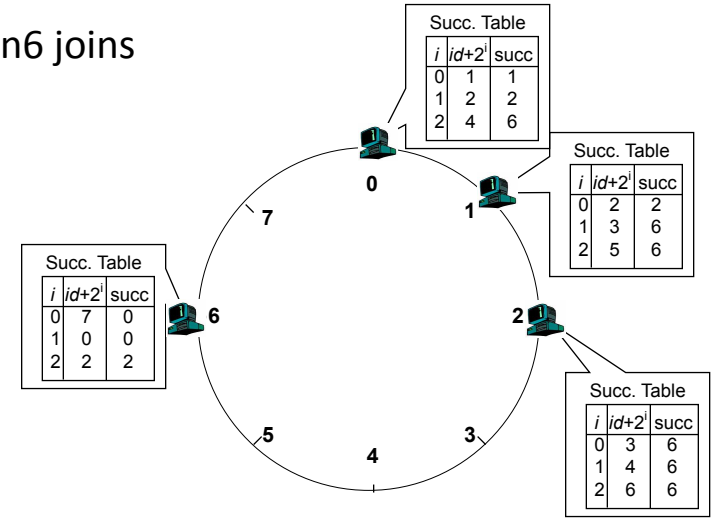
- Node n0 joins



97

DHT: Chord Join

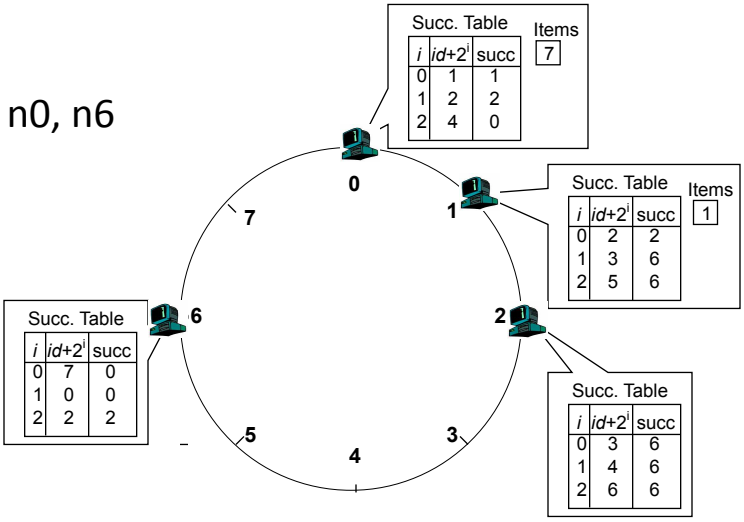
- Node n6 joins



98

DHT: Chord Join

- Nodes:
  n1, n2, n0, n6
- Items:
  f7, f1



99

Scalable Key Location – find_successor()

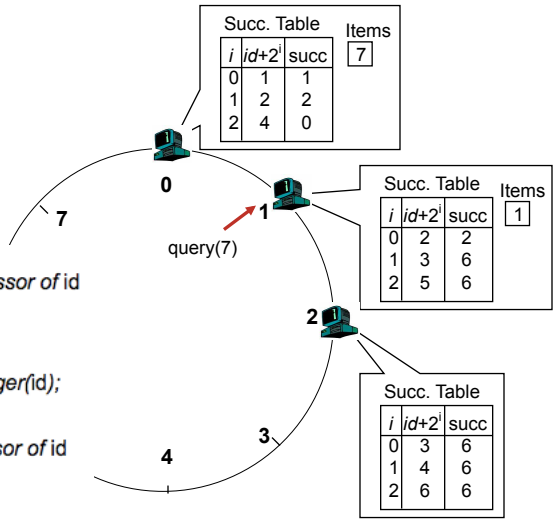
```
// ask node n to find the successor of id
n.find_successor(id)
   if (id ∈ (n, successor])
        return successor;
   else
        n' = closest_preceding_node(id);
        return n'.find_successor(id);

// search the local table for the highest predecessor of id
n.closest_preceding_node(id)
   for i = m downto 1
        if (finger[i] ∈ (n, id))
              return finger[i];
   return n;
```

Scalable Key Location – find_successor()

```
// ask node n to find the successor of id
n.find_predecessor(id):
   n' = n
   while (id ∉ (n', n'.successor])
       n' = n'.closest_preceding_finger(id)
   return n'

n.find_successor(id):
   n' =  find_predecessor(id)
   return n'.successor

// search the local table for the highest predecessor of id
n.closest_preceding_finger(id)
   for i = m downto 1
       if (finger[i] ∈ (n, id))
            return finger[i];
   return n;
```

DHT: Chord Routing

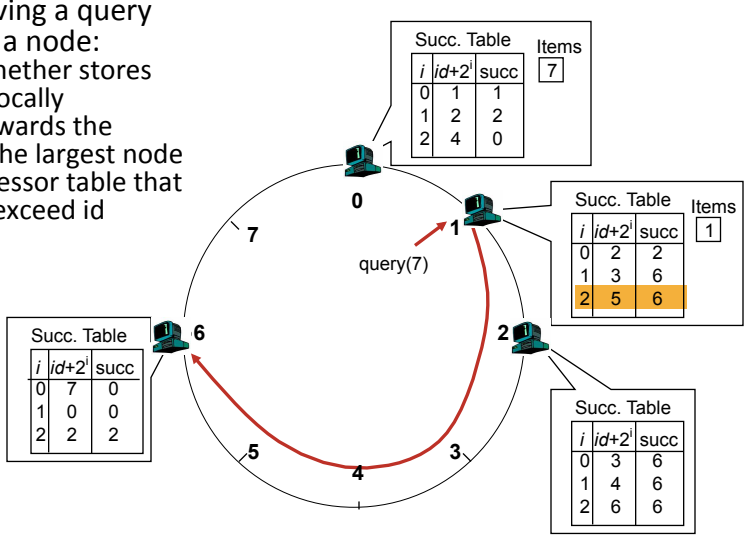
- Upon receiving a query for item id, a node:
  - Checks whether stores the item locally
  - If not, forwards the query to the largest node in its successor table that does not exceed id




```
// ask node n to find the predecessor of id
n.find_predecessor(id)
  n' = n;
  while (id ∉ (n', n'.successor])
    n' = n'.closest_preceeding_finger(id);
  return n'

// ask node n to find the successor of id
n.find_successor(id)
  n' = find_predecessor(id);
  return n'.successor;
```

DHT: Chord Routing

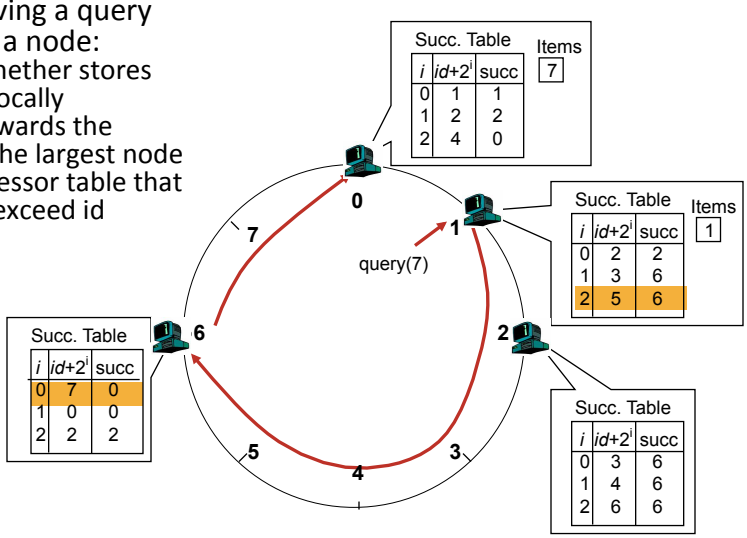
- Upon receiving a query for item id, a node:
  - Checks whether stores the item locally
  - If not, forwards the query to the largest node in its successor table that does not exceed id



103

DHT: Chord Routing

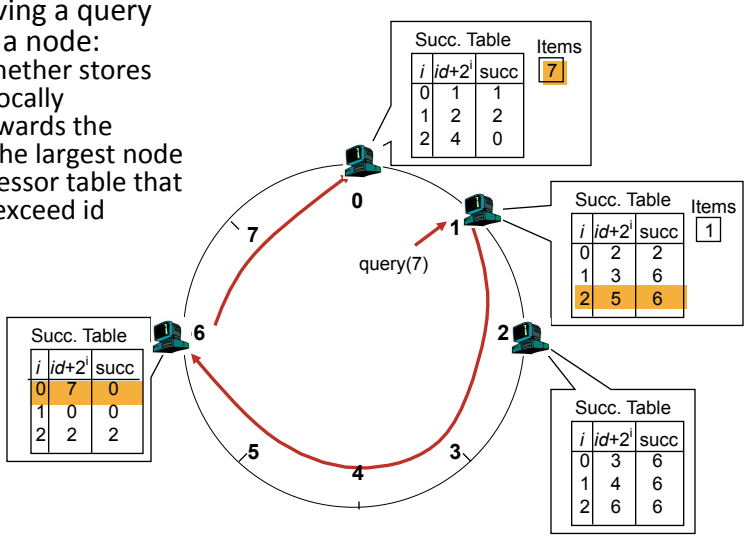
- Upon receiving a query for item id, a node:
  - Checks whether stores the item locally
  - If not, forwards the query to the largest node in its successor table that does not exceed id



104

DHT: Chord Routing

- Upon receiving a query for item id, a node:
  - Checks whether stores the item locally
  - If not, forwards the query to the largest node in its successor table that does not exceed id

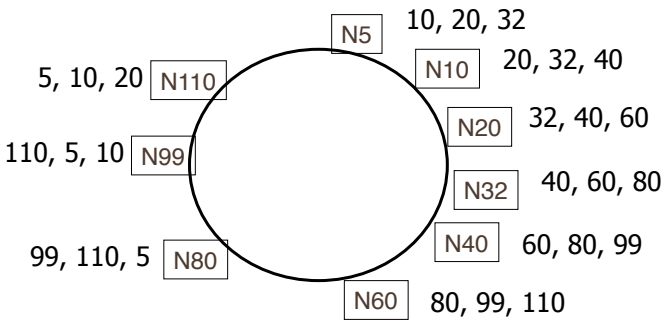


105

Chord reliability

- Correct routing table (successors, predecessors, and fingers)
- Primary invariant: correctness of successor pointers
  - Fingers for performance
  - Algorithm is to "get closer" to the target
  - Successor nodes always do this

Successor Lists Ensure Robust Lookup



- Each node remembers r successors
- Lookup can skip over dead nodes to find blocks
- Periodic check of successor and predecessor links

Maintaining successor pointers

- Periodically run "stabilize" algorithm
  - Node n finds successor's predecessor p
  - Notifies successor of n's existence
  - Successor may make n its predecessor

```
n.stabilize()
  x = sucessor.predecessor;
  if (x ∈ (n, successor) )
     successor = x;
  successor.notify(n);
```

Node Joins – stabilize()

```
// called periodically. verifies n's immediate
// successor, and tells the successor about n.
n.stabilize()
  x = successor.predecessor;
  if (x ∈ (n, successor))
      successor = x;
  successor.notify(n);

// n' thinks it might be our predecessor.
n.notify(n')
if (predecessor is nil or n' ∈ (predecessor, n))
  predecessor = n';
```

Join: Expensive Approach

- New node has to:
  - Fill its own successor, predecessor, fingers
  - Notify other nodes for which it can be successor, predecessor or finger
- Can be done in O(log N) time
- But complex
  - Impractical with high churn rate

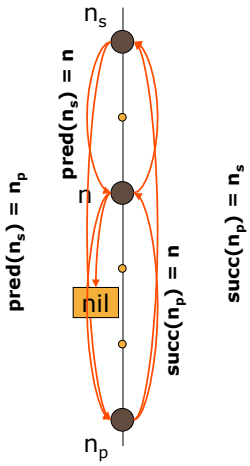
Join: Relaxed Approach

- If ring is correct, then routing is correct
  - Fingers needed for speed only
- Stabilization
  - Each node periodically runs stabilization routine
  - Each node refreshes all fingers by periodically calling find_successor($n+2^{i-1}$) for random I
  - Periodic cost is O(log N) per node due to finger refresh

fix_fingers()

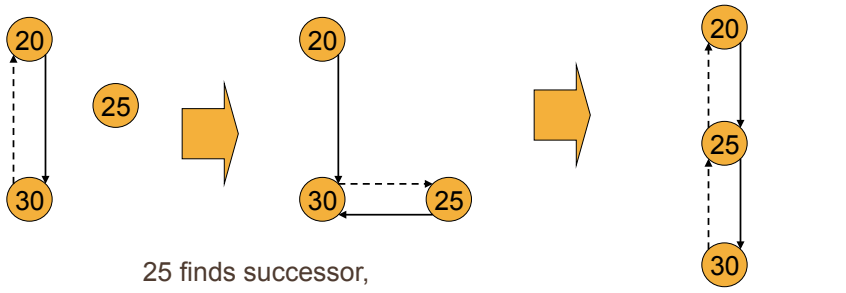
```
// called periodically. refreshes finger table entries.
n.fix_fingers()
  next = next + 1 ;
  if (next > m)
      next = 1 ;
  finger[next] = find_successor(n + 2^(next-1));

// checks whether predecessor has failed.
n.check_predecessor()
  if (predecessor has failed)
      predecessor = nil;
```

Node Joins – Join and Stabilization



- **n joins**
  - predecessor = nil
  - n acquires n_s as successor via some n'
- **n runs stabilize**
  - n notifies n_s being the new predecessor
  - n_s acquires n as its predecessor
- **n_p runs stabilize**
  - n_p asks n_s for its predecessor (now n)
  - n_p acquires n as its successor
  - n_p notifies n
  - n will acquire n_p as its predecessor
- **all predecessor and successor pointers are now correct**
- **fingers still need to be fixed, but old fingers will still work**

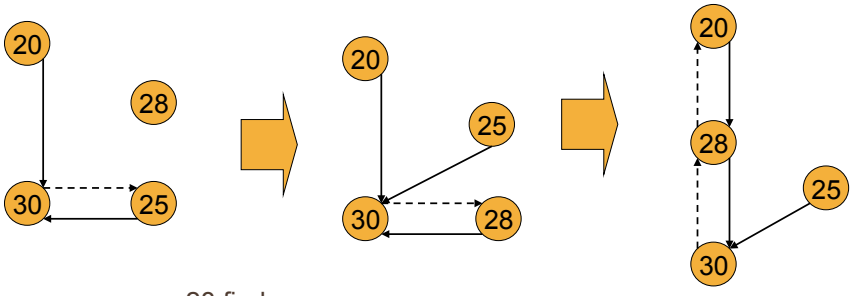
Initial: 25 wants to join correct ring (between 20 and 30)



25 finds successor, and tells successor (30) of itself

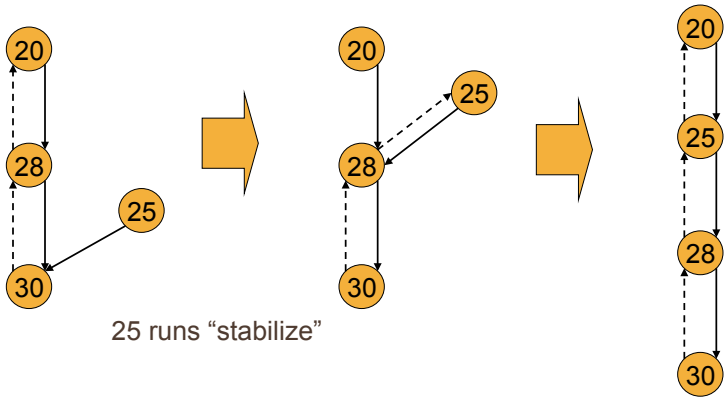
20 runs "stabilize":
20 asks 30 for 30's predecessor
30 returns 25
20 tells 25 of itself

This time, 28 joins before 20 runs "stabilize"



28 finds successor, and tells successor (30) of itself

20 runs "stabilize":
20 asks 30 for 30's predecessor
30 returns 28
20 tells 28 of itself



25 runs "stabilize"

20 runs "stabilize"

DHT: Chord Summary

- Routing table size?
  - Log N fingers
- Routing time?
  - Each hop expects to 1/2 the distance to the desired id => expect O(log N) hops.

- Pros:
  - Guaranteed Lookup
  - O(logN) per node state and search scope
- Cons:
  - No one uses them? (only one file sharing app)
  - Supporting non-exact match search is hard

117

DHT APPLICATIONS

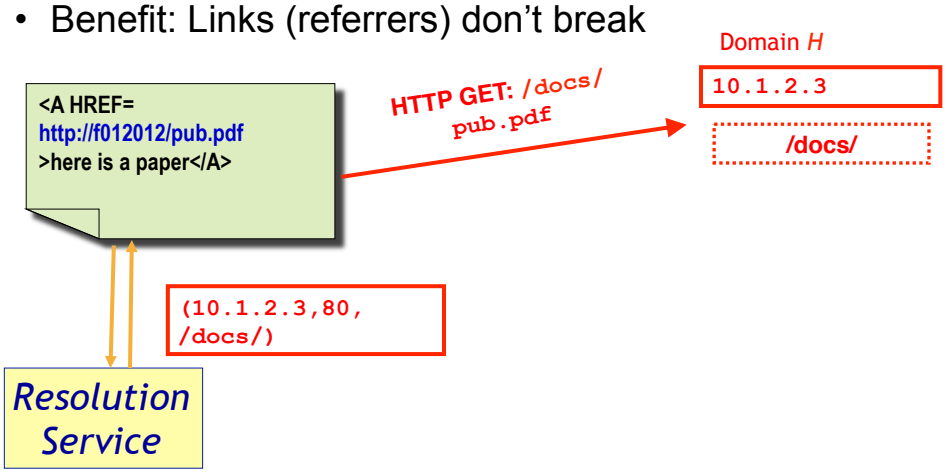
P2P-enabled applications: Flat naming

- Most naming schemes use hierarchical names to enable scaling
- DHT provide a simple way to scale flat names
  - E.g. just insert name resolution into Hash(name)

119

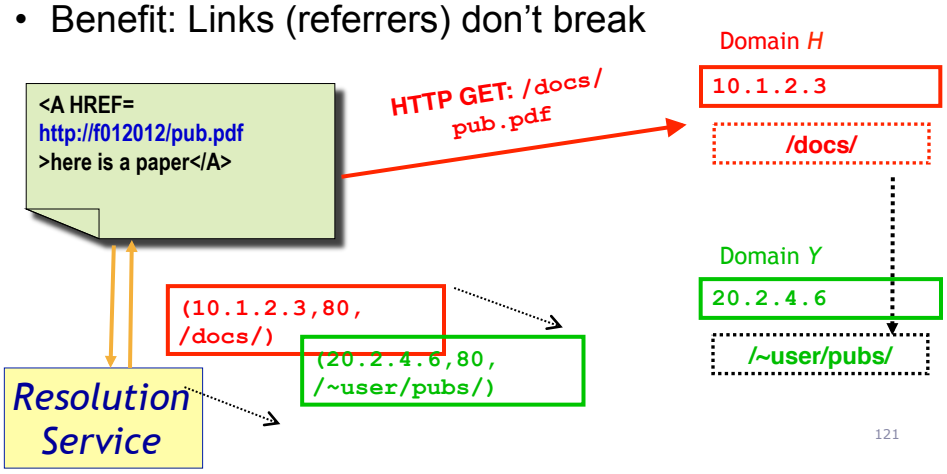
Flat Names Example

- SID abstracts all object reachability information
- Objects: any granularity (files, directories)
- Benefit: Links (referrers) don't break



Flat Names Example

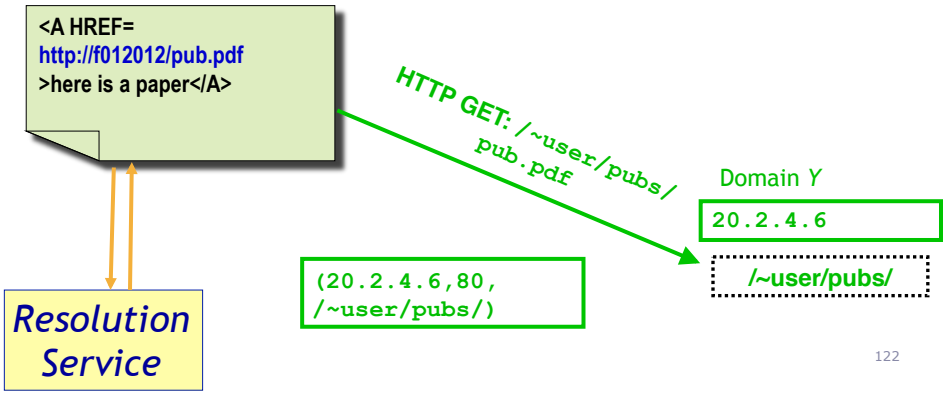
- SID abstracts all object reachability information
- Objects: any granularity (files, directories)
- Benefit: Links (referrers) don't break



121

Flat Names Example

- SID abstracts all object reachability information
- Objects: any granularity (files, directories)
- Benefit: Links (referrers) don't break



122

I3: Motivation

- Today's Internet based on point-to-point abstraction
- Applications need more:
  - Multicast
  - Mobility
  - Anycast
- Existing solutions:
  - Change IP layer
  - Overlays

123

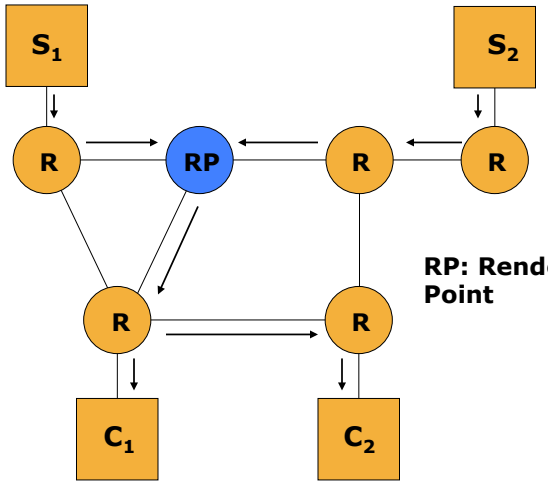
I3: Motivation

- Today's Internet based on point-to-point abstraction
- Applications need more:
  - Multicast
  - Mobility
  - Anycast
- Existing solutions:
  - Change IP layer
  - Overlays

So, what's the problem?
A different solution for each service

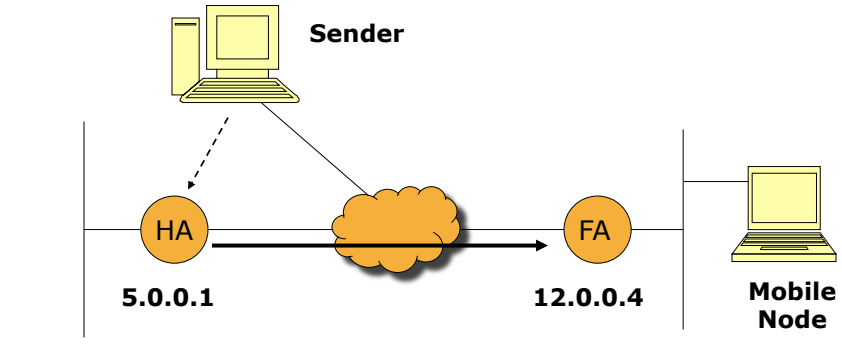
124

Multicast



125

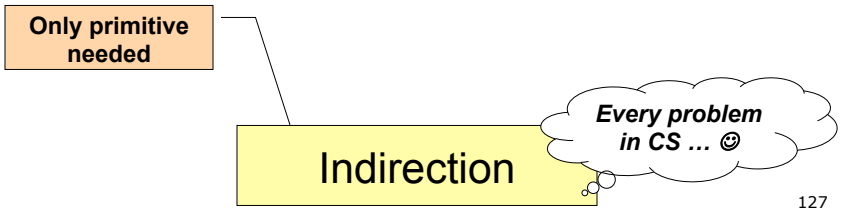
Mobility



126

The i3 Solution

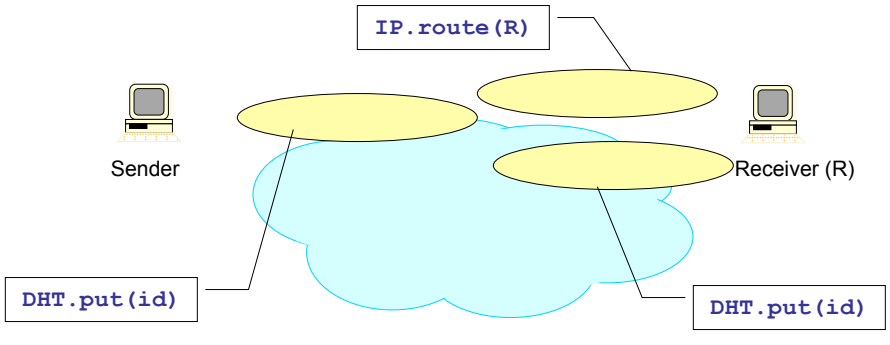
- Solution:
  - Add an indirection layer on top of IP
  - Implement using overlay networks
- Solution Components:
  - Naming using "identifiers"
  - Subscriptions using "triggers"
  - DHT as the gluing substrate



127

i3: Implementation

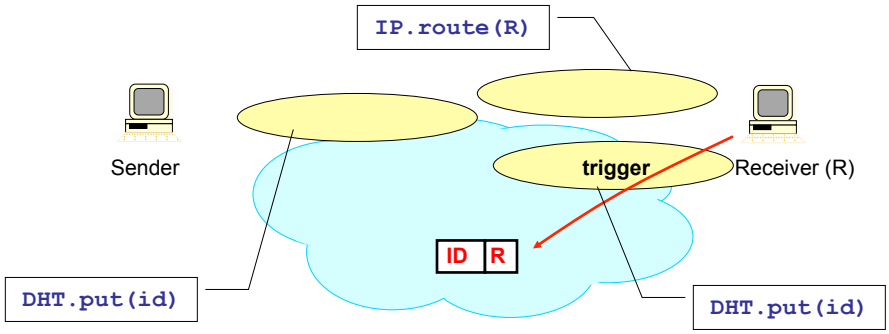
- Use a Distributed Hash Table
  - Scalable, self-organizing, robust
  - Suitable as a substrate for the Internet



128

i3: Implementation

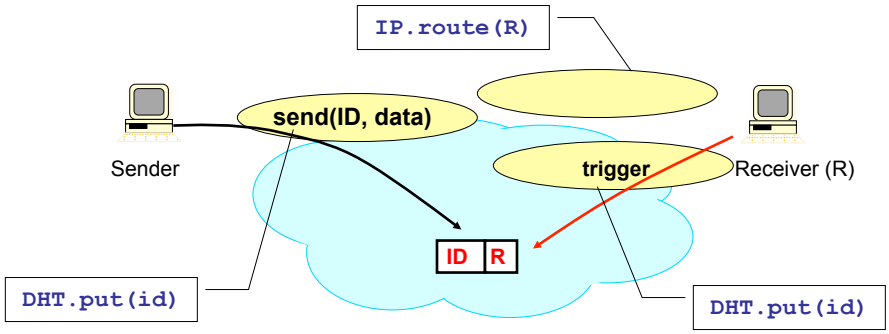
- Use a Distributed Hash Table
  - Scalable, self-organizing, robust
  - Suitable as a substrate for the Internet



129

i3: Implementation

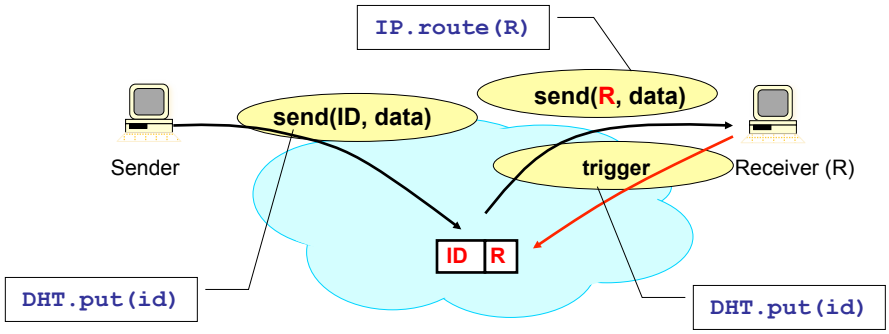
- Use a Distributed Hash Table
  - Scalable, self-organizing, robust
  - Suitable as a substrate for the Internet



130

i3: Implementation

- Use a Distributed Hash Table
  - Scalable, self-organizing, robust
  - Suitable as a substrate for the Internet



131

P2P-Enabled Applications: Self-Certifying Names

- Name = Hash(pubkey, salt)
- Value = <pubkey, salt, data, signature>
  - can verify name related to pubkey and pubkey signed data
- Can receive data from caches or other 3rd parties without worry
  - much more opportunistic data transfer

132

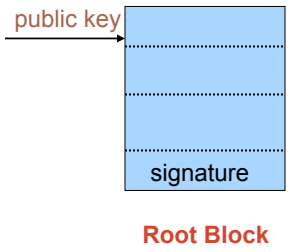
P2P-Enabled Applications: Distributed File Systems

- CFS [Chord]
  - *Block* based read-only storage
- PAST [Pastry]
  - *File* based read-only storage
- Ivy [Chord]
  - *Block* based read-write storage

Chord File System (CFS)

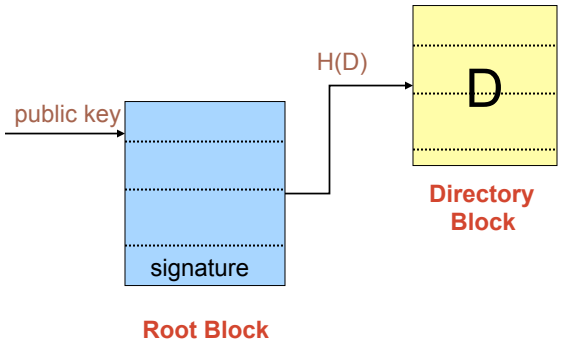
- Blocks are inserted into Chord DHT
  - insert(blockID, block)
  - Replicated at successor list nodes
- Read root block through public key of file system
- Lookup other blocks from the DHT
  - Interpret them to be the file system
- Cache on lookup path

Chord File System (CFS)



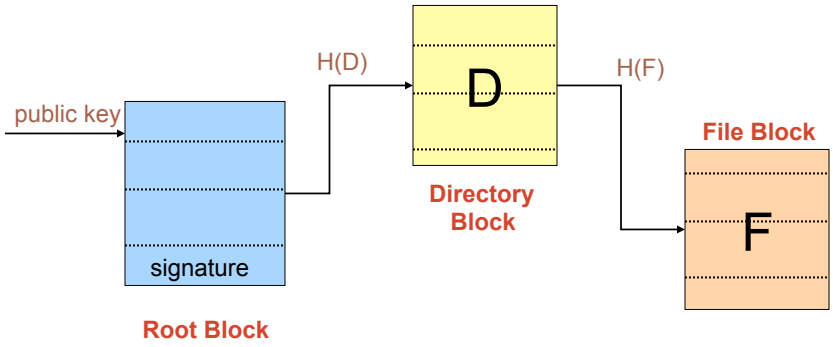
135

Chord File System (CFS)



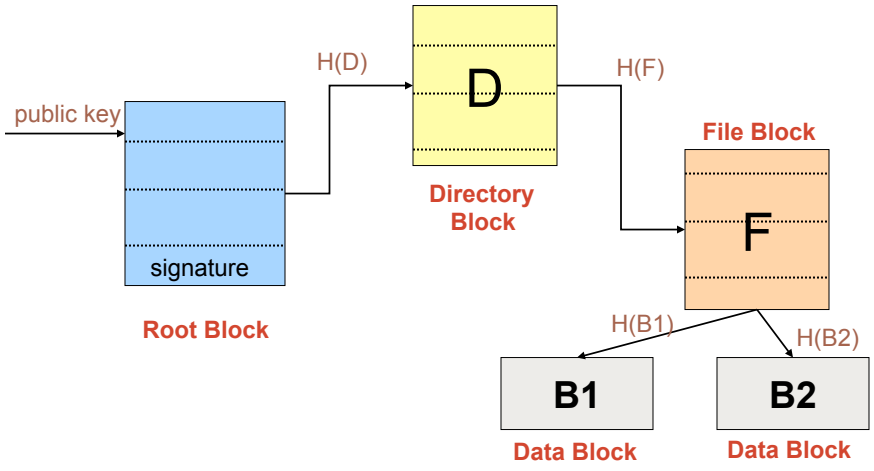
136

Chord File System (CFS)



137

Chord File System (CFS)



138

When are P2P/DHT Useful?

- Caching and "soft-state" data
  - Works well! BitTorrent, KaZaA, etc., all use peers as caches for hot data
- Finding read-only data
  - Limited flooding finds hay
  - DHTs find needles
- BUT

Peer-to-Peer Google?

- Complex intersection queries ("the" + "who")
  - Billions of hits for each term alone
- Sophisticated ranking
  - Must compare many results before returning a subset to user
- Very, very hard for a DHT / p2p system
  - Need high inter-node bandwidth
  - (This is exactly what Google does - massive clusters)

Writable, Persistent P2P

- Do you trust your data to 100,000 monkeys?
- Node availability (aka "churn") hurts
  - Ex: Store 5 copies of data on different nodes
  - When someone goes away, you must replicate the data they held
  - Hard drives are *huge*, but cable modem upload bandwidth is tiny - perhaps 10 Gbytes/day
  - Takes many days to upload contents of 200GB hard drive.
    - Very expensive leave/replication situation!

141

P2P: Summary

- Many different styles; remember pros and cons of each
  - centralized, flooding, swarming, unstructured and structured routing
- Lessons learned:
  - Single points of failure are very bad
  - Flooding messages to everyone is bad
  - Underlying network topology is important
  - Not all nodes are equal
  - Need incentives to discourage freeloading
  - Privacy and security are important
  - Structure can provide theoretical bounds and guarantees

142

MULTI-PLAYER GAMES

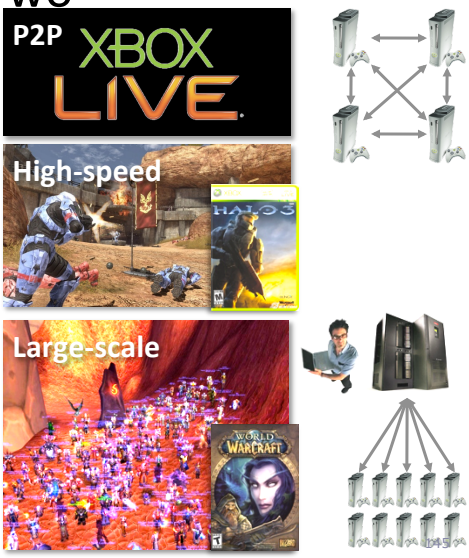
Individual Player's View

- Interactive environment (e.g. door, rooms)
- Live ammo
- Monsters
- Players
- Game state

144

High Speed, Large Scale, P2P – Pick Two

- Many console games are peer hosted to save costs
- Limits high-speed games to 32 players
- 1000+ player games need dedicated servers

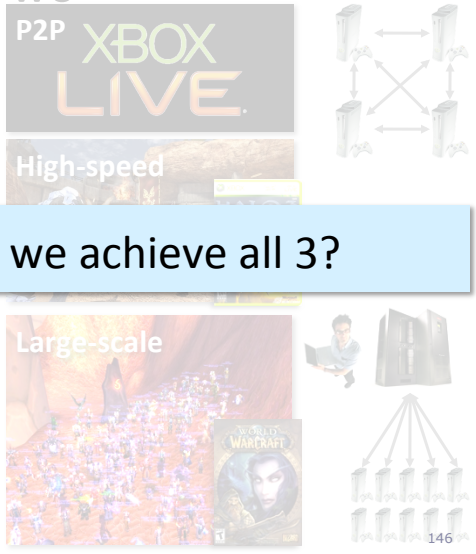


145

High Speed, Large Scale, P2P – Pick Two

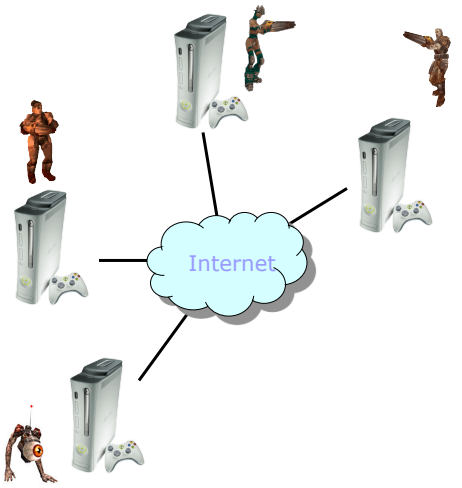
- Many console games are peer hosted to save costs
-
- 1000+ player games need dedicated servers

Question: Can we achieve all 3?

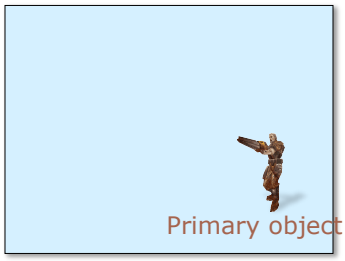


146

P2P

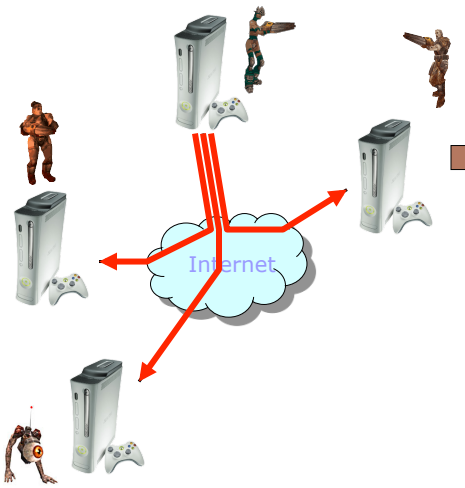


Local View

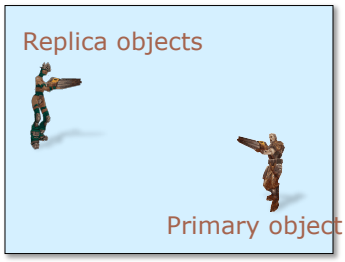


147

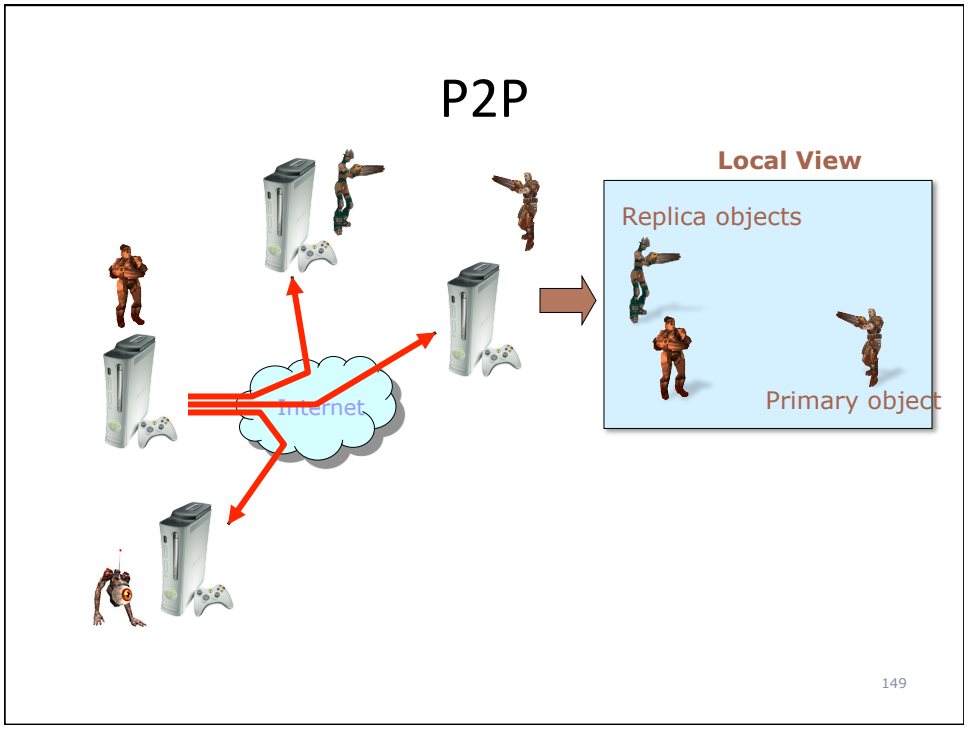
P2P

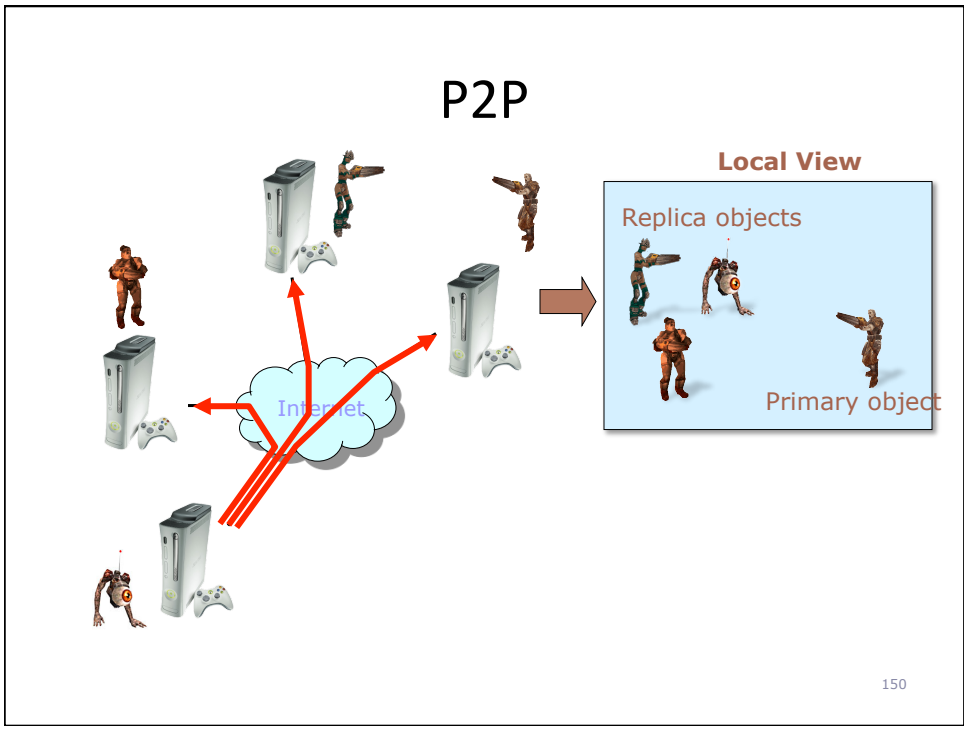


Local View

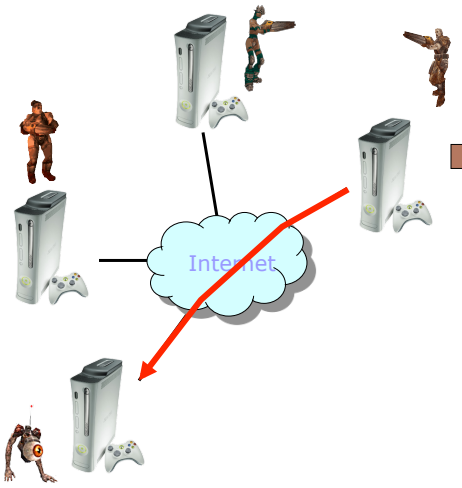


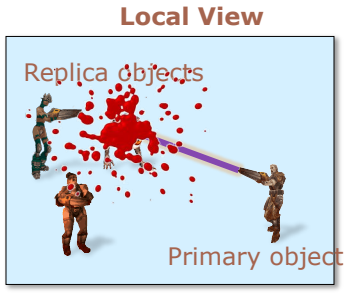
148


P2P
Local View
Replica objects
Primary object
Internet
149


P2P
Local View
Replica objects
Primary object
Internet
150

High Speed

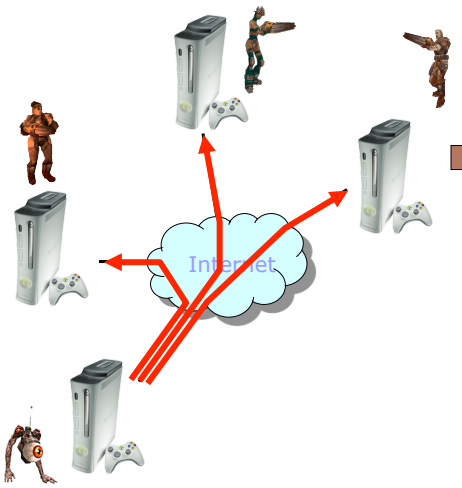


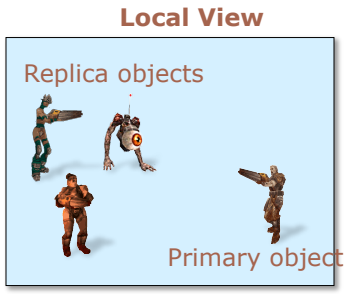


Inter-object writes must be reflected very quickly

151

High Speed



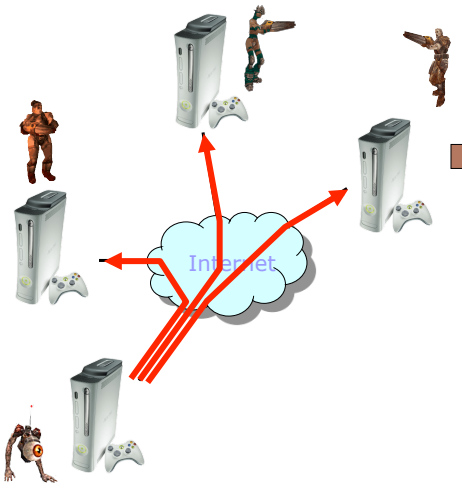


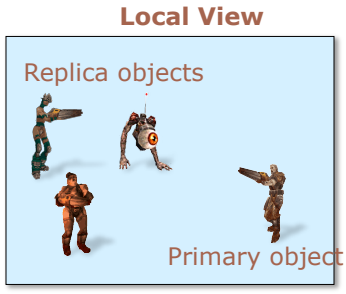
20 updates/sec
≈ 16 kbps per playe

Delay must be < 150
[Beigbeder '04]

152

High Speed



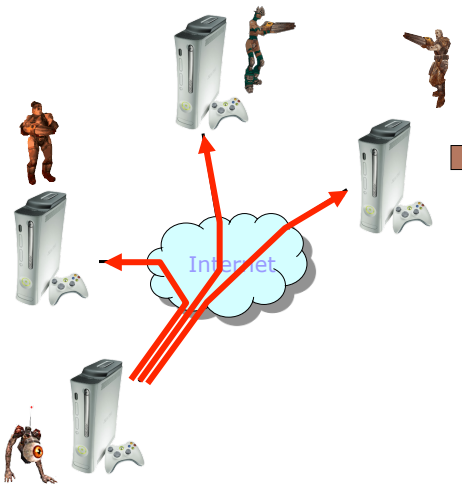


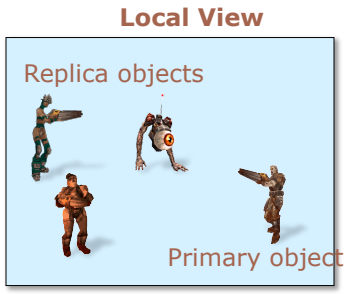
20 updates/sec
≈ 16 kbps per playe

Delay must be < 150
[Beigbeder '04]

153

High Speed



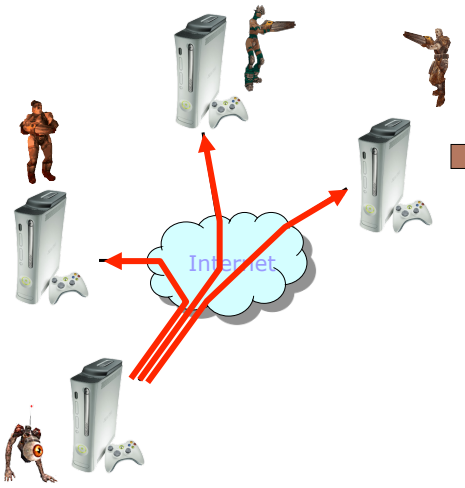


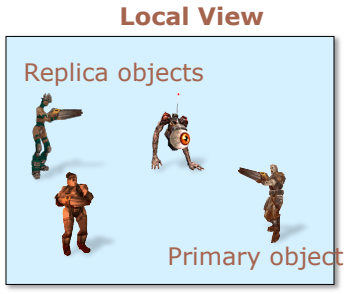
20 updates/sec
≈ 16 kbps per playe

Delay must be < 150
[Beigbeder '04]

154

High Speed



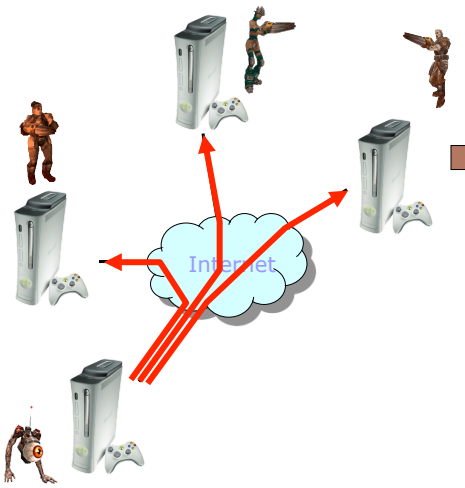


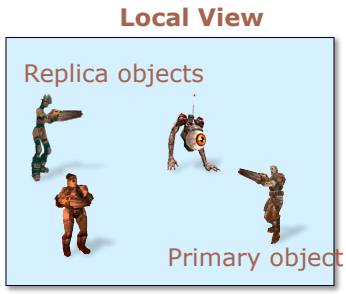
20 updates/sec
≈ 16 kbps per playe

Delay must be < 150
[Beigbeder '04]

155

High Speed



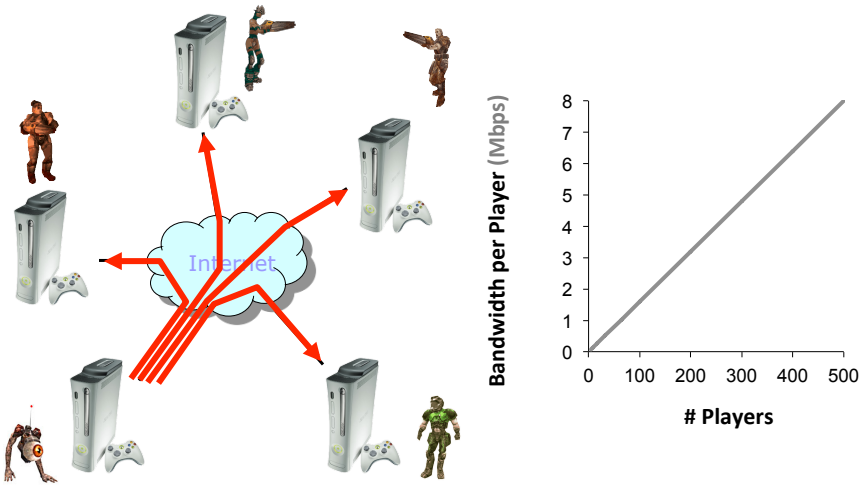


20 updates/sec
≈ 16 kbps per playe

Delay must be < 150
[Beigbeder '04]

156

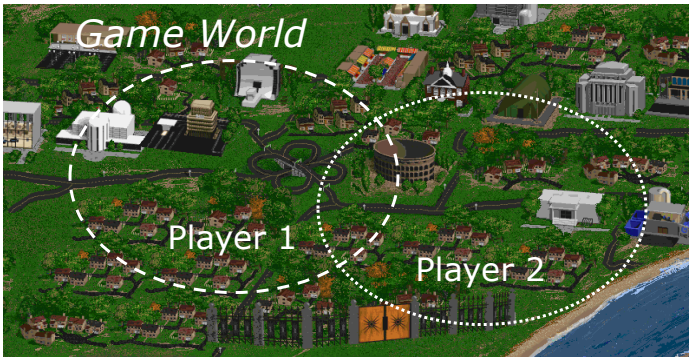
Large Scale



157

Area of Interest (AOI) Filtering

- Large shared world
  - Composed of map information, textures, etc
  - Populated by active entities: user avatars, computer AI's, etc
- Only parts of world relevant to particular user/player



158

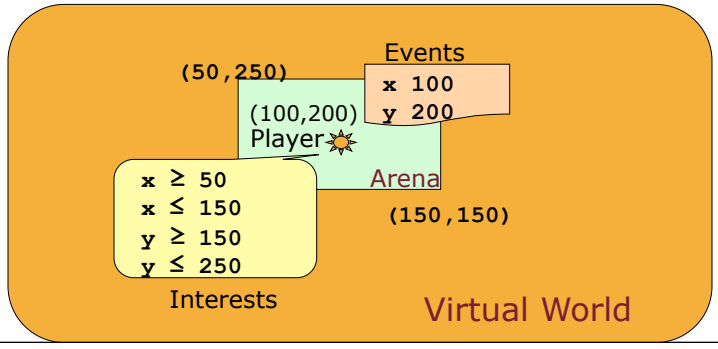
Object Model

- Game world treated as collection of objects
- Single primary copy for each object
  - Maintained at a single owner node
  - Serialization point for updates
  - Determines actions/behavior of object
- Secondary replicas
  - Primary object may need to examine other objects to determine action
  - Need low latency access to object → must replicate object in advance
- How to find set of objects to replicate? → hardest part

159

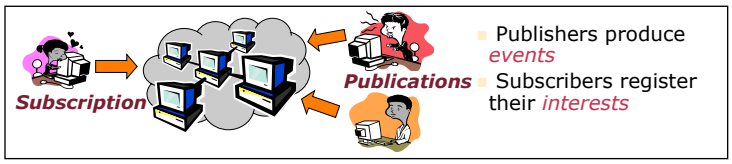
Object Discovery - Solution

- Use publish-subscribe to "register" long-lived distributed lookups
- Publications created each time object is updated (or periodically when no update is done)
  - Publication contents = state of object



160

Publish-Subscribe Overiew



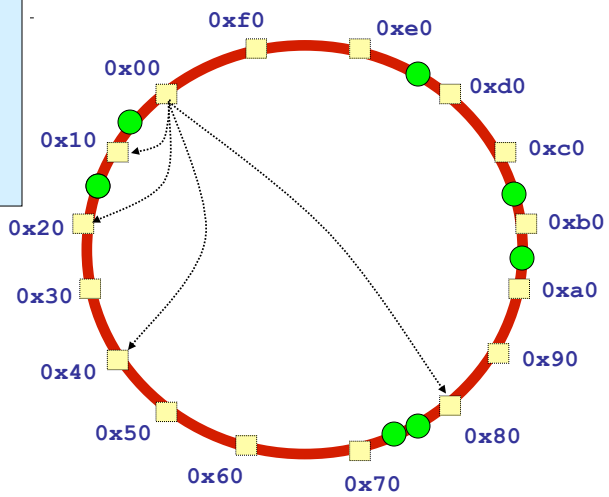
- Key feature → subscription language
  - Subject/channel-based subscriptions (e.g. all publications on the IBM stock channel)
  - Rich database-like subscription languages (e.g. all publications with name=IBM and volume > 1000)
- State-of-the-art
  - Scalable distributed designs with channel-based subscriptions
  - Unscalable distributed designs with rich subscriptions
  - Goal → scalable distributed design with "richer" subscriptions
    - Example: x ≤ 200 & y < 100→ Support for multi-dimensional range predicates

161

Using DHTs for Range Queries

Query: 6 ≤ x ≤ 13

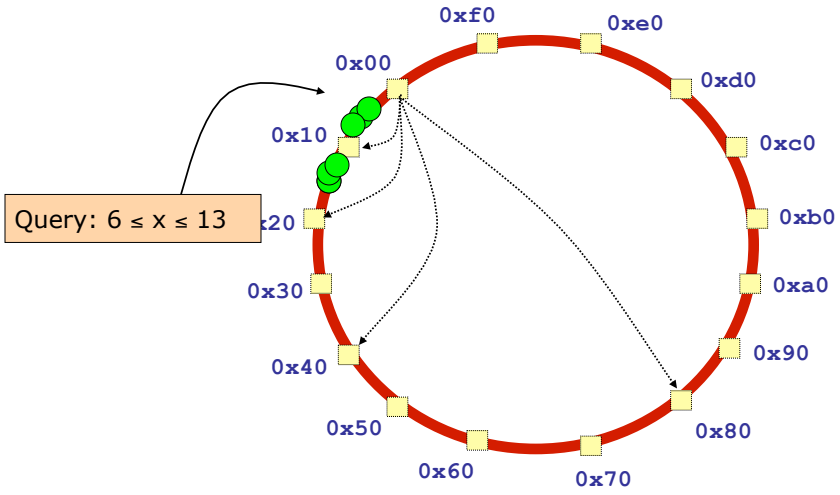
key = 6 **→0xab**
key = 7 **→0xd3**
…
key = 13 **→0x12**



162

Using DHTs for Range Queries

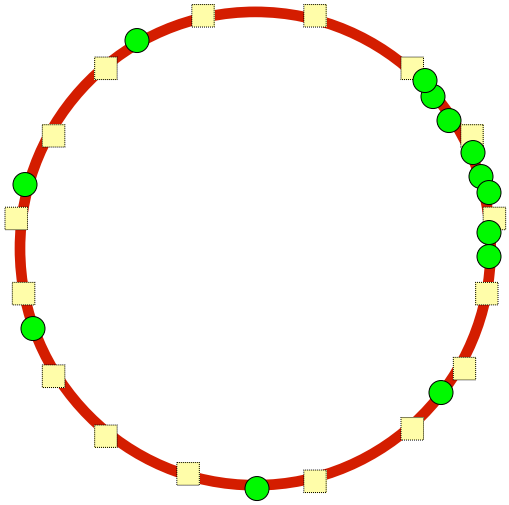
- No cryptographic hashing for key → identifier



163

DHTs with Load Balancing

- Nodes in popular regions can be overloaded
- Load imbalance!

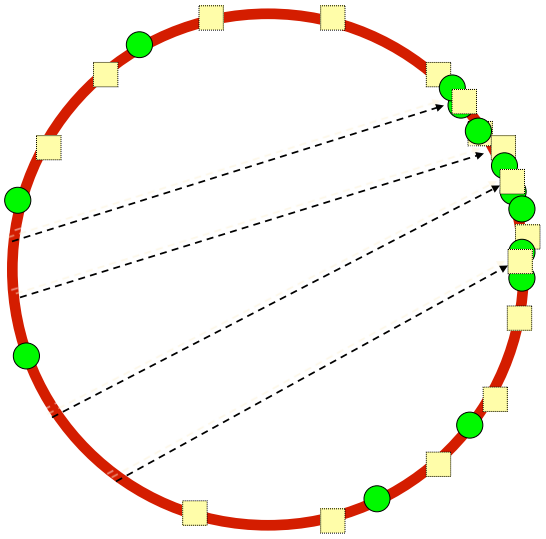


164

DHTs with Load Balancing

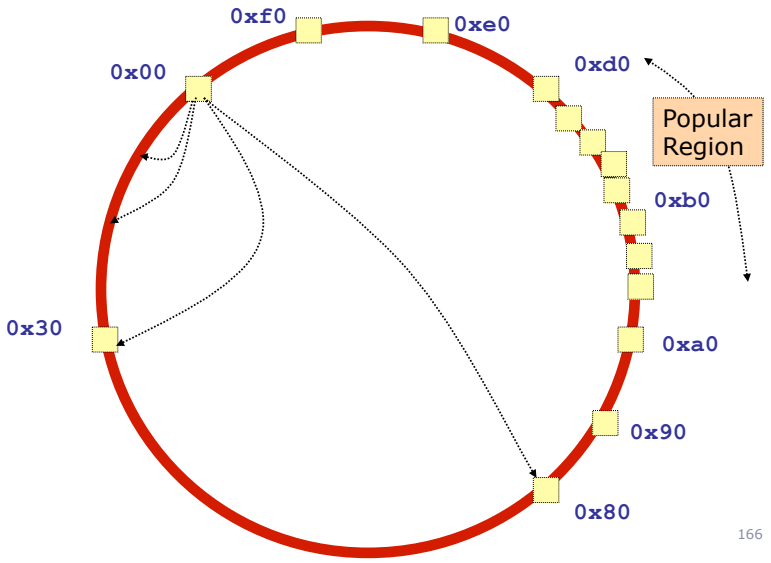
- Mercury load balancing strategy
  - Re-adjust responsibilities

- Range ownerships are skewed!



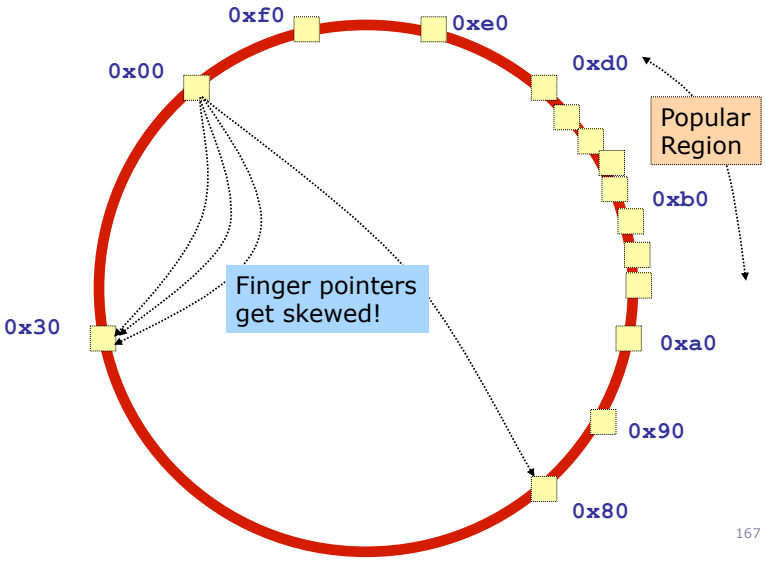
165

DHTs with Load Balancing



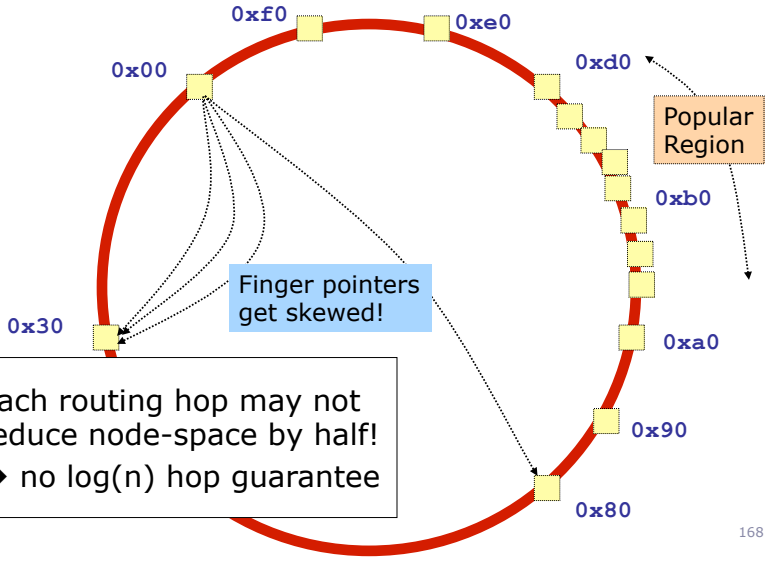
166

DHTs with Load Balancing



167

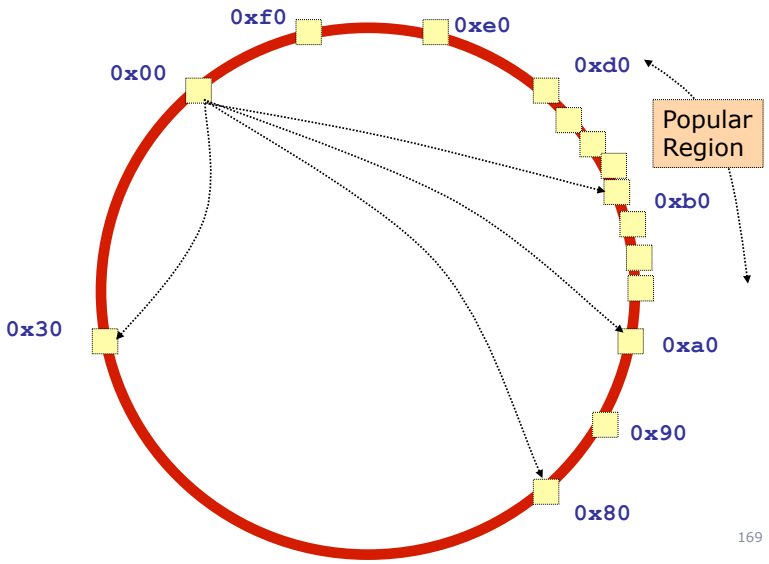
DHTs with Load Balancing



- Each routing hop may not reduce node-space by half!
- ➔ no log(n) hop guarantee

168

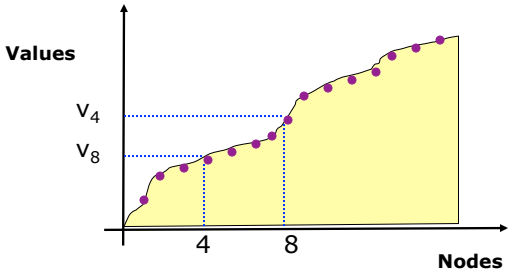
Ideal Link Structure



169

Mercury

- Need to establish links based on node-distance

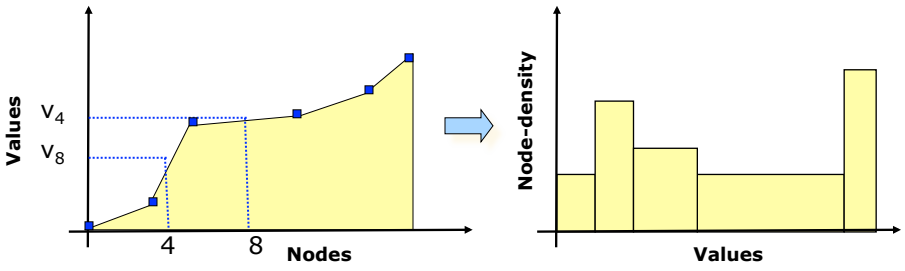


- If we had the above information…
- For finger *i*
  - Estimate value **v** for which **2^i th** node is responsible

170

Mercury

- Need to establish links based on node-distance



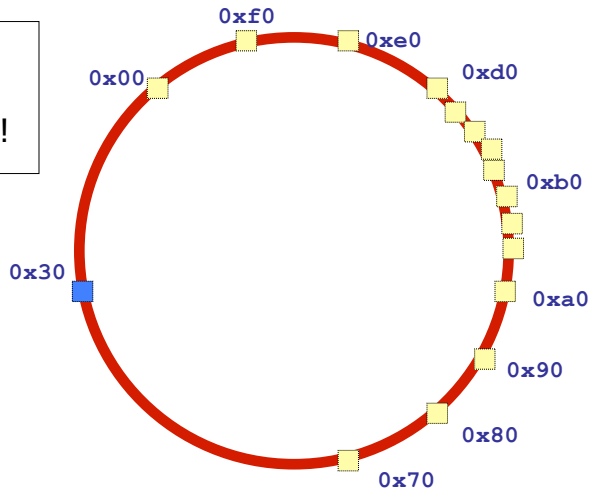
Piece-wise linear approximation

Histogram

171

Histogram Maintenance

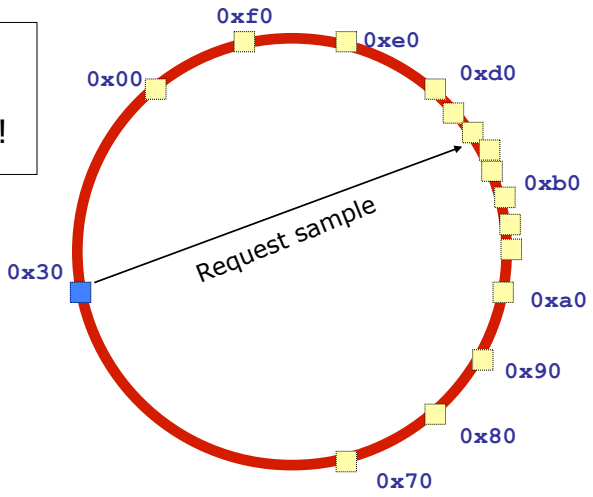
- Measure node-density locally
- Gossip about it!



172

Histogram Maintenance

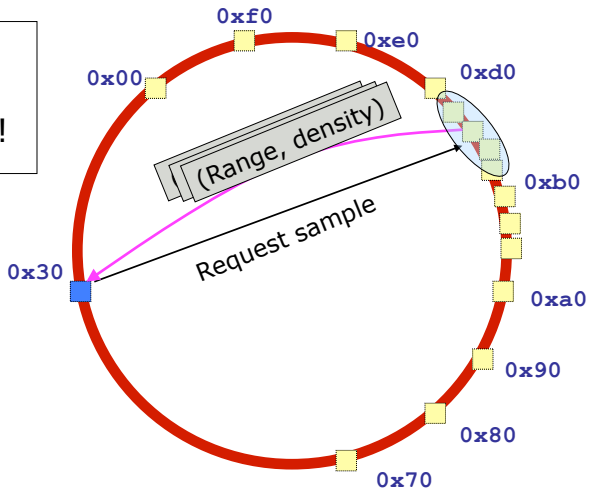
- Measure node-density locally
- Gossip about it!



173

Histogram Maintenance

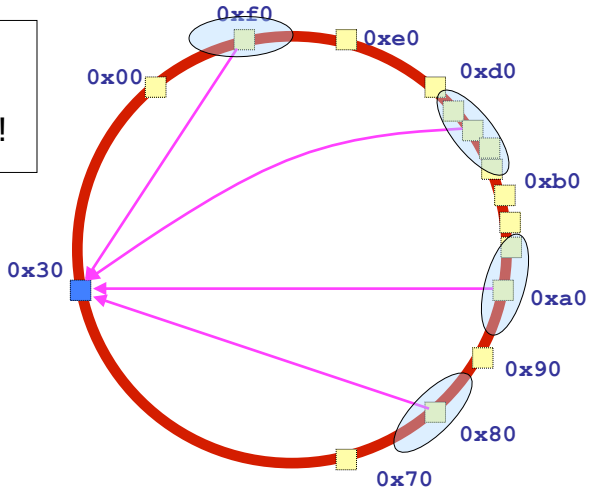
- Measure node-density locally
- Gossip about it!



174

Histogram Maintenance

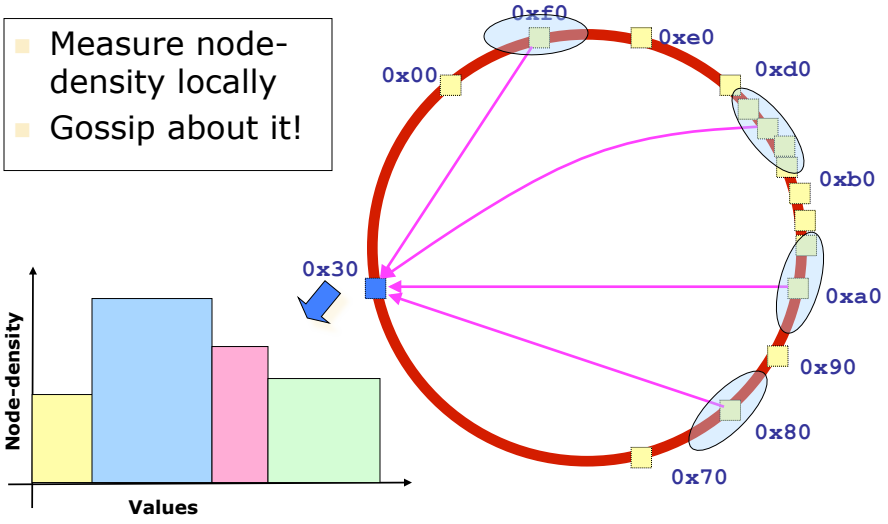
- Measure node-density locally
- Gossip about it!



175

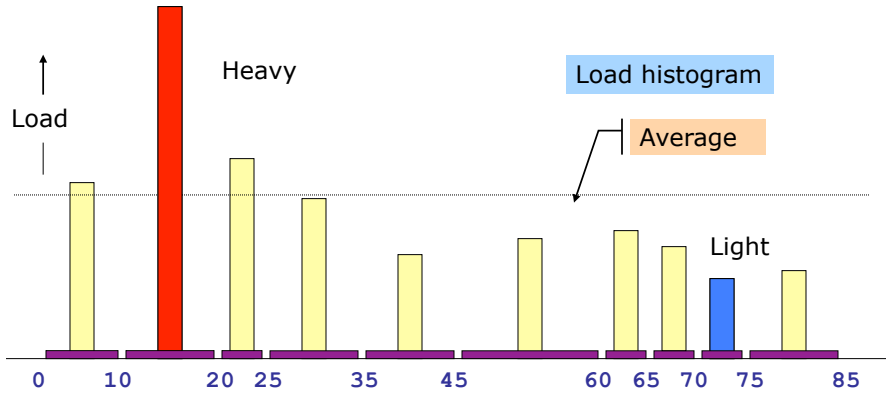
Histogram Maintenance

- Measure node-density locally
- Gossip about it!



176

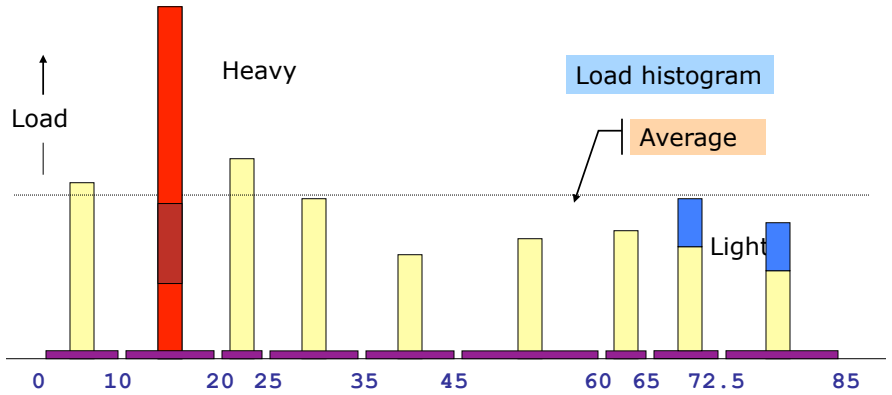
Load Balancing



- Basic idea: leave-rejoin
- Steps
  - Find average, check if heavy or light
  - Light nodes perform a leave and rejoin

177

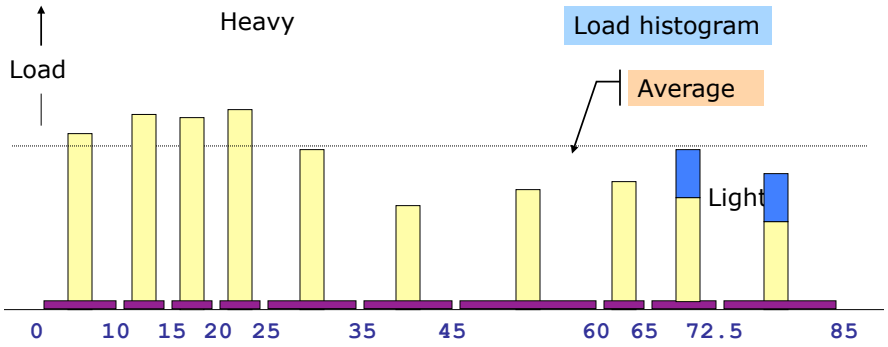
Load Balancing



- Basic idea: leave-rejoin
- Steps
  - Find average, check if heavy or light
  - Light nodes perform a leave and rejoin

178

Load Balancing

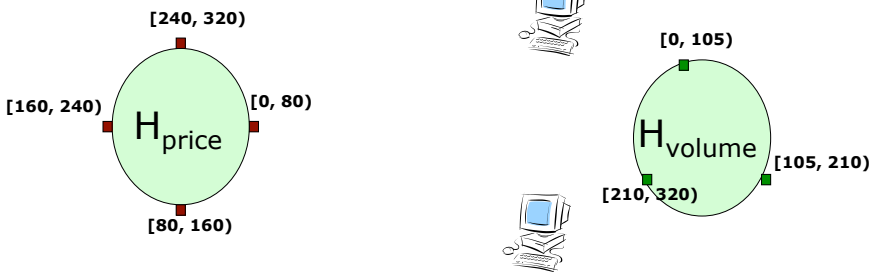


- Basic idea: leave-rejoin
- Steps
  - Find average, check if heavy or light
  - Light nodes perform a leave and rejoin

179

Mercury Routing

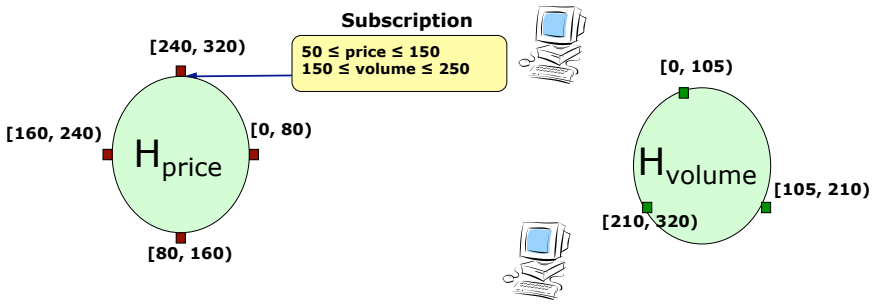
- Send subscription to any one attribute hub
- Send publications to all attribute hubs
- Tunable number of long links → can range from full-mesh to DHT-like



180

Mercury Routing

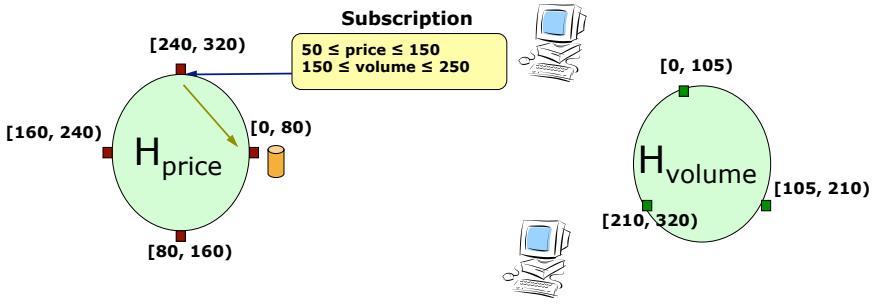
- Send subscription to any one attribute hub
- Send publications to all attribute hubs
- Tunable number of long links → can range from full-mesh to DHT-like



181

Mercury Routing

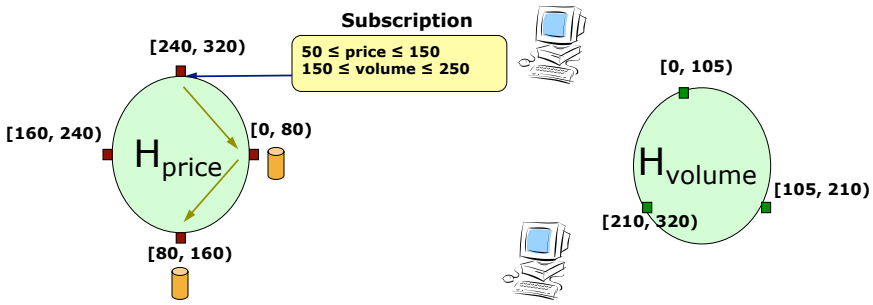
- Send subscription to any one attribute hub
- Send publications to all attribute hubs
- Tunable number of long links → can range from full-mesh to DHT-like



182

Mercury Routing

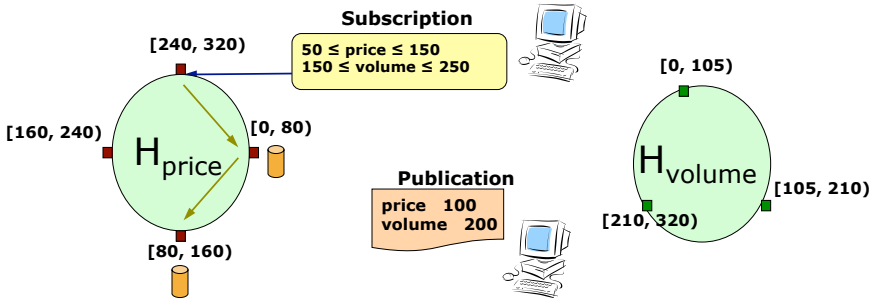
- Send subscription to any one attribute hub
- Send publications to all attribute hubs
- Tunable number of long links → can range from full-mesh to DHT-like



183

Mercury Routing

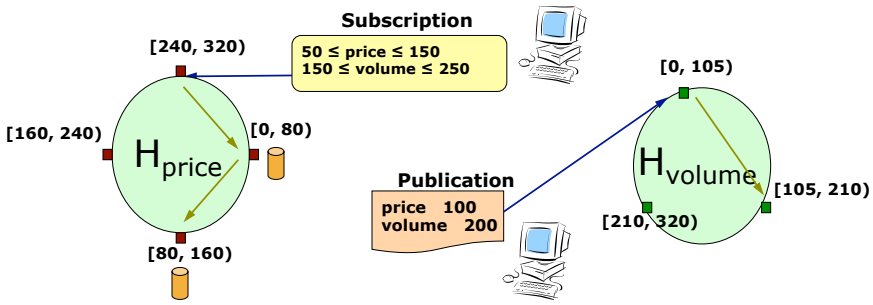
- Send subscription to any one attribute hub
- Send publications to all attribute hubs
- Tunable number of long links → can range from full-mesh to DHT-like



184

Mercury Routing

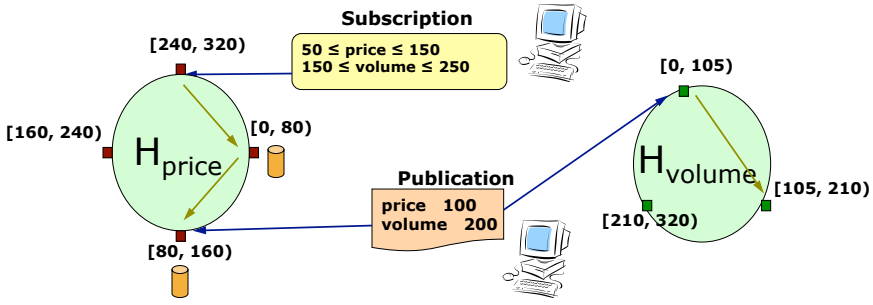
- Send subscription to any one attribute hub
- Send publications to all attribute hubs
- Tunable number of long links → can range from full-mesh to DHT-like



185

Mercury Routing

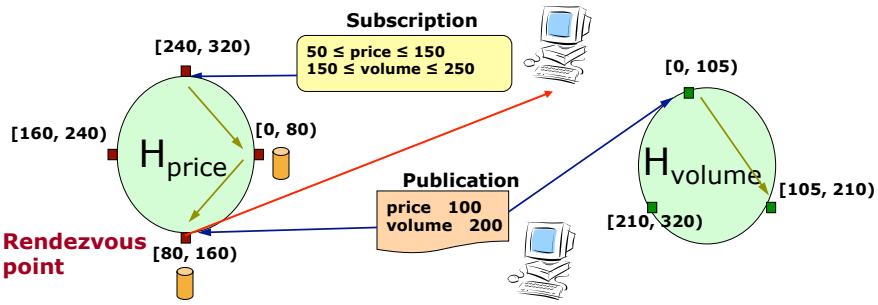
- Send subscription to any one attribute hub
- Send publications to all attribute hubs
- Tunable number of long links → can range from full-mesh to DHT-like



186

Mercury Routing

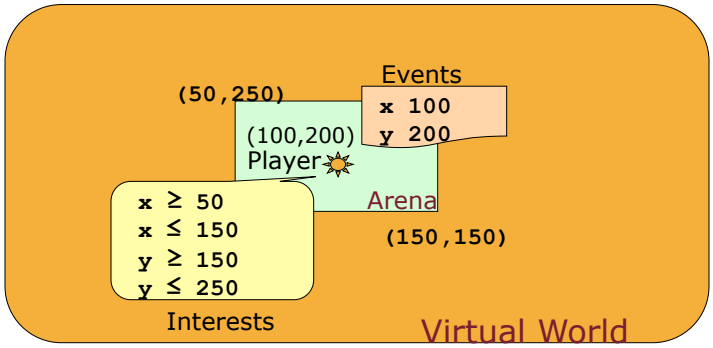
- Send subscription to any one attribute hub
- Send publications to all attribute hubs
- Tunable number of long links → can range from full-mesh to DHT-like



187

Object Discovery – basic design

- Use publish-subscribe to "register" long-lived distributed lookups
- Publications created each time object is updated (or periodically when no update is done)
  - Publication contents = state of object



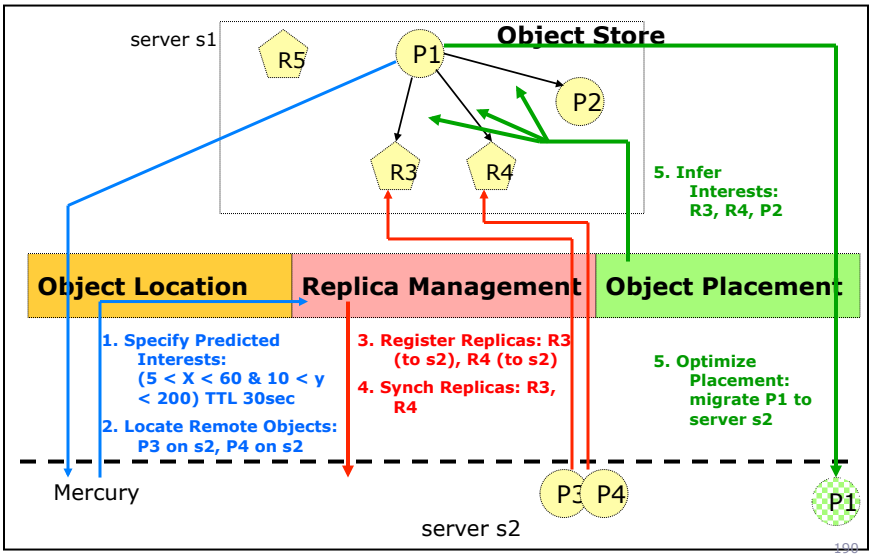
188

Prefetching and Persistence

- Basic design has problems
  - Collecting/updating existing state is too slow
  - High subscription update rate
- 1st fix → use Mercury only for object discovery and not state update
- 2nd fix → persistent publications/subscriptions
  - Publication lifetimes → subsequent subscriptions would immediately trigger transmission
  - Subscription lifetimes → enabled soft-state approach to subscriptions
- 3rd fix → predict future subscriptions
- Creates a new hybrid of persistent storage and pubsub

189

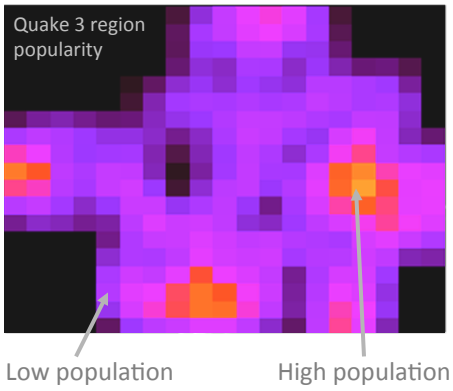
Colyseus Architecture Overview



190

Area of Interest (AOI) Filtering

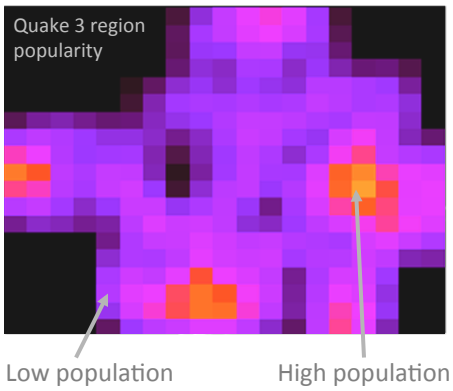
- Only receive updates from players in your AOI
  - Colyseus[Bharambe '06]
  - VON [Hu '06]
  - SimMUD[Knutsson '04]



191

Area of Interest (AOI) Filtering

- Only receive updates from players in your AOI
  - Colyseus[Bharambe '06]
  - VON [Hu '06]
  - SimMUD[Knutsson '04]
- Problems:
  - Open-area maps, large battles
  - Region populations naturally follow a power-law [Bharambe '06, Pittman '07]



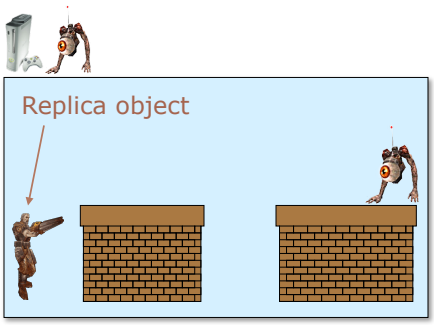
Requirement: ~1000 players in *same* AOI

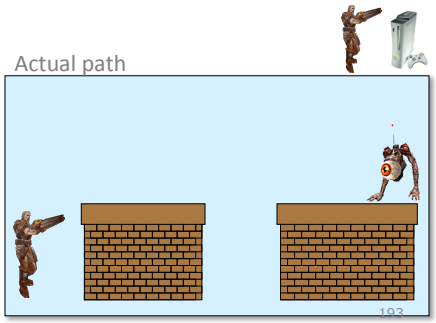
192

Smoothing Frequest Updates

- Send *guidance* (predictions) instead of state updates
- *Guidable AI* extrapolates transitions between points
  - E.g., game path-finding code

- **Problem**: Predictions are not always accurate
  - Interactions appear inconsistent
  - Jarring if player is paying attention

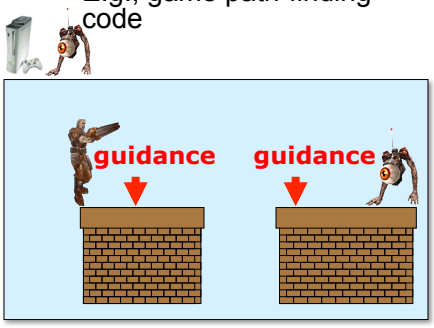


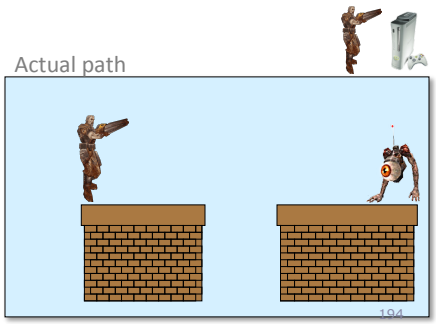


Smoothing Frequent Updates

- Send *guidance* (predictions) instead of state updates
- *Guidable AI* extrapolates transitions between points
  - E.g., game path-finding code

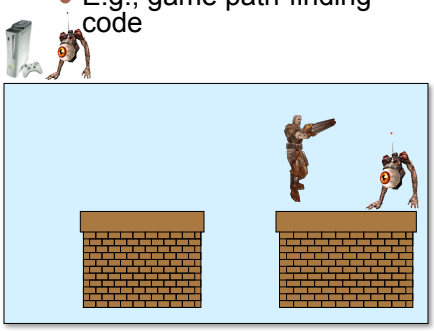
- **Problem**: Predictions are not always accurate
  - Interactions appear inconsistent
  - Jarring if player is paying attention



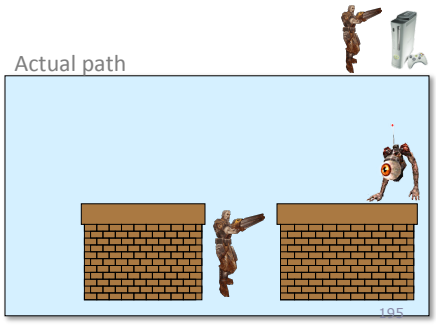


Smoothing Frequest Updates

- Send *guidance* (predictions) instead of state updates
- *Guidable AI* extrapolates transitions between points
  - E.g., game path-finding code

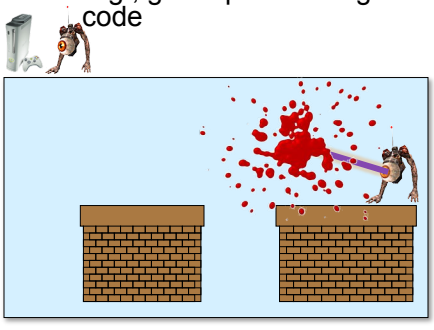


- **Problem**: Predictions are not always accurate
  - Interactions appear inconsistent
  - Jarring if player is paying attention

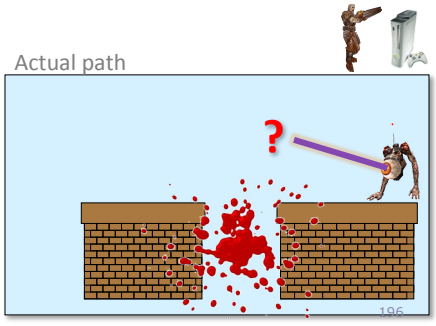


Smoothing Frequent Updates

- Send *guidance* (predictions) instead of state updates
- *Guidable AI* extrapolates transitions between points
  - E.g., game path-finding code

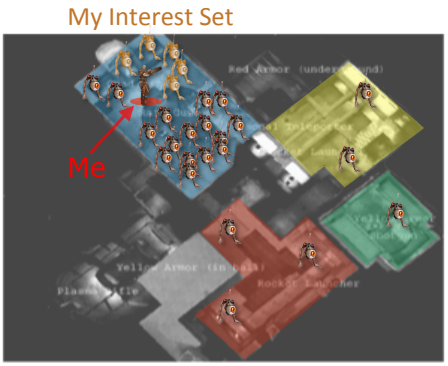


- **Problem**: Predictions are not always accurate
  - Interactions appear inconsistent
  - Jarring if player is paying attention



Donnybrook: Interest Sets

- **Intuition**: A human can only focus on a constant number of objects at once [Cowan '01, Robson '81]
  ⇒Only need a constant number of high-accuracy replicas
- **Interest Set**: The 5 players that I am most interested in
  – *Subscribe* to these players to receive 20 updates/sec
  – Only get 1 update/sec from everyone else



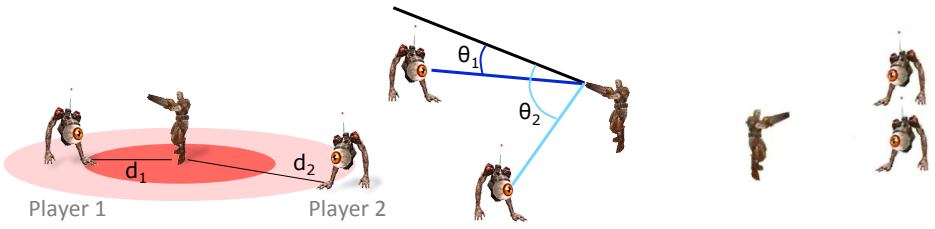
197

Donnybrook: Interest Sets

- How to estimate human attention?
  – Attention(**i**) = how much I am focused on player **i**

Attention(**i**) =

$f_{proximity}(d_i)$ + $f_{aim}(\theta_i)$ + $f_{interaction\text{-}recency}(t_i)$



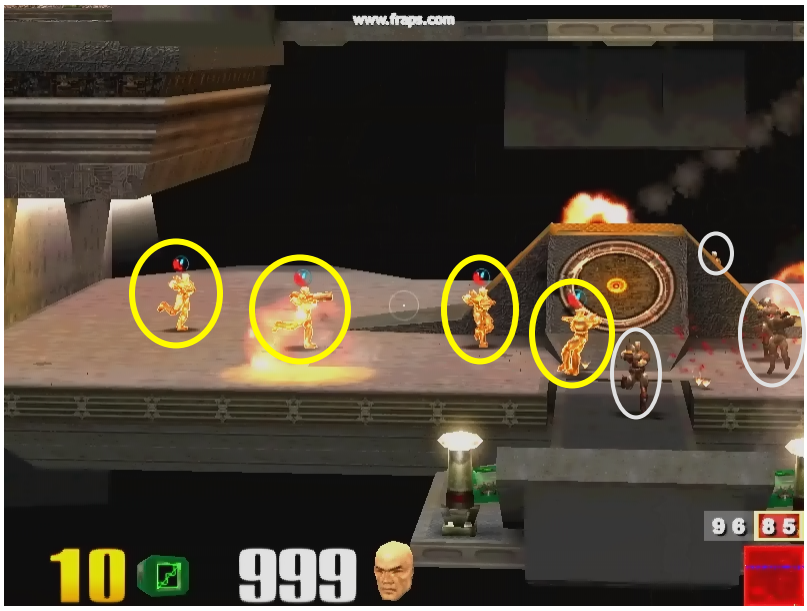
198


www.fraps.com
Not in Interest Set
= Interest Set
11
999
90 79

199


www.fraps.com
9
999
96 85

200



201

202