Peer-to-Peer Communication

Colin Perkins

http://csperkins.org/teaching/2004-2005/gc5/

UNIVERSITY
of
GLASGOW

Lecture Outline

- Peer-to-Peer Systems for Grid Computing
- Distributed Hash Tables
  - Finding stuff
  - For File Sharing/data Storage
  - For Event Notification
- Distributed Monitoring and Data Aggregation
- Deployment Considerations
  - NAT
  - Firewalls
  - Overlay Networks

What are Peer-to-Peer Systems?

- Every participating host acts as both a client and a server
- Properties:
  - No central coordination; no global knowledge
  - All existing data and services are accessible from any peer
  - Peer nodes may come and go at any time
  - Data stored at peers may change dynamically
- Requirements:
  - Scalability to networks with arbitrary sizes
  - Performance: low lookup latency; small traffic load
  - Adaptive to constant topology/data content changes without incurring high maintenance overhead
  - Tolerance to heterogeneity of resources (bandwidth, etc.) across peers
  - Load balancing
  - Security

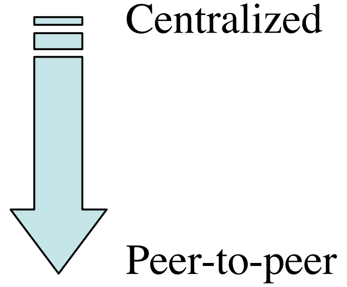
Peer-to-Peer Systems and Grid Computing

- Current grid computing applications often use existing protocols in a peer-to-peer mode
  - Dynamic virtual organizations
  - Services instantiated on various machines
  - Data sharing, etc.
- Assumes the network provides transparent end-to-end connectivity
  - Issues with NAT and firewalls; otherwise well understood…
- Can we use newer peer-to-peer technologies to improve grid computing systems?
  - To efficiently find stuff
  - For file sharing/data dissemination
  - For large scale event notification
  - To efficiently monitor large scale distributed applications

Finding Stuff

- The problem:
  - Given the name of an object, which could be located anywhere in a distributed system, efficiently locate that object
  - Desirable features:
    - Scalable to large systems, many objects
    - Fault tolerant, degrades gracefully
    - Allows unstructured names (to support any type of data)

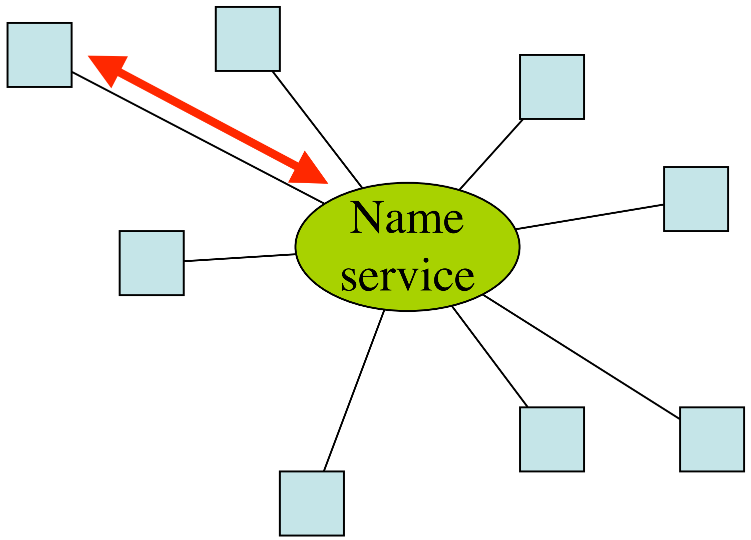
- Solutions:
  - Centralized name service
  - Distributed hierarchical name service
  - Distributed flooding
  - Distributed hash table



More discussion in [1]

Centralized Name Service

- Nodes advertise names of the objects they hold to a central name service
- All searches resolved by that central (replicated?) service
  - Allows unstructured names; any host can hold any object



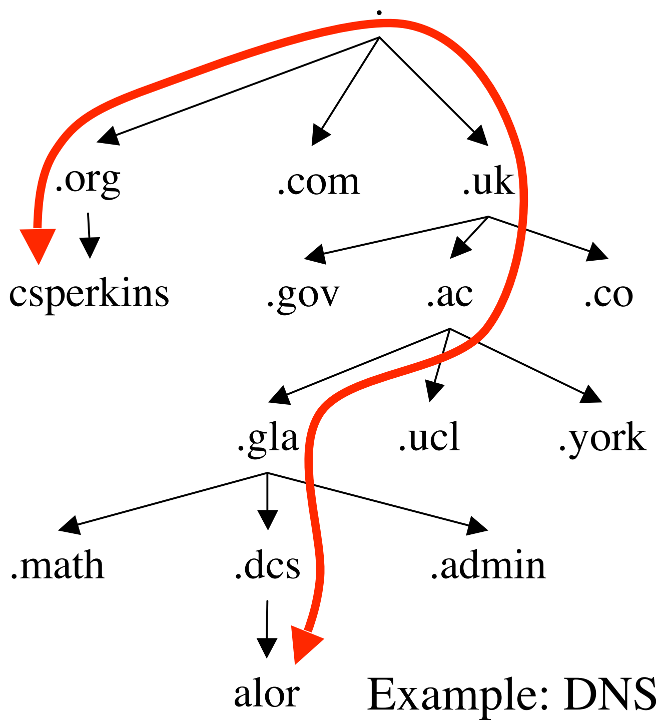
- Problems:
  - Doesn’t scale
  - Single point of failure
  - Centralized control

  } Want a distributed name service

- E.g. Globus, CORBA, etc.

Distributed Hierarchical Naming

- Assign hierarchical names to objects
  - Delegate portions of the namespace to different entities/organizations
- Build a search tree
  - Each node knows its parent and children
  - Search for key ascends up towards the root, then descends into the tree
  - Value returned via reverse search path



Example: DNS

- Assumptions:
  - Objects can be named hierarchically
  - Object ownership can be delegated to different organizations matching the hierarchical naming
  - Single root ⇒ centralized control

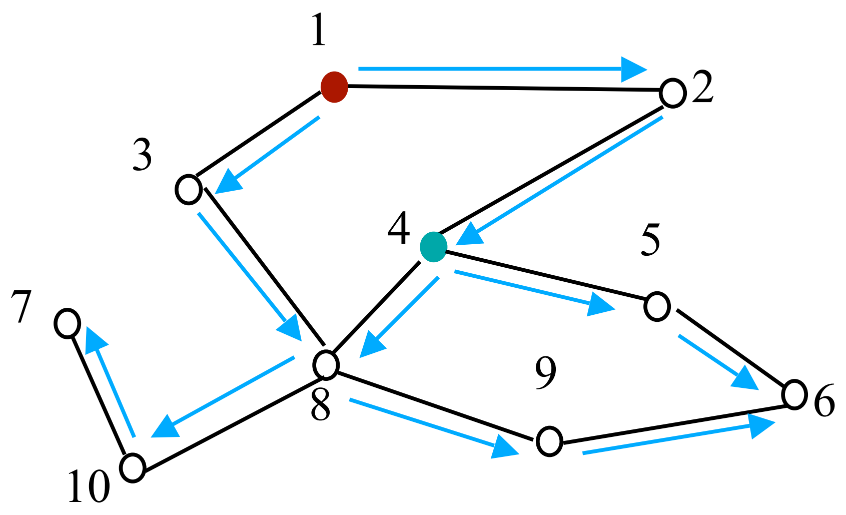
} Still not ideal; prefer decentralized system with unstructured names

Distributed Flooding

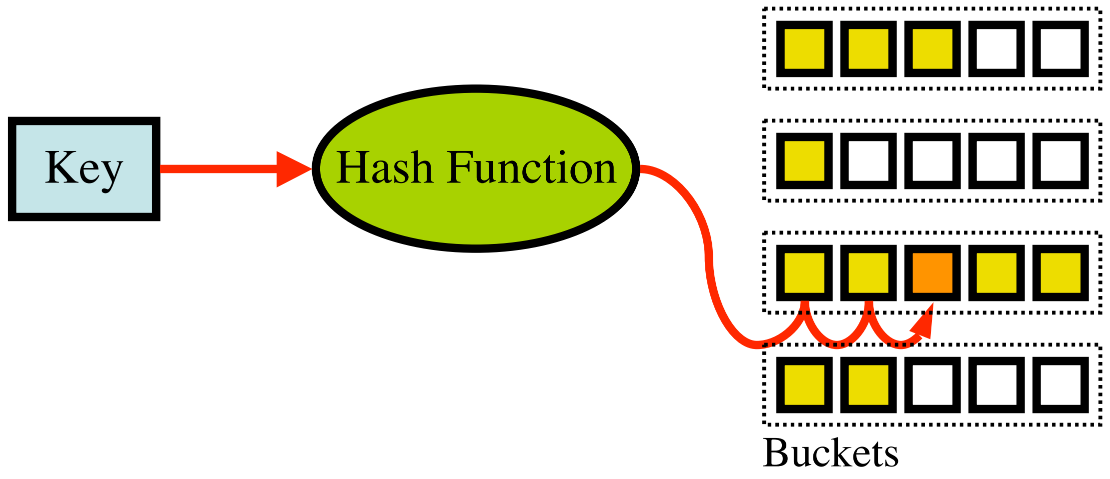
- Every node forwards packets to all of its neighbors
- Lifetime of packets are limited by time-to-live
- Packets have unique identifiers to detect loops
- Allows unstructured names
- Simple, robust, but generates *huge* amounts of traffic
- Example: Gnutella

Node 1 initiates search

Data floods throughout network, even though result found early (at node 4)

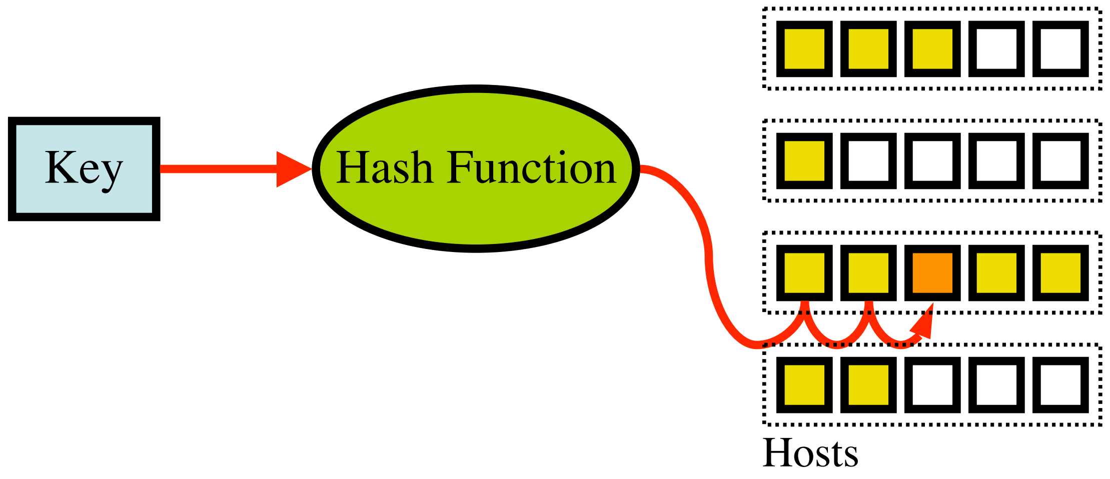


A Distributed Hash Table (DHT)



- A classical hash table efficiently returns a *value* given a *key*
- Passes the key through a *hash function* which maps it onto a fixed bucket address
  - Choice of hash function important, to evenly distribute keys to buckets
- Iterate through items in the bucket to find value corresponding to the key; return that value
- Space-time trade off to determine number and size of buckets

A Distributed Hash Table



- A DHT is similar, but distributed across a group of hosts
  - Efficient lookup of data that is located on one of a set of hosts
  - Each bucket located on a different host
- Each host can use the hash table to retrieve values for any key
- Scaling to large numbers of nodes and keys desirable
  - Cannot assume global knowledge
  - Must be fault tolerant

Key Properties of a DHT

- Keys are *unstructured*
  - No need for hierarchical names
  - Works with any sort of data
- Data is distributed
  - Each node responsible for a portion of the data space
- Queries are routed efficiently
- No central server or control
  - No node has global state
  - No node has a special position
  - Relies on hash function to provide implicit global knowledge

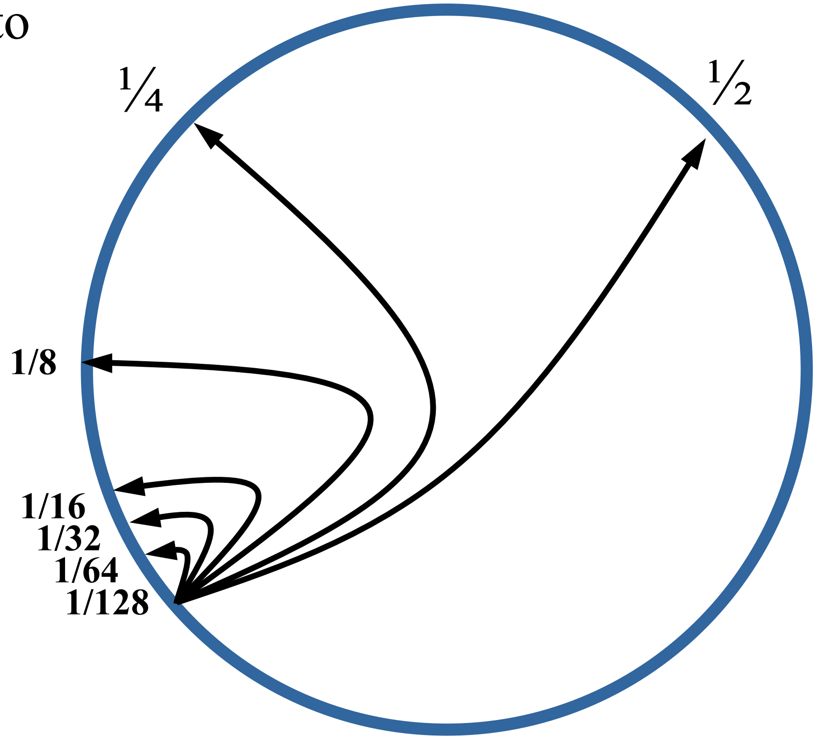
DHT Examples

- Many examples of DHT in the literature, trying to formalize the structure of peer-to-peer name resolution
  - Compared to the many unstructured file-trading systems with ad-hoc name lookup, flooding or centralized schemes
  - Aiming to develop systems that can be reasoned about; have known lookup latency, state requirements, etc.
- Three representative examples:
  - Chord
  - CAN
  - Tapestry

  - Will show basic routing algorithm for each; ignore details of neighbour discovery, handling joins and leaves, etc.

Example: Chord

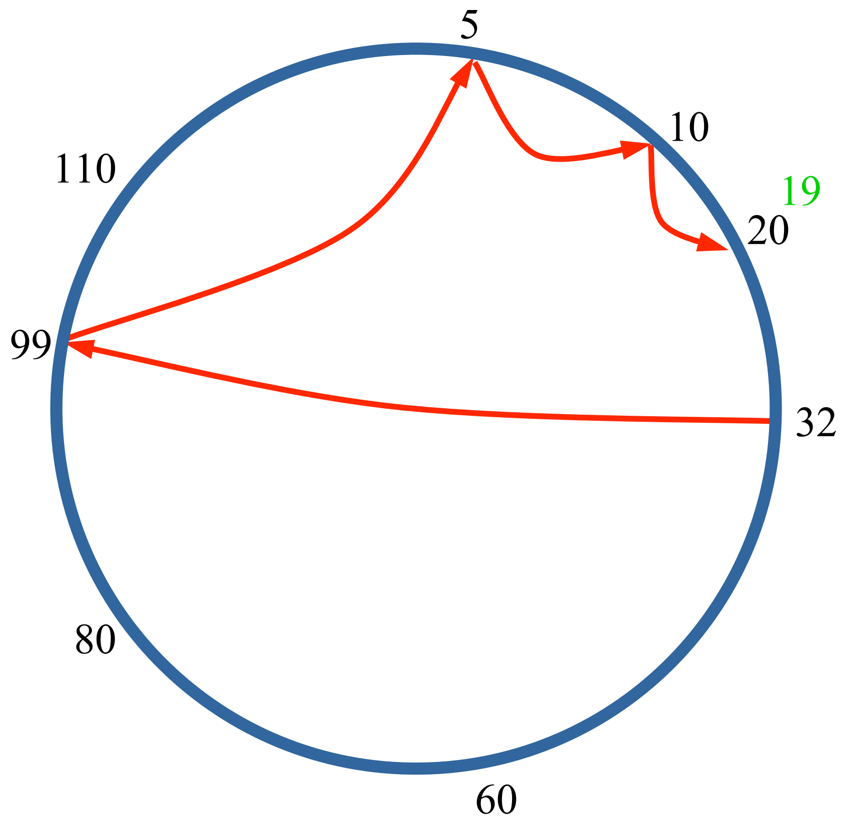
- Distributed name lookup, can be used to build a DHT:
  - Lookup(key) → IP address
  - Chord does not store the data
- Nodes, keys identified by hash value:
  - Node ID is hash of IP address
  - Key ID is hash of key
  - Both share the same numeric space
    - 160 bit SHA-1 hashes
- *N* Nodes arranged in a virtual ring
  - Hash values under modulo arithmetic
  - O(*log(N)*) links to other nodes
    - Links to nodes placed $^1/_2$, $^1/_4$, $^1/_8$, $^1/_{16}$, … way around the ring
    - More links to nodes with similar node ID
  - A node manages all keys with key ID less than its node ID, but greater than the previous node's ID



Details in [2]

Example: Chord

- Nodes maintain a routing table:
  - (Node ID, IP address) for each link
- Each hop routes queries along the link to the node with the greatest node ID less than key hash (modulo arithmetic)
  - Each hop halves the distance - in the hash space - to the node with the key
  - Actual network distance unknown at each hop
- Reaches destination in O(*log(N)*) hops
  - Efficient loopup in a large space

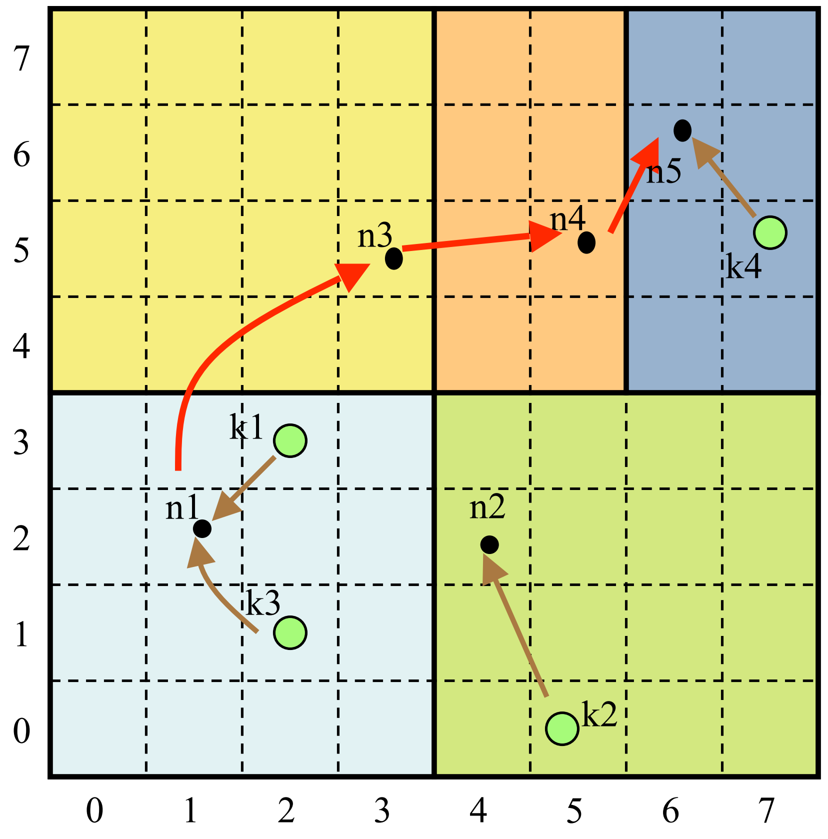


- Robust to node failures; simply choose a different (longer) path around ring

Details in [2]

Example: CAN

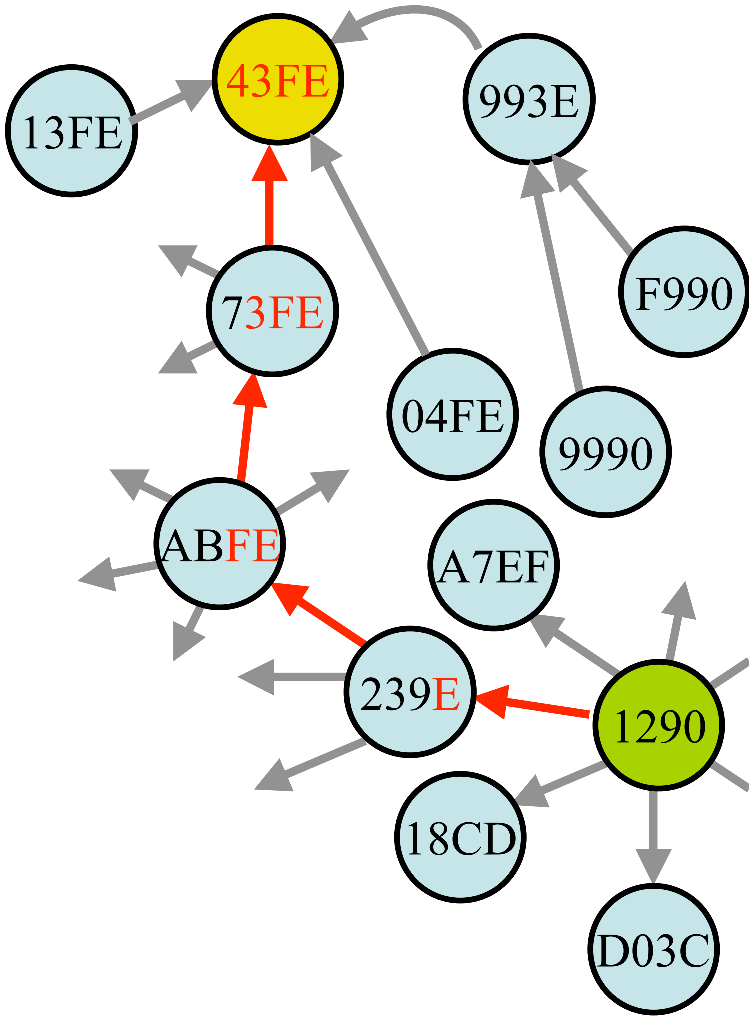
- Uses an d dimensional coordinate space on a torus ($d \geq 2$)
- Entire space divided between nodes
  - Each node owns a hyper-rectangle in the space, based on hash of node address
- Key-value pairs are stored in the CAN:
  - Keys mapped to points in coordinate space using a hash function
  - Values stored at the node owning that part of the space
- Each node knows its neighbours
  - *2d* neighbours (one in each direction for each dimension)
- Forward query to the neighbour node towards the location of the key in the coordinate space
  - Reaches destination in $O(N^{1/d})$ hops



Details in [4]

Example: Tapestry

- Node and key ID assigned based on hash, converted to digits base *b* (e.g. base 16)
- Global mesh; each node has links to *b* nodes matching each possible suffix of its address
  - $b \log_b N$ neighbours
- Routes to the *closest* neighbour with longer match to the desired address, digit-by-digit
  - Reaches destination in $O(\log_b N)$ hopes
  - Will match several digits in one hop, if there is a matching neighbour
- Efficient name lookup; topology based routing



Details in [3]

Comparison: Chord *vs.* Tapestry *vs.* CAN

	Model	**Search Time**	**Node State**
Chord	1 dimensional	$O(\log N)$	$O(\log N)$
CAN	*d* dimensional	$O(N^{1/d})$	$2d$
Tapestry	Mesh	$O(\log_b N)$	$O(b \log_b N)$

- Each makes a different trade-off between search time and amount of state required
- Different fault tolerance and robustness properties
- Different degrees of complexity in maintaining the overlay
- Different behaviour when membership changes rapidly

⇒ Robustness issues as systems scale *not* yet well understood

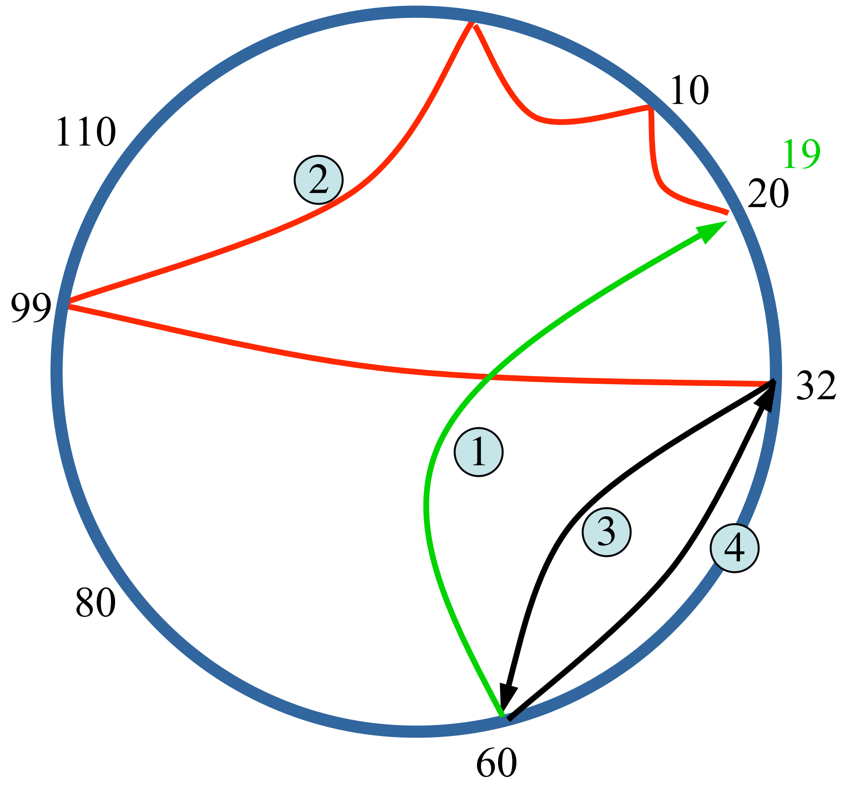
Uses of Distributed Hash Tables

- A DHT maps from key to value
  - Efficient and location transparent lookup
  - Scalable to very large distributed systems

- Can be used for:
  - File sharing and data dissemination
  - Publish/subscribe based event notification
  - Distributed object location
  - Etc.

Peer-to-Peer File Sharing/Data Storage

- A DHT can be used to build distributed indexes for file sharing or data dissemination
  - Keys are hashed “filenames”, values returned are URLs for the actual data
  - Nodes keep the data, but publish location into the DHT for lookup



1. Name published in the DHT
2. DHT name lookup to find location of data item
3. Request data from peer
4. Data transferred directly

Note: name stored in DHT based on hash function; not data location

OceanStore

- An example of a distributed file system, built using a DHT [5, 6]
- Un-trusted infrastructure
  - Extensive use of cryptography to ensure privacy; enforce access rules
  - Extensive use of caching for robustness and performance
- Files identified by a hash of the filename + path
  - Files split into blocks, returned data structure is pointer to a table of hashes for blocks
    - Blocks indexed by cryptographic hash of contents
    - Blocked pushed into the network, located using Tapestry
  - Copy-on-write semantics for block; old versions retained forever
    - Efficient: only changes between versions stored
    - Efficient: files that share content automatically share storage since they hash to the same block, closest replica of the block located by Tapestry

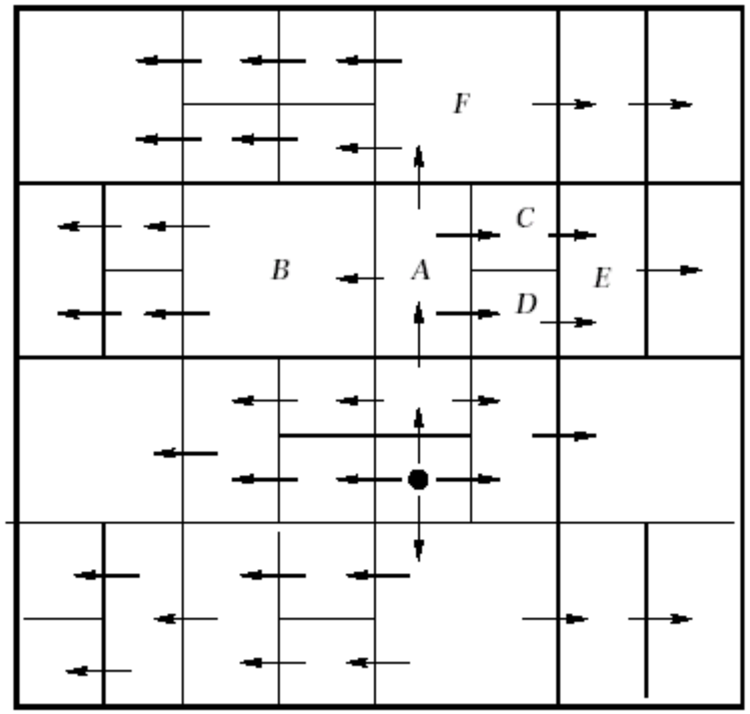
- [Lots of details skipped: see the papers…]

Publish/Subscribe and Event Notification

- Want to distribute notifications of events to a group of subscribers in a scalable manner
- Two problems:
  - How to locate the publisher of the events ⇒ DHT name lookup
  - How to notify subscribers of new events ⇒ application level multicast
    - Often built-in to the operation of a CAN using reverse path forwarding down the lookup tree (e.g. Bayeux); very similar to IP multicast
- Examples:
  - Multicast on CAN
  - Scribe on Pastry
  - Bayeux on Tapestry
  - Etc.

Example: Multicast on CAN

- Group address hashes to node in the CAN
  - Other nodes contact that node and build a separate mini-CAN
  - Instead of building a multicast distribution tree, build the mini-CAN
- Flood packets from source to all nodes on the mini-CAN
  - Straight forward operation, since CAN has regular structure
  - Easier than a multicast tree; assumes building a CAN is light-weight



See [7] for details

Distributed Monitoring and Aggregation

- Problem: How to effectively monitor state of a large system?
  - Distributing full system state very inefficient
  - Full state details not interesting, provided everything working
  - Often sufficient to see a summary of the results
- Examples:
  - Network management
  - Distributed data mining
- Desirable to use a peer-to-peer protocol to distribute and aggregate results
  - Spread processing load across the network
  - Keep state local unless explicitly requested; distribute summaries widely
  - Scalable communication

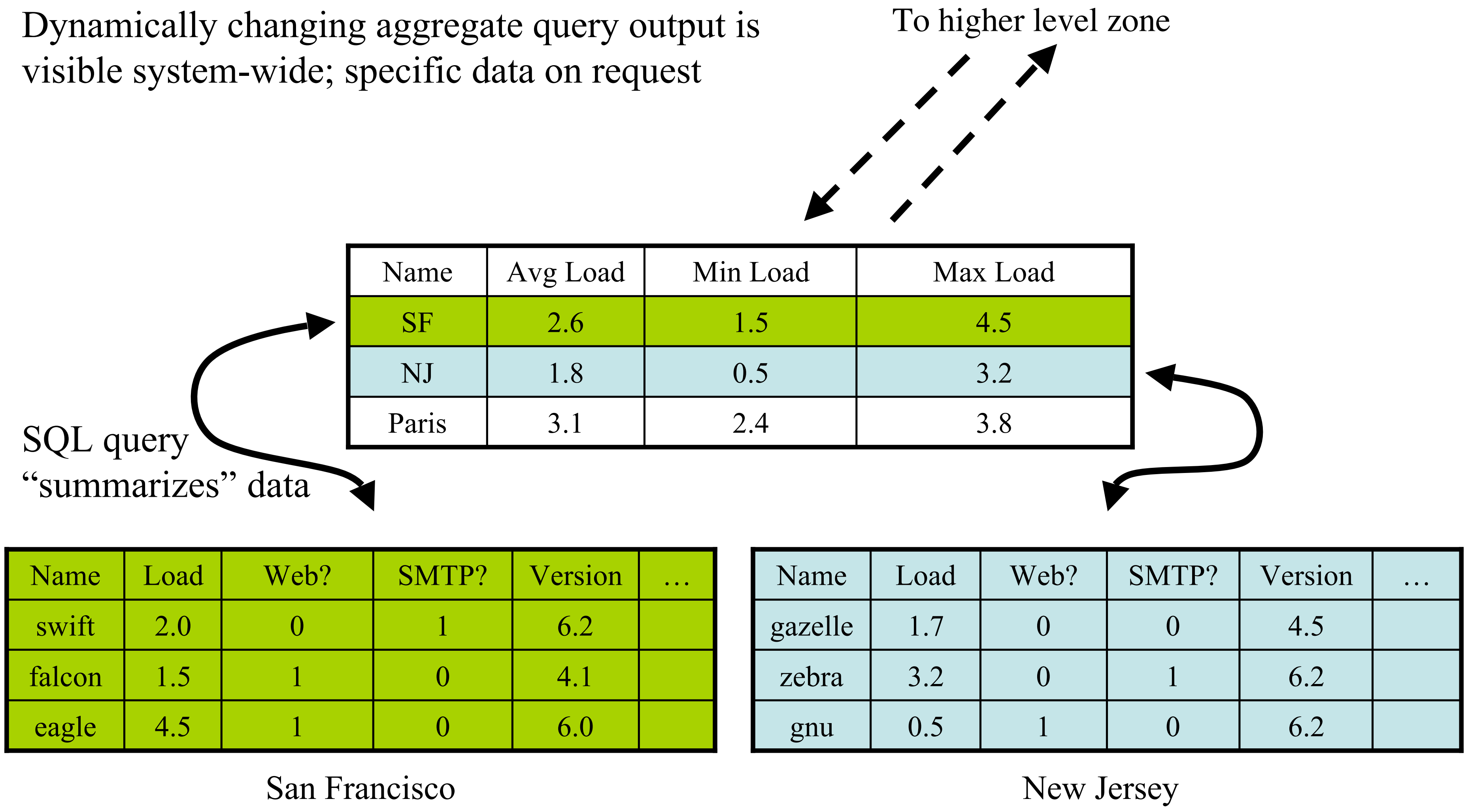
Example: Astrolabe

- Goal is to create a dynamic peer-to-peer database that shows the continuously evolving state of some system
  - Pass SQL queries down through a peer-to-peer tree
    - Each level of the tree is a *zone*
  - Compute results locally
  - Aggregate and summarize data flowing back up the tree
- Approach: "peer to peer gossip"
  - Each machine has a piece of a jigsaw puzzle.
  - Periodically exchange state with a randomly chosen peer from your zone
    - Ensures data diffuses throughout the network with high probability
    - Aggregated data derived at hosts within a zone
  - Periodically query random child zone for it's aggregated data
- Provides aggregate data for each zone; specific queries can then be issued to find details

See [8] for details

Example: Astrolabe

Dynamically changing aggregate query output is visible system-wide; specific data on request



Name	Avg Load	Min Load	Max Load
SF	2.6	1.5	4.5
NJ	1.8	0.5	3.2
Paris	3.1	2.4	3.8

SQL query "summarizes" data

Name	Load	Web?	SMTP?	Version	…
swift	2.0	0	1	6.2	
falcon	1.5	1	0	4.1	
eagle	4.5	1	0	6.0	

San Francisco

Name	Load	Web?	SMTP?	Version	…
gazelle	1.7	0	0	4.5	
zebra	3.2	0	1	6.2	
gnu	0.5	1	0	6.2	

New Jersey

Computation done locally; query results flow back up

Distributed Monitoring and Aggregation

- Very active research area
- Lots of potential for efficiently managing large scale systems
  - Networks
  - Web server farms
  - Clusters

…no off-the-shelf solutions yet; just research prototypes

Deployment Considerations

- Peer-to-peer applications assume network provides transparent end-to-end connectivity
  - The original IP model

- Problem: widespread deployment of NAT and Firewalls breaks this transparency
  - NAT prevents inbound connections, since cannot address hosts behind the NAT
  - Firewalls can prevent both in- and out-bound connections
  - Makes it difficult to deploy peer-to-peer applications

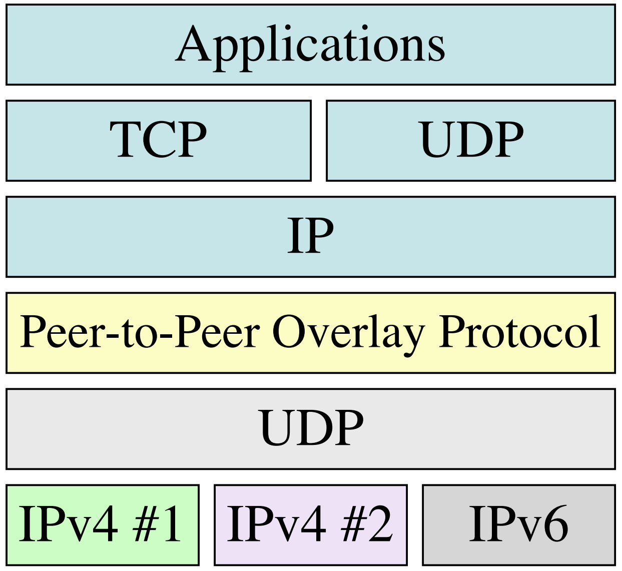
NAT Traversal

- Growing prevalence of Network Address Translation (NAT) is fragmenting the Internet
  - Complicates applications since they cannot easily name/access peers
  - Hosts no longer have unique addresses
  - Bidirectional connectivity not assured, may vary by protocol or direction
  - Especially affects protocols with dynamic connections ⇒ peer-to-peer
- Unintentional breakage, resulting from working around shortage of IPv4 addresses
- Problem widely noticed in IP telephony world
  - Numerous solutions ("kludges") under development in IETF:
    - STUN
    - TURN
    - ICE

    } Methodologies for detecting presence of NAT, deducing it's behaviour, and establishing connectivity
  - Signalling driven NAT detection and peer-to-peer connection establishment
  - (more on Monday)

NAT Traversal: Overlay Routing

- Want a generic way to traverse NAT boxes; re-establish end-to-end and transparent connectivity
- Desirable that this…
  - be implemented as reusable middleware
  - be application independent
  - allows existing applications to run unchanged
  - be simple for network administrators who must deploy it and enforce security policy



- Various proposals use middleware to build an overlay IP network, with NAT traversal and multicast support, run applications on that
- Ugly, but solves the problem…

Firewalls and Security

- Firewalls *intentionally* break connectivity for security reasons
- Many peer-to-peer applications try to work around this:
  - Dynamically chosen ports
  - Tunnelling in HTTP or other protocols
- *This is bad!*

- Leads to an arms race:
  - Peer-to-peer application evades firewall by tunnelling
  - Firewall gets more sophisticated, looks inside higher level protocol
  - Higher level protocol later modified; can't be deployed because firewalls think the new version is an attempt to tunnel a p2p application
    - E.g. how could we modify HTTP today?

- Firewall traversal a social problem; technical solutions don't work

Lecture Summary

- You should know…
  - How peer-to-peer might be used by Grid computing systems
  - Outline of operation of distributed hash tables
    - To build object location systems
    - To build file sharing applications
    - To build publish/subscribe event notification systems
  - Outline of operation of distributed monitoring and aggregation systems
  - How NAT and firewalls affect peer-to-peer application deployment

- Tutorial tomorrow: XCP
  - *Read the paper!*

References

1. H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris and I. Stoica, “Looking up Data in P2P Systems”, Communications of the ACM, Vol. 46, No. 2, February 2003.
2. I. Stoica, R. Morris, D. Karger, M. F. Kaashoek and H. Balakrishnan, “Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications”, Proceedings of ACM SIGCOMM’01, San Diego, CA, USA, August 2001.
3. B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph and J. D. Kubiatowicz, “Tapestry: A Resilient Global-Scale Overlay for Service Deployment”, IEEE Journal on Selected Areas in Communications, Vol. 22, No. 1, January 2004.
4. S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Shenker, “A Scalable Content-Addressable Network”, Proceedings of ACM SIGCOMM’01, San Diego, CA, USA, August 2001.
5. J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells and B. Zhao, “OceanStore: An Architecture for Global-Scale Persistent Storage”, Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, USA, November 2000.
6. S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao and J. Kubiatowicz, “Pond: The OceanStore Prototype”, Proceedings of the 2nd USENIX Conference on File and Storage Technologies, San Francisco, CA, USA, April 2003.
7. S. Ratnasamy, M. Handley, R. Karp and S. Shenker, “Application-level Multicast Using Content-Addressable Networks”, Proceedings of 3rd International Workshop on Networked Group Communication, London, UK, November 2001.
8. R. van Renesse, K. P. Birman, W. Vogels, “Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining”, ACM Transactions on Computer Systems, Vol. 21, No. 2, May 2003.

Remember: read to understand the concepts, not all the details