Large Scale Systems Architecture (2)

Grid Computing (M)

Lecture 8

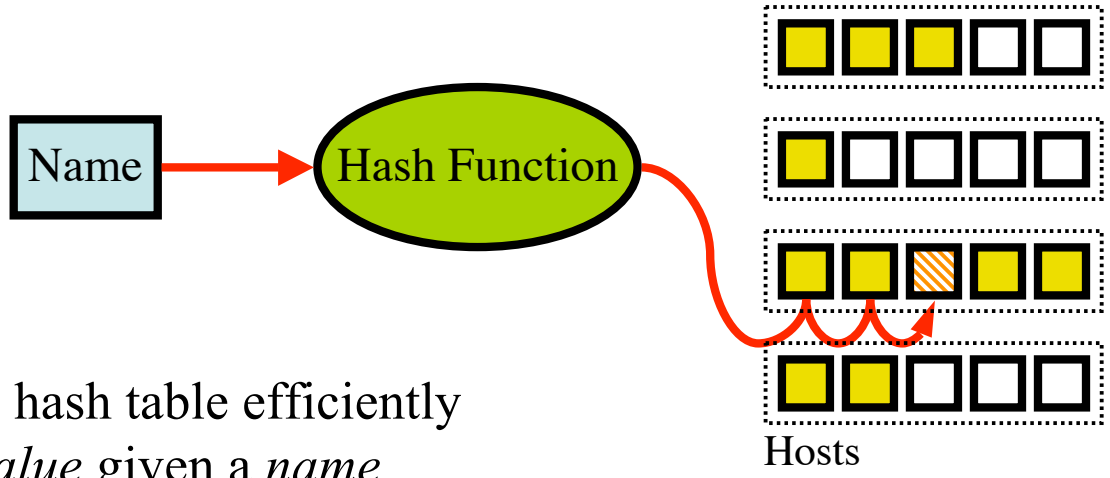
UNIVERSITY
of
GLASGOW

Lecture Outline

- The distributed hash table abstraction
  - Chord
  - Tapestry
- Example systems
  - Distributed file system: OceanStore
  - Event notification
- Deployment considerations
  - NAT
  - Firewalls

- *Future venues:*
  - *Tutorials take place in Kelvin Building, room 246B, starting on Friday*
  - *Future lectures take place in F121, Lilybank Gardens, except 14 February, when Maths 325 will be used*

A Distributed Hash Table (DHT)



- A classical hash table efficiently returns a *value* given a *name*
  - Passes name through a *hash function* mapping it to 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 hashes the name to map it to a *host*
  - Potentially flat unstructured names; location encoded via hash function
  - Iterate from host to locate object
    - Relies on a structured network protocol to point to the next host

Key Properties of a DHT

- Keys are *unstructured*
  - No need for hierarchical names
  - Works with any sort of data
- Data is distributed using a *structured* protocol
  - 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.

- Two representative examples:
  - Chord [http://pdos.csail.mit.edu/chord/]
  - Tapestry [http://www.cs.ucsb.edu/~ravenben/tapestry/download/tapestry-2.0.1.tar.gz]

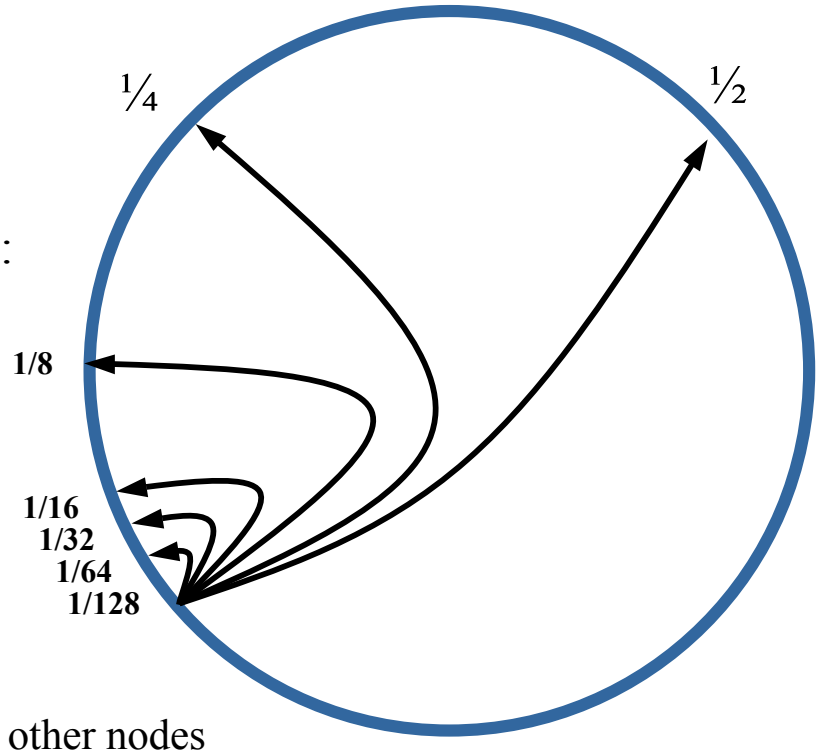
  - Will show basic routing algorithm for each
    - Details in the papers referenced on final slide
  - Each is a structured peer-to-peer system; but with very different structure

Chord

- A scalable distributed name lookup protocol
  - Lookup(*key*) → *IP address*
  - Provides an efficient lookup service, but *does not* store data
    - The Chord library will tell you where a key should be located
    - The application using Chord is responsible for storing the data at the specified location, and for contacting the returned location to retrieve data after lookup

- One of the first structured DHT algorithms
  - Relatively simple protocol; predictable behaviour
  - Widely studied with known properties
  - Representative of a large class of similar algorithms
    - Pastry
    - Bamboo *a.k.a.* OpenDHT [http://bamboo-dht.org/]
    - Kademlia
      - Overnet, eDonkey, tracker-less BitTorrent, etc.

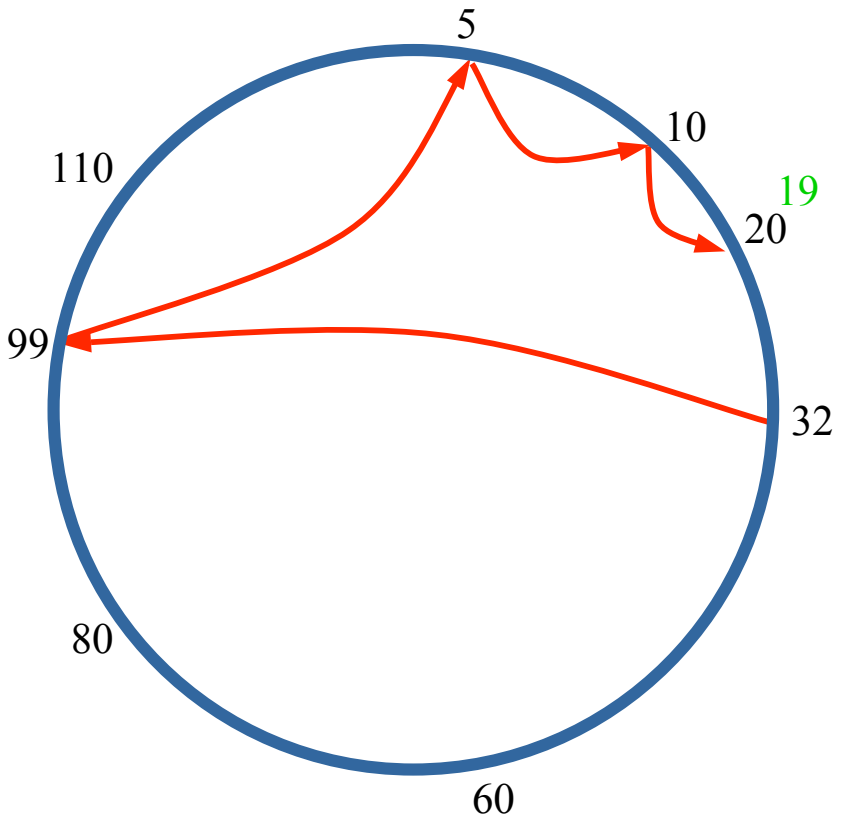
Chord: Basic Structure



- A *structured* distributed hash table
  - Nodes and 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
      - Flat, uniform, namespace
  - *N* nodes arranged in a virtual ring
    - Hash values under arithmetic modulo *N*
    - Links to neighbour nodes and O(*log(N)*) 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
      - The "finger table"
  - Each node manages all keys with key ID less than its node ID, but greater than the previous node's ID, modulo *N*

Chord: Key Lookup

- 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 *N*)
  - Each hop halves the distance - in the hash space - to the node with the key
  - Eventually, successor node owns the key, so pass to successor
- Reaches destination in *O*(*log N*) hops
  - Efficient in terms of hop count
  - Makes no attempt to minimize network distance covered by each hop



- Robust to node failures or incorrect finger tables
  - Simply choose a different (longer) path around ring

Chord: Maintenance

- Nodes may join, leave or fail at any time
- Behaviour on joining:

```
Join:
1.  Contract bootstrap node; lookup own ID to get successor node
2.  Link with neighbouring nodes; initialise own finger table
3.  Transfer ownership of keys from successor
4.  Update finger tables of existing nodes
```

  - For correctness, must ensure that at all times:
    - Each node's successor is correctly maintained
    - For every key *k*, node *successor*(*k*) is responsible for *k*
  - Desirable finger tables are correct, to improve lookup speed

Race conditions with concurrent joins can cause slow lookup, or occasional transient failure

- Behaviour on leaving:

```
Leave:
1.  Transfer ownership of keys to successor
2.  Unlink from neighbouring nodes
```

Failure - unplanned leave - handled by replicating keys

- Periodic *stabilization* algorithm runs to check successor and predecessor links and update finger tables

Chord: Discussion

- Chord works well for stable, long-lived systems, where lookup latency is not time critical:
  - Nodes close in the ring not necessarily close in the network
    - Relatively large lookup latency, even though number of hops low
  - Churn is a significant problem
    - Large peer-to-peer networks exhibit frequent joins and leaves ("churn")
    - System never reaches equilibrium given sufficient churn
    - Incorrect finger tables cause Chord to perform a linear search
    - Leads to excessive lookup times and transient failures

- Many extensions/variants developed to address these issues, at the expense of considerable extra complexity
  - Bamboo and Kademlia best developed in the Chord family

Tapestry

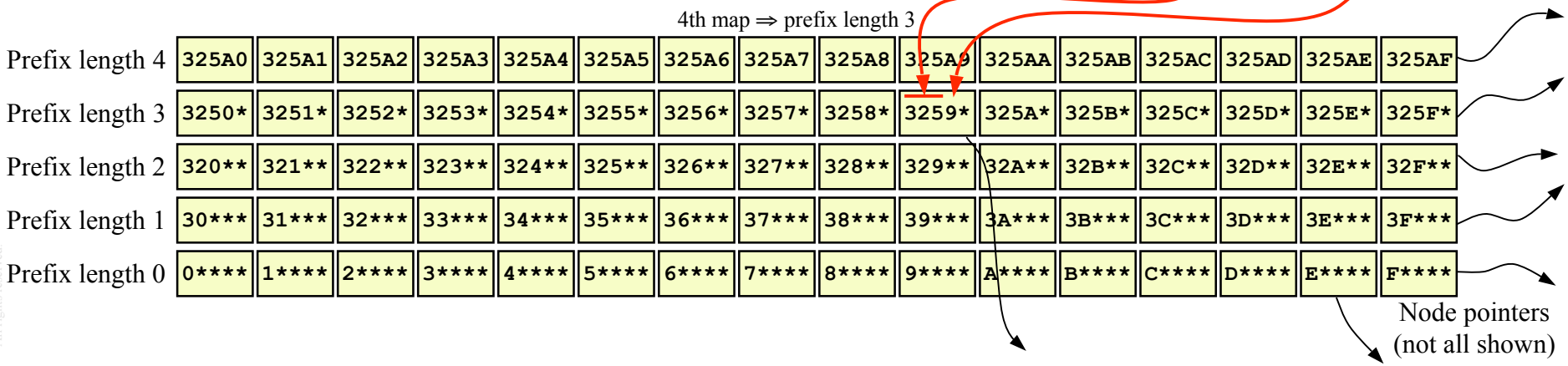
- A distributed object location and routing protocol
  - High-performance, scalable, location independent routing of message to nearby copies of an object, O_G
  - Supports multiple applications, A_{id}, running on nodes, N
  - More extensive API than Chord:

```
PublishObject(O_G, A_id)
UnpublishObject(O_G, A_id)
RouteToObject(O_G, A_id)
RouteToNode(N, A_id, Exact?)
```

- A 2nd generation peer-to-peer system
  - More complex and feature-full than Chord
  - Lower latency and less sensitive to churn

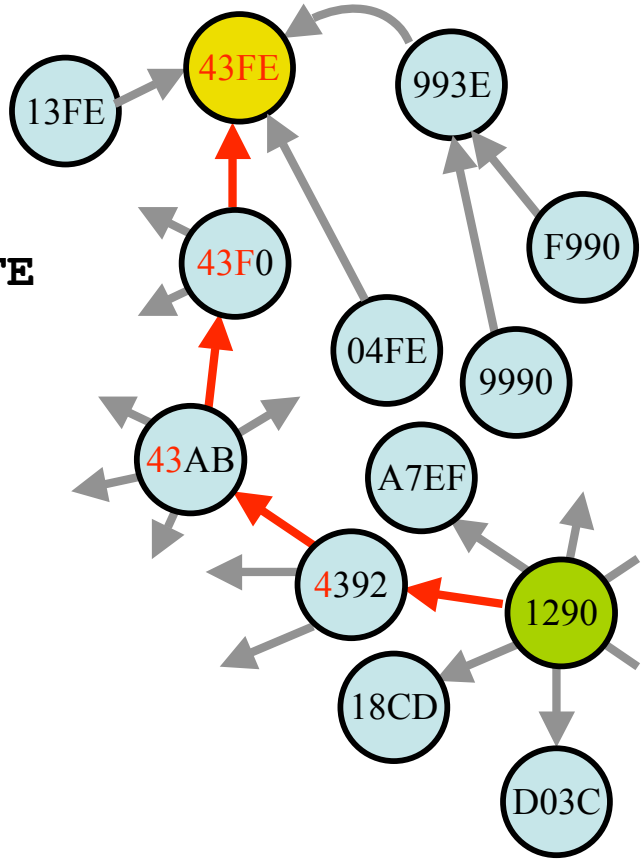
Tapestry: Basic Structure

- Nodes and objects share a flat namespace
  - 160 bit SHA-1 hash expressed as 40 digit hexadecimal identifier
    - Radix of the system, $b = 16$, a key parameter
- Nodes arranged in a highly connected mesh
  - Each node has a neighbour map for each prefix of its node identifier
    - Each map contains entries for b nodes ($\Rightarrow$ total $40 \times 16 = 640$ routing entries)
    - The ith entry in the jth map is a bidirectional link to the closest node with an identifier that begins **`prefix`**$(N, j - 1)$ + "i"
    - Example:
      - Consider nodes with 5 digit identifiers; the 9th entry in the 4th map for node 325AE is a pointer to the closest node with an identifier that begins 3259



Tapestry: Routing

- Routes to the *closest* neighbour with longest match to the desired address, digit-by-digit
  - `1290` ⇒ `4***` ⇒ `43**` ⇒ `43F*` ⇒ `43FE`
  - Can match several digits in one hop, when there is a matching neighbour
  - Reaches destination in at most $log_b N$ hops
    - 40 hops for $N = 2^{160}$ and b = 16



- Efficient topology based routing to objects
  - In addition to closest neighbour matching prefix, redundant links to further matching neighbours exist for robustness

Tapestry: Maintenance

- Nodes may join at any time:

Node *N* joins:
1. **Need-to-know nodes are notified of *N*, because *N* fills a null entry in their routing table**
   - **Uses directed multicast to find all nodes matching the common prefix of *N* and *S* (where *S* was the node previously responsible for node ID *N*)**
   - **Those nodes add *N* as a neighbour, if necessary**
2. **Node *N* might become the new object root for existing objects; need to migrate those objects to node *N***
3. **Must construct a near-optimal routing table for node *N***
   - **Nodes found in step 1 bootstrap the table**
4. **Nodes near *N* are notified, and may consider using *N* in their routing table as an optimization**

- Richly connected mesh makes leave operations simple:

Node *N* leaves voluntarily:
1. **Inform all neighbours of intent to leave, suggesting an replacement node for the neighbours to link with.**

Failures handled by redundant links (to non-closest peers)

Tapestry: Locating The Closest Neighbour

- How to find closest neighbour matching prefix?
  - Probe all possibilities, measuring RTT, to pick closest
    - Needs many probes ⇒ high overhead
    - Prohibitively expensive for large scale systems
  - Predict latency, based on virtual coordinates
    - Assume the Internet can be modelled by a geometric space
      - e.g. a two-dimensional grid (although practical systems use a more complex space)
    - Assign each node coordinates in that space
      - e.g. a position on the grid
      - Might assign coordinates based on distance to well-known landmark nodes; might be based on distance to other nodes in the peer-to-peer system measured during normal operation
      - Disseminate positions piggybacked onto other application messages
    - Calculating distance between any two nodes, whether or not direct communication has taken place, done by simple geometry

T. S. Eugene Ng and Hui Zhang, "Predicting Internet Network Distance with Coordinates-Based Approaches", IEEE Infocom 2002.

Cox *et al.*, “Practical, Distributed Network Coordinates”, ACM HotNets II, 2003.

Tapestry: Discussion

- Richly connected mesh makes Tapestry more robust than Chord
  - Requires more state at each node
  - Implementation is more complex
    - 57000 lines of Java
    - Compare to 7900 lines of C++ for Chord

- Closest neighbour selection helps to ensure Tapestry is efficient in network distance covered
  - Requires many control messages to determine distance to hosts
  - Note: Tapestry and Chord both $O(\log N)$ hops, but Tapestry finds shorter hops in general

Comparison of Chord and Tapestry

- Two very different approaches to peer-to-peer lookup
  - Provide related, but somewhat different, lookup services
  - Unstructured namespace
    - SHA-1 hash
  - Structured object lookup
    - Topology agnostic ring structure *vs.* highly connected closest neighbour mesh
    - Similar performance in terms of lookup hop count: both $O(\log N)$
    - Tapestry keeps more state, more complexity to optimise lookups in terms of network topology

- Neither is the final solution - algorithms still evolving rapidly
  - Scaling, churn, and topology awareness still issues
  - Security a major unsolved problem

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
    - OceanStore, Kademlia, etc.
  - Distributed object location
    - Skype user location
  - Etc.

  - Potential basis for future grid computing systems

OceanStore

- An example of a global file system, built using a DHT
  - Aim: support 10 billion users each with 10,000 files
- Public, untrusted, infrastructure
  - Extensive use of cryptography to ensure privacy; enforce access rules
  - Extensive use of caching and FEC for robustness and performance
- File identified by secure hash of owner's key and filename
  - Files split into blocks, returns a list of identifiers for data blocks
    - Blocks identified by cryptographic hash of contents
    - Blocked pushed somewhere into the network, located a Tapestry-like protocol
      - Uses the DHT for data storage
    - Robust: makes multiple copies for availability
  - Copy-on-write semantics for blocks; 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

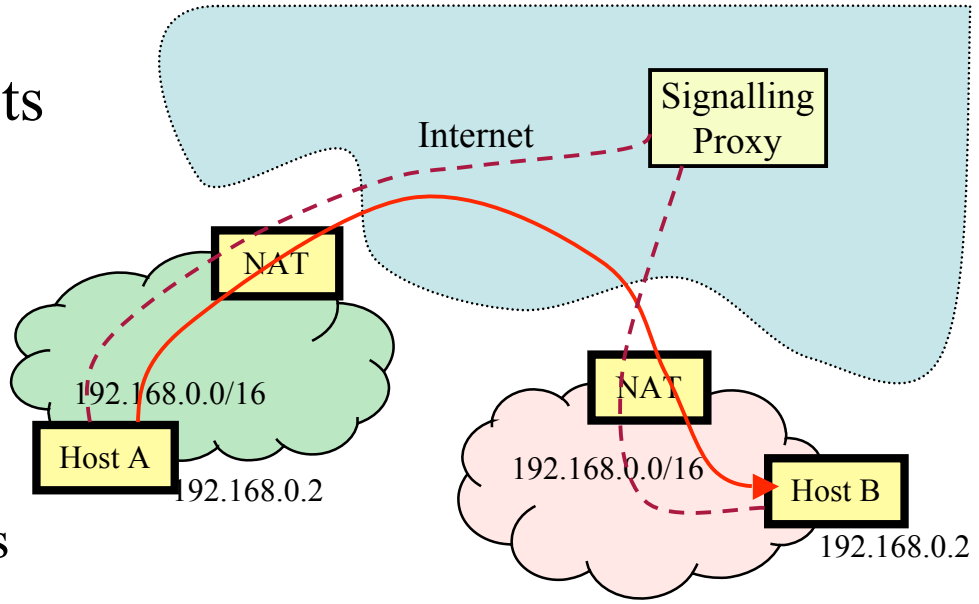
[Lots of details skipped: see the paper]

Deployment Considerations

- Peer-to-peer applications assume network provides transparent end-to-end connectivity
- Wide deployment of NAT and Firewalls breaks this transparency
  - NAT prevents inbound connections; cannot address hosts behind NAT
    - 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
  - Firewalls can prevent both in- and out-bound connections
  - Makes it difficult to deploy peer-to-peer applications
    - Sometimes intentionally, sometimes unfortunate side-effect
    - Need both political and technical fixes

Deployment Considerations: NAT

- How to enable bidirectional communication between hosts behind NAT?
  - A host outside a NAT can see the external source address of the host inside the NAT
  - Outbound communication ok
  - Can usually send to an address from which you've received
    - Sending opens a bidirectional NAT pinhole
    - Sometimes for all traffic, sometimes only for symmetric traffic
  - Talk to well known "signalling proxy"
    - Proxy learns external addresses, communicates to desired peers
    - Peers try to initiate direct flow, relay via proxy if fails



J. Rosenberg, "ICE: A Methodology for NAT Traversal for Offer/Answer Protocols", http://www.ietf.org/internet-drafts/draft-ietf-mmusic-ice-13.txt

Deployment Considerations: Firewalls

- 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 peer-to-peer application
    - E.g. how could we modify HTTP today?

- A social problem; no technical solution

Summary

- The distributed hash table abstraction
  - Concepts
  - Example protocols:
    - Chord
    - Tapestry
- Uses and motivating example system:
  - OceanStore
- Deployment considerations

Peer-to-peer protocols represent interesting design evolution, potentially useful for grid computing systems

Further Reading

1. 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 2001, San Diego, CA, USA, August 2001. http://acm.org/sigcomm/sigcomm2001/p12-stoica.pdf
2. 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. http://srhea.net/papers/tapestry_jsac.pdf
3. 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. http://oceanstore.cs.berkeley.edu/publications/papers/pdf/asplos00.pdf

Read to understand the concepts, not all the details