6.824 2006: Scalable Lookup

Prior focus has been on traditional distributed systems
  e.g. NFS, DSM/Hypervisor, Harp
  Machine room: well maintained, centrally located.
  Relatively stable population: can be known in entirety.
  Focus on performance, semantics, recovery.
  Biggest system might be Porcupine.

Now: Internet scale systems
  Machines owned by you and me: no central authority.
  Huge number of distributed machines: can't know everyone
  e.g. From e-mail to Napster.

Problems
  How do you name nodes and objects?
  How do you find other nodes in the system (efficiently)?
  How should data be split up between nodes?
  How to prevent data from being lost?  How to keep it available?
  How to provide consistency?
  How to provide security? anonymity?

What structure could be used to organize nodes?
  Central contact point: Napster
  Hierarchy: DNS for E-mail, WWW
  Flat?  Let's look at a system with a flat interface: DHT

Scalable lookup:
  Provide an abstract interface to store and find data
  Typical DHT interface:
    put(key, value)
    get(key) -> value
  loose guarantees about keeping data alive
  log(n) hops, even for new nodes
  guarantees about load balance, even for new nodes

Potential DHT applications:
  publishing: DHT keys are like links
  file system, use DHT as a sort of distributed disk drive
    keys are like block numbers
    Petal is a little bit like this
  location tracking
    keys are e.g. cell phone numbers
    a value is a phone's current location

Basic idea
  Two layers: routing (lookup) and data storage
  Routing layer handles naming and arranging nodes and then finding them.
  Storage layer handles actually putting and maintaining data.

What does a complete algorithm have to do?
  1. Define IDs, document ID to node ID assignment
  2. Define per-node routing table contents
  3. Lookup algorithm that uses routing tables
  4. Join procedure to reflect new nodes in tables
  5. Failure recovery

```
  6. Move data around when nodes join
  7. Make new replicas when nodes fail

Typical approach:
  Give each node a unique ID
  Have a rule for assigning keys to nodes based on node/key ID
    e.g. key X goes on node node with nearest ID to X
    Now how, given X, do we find that node?
  Arrange nodes in an ID-space, based on ID
    i.e. use ID as coordinates
    Build a global sense of direction
    Examples: 1D line, 2D square, Tree based on bits, hypercube, or ID
circle
  Build routing tables to allow ID-space navigation
    Each node knows about ID-space neighbors
      I.e. knows neighbors' IDs and IP addresses
    Perhaps each node knows a few farther-away nodes
      To move long distances quickly

The "Chord" peer-to-peer lookup system
  By Stoica, Morris, Karger, Kaashoek and Balakrishnan
  http://pdos.csail.mit.edu/chord/
  An example system of this type

ID-space topology
  Ring: All IDs are 160-bit numbers, viewed in a ring.
  Everyone agrees on how the ring is divided between nodes
    Just based on ID bits

Assignment of key IDs to node IDs?
  Key stored on first node whose ID is equal to or greater than key ID.
    Closeness is defined as the "clockwise distance"
  If node and key IDs are uniform, we get reasonable load balance.
  Node IDs can be assigned, chosen randomly, SHA-1 hash of IP
address...
  Key IDs can be drived from data, or chosen by user

Routing?
  Query is at some node.
  Node needs to forward the query to a node "closer" to key.
  Simplest system: either you are the "closest" or your neighbor is
closer.
    Hand-off queries in a clockwise direction until done
    Only state necessary is "successor".

    n.find_successor (k):
      if k in (n,successor]: return successor
      else: return successor.find_successor (k)

Slow but steady; how can we make this faster?
  This looks like a linked list: O(n)
  Can we make it more like a binary search?
    Need to be able to halve the distance at each step.

Finger table routing:
  Keep track of nodes exponentially further away:
    New state: succ(n + 2^i)
```

```
    Many of these entries will be the same in full system: expect O(lg
N)
    n.find_successor (k):
      if k in (n,successor]: return successor
      else:
        n' = closest_preceding node (k)
        return n'.find_successor (k)

Maybe node 8's looks like this:
  1: 14
  2: 14
  4: 14
  8: 21
  16: 32
  32: 42

There's a complete tree rooted at every node
  Starts at that node's row 0
  Threaded through other nodes' row 1, &c
  Every node acts as a root, so there's no root hotspot
    This is *better* than simply arranging the nodes in one tree

How does a new node acquire correct tables?
  General approach:
    Assume system starts out w/ correct routing tables.
    Use routing tables to help the new node find information.
    Add new node in a way that maintains correctness.
  Issues a lookup for its own key to any existing node.
    Finds new node's successor.
    Ask that node for its finger table.
  At this point the new node can forward queries correctly:
    Tweak its own finger table as necessary.

Does routing *to* us now work?
  If new node doesn't do anything,
    query will go to where it would have gone before we joined.
    I.e. to the existing node numerically closest to us.
  So, for correctness, we need to let people know that we are here.
    Each node keeps track of its current predecessor.
    When you join, tell your successor that its predecessor has
changed.
    Periodically ask your successor who its predecessor is:
      If that node is closer to you, switch to that guy.
  Is that enough?

Everyone must also continue to update their finger tables:
  Periodically lookup your n + 2^i-th key

What about concurrent joins?
  E.g. two new nodes with very close ids, might have same successor.
  e.g. 44 and 46.
  Both may find node 48... spiky tree!
    Good news: periodic stabilization takes care of this.

What about node failures?
  Assume nodes fail w/o warning. Strictly harder than graceful
departure.
```

```
  Two issues:
    Other nodes' routing tables refer to dead node.
    Dead nodes predecessor has no successor.
  If you try to route via dead node, detect timeout, treat as empty
table entry.
    I.e. route to numerically closer entry instead.
    Repair: ask any node on same row for a copy of its corresponding
entry.
      Or any node on rows below.
      All these share the right prefix.
  For missing successor
    Failed node might have been closest to key ID!
    Need to know next-closest.
    Maintain a _list_ of successors: r successors.
      If you expect really bad luck, maintain O(log N) successors.
    We can route around failure.
  The system is effectively self-correcting.

Locality
  Lookup takes log(n) messages.
    But they are to random nodes on the Internet!
    Will often be very far away.
  Can we route through nodes close to us on underlying network?
  This boils down to whether we have choices:
    If multiple correct next hops, we can try to choose closest.
  Chord doesn't allow much choice.
    Observe: Strict successor for finger not necessary.
    Sample nodes in successor list of true finger, pick closest.

What's the effect?
  Individual hops are lower latency.
  But less and less choice (lower node density) as you get close in ID
space.
  So last few hops likely to be very long.
  Thus you don't *end up* close to the initiating node.
    You just get there quicker.

How fast could proximity be?
  1 + 1/4 + 1/16 + 1/64
  Not as good as real shortest-path routing!

Any down side to locality routing?
  Harder to prove independent failure.
    Maybe no big deal, since no locality for successor lists sets.
  Easier to trick me into using malicious nodes in my tables.

What about security?
  Self-authenticating data, e.g. key = SHA1(value)
    So DHT node can't forge data
    Of course it's annoying to have immutable data...
  Can a DHT node claim that data doesn't exist?
    Yes, though perhaps you can check other replicas
  Can a host join w/ IDs chosen to sit under every replica?
    Or "join" many times, so it is most of the DHT nodes?
    How are IDs chosen?

Why not just keep complete routing tables?
```

So you can always route in one hop?
Danger in large systems: timeouts or cost of keeping tables up to date.
Accordion (NSDI '05)
trade off between keeping complete state and performance
subject to a budget

Are there any reasonable applications for DHTs?
For example, could you build a DHT-based Gnutella?

Next time: wide area storage.