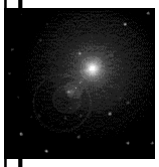


A Course on Foundations of Peer-to-Peer Systems & Applications

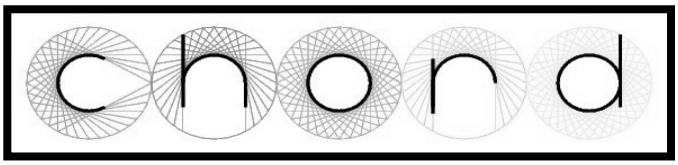
CS 6/75995
Foundation of Peer-to-Peer

Applications & Systems

Kent State University

Dept. of Computer Science
www.cs.kent.edu/~javed/class-P2P08/

Structured Systems: Chord



Background: Consistent Hashing

[class mechanics]

- [2/17/10, took 1 class hashing principles+ routing, 1 class to explain join/leave+stability+performance] [2/14/08] 0.5 classes on first day (take time to explain the hashing principles, has identifier space, consistent hashing, hash space, since this is the first class on DHT)
- [2/19/08]1 class on 2nd day [routing +node join+node departure+ performance from paper]
- [some problem in sequencing needs fixing, problem explaining short jump in last stage of example???]
- [add stability routing-added]+ used paper to explain performance on load balancing, node aggregation+hops+ stability]
- With the stability it should be two class material [added 2/17/10]

Overview

1. Chord
   1. Topology
   2. Routing
   3. Self-Organization
2. Pastry
   1. Topology
   2. Routing
   3. Self-Organization
3. CAN (Content Addressable Network)
   1. Topology
   2. Routing
   3. Self-Organization
4. Symphony
5. Viceroy
6. Kademlia
7. Summary

Chord: Overview

- Chord is one of the original distributed hash table protocols being developed at MIT (2001).
- Chord source code can be downloaded and used under the MIT License.
- Simple & elegant
  - easy to understand and implement
  - many improvements and optimizations exist
- Main responsibilities:
  - Routing
    - Flat logical address space: l-bit identifiers instead of IP addresses
    - Efficient routing in large systems: log(N) hops with N total nodes
  - Self-organization
    - Handle node arrival, departure, and failure

FOUNDATION OF PEER-TO-PEER SYSTEMS

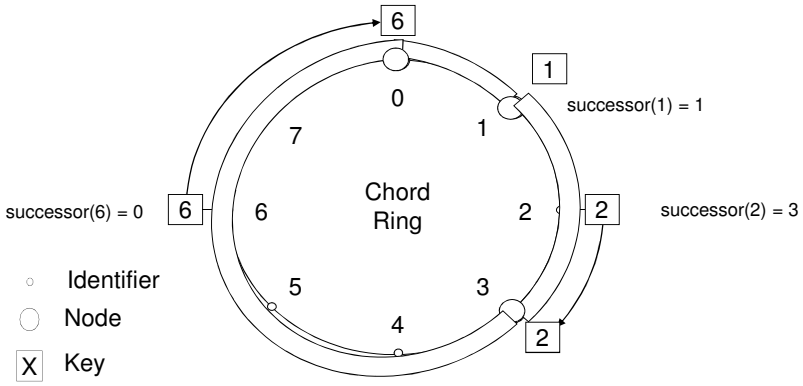
Chord: Topology

- Hash-table storage
  - Put (key, value) inserts data to Chord
  - Value = get (key) retrieves data from Chord
- Identifiers
  - Derived from hash function
    - E.g. SHA-1, 160-bit output → 0 <= identifier < 2^160
  - *Key* associated with data item
    - E.g. key = sha-1(value)
  - *ID* associated with host
    - E.g. id = sha-1 (IP address, port)

FOUNDATION OF PEER-TO-PEER SYSTEMS

Chord: Topology

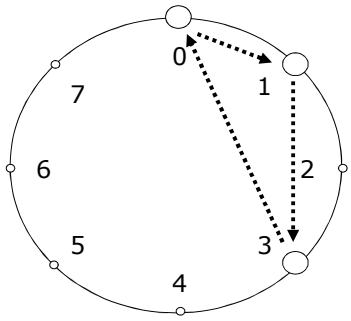
- Keys and IDs on ring, i.e., all arithmetic modulo 2^160
- (key, value) pairs managed by clockwise next node: ***successor***



FOUNDATION OF PEER-TO-PEER SYSTEMS

Chord: Topology

- The basic topology of chord is a circular linked list. Each node remembers just one piece of information about next node in clockwise direction.

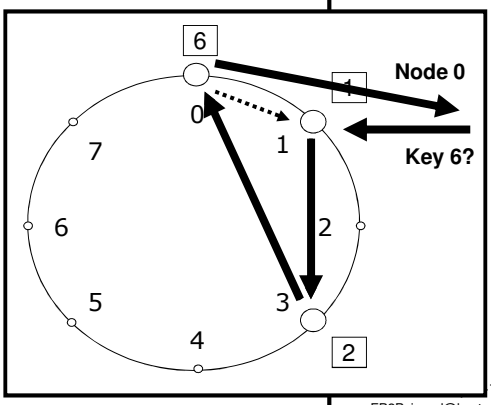


FOUNDATION OF PEER-TO-PEER SYSTEMS

Chord: Routing

FOUNDATION OF PEER-TO-PEER SYSTEMS

- Primitive routing:
  - Forward query for key x until successor(x) is found
  - Return result to source of query
- Pros:
  - Simple
  - Little node state
- Cons:
  - Poor lookup efficiency: O(1/2 * N) hops on average (with N nodes)
  - Node failure breaks circle



FP2P, javed@kent.edu
Javed I. Khan@2008

Chord: Routing

- [VVI: it does not forward query rather keeps on asking about the predessor of k to nodes closer and closer to k]
- Advanced routing:
  - Store links to z next neighbors
  - Forward queries for k to farthest known predecessor of k
  - For z = N: fully meshed routing system
    - Lookup efficiency: O(1)
    - Per-node state: O(N)
  - Still poor scalability
- Scalable routing:
  - Linear routing progress scales poorly
  - Mix of short- and long-distance links required:
    - Accurate routing in node's vicinity
    - Fast routing progress over large distances
    - Bounded number of links per node

FOUNDATION OF PEER-TO-PEER SYSTEMS

FP2P, javed@kent.edu
Javed I. Khan@2008

Corrected Routing

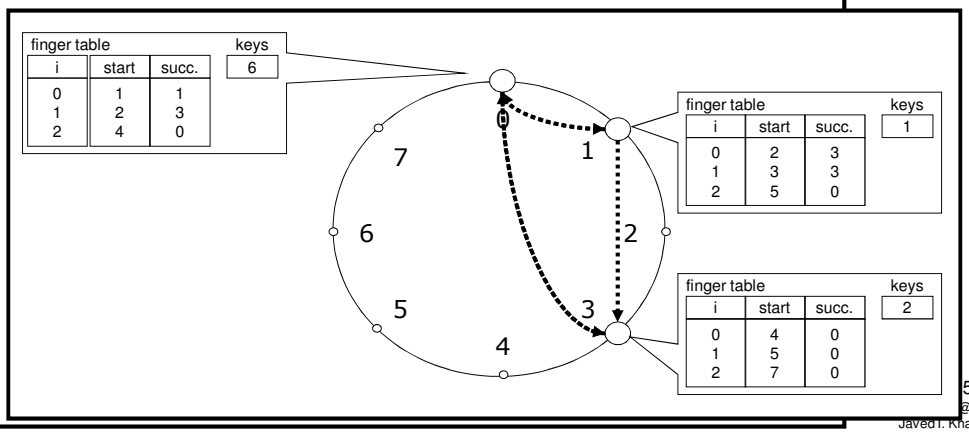
```
// ask node n to find id's successor
n.find_successor(id)
   n' = find_predecessor(id);
   return n'.successor;

// ask node n to find id's predecessor
n.find_predecessor(id)
   n' = n;
   while (id ∉ (n', n'.successor])
      n' = n'.closest_preceding_finger(id);
   return n';

// return closest finger preceding id
n.closest_preceding_finger(id)
   for i = m downto 1
      if (finger[i].node ∈ (n, id))
         return finger[i].node;
   return n;
```

FOUNDATION OF PEER-TO-PEER SYSTEMS

Chord: Routing

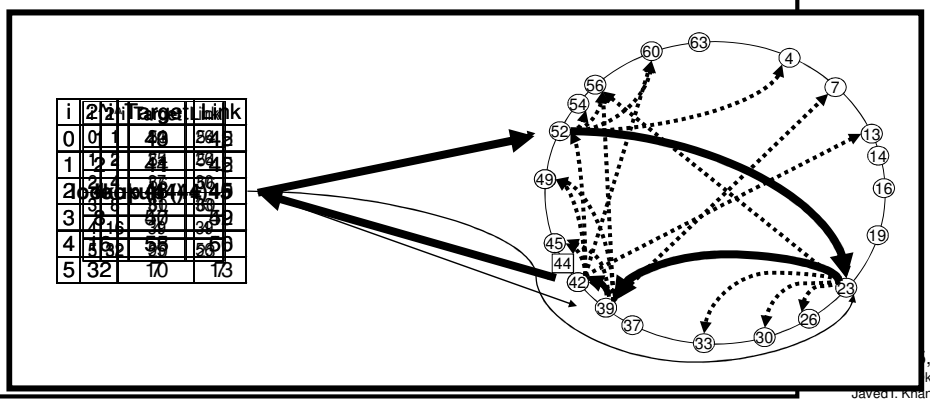
- Chord's routing table: *finger table*
  - Stores log(N) links per node
  - Covers exponentially increasing distances:
    - Node n: entry i points to successor(n + 2^i) (*i-th finger*)

FOUNDATION OF PEER-TO-PEER SYSTEMS



Chord: Routing

- Chord's routing algorithm:
  - Each node n forwards query for key k clockwise
    - To farthest finger preceding k
    - Until n = predecessor(k) and successor(n) = successor(k)
    - Return successor(n) to source of query



FOUNDATION OF PEER-TO-PEER SYSTEMS

Chord: Self-Organization

- Handle changing network environment
  - Failure of nodes
  - Network failures
  - Arrival of new nodes
  - Departure of participating nodes
- Maintain consistent system state for routing
  - Keep routing information up to date
    - Routing correctness depends on correct successor information
    - Routing efficiency depends on correct finger tables
  - Failure tolerance required for all operations

FOUNDATION OF PEER-TO-PEER SYSTEMS

Chord: Failure Tolerant Storage

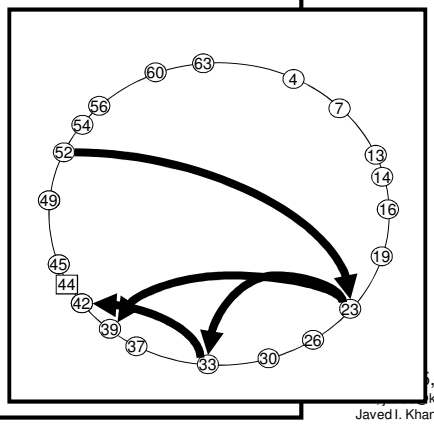
FOUNDATION OF PEER-TO-PEER SYSTEMS

- Layered design
  - Chord DHT mainly responsible for routing
  - Data storage managed by application
    - persistence
    - consistency
    - fairness
- Chord soft-state approach:
  - Nodes delete (key, value) pairs after timeout
  - Applications need to refresh (key, value) pairs periodically
  - Worst case: data unavailable for refresh interval after node failure

Chord: Failure Tolerance Routing

FOUNDATION OF PEER-TO-PEER SYSTEMS

- Finger failures during routing
  - query cannot be forwarded to finger
  - forward to previous finger (do not overshoot destination node)
  - trigger repair mechanism:
    replace finger with its successor
- Active finger maintenance
  - periodically check liveness of fingers
  - replace with correct nodes on failures
  - trade-off: maintenance traffic vs. correctness & timeliness



Chord: Failure Tolerance Routing

- Successor failure during routing
  - Last stop of routing returns failed node to source of query.
  - All queries for successor fail
  - Store m successors in a ***successor list***
    - successor[0] fails -> use successor[1] etc.
    - routing fails only if m consecutive nodes fail simultaneously
- Also add one ***predecessor*** in table.
- Active maintenance of *successor list*
  - periodic checks similar to finger table maintenance
  - crucial for correct routing

FOUNDATION OF PEER-TO-PEER SYSTEMS

finger table

i	start	succ.
0	1	1
1	2	3
2	4	0

keys: 6

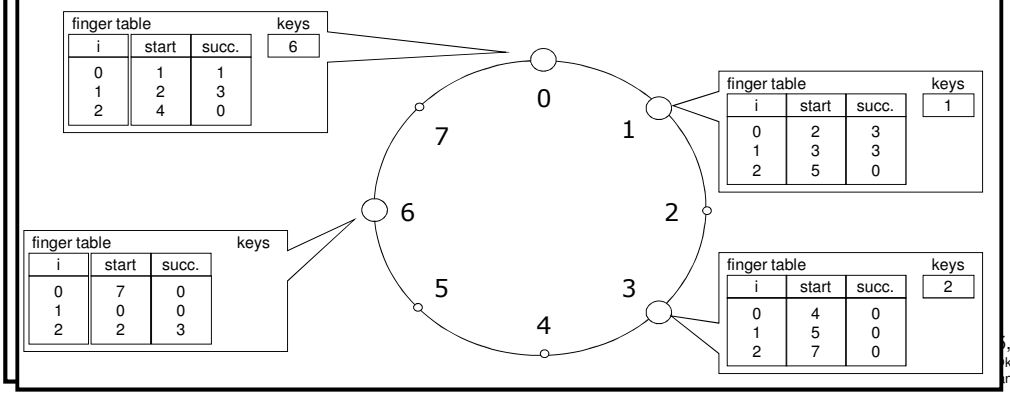
Successor List

Predecessor

Chord: Node Join

- New node n picks ID
- Contact an existing node m
- Construct finger table standard routing/lookup() [send query message via m to each start table entry and see who responds-javed]
- Retrieve (key, value) pairs from successor

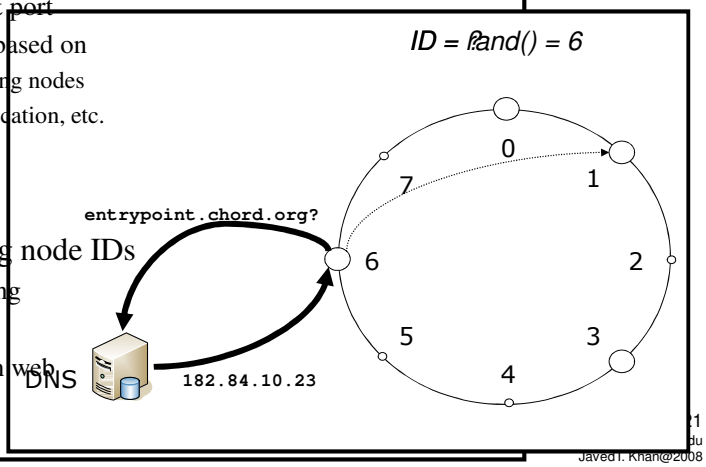
FOUNDATION OF PEER-TO-PEER SYSTEMS



Chord: Node Arrival

FOUNDATION OF PEER-TO-PEER SYSTEMS

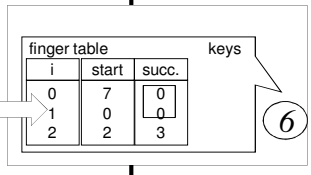
- Examples for choosing new node IDs
  - random ID: equal distribution assumed but not guaranteed
  - hash IP address & port
  - place new nodes based on
    - load on existing nodes
    - geographic location, etc.
- Retrieval of existing node IDs
  - Controlled flooding
  - DNS aliases
  - Published through web
  - etc.



Chord: Node Arrival

FOUNDATION OF PEER-TO-PEER SYSTEMS

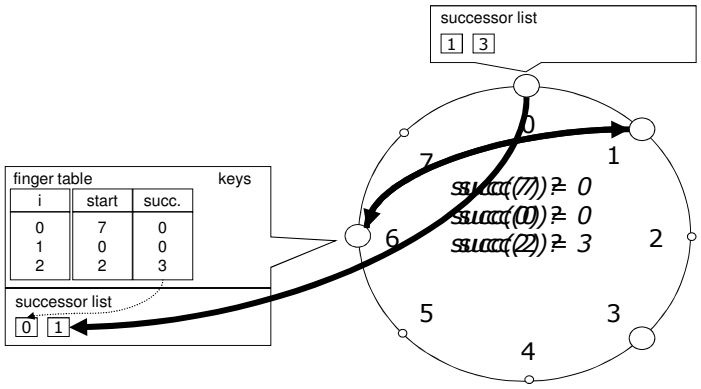
- Construction of new node's finger table
  - iterate over finger table rows
  - for each row: query entry point m for successor
  - Standard Chord routing propagated by entry point.
  - Returned results tells new node who is the successor for each finger. O(m.log N) ≈ O($\log^2$ N)
- Optimization #1:
  - Check if i-th finger is also the finger for (i+1) th finger.
  - i.e if finger[i] node >= finger [i+1].start, then the (i+1)-finger will have the same as i-th finger. So skip query.
  - Example: Finger (i=0) is 0 which is also higher that finger (i=1)'s start 0.
- Optimization#2:
  - Query immediate successor for its entire finger table. Then query to verify. Actual corrected fingers should be very close. Practically O(log N)



Chord: Node Arrival (contd..)

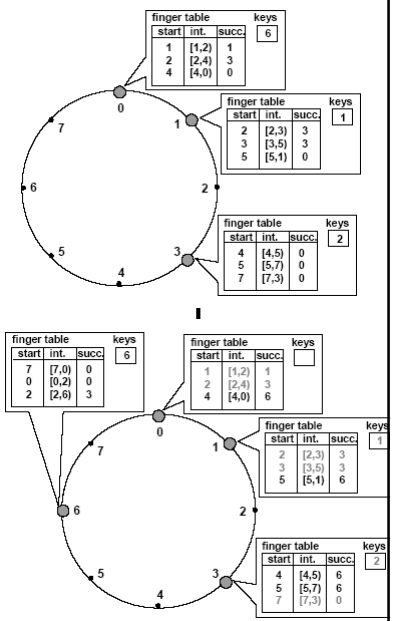
FOUNDATION OF PEER-TO-PEER SYSTEMS

- Construction of new node's successor list
  - add immediate successor from finger table
  - request successor list from successor



Chord: Node Arrival (contd.

- Reconstruction of fingers in other nodes:
- Conditions for updates:
  - Node n will be i-th finger in p if: (a) p preceds n by at least 2^{i-1}. and (b) the i-th finger of p succeeds n.
- Process:
  - Start with immediate predecessor within $n-2^{i-1}$ distance backward which is also the i-th finger.
  - So go there are correct all the required entries. These are generally low i entries.
  - Then continue walk backward (counter-clockwise) check its higher distance entries. Update the entries until it hits node p whose i-th finger does not reach n.



Chord: Stability

- If joining nodes have affected some region of the Chord ring, a lookup that occurs before stabilization has finished can exhibit one of three behaviors:
-
  - The common case is that all the finger table entries involved in the lookup are reasonably current, and the lookup finds the correct successor s.
  - The second case is where successor pointers are correct, but fingers are inaccurate. This yields correct lookups, but they may be slower.
  - In the final case, the nodes in the affected region have incorrect successor pointers, or keys may not yet have migrated to newly joined nodes, and the lookup may fail.

Chord uses a period stabilization algorithm to correct the network.

FOUNDATION OF PEER-TO-PEER SYSTEMS

Chord: Stability

- When node n first starts, it send a Join() message to any known Chord node m:
  - Join() asks m to find the immediate successor of n. n adds this to its successor table.
  - By itself, Join() does not make the rest of the network aware of n in the network.
- Rather, every node runs Stabilize () periodically:
  - When node n runs stabilize, it asks it's successor for the successor's predecessor p.
  - Upon receiving p, n decides if p should be n's successor instead.
  - If that's the case then n also sends message to p of n's existence, giving the successor a chance to change its predecessor to n.
  - The successor p updates its predecessor only if it knows of no closer predecessor that n.

FOUNDATION OF PEER-TO-PEER SYSTEMS

Chord: Stability

- The stabilization scheme guarantees to add nodes to a Chord ring in a way that preserves reachability of existing nodes, even in the face of concurrent joins and lost and reordered messages.
- THEOREM 6. *If we take a stable network with n nodes, and another set of up to nodes joins the network with no finger pointers (but with correct successor pointers), then lookups will still take O(log N) time with high probability.*
- Stabilization by itself won't correct a Chord system that has split into multiple disjoint cycles, or a single cycle that loops multiple times around the identifier space. These pathological cases cannot be produced by any sequence of ordinary node joins.
- It is unclear whether they can be produced by network partitions and recoveries or intermittent failures. If produced, these cases could be detected and repaired by periodic sampling of the ring topology.

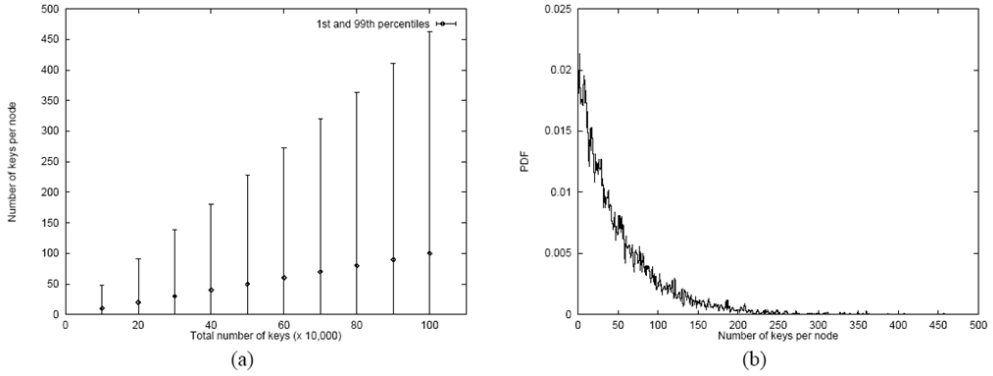
Chord: Node Departure

- Deliberate node departure
  - clean shutdown instead of failure
- For simplicity: treat as failure
  - system already failure tolerant
  - soft state: automatic state restoration
  - state is lost briefly
  - invalid finger table entries: reduced routing efficiency
- For efficiency: handle explicitly
  - notification by departing node to
    - successor, predecessor, nodes at finger distances
  - copy (key, value) pairs before shutdown

FOUNDATION OF PEER-TO-PEER SYSTEMS

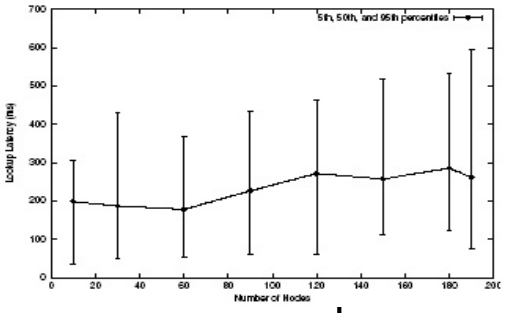
Performance: Load Distribution

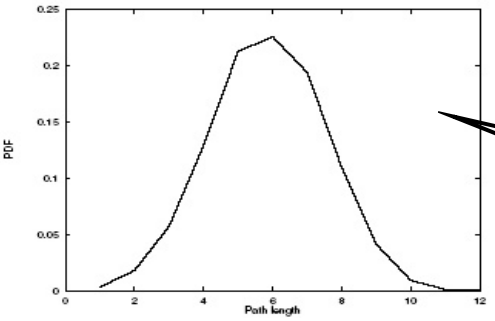
FOUNDATION OF PEER-TO-PEER SYSTEMS



Performance: Average Path Length

Moderate impact of number of nodes on lookup latency



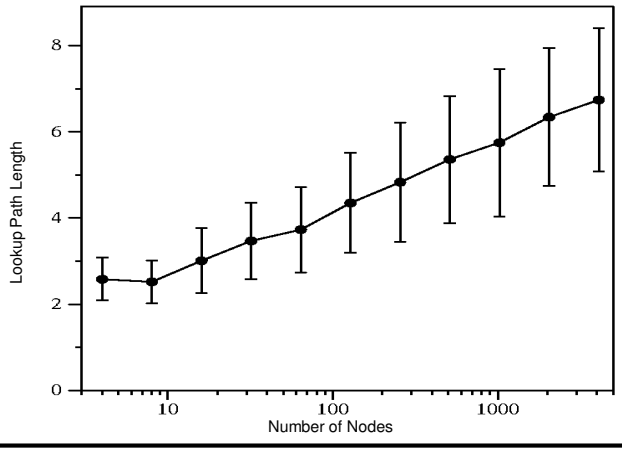


Consistent average path length

```
n.join(n')
    predecessor = nil;
    successor = n'.find_successor(n);

// periodically verify n's immediate successor,
// and tell 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';

// periodically refresh finger table entries.
n.fix_fingers()
    i = random index > 1 into finger[];
    finger[i].node = find_successor(finger[i].start);
```

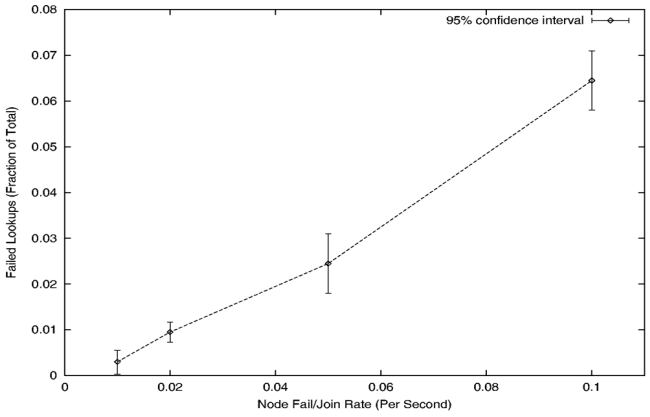
Performance: Lookup Path Length

- Lookup latency (number of hops/messages): average, 1st and 9th percentile average is around ~ 1/2 $\log_2(N)$
- Confirms theoretical estimation



Performance: Node Join/Leave vs. Failed Lookup

- 104 nodes storing 106 keys, and fraction p of randomly selected node fails/joins using Poissons distribution. Each 30 second a stabilization is runs. The network starts with 500 nodes.



Chord: Summary

- Complexity
  - Messages per lookup: O(log N)
  - Memory per node: O(log N)
  - Messages per management action (join/leave/fail): O(log² N)
- Advantages
  - Theoretical models and proofs about complexity
  - Simple & flexible
- Disadvantages
  - No notion of node proximity and proximity-based routing optimizations
  - Chord rings may become disjoint in realistic settings
- Many improvements published
  - e.g. proximity, bi-directional links, load balancing, etc.

Next Class:
Pastry & CAN