Advanced Topics in Computer Systems, CS262B
Prof Eric A. Brewer

Chord and Pastry

March 18, 2004

I. Background

Consistent hashing (Karger, Leighton, Lewin, et al.)

- Want to hash key *x* onto a group of *k* servers
  - with even spread
  - and such that you can change k and only move O(1/k) of the data (optimal)
- One solution: hash x into a uniform one-dimensional space
  - map k buckets into the same space
  - nearest bucket is the owner of the key (=> even spread)
  - new buckets only move nearby items (and vice versa for deleted buckets)
  - load can vary by O(log n)
  - takes O(log n) to find the nearest bucket using binary search

II. Chord

Goals:

- key -> responsible node mapping (under changing set of nodes)
- load balance
- decentralized
- scalable
- available
- flexible naming (use your own hash function into the 1D space)

Uses consistent hashing onto a circular space (e.g. 128-bit integers)

- owner node is the first one clockwise from the hash value of the key
- how many bits? (enough to probabilistically avoid collisions)
- SHA-1 is the hash function
  - key idea: make it hard for an attacker to *cause* collision or uneven load
- virtual nodes are just "over sampling" to reduce the variance of the load

Don't want to have to know all of the bucket locations:

- keep track of nearby buckets plus O(log n) fingers (chords) to distant buckets (like a tree)

- new nodes pick a random location, then take over that part of the space
- keys pick a random location and put the data there (users of data must agree on the key)

Scalable key location:

- worst case: just go around the circle from node to node = O(n) lookup
- add log n fingers to nodes at rough distance 2^i (for the log n values of i)
- => O(log n) storage for fingers, O(log n) messages to reach a given key

Adding a node:

- three steps:
  - initialize fingers and predecessor link for new node
  - update fingers/pred that should now point to this node
  - move some data from neighbors
- to get the new fingers: you can do log n searches => $O(\log^2 n)$ overall
  - easier: just copy your neighbors and check it, many entries will be the same
- to update others fingers is harder
  - do an O(log n) search for class of finger to find the first node that could be the i^{th} finger that points to you. Then check it and walk backward to check its predecessors
  - this is $O(\log^2 n)$
  - but in practice may not need to update the fingers that point to you, better to do it lazily

Stabilization:

- idea: lazily update fingers, to simplify concurrent operation. Eventually consistent
- stale fingers cause extra hops, stale successor pointer could cause failures that should work if retried later
- theorem: as long as consistency is reached in less time than it takes to double the network, then lookups are still O(log n) (becuase on average you are only adding about 1 node to each existing interval, which adds 1 hop on average)

Fault tolerance:

- replication: can replicate at successsor node (or at some fixed distance, or rehash)
- keep list of r nearest successors, so you can easily skip over failed nodes
- pick r such that you probabilistic expect at least one of your r sucessors to be alive

Issues:

- partitions?
- malicious attacks (sybil attack?)
- what base for log?
- can you have constant fingers and log n hops? or log n fingers and constant hops?
- locality?

III. Pastry

Similar goals + locality

Based on radix-r search: each step (usually) makes progress on digit, thus log N steps (base r)

Basic routing:

- o Assume k digits in base r
- o k rows, r columns
- o Each row matches on prefix for the higher rows
  - i.e. row 0 has no matches, row 3 matches on the first 3 digit
  - the r columns are the r choices for that row, with one being n/a since it matches this node's id
- o At lower rows (longer prefixes), their may be empty slots
- o Leaf set is a set of nearby nodes (numerically) which you jump to when you get close
- o Leaf set makes up for the empty slots near the bottom of the table
  - we may be the nearest node if the slot is empty
  - probability of empty slot, but not covered by leaf set depends on the size of the leaf set, but varies from 2% to 0.6%

Node arrival of node X (join):

- o get an ID (such as a hash value)
- o start with a physically nearby node, A
- o join starting at A using the routing algorithm until you get to node Z (the nearest node for that ID)
- o use Z's leaf set to init X
- o use A's neighborhood set (since it is physically close)
- o simple routing table:
  - get the ith row from the i^{th} node on the path to Z
  - slightly more accurate: copy a row from a node if it is a better version of the row you have; this works if you have more than log n steps, or less than log n steps
- o send resulting table to path nodes, to fill in their holes
- o improvement: looks at the tables of nodes referenced in others' tables

Repairs:

- o leaf set: ask other leafs for more options, verify via contact, and add
- o routing table: route around at first, then lazily update
  - ask other nodes from that row about their entries
  - else, try the row below (which also qualify)
- o neighborhood:
  - first, perioically check liveness
  - ask other neighbors for their sets, and check those distances

Locality:

- key idea: early rows contain close nodes, lower nodes are spread far apart
- since with a short prefix there are *many* possible choices, we can choose some that are close
- leaf set nodes are NOT close (spread uniformly over whole internet)
- proof by induction: assume that we have locality and show that we keep it as we add nodes
  - X's row 0 = A's row 0, which is close by the transitive property
  - B's is closer to A than it is to is row 1 partners (since there are less of them!), and therefore its row 1 is a good choice for X as well
- need second stage of join: X looks at entries from all of the nodes in its routing table and their neighborhood sets (the WTF optimization)

Can also find one of the nearest k nodes