Peer-to-Peer (P2P) Systems

Chapter 7:

Case study: Kademlia

Outline

§ 1. XOR Topology

§ 2. Routing algorithm with multiple nodes

General

- Authors:
  - Petar Maymounkov and David Mazieres
- Design focuses on:
  - Minimization of control traffic nodes send to learn about each other
  - Least-recently seen eviction policy but preserving alive nodes
  - Constrained multicast (parallel, asynchronous queries) to avoid timeout delays from failed nodes
- BitTorrent applies Kademlia

§1. XOR Topology

- IDs and keys are 160-bit integers
- XOR distance metric

$d(x,y) = x \text{ XOR } y$

1. Positive definiteness
   $d(x,x) = 0$ and if $x<>y$, $d(x,y) > 0$
2. Symmetry
   $d(x,y) = d(y,x)$
3. Triangle inequality
   $d(x,z) + d(z,y) >= d(x,y)$

Comparison

ID space and distance metric in popular greedy DHTs:

DHT design	ID space S	Distance metric $\rho(u,k)$
Chord [11]	n-bit numbers uniformly projected to the ring ($b = 2$)	$\rho(u,k) = \begin{cases} k-u, & \text{if } u \le k, \\ 2^n - (u-k), & \text{otherwise.} \end{cases}$ Clockwise arc length between u and k
Tapestry [14]	n-digit numbers in base $b \ge 2$.	$\rho(u,k) = \sum_{j=0}^{n-1} \lvert u_j - k_j \rvert b^j$ where $u = \sum_{j=0}^{n-1} u_j b^j$ and $k = \sum_{j=0}^{n-1} k_j b^j$. Longer the prefix (in digits) shared among u and k, closer these IDs
Pastry [13]	Similarly to tapestry with $b = 2^c$ for $c \ge 1$.	The same as in tapestry.
Kademlia [12]	Similarly to tapestry with $b = 2$.	The same as in tapestry. Equivalent to the bitwise exclusive OR (XOR), $\rho(u,k) = \sum_{j=0}^{n-1} (u_j \oplus k_j) b^j$.

Routing state: *k*-buckets

- For each i in [0…160)
  - Node keeps a list (***k*-bucket**) of k neighbors of distance between 2^i and 2^{i+1} from itself
  - k is upper bound for the bucket size (k=20)
- Each k-bucket is sorted by time last seen
  - Least-recently seen at the head
  - Most-recently seen at the tail
- For small i, the k-buckets are empty or small

Maintenance (1)

When a node receives any message, it updates the appropriate *k*-bucket

- If the sending node already exists in *k*-bucket, it is moved to the tail
- If the sending node is not already in *k*-bucket,
  - If |*k*-bucket|<*k*, then it is inserted to the tail
  - Otherwise, ping the least-recently seen node
    - No response: remove it and insert the new one
    - Response: move it to the tail, and the new one is not inserted

Maintenance (2)

- Small amount of control traffic
  - Pings only
- Least-recently seen eviction policy, except that alive nodes are never removed
- The preference for old contacts are from analysis of Gnutella traces
- Resistance to DoS attacks
  - One cannot corrupt routing states of nodes by flooding the system with new (zombie) nodes
  - New nodes appear in routing states when old nodes have leaved the system

§2. Routing algorithm with multiple nodes

Progressive routing (cf. Chord)

- If *u* keeps a neighbor up to distance 2^i
  - for any key s.t. $2^{i-1}<\rho(u,\text{key})\le 2^i$
    $\rho(v,\text{key}) < 2^i-2^{i-1}=2^{i-1}$ =>
    $\rho(u,\text{key}) / \rho(v,\text{key}) > 2^{i-1} / 2^{i-1} = 1$
- If *u* keeps several neighbors for 2^i
  - high probability of existence of *v* s.t. $\rho(v,\text{key}) < 2^{i-2}$
  - $\rho(u,\text{key}) / \rho(v,\text{key}) > 2^{i-1} / 2^{i-2} = 2$

Multiple nodes

In routing a lookup:

- *Select n* closest nodes to the key
- Multicast to *n* nodes at every hop,
- Receive back *kn* nodes

Constrained multicast

- Lookup(key) is implemented as RPC (remote procedure call)
- Recipient must return *k* nodes
  - Either from a single *k*-bucket,
  - or from multiple *k*-buckets if the closest *k*-bucket is not full enough
  - Unless there are fewer than *k* nodes in all recipient's *k*-buckets combined

Routing steps

Let *n* be a system-wide concurrency parameter

- Initial (lookup source node)
  - Initiator starts by picking *n* nodes from its closest non-empty *k*-bucket
  - The initiator sends parallel, asynchronous FindNode RPCs to these *n* nodes
- Forwarding (iterative routing)
  - Initiator resends FindNode RPCs to *n* nodes it has learned about from previous RPCs
    - This can begin before all *n* of the previous RPCs have returned
  - Select *n* nodes from *nk* returned ones that are closest to the target and have not yet queried
- Nodes that fail to respond quickly are eventually removed

Termination

- When a round of resending fails to return a node that is closer than the closest already seen
- As result, initiator has found the *n* closest nodes to the given key ID

Chord-like routing

- For n=1 Kademlia routing is similar to Chord routing
- Chord extensions also use k>1 neighbors for each distance range 2^i
- Kademlia distance is symmetric in contrast to asymmetric Chord

Data publication and replication

- To publish data,
  - Node finds *n* closest nodes to the data key
  - All these *n* nodes are used to store the data item
    - replication among the closest nodes!