Structured networks: search

Márk Jelasity
Outline

- Hash tables and distributed hash tables (DHT): the abstraction
- An example implementation: Chord
- Implementing keyword search on a DHT
- Some other other DHTs: Pastry and CAN
- Summary of DHT complexity results
- Hybrid (structured/unstructured) approaches to search
Motivation

- We have seen search does well in unstructured networks except when items are rare
- Can we come up with a technique that provides efficient search (lookup) for rare items?
  - Yes: distributed hash tables (DHT)
- What is the ultimate solution that is robust, cheap and works for popular and rare items too?
  - Hybrid solutions?
  - Something not yet invented?
- DHTs are good for other things too
Hash tables

- Store arbitrary keys and satellite data (value)
  - put(key, value)
  - value = get(key)
- Lookup must be fast
  - Calculate hash function h() on key that returns a storage cell
  - Chained hash table: Store key (and optional value) there

Allocated array: indexed by hash values

<table>
<thead>
<tr>
<th>1 = h(k₁) = h(k₂)</th>
<th>k₁</th>
<th>v₁</th>
<th>k₂</th>
<th>v₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 = h(k₃) = h(k₄)</td>
<td>k₃</td>
<td>v₃</td>
<td>k₄</td>
<td>v₄</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 = h(k₅)</td>
<td>k₅</td>
<td>v₅</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Why a hash table?

- Most often the point of a hash table is **fast and cheap** lookup of data indexed by a key.
- When used for search, the issue of query richness comes up:
  - In random walk/flooding, a query can be arbitrarily complex (even full text search with regular expressions).
  - If we use only key based lookup, we must be creative and work more to allow for non-trivial queries.
    - **Inverse indexing, etc**
- The idea is trading some flexibility and simplicity off for efficiency and effectivity.
Distributed hash table

• We want hash table functionality in a p2p network: lookup of data indexed by keys

• Assume the storage space is a distributed set of nodes (not an array)
  – Note that in all cases we will have an overlay network that connects these nodes in tricky ways
  – The exact set of nodes is not known locally and can change all the time
  – We work with an idealized storage space,
    • Hash function maps to this ideal space
    • We assign parts of the space to nodes in a distributed way dynamically: extra complications
Distributed hash tables

Abstract “allocated array” called ID space, indexed by hash values

<table>
<thead>
<tr>
<th>1 = h(k₁) = h(k₂)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
<tr>
<td>3 = h(k₃)</td>
</tr>
<tr>
<td>4 = h(k₄)</td>
</tr>
<tr>
<td>5 = h(k₅)</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

Actual nodes in the network (dynamic)

<table>
<thead>
<tr>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>k₁ \ v₁</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>k₃ \ v₃</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>k₅ \ v₅</td>
</tr>
</tbody>
</table>

Stored entries

consistent hashing of keys to nodes typically two step, as shown above
Distributed has tables: main functions

- **Key-hash ↔ node mapping**
  - Assign a unique live node to a key
  - Find this node in the overlay network quickly and cheaply (routing)

- **Maintenance, optimizations**
  - Implement DHT API on top of routing
  - Load balancing: maybe even change the key-hash ↔ node mapping on the fly
  - Replicate entries on more nodes to increase robustness
  - etc
Chord

• Most cited DHT implementation (3000+ citations to date!!!)

• Advantages
  – Simple
  – Good storage and message complexity

• Consistent hashing based on an ordered ring overlay
  – This is why it is “structured”
Hashing in the Chord ring

- Identifier circle
  - 10 nodes
  - 5 keys

- Both keys and nodes are hashed to 160 bit IDs (SHA-1)

- Then keys are assigned to nodes using consistent hashing
  - Successor in ID space
Chord hashing properties

- **Consistent hashing**
  - randomized
    - All nodes receive roughly equal share of load
  - Local
    - Adding or removing a node involves an \( O(1/N) \) fraction of the keys getting new locations

- **Actual lookup**
  - Chord needs to know only \( O(\log N) \) nodes in addition to successor and predecessor to achieve \( O(\log N) \) message complexity for lookup
A primitive lookup algorithm

// ask node n to find the successor of id
n.find_successor(id)
    if (id ∈ (n, successor])
        return successor;
    else
        // forward the query
        // around the circle
        return successor.find_successor(id);
A scalable lookup algorithm
A scalable lookup algorithm

// ask node n to find the successor of id
n.find_successor(id)

n' = find_predecessor(id);
return n'.successor;

// ask node n to find the predecessor of id
n.find_predecessor(id)

n' = n;
while (id $\notin$ (n', n'.successor])
    n' = n'.closest_preceeding_finger(id);
return n'

• Jump to the closest preceeding finger
• $O(\log N)$ jumps
• $O(\log N)$ neighbors stored at each node
• This formulation assumes one node coordinates the lookup (not recursive) but could be
Join: an expensive approach

- A new node has to
  - Fill its own successor, predecessor and fingers
  - Notify other nodes for which it can be a successor, predecessor of finger
- With several optimizations this can be done on $O(\log N)$ time
- But the resulting protocol is complex
- Can be done simpler, using a relaxed and simple stabilization protocol, used also for error correction
Join: a relaxed approach

- If the ring is correct, then routing is correct, fingers are needed for the speed only

- Stabilization
  - Each node periodically runs the stabilization routine
  - Each node refreshes all fingers by periodically calling find_successor(n+2^{i-1}) for a random i
  - Periodic cost is O(logN) per node due to finger refresh

```
n.stabilize()
  x = successor.predecessor;
  if (x ∈ (n, successor))
    successor = x;
    successor.notify(n);

  n.join(n')
    predecessor = nil;
    successor = n'.find_successor(n);
```
Join: a relaxed approach

- Node join: find successor and then stabilize
  - Ring is immediately joined: routing works
  - Routing also fast enough if not too many nodes join concurrently, but eventually fingers will be ok too
Failure and replication

- Failed nodes are handled by
  - Replication: instead of one successor, we keep r successors
    - More robust to node failure (we can find our new successor if the old one failed)
  - Alternate paths while routing
    - If a finger does not respond, take the previous finger, or the replicas, if close enough

- At the DHT level, we can replicate keys on the r successor nodes
  - The stored data becomes equally more robust
Virtual nodes

- A physical node acts as if it was many nodes
  - The Chord network appears to be larger
  - One physical node gets a much more balanced number of keys
  - Maintenance cost grows
  - Path length does not grow significantly
Path length in simulations

![Graph showing path length vs number of nodes with error bars representing 1st and 99th percentiles.](image)
Conclusions

- The DHT abstraction can be implemented in a fairly simple and efficient way
- All implementations are based on a distributed datastructure, a so called “structured overlay”
  - Chord used an ordered ring, with fingers (shortcuts)
- Some remaining issues to consider
  - Can more complex and more flexible applications be implemented such as keyword search (yes)
  - Can the storage or message complexity improved (yes)
  - So, what is the best way to implement a file sharing system?
Keyword search in DHTs

- DHTs support only key lookup by default
- We need to perform complex queries as in unstructured networks
- We need to be creative: here we discuss an inverted index-based approach
  - Document identifiers are stored in a DHT with all contained keywords as keys
  - All keywords are looked up and the intersection of matches is calculated
  - A few techniques to optimize the cost of all this
Inverted index approach

- Inverted index usual in search engines
  - For all keywords collect the documents that contain that keyword
  - Create intersection, union, etc, base on keyword based query
- Do that P2P style
Distributing the inverted indices

Mainly centralized services
cheap update, expensive lookup

Better if update is rare but
communication is expensive
DHT for storing documents sets

- A DHT is used to map keywords to nodes
  - A node is assigned a set of keywords, and stores sets of pointers to documents that contain the given keyword

- The retrieval procedure needs to AND sets
  - Naive procedure shown

  - Set $A$ on server $s_A$ contains documents that have keyword $k_A$

Request is “$k_A \& k_B$"
Optimizations: Bloom filters

- Bloom filter of $A$ is sent to $s_B$ (2)
- $s_A$ removes false positives ("6" in this example)
- It saves bandwidth if set is large enough
  - We use filters for more than 300 elements only
- Smaller set should be visited first (natural thing)
- Works for more keywords too
  - All servers need to see the final result to remove false positives
Optimizations: Caches

- Bloom filters or unencoded keyword match sets can be cached
  - Some measurements indicate there are very popular keywords (power law distr) so hit rate can be good

- Utilization of caches
  - A server checks if it has cached info on a next keyword to be intersected
  - If yes, performs intersection locally, skips the corresponding server
Optimizations: virtual nodes

- Same idea as in Chord
- Assign virtual nodes proportional to capacity
  - Number of keywords proportional to capacity
  - Variance due to random hashing is reduced (as in Chord)
- Load balancing still a problem
  - Keyword popularity is not equal
    - Number of keywords is not a good measure, popularity needs to be considered too
Experiments

- **Network types**
  - All backbone, all modem, and gnutella trace

- **Search trace:**
  IRCache log file

- **Parameters**
  - Bloom filter threshold 300, Bloom filter size: 18/24 with cache on/off
Other DHT designs

- A DHT is an abstraction
  - Eg previous keyword search technique used a generic DHT
- A DHT has many popular implementations, we review two briefly: CAN and Pastry
- Different implementations have different tradeoffs and complexity properties, we review these
Content addressable network (CAN)

- CAN became the name of a specific algorithm, although it is in fact a synonym to DHT
- Logical space to which keys are mapped by a hash function
  - D-dimensional real space $[0,1]^d$
- All nodes are assigned a partition of this space
  - At any point in time the set of current nodes cover the space
- Compare with Chord!
  - Logical space is different; partitioning of this space is implicit (but nevertheless well defined)
CAN logical space

node B’s virtual coordinate zone
Routing and node join

• Greedy routing to neighbor that is closest to destination
  - Hop count is $O(dN^{1/d})$
  - Number of neighbors is $O(d)$
  - If $d=O(\log N)$, then roughly same as Chord

• Join
  - Create random point in virtual space
  - Find the node that is responsible for that point
  - Split the block of that node and update neighbors appropriately
Node join in CAN

1’s coordinate neighbor set = \{2, 3, 4, 5\}
7’s coordinate neighbor set = \{\}

1’s coordinate neighbor set = \{2, 3, 4, 7\}
7’s coordinate neighbor set = \{1, 2, 4, 5\}
Node departure and recovery

- Failure detection through missing heartbeat
- Neighbors of failed node independently try to take over the zone of the failed node
- The winning node merges the failed zone if possible, or simply holds it if not possible
- Background repair mechanism reassigns zones to prevent fractioning
- Perhaps this is the weakest point of CAN
  - Possibility for inconsistency, complex repair and failure handling procedure
Optimizations

• Increasing d
  – Shorter path length, more fault tolerance (more paths) but more neighbors

• More realities
  – Maintain many virtual spaces (CANs) in parallel
  – Replicate stored data on all realities
  – Improves path lengths (jumps inside a node) and fault tolerance (replication, more paths)

• Uniform partitioning: more balanced zone sizes
  – When joining, the selected random node replaces itself with the neighbor with the largest zone
Optimizations

- Improved routing taking proximity into account
  - When selecting a neighbor, use network latency also

- Overloading zones: more nodes in the same zone
  - When joining, zones are not split, only if enough nodes are in the zone
  - Reduces path length (fewer zones)
  - Reduces latency (possibility to select neighbor that has smallest latency)
  - Improved fault tolerance due to redundancy
Pastry: another DHT

- Applies a sorted ring in ID space like Chord
- Virtual space: same as Chord
  - We interpret IDs as sequences of digits with base $2^b$
- Applies Finger-like shortcuts to speed up routing
- The node that is responsible for a key is the numerically closest (not the successor)
  - Pastry is bidirectional and uses numeric distance
Pastry routing

- If destination is among the leafs, stop
- Otherwise Pastry either forwards the message to a node which
  - has a longer common prefix with the destination or
  - has an equally long prefix but is numerically closer
- Routing is successful if no L/2 consecutive nodes fail (ring is intact)
Pastry maintenance

• **Join**
  - Use routing to find numerically closest node already in network
  - Ask state from all nodes on the route and initialize own state

• **Error correction**
  - Failed leaf node: contact a leaf node on the side of the failed node and add appropriate new neighbor
  - Failed table entry: contact a live entry with same prefix as failed entry until new live entry found, if none found, keep trying with longer prefix table entries
Proximity in Pastry

- All routing table entries are drawn from rather large sets (unlike with Chord)
  - Pastry puts emphasis on optimizing the actual entry based on proximity
  - Entries can be selected based on other criteria as well (semantic proximity, capacity, etc)

- The shorter the common prefix, the larger the set of potential entries (exponentially)

- Original Pastry approach for actually implementing the proximity bias can be improved (not discussed here)
Are Pastry and Chord a different protocol?

- Chord and Pastry are variations of the same idea and can be transformed into each other smoothly
- What is not different
  - Basic idea: ring + shortcuts to exponentially increasing distance
  - Leaf set/successor list: Chord also uses $r$ successors/predecessors
  - Chord can also use more fingers to achieve the same hop count and model a b letter alphabet ID space
  - Same lazy repair protocol for leafs/successors
Are Pastry and Chord a different protocol?

• What is different?
  – A Chord finger is a unique node, whereas with Pastry a routing table entry can come from a large set
    • Chord could define fingers more loosely, but that needs a different update protocol for fingers
  – Chord routing is unidirectional, Pastry is direction independent
    • Chord could easily be bidirectional too with fingers into two directions
A final note on complexity

- Chord and Pastry have $O(\log N)$ storage and hop count complexity
- CAN have $O(dN^{1/d})$ hop count complexity and $O(d)$ storage
- It is possible to have $O(1)$ storage complexity with $O(\log N)$ hop count (Viceroy) or with $O(\log^2 N)$ hop count (Symphony)
  - Sounds good but more complex protocols, less reliability and $\log N$ is small enough: is it worth it?
So, how to implement filesharing?

- Get the best of both worlds: hybrid approaches
- Use DHT for rare items, random walk for popular items
- What about the topology of the overlay network?
  - Unstructured networks are easy to build and maintain, and robust to churn
  - Are DHT-s really more complicated or expensive or less robust? Not necessarily
- We overview two hybrid approaches along the lines above
Gnutella: observing the long tail

• Gnutella (latest version with ultrapeers and dynamic query) is excellent for locating popular items (reliable, fast)

• Gnutella is not so good at locating rare items
  – 41% of queries receive <10 results, 18% none at all
  – Queries that return a single result take 73s on average, and for <10 results, first is 50s on average
  – Very often results are not found that actually exist (eg the 18% failure can be reduced to 6%)

• Lots of room (we knew that) and need (this is new info) for improvement for rare items
Hybrid approach

- Inverted index for popular keywords is
  - expensive to compute (many messages to the responsible node)
  - Expensive to use (the distributed join (i.e., intersection of matches for keywords in query) is expensive)

- For rare keywords all that is cheap
  - We need to identify rare files and rare keywords and publish those to the DHT
  - When a query has no result for some tome (~30s), we ask the DHT
  - Rarity can be determined by seeing a file in a small result set, and by other heuristics
Another kind of hybrid

• Common wisdom
  – Structured overlays are more expensive and less robust to churn and failures

• Is this true?
  – Comparison is very difficult: too many factors, not clear how to be fair
  – But there are indications it is NOT necessarily true

• If it is indeed not true, they are actually (much) better to support “unstructured” search algorithms, such as flooding and random walks
Busting a myth?

- On some real traces maintenance cost of MS Pastry appears to be better than that of Gnutella
  - Heartbeat messages only to one node: the left neighbor in ring (as opposed to gnutella)

- Heterogeneity can also be captured
  - Super Pastry: similar to Gnutella, but ultrapeers form a Pastry network
  - Hetero Pastry: similar to GIA: routing table entries are optimized to prefer high capacity nodes, and a bound on the in-degree can also be set
  - Maintenance overhead is still fine here
Flooding and random walk in structured networks

- Exploiting the structure of the overlay, broadcast can be optimized to have almost no wasted traffic

- Restricted flooding: a given number of nodes can be visited effectively in parallel
  - Same mechanism for random walk: sequential instead of parallel traversal

- Compare some algorithms
  - using an eDonkey trace
  - max 128 node random walk, one hop replication in all cases (in Pastry, on routing table entries)
Experimental results

Graph showing the success rate over time for different networks:

- Black line: HeteroPastry
- Red line: Gia
- Blue line: Gnutella 0.6
- Green line: SuperPastry
Experimental results

- Gnutella 0.6
- SuperPastry
- Gia
- HeteroPastry

Delay (ms) vs Time (hours)
Conclusions

● DHTs are an alternative to support search
  – They are very efficient
  – They support key based lookup but
  – They can be adapted to support more complex queries as well

● Restricted flooding and random walk is still better for not-so-rare items

● Hybrid approaches
  – Use DHT for rare items only
  – Use structured network to support flooding-style queries instead of random network
Some refs


- http://www.inf.u-szeged.hu/~jelasity/p2p/