Distributed Hash Tables

Christian Grothoff

Technische Universität München

April 11, 2013

Distributed Hash Tables (DHTs)

- Distributed **index**
- GET and PUT operations like a hash table
- JOIN and LEAVE operations (internal)
- Trade-off between JOIN/LEAVE and GET/PUT costs
- Typically use exact match on cryptographic hash for lookup
- Typically require overlay to establish particular connections

DHTs: Key Properties

To know a DHT, you must know (at least) its:

- routing table structure
- lookup procedure
- join operation process
- leave operation process

... including expected costs (complexity) for each of these operations.

A trivial DHTs: The Clique

- routing table: hash map of all peers
- lookup: forward to closest peer in routing table
- join: ask initial contact for routing table, copy table, introduce us to all other peers, migrate data we're closest to to us
- leave: send local data to remaining closest peer, disconnect from all peers to remove us from their routing tables

Complexity?

A trivial DHTs: The Circle

- routing table: left and right neighbour in cyclic identifier space
- lookup: forward to closest peer (left or right)
- join: lookup own peer identity to find join position, transfer data from neighbour for keys we are closer to
- leave: ask left and rigt neighbor connect directly, transfer data to respective neighbour

Complexity?

Additional Questions to ask

- Security against Eclipse attack?
- Survivability of DoS attack?
- Maintenance operation cost & required frequency?
- Latency? ($\neq$ number of hops!)
- Data persistence?

Content Addressable Network: CAN

- routing table: neighbours in *d*-dimensional torus space
- lookup: forward to closest peer
- join: lookup own peer identity to find join position, split quadrant (data areas) with existing peer
- leave: assign quadrant space to neighbour (s)

Interesting CAN properties

- CAN can do range queries along $\leq n$ dimensions
- CAN's peers have $2d$ connections (independent of network size)
- CAN routes in $O(d\sqrt[d]{n})$

Chord

- routing table: predecessor in circle and at distance 2^i, plus r successors
- lookup: forward to closest peer (peer ID after key ID)
- join: lookup own peer identity to find join position, use neighbor to establish finger table, migrate data from respective neighbour
- leave: join predecessor with successor, migrate data to respective neighbour, periodic stabilization protocol takes care of finger updates

Interesting Chord properties

- Simple design
- $\log_2 n$ routing table size
- $\log_2 n$ lookup cost
- Asymmetric, inflexible routing tables

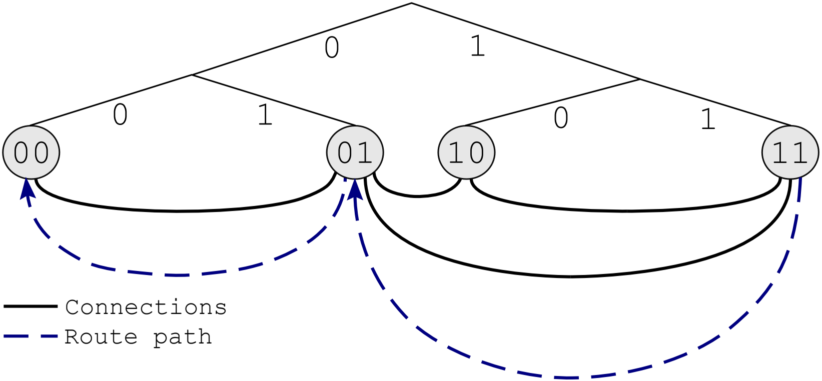
Kademlia

- routing table:
- lookup: iteratively forward to α peers from the "best" bucket, selected by latency
- join: lookup own peer identity, populate table with peers from iteration
- maintenance: when interacting with a peer, add to bucket if not full; if bucket full, check if longest-not-seen peer is live first
- leave: just drop out

Interesting Kademlia properties

- XOR is a symmetric metric: connections are used in both directions
- α replication helps with malicious peers and churn
- Iterative lookup gives initiator much control,
- Lookup helps with routing table maintenance
- Bucket size trade-off between routing speed and table size
- Iterative lookup is a trade-off:
  - good UDP (no connect cost, initiator in control)
  - bad with TCP (very large number of connections)

Kademlia



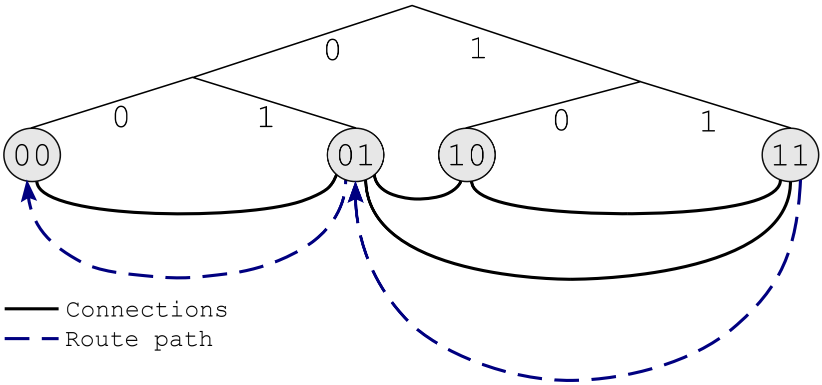
Additional Questions to ask

- Possibility of link-encryption?
- Risks of topology exposure / participant visibility?
- UDP and NAT?

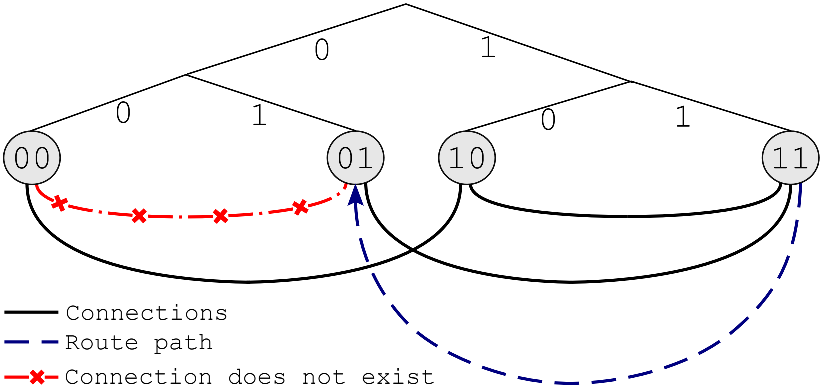
Additional Questions to ask

- Possibility of link-encryption?
- Risks of topology exposure / participant visibility?
- UDP and NAT?
- NAT?

Kademlia



Kademlia and Restricted Routes



Network Size Estimation: Structured Methods [6]

- Assume DHT with equal key distribution between peers
- (average) distance between keys is $\frac{1}{n}$

Network Size Estimation: Non-local Structured Methods

- Each iteration, perform a “GET” request for a random key
- Observe distance d to closest peers to the key
- Calculate average $n \approx \frac{1}{d}$ over many rounds
- Cost: $O(n \cdot \log n)$ per round for the network

Network Size Estimation: Local Structured Methods [1]

Basic Idea

- Observe DHT routing table
- Suppose there are p_k entries in bucket k
- Calculate size $n \approx p_k \cdot 2^k$
- Average over all non-full buckets
- Cost: no network overhead

Problems

- The formula above is intuitive but wrong.

[1] Bartlomiej Polot: "Adapting blackhat approaches to increase the resilience of whitehat application scenarios", MS Thesis, TUM, 2010

Bloom Filters

- Probabilistic data structure to answer the question "is element X in set S" with "no" or "maybe"
- If an element is not in the set, the probability is high that the answer is "no"
- Uses a bit-array where k bits based on $H(X)$ are set to 1 for each element $X \in S$.

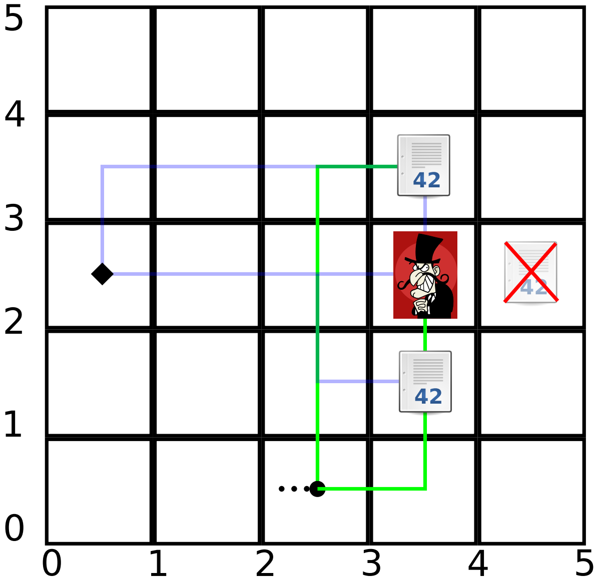
The R^5N DHT

- Designed to work well in restricted route networks (many nearest peers) and reduce the impact of malicious peers.
- Requires recursive routing; less control for initiator, better performance; stateful return routing
- Kademlia style routing table — so-called "k-buckets" storing k peers; such that the i^{th} k-bucket stores peers with XOR distance between $[2^i, 2^{i+1})$

The R^5N Routing Algorithm

- Random- and Kademlia-style routing phases
  $\Rightarrow$ combines path *diversity* with *efficient* routing
  - Random phase: “start” Kademlia routing from random location.
  - Kademlia phase: efficiently find nearest peers.
- Requests have desired replication level r; the number of nearest peers a request *should* reach.
- Achieved by probabilistic path branching, at each hop a request may be forwarded to one or more peers.

The R^5N Routing Algorithm



The R^5N Routing Algorithm

PUT Request

```
if nearest(r) then
    store_data(r)
else
    for i = 0 → num_forwards(r) do
        p = get_forward_peer(r)
        forward_request(r, p)
    end for
end if
```

GET Request

```
if NULL ≠ (d = find_data(r)) then
    route_result(r, d)
end if
for i = 0 → num_forwards(r) do
    p = get_forward_peer(r)
    store_route(p, r)
    forward_request(r, p)
end for
```

Routes with Loops

- R^5N cannot loop forever due to the hop counter
- Looping is still inefficient

⇒ R^5N uses a Bloom filter to avoid loops

Performance Analysis for R^5N

- Randomized routing takes c steps, $c \sim \log n$
- Kademlia-style routing takes $O(\log n)$ steps

$\Rightarrow$ Finding a nearest peer is $O(\log n)$

Performance Analysis for R^5N

- There are $\frac{|N|^2}{|E|} \in O(|N|)$ nearest peers
- For a 50% success rate for a single GET, we need $O(\sqrt{|N|})$ replicas
- Then repeat GET $O(\sqrt{|N|})$ times for "high" success rate

$\Rightarrow$ Total routing cost is $O(\sqrt{n} \log n)$

Absolute Performance

Size of network	Average hops per PUT		Average hops per GET	
	R-Kademlia	R^5N	R-Kademlia	R^5N
100	2.70 ± 0.06	3.96 ± 0.06	2.54 ± 0.03	4.63 ± 0.17
250	3.06 ± 0.10	4.26 ± 0.10	3.10 ± 0.06	5.96 ± 0.27
500	3.08 ± 0.46	4.38 ± 0.45	3.38 ± 0.06	6.17 ± 1.14
750	3.19 ± 0.74	4.37 ± 0.83	3.50 ± 0.04	6.29 ± 1.04
1000	3.63 ± 0.07	4.47 ± 0.93	3.64 ± 0.04	7.29 ± 0.95

The DHT API

- `GNUNET_DHT_connect`, `GNUNET_DHT_disconnect`
- `GNUNET_DHT_put`, `GNUNET_DHT_put_cancel`
- `GNUNET_DHT_get_start`, `GNUNET_DHT_get_stop`
- `GNUNET_DHT_monitor_start`, `GNUNET_DHT_monitor_stop`

Special GET Options

GET requests can be given the following optional options:

- Bloom Filter: filter known results (duplicates)
- Bloom Filter Mutator: change hash function of Bloom Filter
- eXtended Query: additional query information beyond the hash

Options for GET and PUT

- `GNUNET_DHT_RO_DEMULTIPLEX_EVERYWHERE`
- `GNUNET_DHT_RO_RECORD_ROUTE`
- Replication level
- Expiration time (provided to PUT, returned by GET)
- Block type $\Rightarrow$ for content validation

Monitoring

DHT monitoring is useful for...

- Testing / debugging
- Performance analysis
- Application development!

The BLOCK API

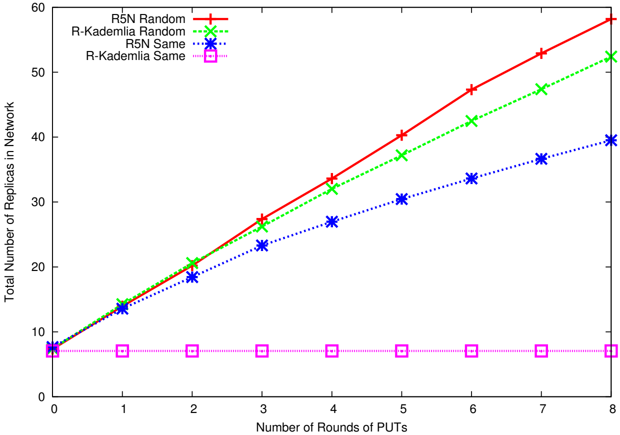
- Block type determines responsible Block plugin
- Configuration option `[block] PLUGINS` specifies supported plugins
- Implement a new plugin based on the `gnunet_block_plugin.h` header
- "fs" for file-sharing, "dht" for DHT internals, "test" for no verification (any data can match any key)

The BLOCK Plugin API

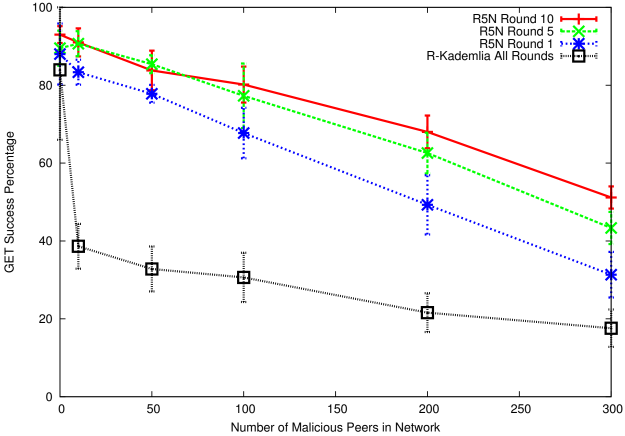
Each plugin must provide two functions:

- `GNUNET_BLOCK_EvaluationFunction`: does the given block satisfy the requirements of the given query? Possible answers include: Yes, and other replies can exist; yes, and this is the only answer; no, duplicate reply; no, invalid reply
- `GNUNET_BLOCK_GetKeyFunction`: given a block, what key should it be stored under? Possible answers are: A key; bad block; not supported

Experimental Results: Replication



Experimental Results: Sybils



Questions?

?

Searching in DHT-based Peer-to-Peer Networks

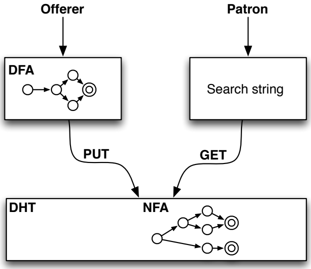
- Distributed *key/value* storage, typically hashes for *keys*
- Range queries (PastryStrings [1], PHT [5])
- Pattern matching (Cubit [3], DPMS [2])
- Similarity queries (Karnstedt et al. [4])

Searching in DHT-based Peer-to-Peer Networks

- Distributed *key/value* storage, typically hashes for *keys*
- Range queries (PastryStrings [1], PHT [5])
- Pattern matching (Cubit [3], DPMS [2])
- Similarity queries (Karnstedt et al. [4])
- Today: regular expressions (Szengel et al.)

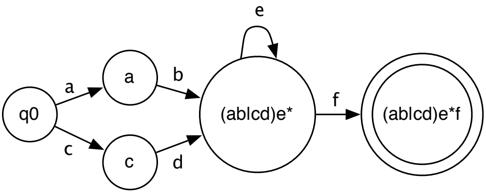
Approach: Idea

1. Offerer creates regular expression describing service
2. Regular expression is converted to a DFA
3. DFA is stored in the DHT
4. Patron matches using a string



Problem: Mapping of States to Keys

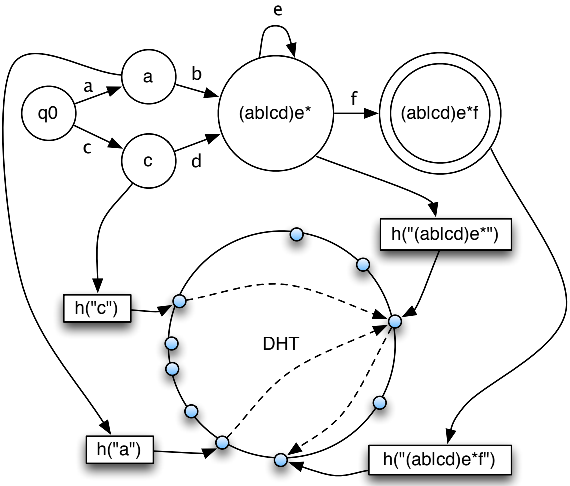
Regular expression $(ab|cd)e^*f$ and corresponding **DFA**



A regular expression is assigned to each state as its identifier.
The hash of the identifier is used as the *key* for DHT PUT.

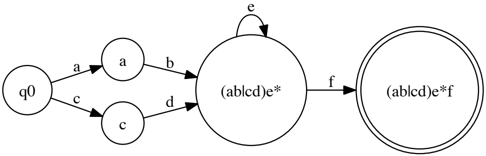
Problem: Mapping of States to Keys

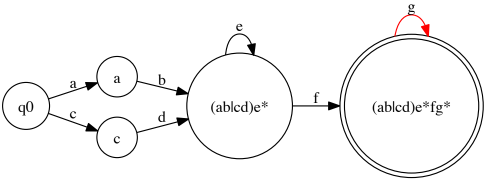
Regular expression $(ab|cd)e^*f$ and corresponding **DFA**



Problem: Merging of DFAs

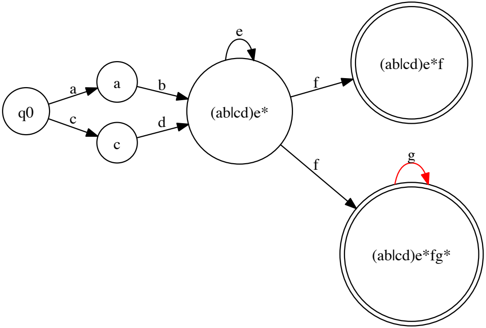
Regular expressions $(ab|cd)e^*f$ and $(ab|cd)e^*fg^*$ with corresponding **DFAs**





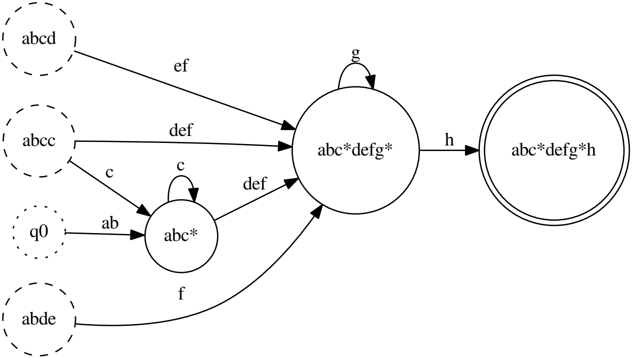
Problem: Merging of DFAs

Merged **NFA** for regular expressions $(ab|cd)e^*fg^*$ and $(ab|cd)e^*f$



Problem: Decentralizing the Start State

Regular expression: abc^*defg^*h and $k = 4$.



Future Work

RegEx search is implemented in GNUnet.

Future Work

- Use regular expression search in new applications
- Open problem: searching using a regular expression

Questions?

?

References

Ioannis Aekaterinidis and Peter Triantafillou.
PastryStrings: a comprehensive content-based publish/subscribe DHT network.
In *Proc. 26th IEEE Int. Conf. on Distributed Computing Systems (ICDCS '06), Lisboa, Portugal*, page 23, 2006.

Reaz Ahmed and Raouf Boutaba.
Distributed pattern matching: A key to flexible and efficient P2P search.
In *Proc. 10th IEEE/IFIP Network Operations and Management Symposium (NOMS 2006)*, pages 198–208.

Aleksandrs Slivkins Bernard Wong and Emin Gn Sirer.
Approximate matching for Peer-to-Peer overlays with Cubit.
Technical report, Cornell University, Computing and Information Science, 2008.

Marcel Karnstedt, Kai-Uwe Sattler, Manfred Hauswirth, and Roman Schmidt.
Similarity queries on structured data in structured overlays.
In *Proc. 22nd Int. Conf. on Data Engineering Workshops (ICDEW '06)*, page 32, Washington, DC, USA, 2006. IEEE Computer Society.

Sylvia Ratnasamy, Joseph M. Hellerstein, and Scott Shenker.
Range queries over DHTs.
Technical Report IRB-TR-03-009, Intel Research, 2003.

Tallat Shafaat, Ali Ghodsi, and Seif Haridi.
A practical approach to network size estimation for structured overlays.
In Karin Hummel and James Sterbenz, editors, *Self-Organizing Systems*, volume 5343 of *Lecture Notes in Computer Science*, pages 71–83. Springer Berlin / Heidelberg, 2008.