Designing a DHT for low latency and high throughput

Robert Vollmann

30.11.2004

P2P Information Systems

Overview

I. Introduction to DHT‘s

II. Technical Background

III. Latency

IV. Throughput

V. Summary

Distributed Hash Tables

Hash table

- Data structure that uses hash function to map keys to hash values to determine where the data is stored
- Allows quick access to keys through lookup algorithm

Distributed hash table

- Hash table is distributed over all participating nodes
- Lookup algorithm determines the node responsible for a specific key

Requirements

- Find data
- High availability
- Balance load
- Fast moving of data

Problems

- Link capacity of nodes
- Physical distance of nodes
- Congestion of network and packet loss

Technical Background

DHash++

- Values are mapped to keys using SHA-1 hash function
- Stores key/value pairs (so-called blocks) on different nodes
- Uses Chord and Vivaldi

Chord

- Lookup protocol to find keys with runtime O(log N)

Vivaldi

- Decentralized Network Coordinate System to compute and manage synthetic coordinates which are used to predict inter-node latencies
- No additional traffic because synthetic coordinates can piggy-back on DHash++'s communication patterns

Hardware

- Test-bed of 180 hosts which are connected via Internet2, DSL, cable or T1

Additional testing

- Simulation with delay matrix filled with delays between 2048 DNS Servers

Low latency

Data layout

DHash++ is designed for read-heavy applications that demand low-latency and high throughput.

Examples

SFR (Semantic Free Referencing system)

- Designed to replace DNS (Domain Name System)
- Uses DHT to store small data records representing name bindings

UsenetDHT

- Distributed version of Usenet which splits large articles into small blocks to achieve load balancing

DHash++:

- Can be seen as a Network Storage System with shared global infrastructure
- Uses small blocks of 8kb length
- Uses random distribution of blocks via hash Function

Low latency

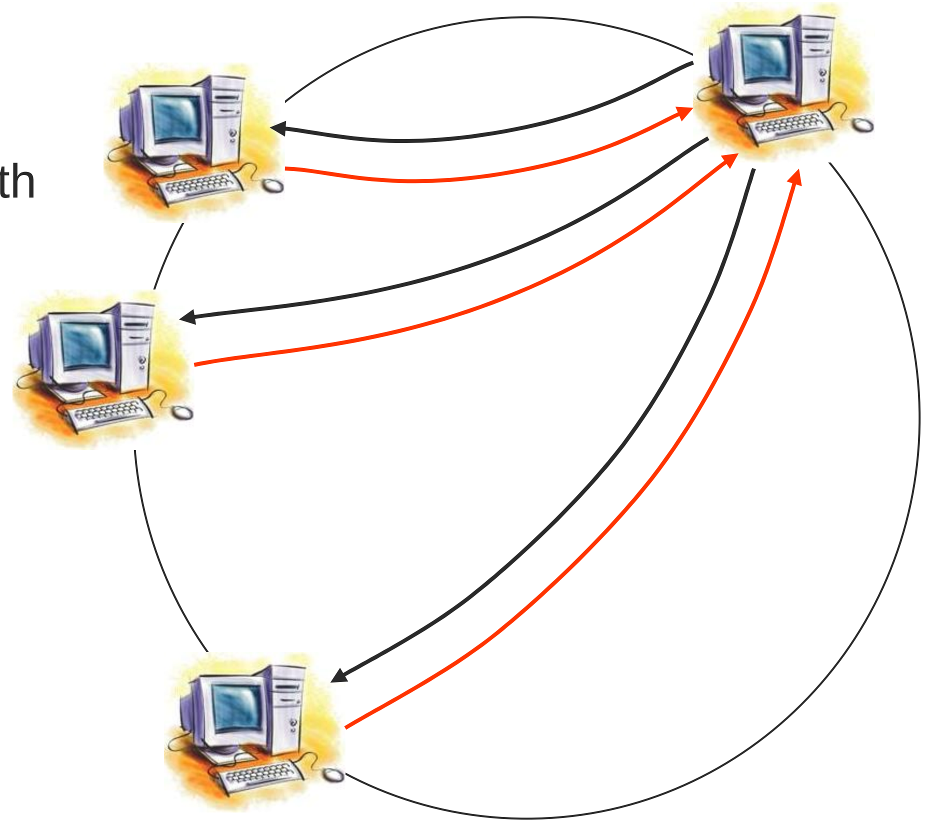
Recursive vs. iterative

Iterative lookup

- Send lookup query to each successive node in lookup path
- Can detect node failure

But

- Must wait for response before proceeding



Low latency

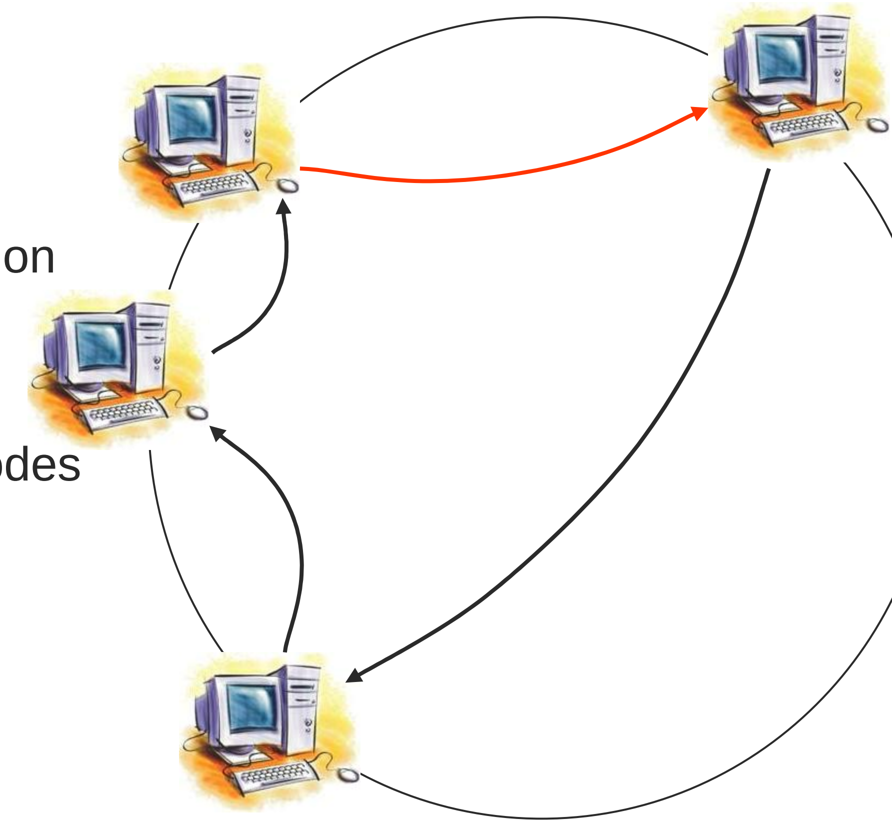
Recursive vs. iterative

Recursive lookup

- Direct query forwarding to next node
- Less queries -> less congestion

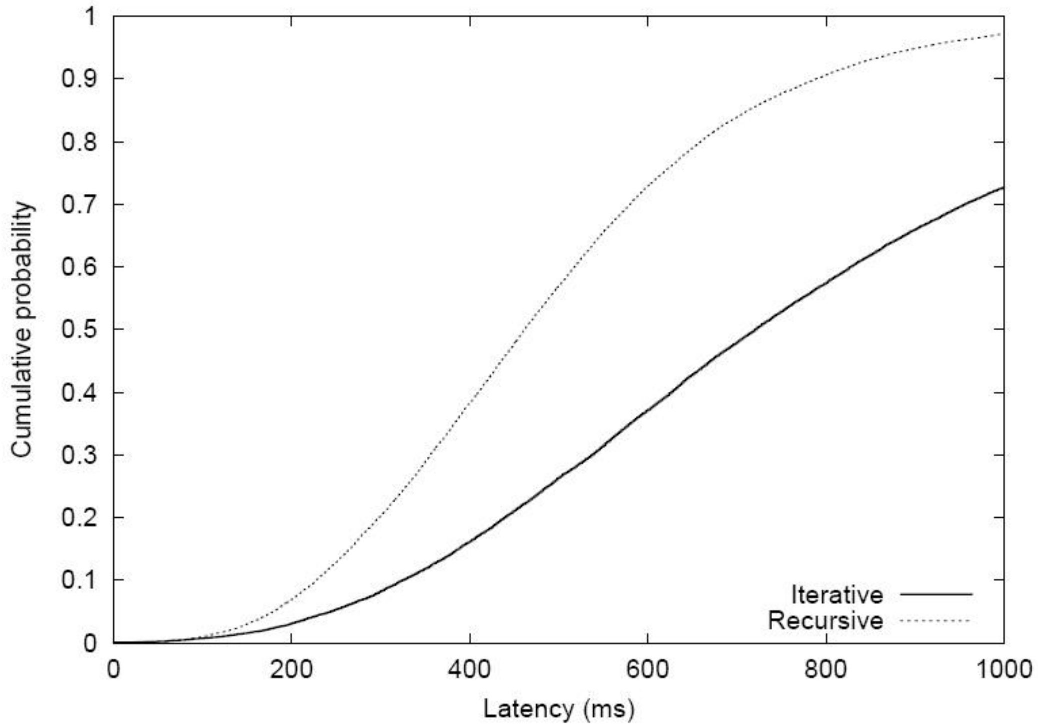
But

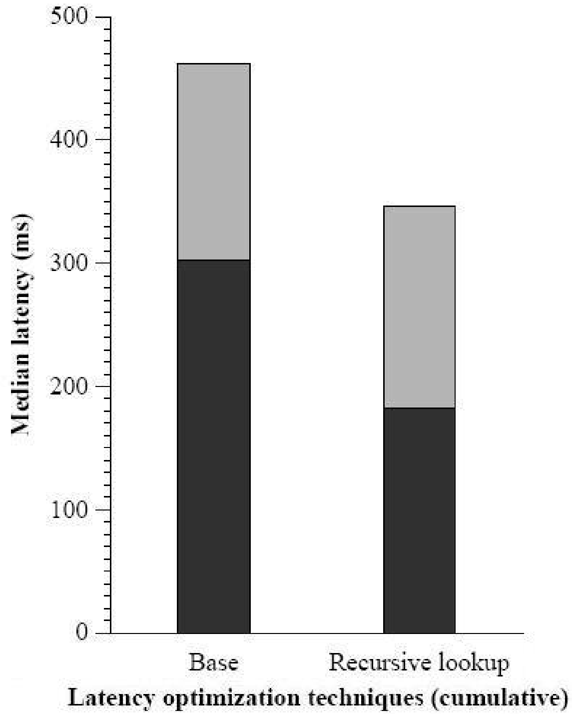
- Impossible to detect failed nodes



Low latency

Recursive vs. iterative





Left Figure: Simulation of 20,000 lookups with random hosts for random keys

- Recursive lookup takes 0.6 times as long as iterative

Right Figure: 1,000 lookups in test-bed

- Result of simulation confirmed

Trade-off: DHash++ uses recursive routing but switches to iterative routing after persistent link failures

Low latency

Proximity neighbor selection

Idea

- Chose nearby nodes to decrease latency

Realisation

- ID-Space range of ith finger table entry of node a:

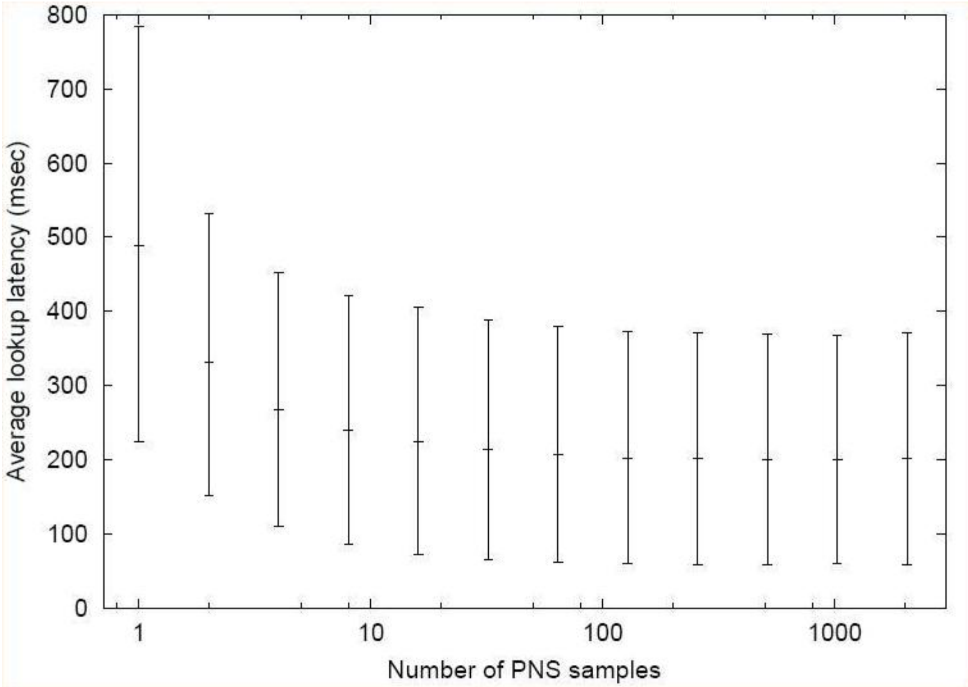
$$a + 2^i \text{ to } a + 2^{i+1} - 1$$

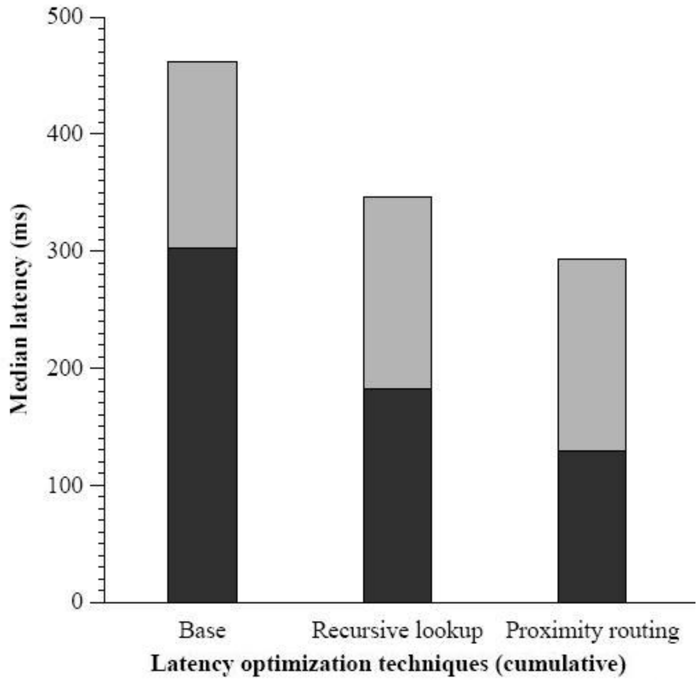
- Every finger table entry points to first available ID in this range
- Get the latency of the the first x available nodes in this ID-Space range from successor list
- Route lookups through node with lowest latency

What is a suitable value for x?

Low latency

Proximity neighbor selection





Right Figure: Simulation of 20 000 lookups with random hosts for random keys

- 1 – 16 Samples: highly decreasing latency
- 16 – 2048 Samples: barely decreasing latency

Right figure: 1 000 lookups in test-bed

- Decreased lookup latency

→ DHash++ uses 16 Samples

Low latency
Erasure-coding vs. replication

Erasure-coding
- Data block split into l fragments
- m different fragments are necessary to reconstruct the block
- Redundant storage of data

Replication
- Node stores entire block
- Special case: $m = 1$ and l is number of replicas
- Redundant information spread over fewer nodes

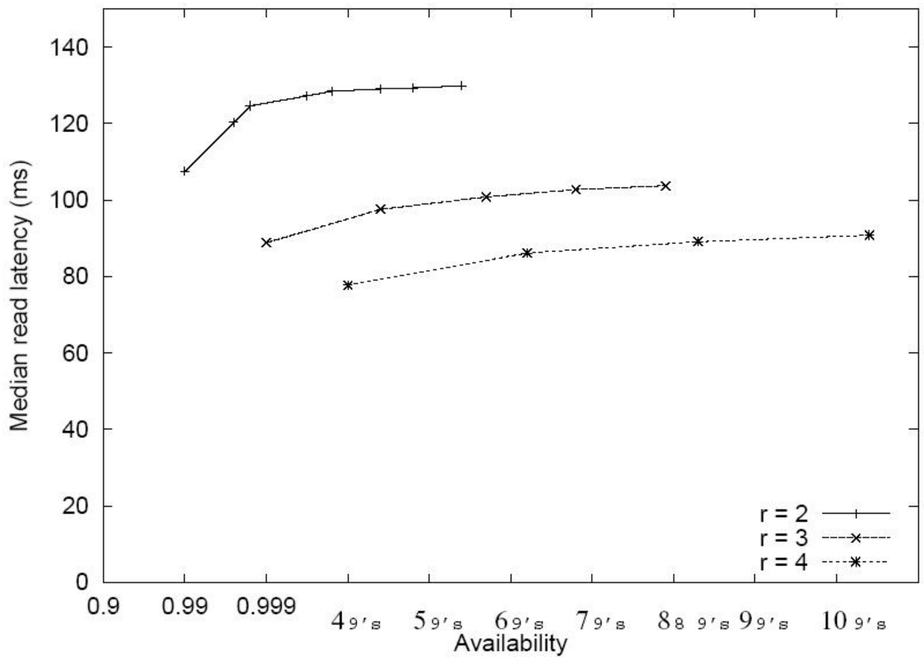
Comparison of both methods
- $r = l / m$ amount of redundancy

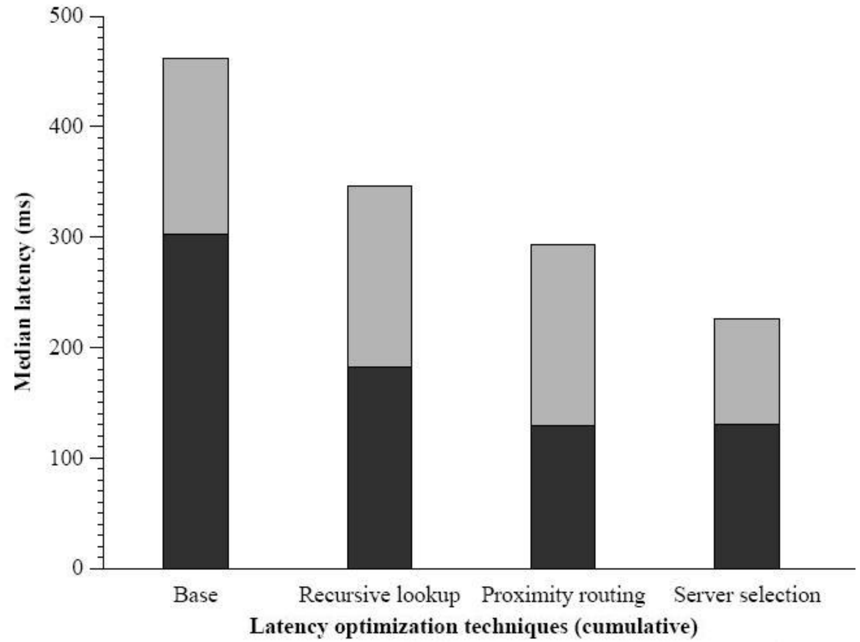
Probability that a block p is available:

$$p_{avail} = \sum_{i=m}^{l} \binom{l}{i} p_0^i (i - p_0)^{l-i}$$

Low latency

Erasure-coding vs. replication





Replication

- Only slightly lower latency than erasure-coding for same r if r is high
- Less congestion than erasure-coding because less files have to be distributed

Erasure-coding

- Higher availability of fragments
- More choice because of more fragments

DHash++ uses Erasure-coding with m = 7 and l = 14

Low latency

Integration

Remember proximity neighbor selection

- Little advantage in the last steps

Also remember last step in lookup procedure

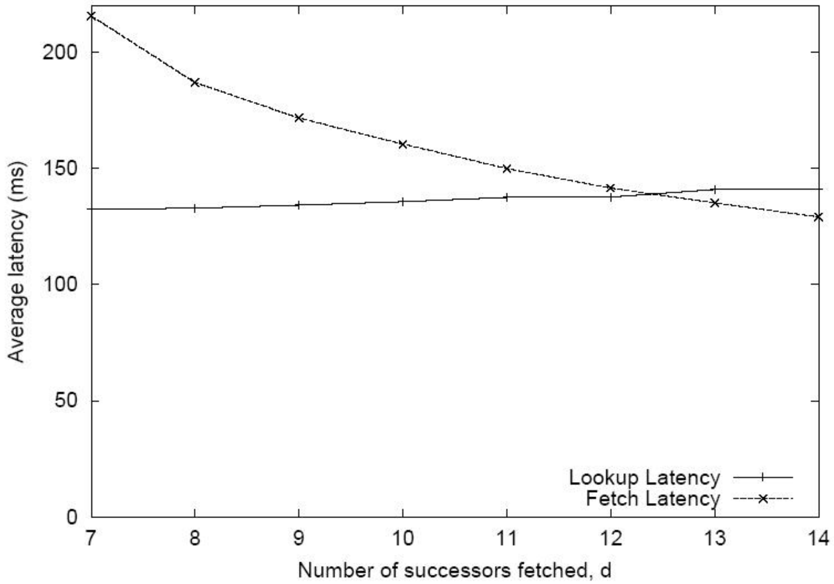
- Originator contacts a key‘s direct predecessor to obtain successor list

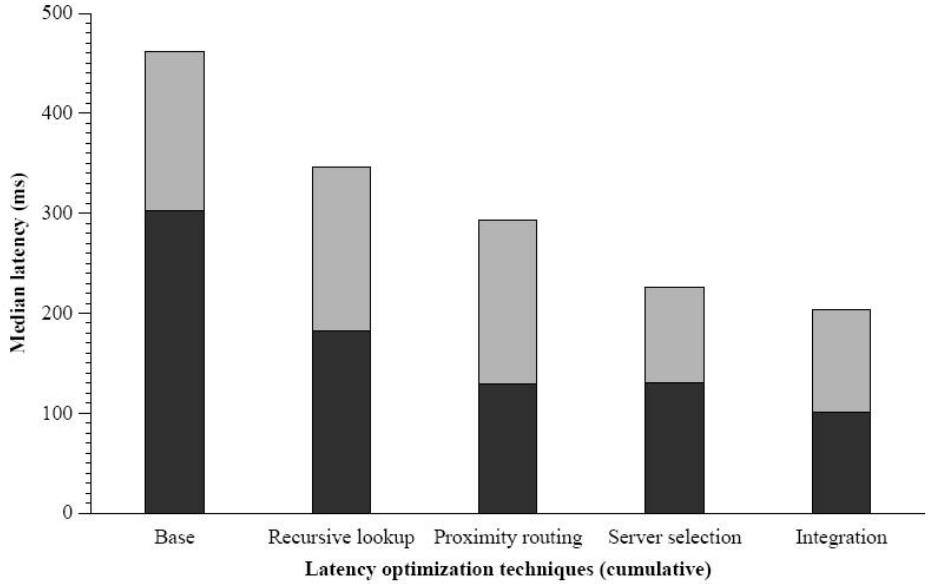
- But full successor list not neccessarily needed

Why?

- List has length s
- l successors store fragments of the block
- m fragments are needed
- s-m predecessors of a key have lists with at least m nodes

Low latency
Integration





d: Number of successors in successor list with needed fragment

Trade-off between lookup time and fetch latency while choosing *d*

- Large *d*: more hops for lookup but more choice between nodes
- Small *d*: less hops but higher fetch latency

Optimum: $d = l$

High throughput

Overview

Requirements for a DHT

- Parallel sending and receiving of data
- Congestion control to avoid packet loss and re-transmissions
- Recover from packet loss
- **Difficulty**: Data is spread over a large set of servers

→ Efficient transport protocol needed

First possibility: Use existing transport protocol

TCP (Transport Control Protocol)

- Provides congestion control, but
- Optimal congestion control and timeout estimation require some time
- Imposes start-up latency
- Number of simultaneous connections limited

Approach by DHash++

- Small number of TCP connections to neighbours
- Whole communication over these neighbours

High throughput
STP

Second Possibility: Design alternative transport protocol

STP (Striped Transport Protocol)
- Receiving and transmitting data directly to other nodes in single instance
- No per-destination states, decisions based on recent network behavior and synthetic coordinates (Vivaldi)

Remote Procedure Call (RPC)
- Calls a procedure that is located at another host on the network
- Example: lookup calls procedure on host to retrieve finger table entry

Round-trip-time (**RTT**)
- The time it takes to send a packet to a host and receive a response
- Used to measure delay on a network

High throughput
STP

Congestion Window Control

- *w* simultaneous RPCs
- New RPC only when old is finished
- Congestion: If RPC is answered *w* is increased by 1/*w*, otherwise *w* is decreased by *w*/2

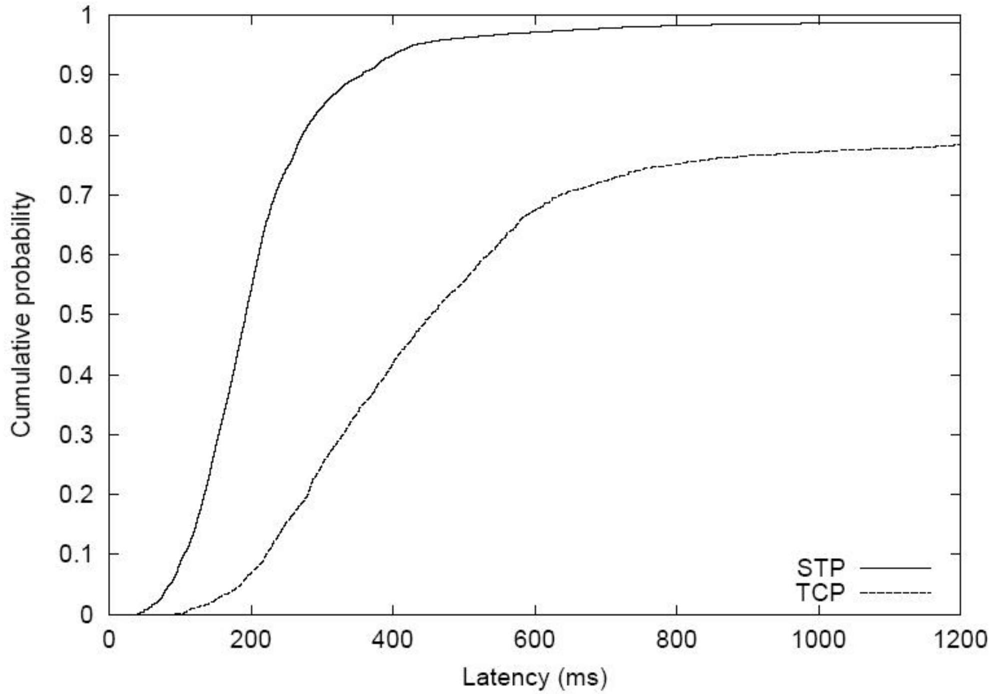
Retransmit timers

- TCP predicts new RTT with deviation of old RTTs
- In general no repeated sending of RPCs to the same node
  → must predict RTT before sending, therefore uses Vivaldi

Retransmit policy

- No direct resending if timer expires
- Notifies application (DHash++)
  → On lookup: send to next-closest finger
  → On fetch: query successor that is next-closest in predicted latency

High throughput
TCP vs. STP



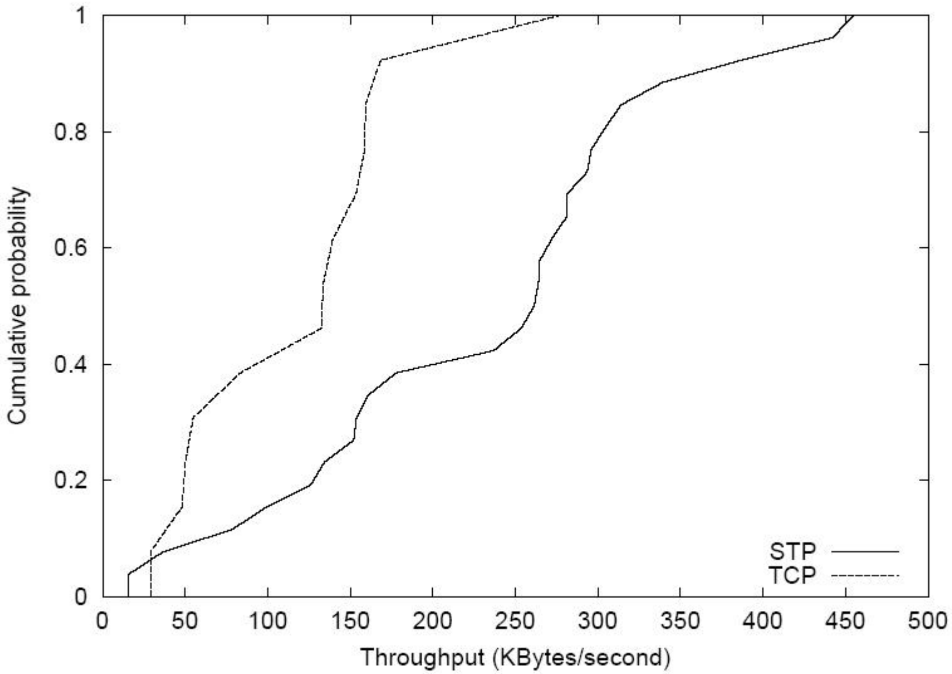
- Sequence of single random fetches of 24 nodes
- Median fetch time: 192 ms with STP, 447ms with TCP
- On average: 3 hops to complete lookup

But

- TCP fetch through 3 TCP connections consecutively
- STP fetch through 1 single STP connection

High throughput

TCP vs. STP



- Simultaneous fetches of different nodes
- Median throughput: 133 KB/s with TCP, 261 KB/s with STP
- Same reason for result as in previous slide

Summary

Different design decisions

- **Recursive routing** → reducing number of sent packets
- **Proximity neighbor selection** → searching nearby nodes
- **Erasure-coding** → increasing availability of data
- **Integration** → reducing number of hops for lookup

Result:

Reduced the lookup and fetch latency up to a factor of 2

Alternative Transport Protocol **STP**

- Fitted to the needs of DHash++
- Direct connection between nodes
  →Higher throughput than TCP
  →Lower latency than TCP

Result:

Further reduced latency and optimized throughput by factor 2

Thanks for Your Attention