KTH ROYAL INSTITUTE
OF TECHNOLOGY

Distributed Hash Tables

Johan Montelius and Vladimir Vlassov

Distributed Hash Tables

- Large-scale databases (key-value stores)
  - hundreds of servers
- High churn rate
  - servers will come and go
- Benefits
  - fault tolerant
  - high performance
  - self administrating

A key-value store

Associative array to store ***key-value pairs***, a data structure known as a ***hash table*** (array of buckets) that maps keys to values.

Operations:

put (key, object) – store a given object with a given key

object: = get (key) – read an object given key.

Design issues:

- Identify: how to uniquely identify an object
- Store: how to distribute objects among servers
- Route: how to find an object

Unique identifiers

We need ***unique identifiers*** to identify objects, i.e., to find a bucket to get/put an object with a given key

identifier = f(key, size_of_hash_table)

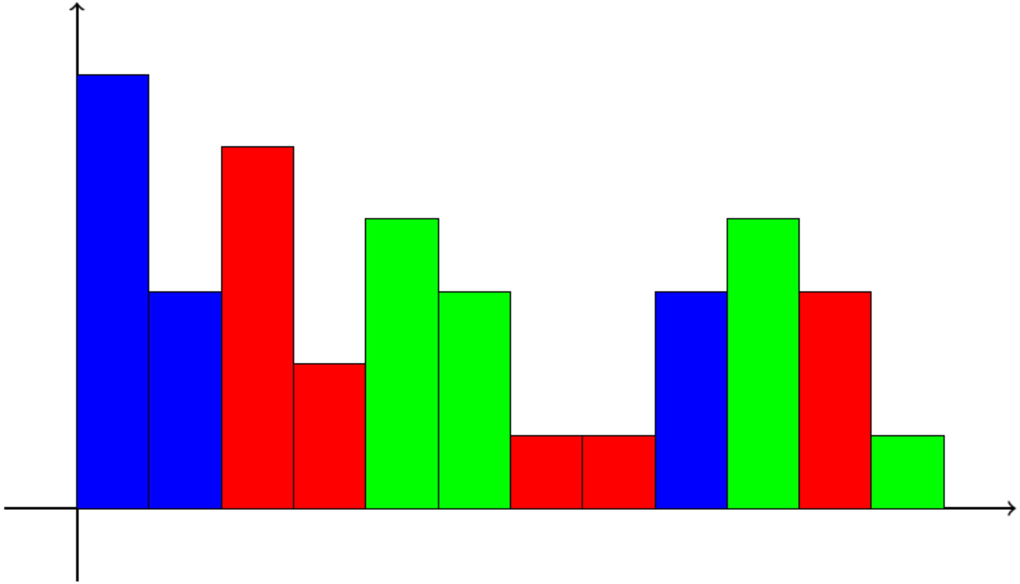
How to select identifiers:

- use a key (a name)
- a cryptographic hash of the key
- a cryptographic hash of the object

Why hash?

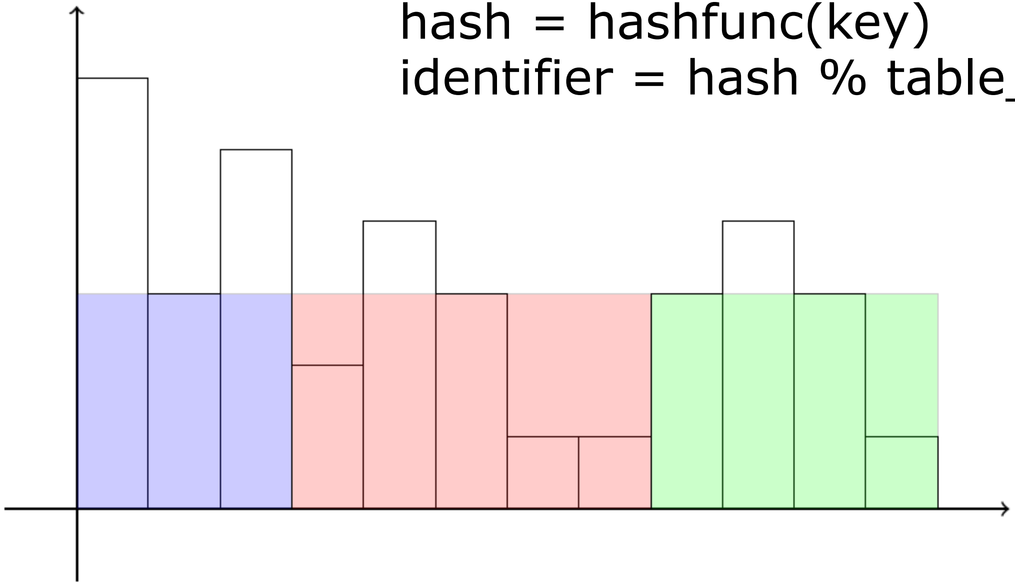
Key distribution – direct map

A direct map of keys to identifiers (buckets) might give a non-uniform (uneven) distribution of keys among buckets.



Key distribution – hashing keys

A cryptographic hash function gives a uniform (even) distribution of the keys among buckets

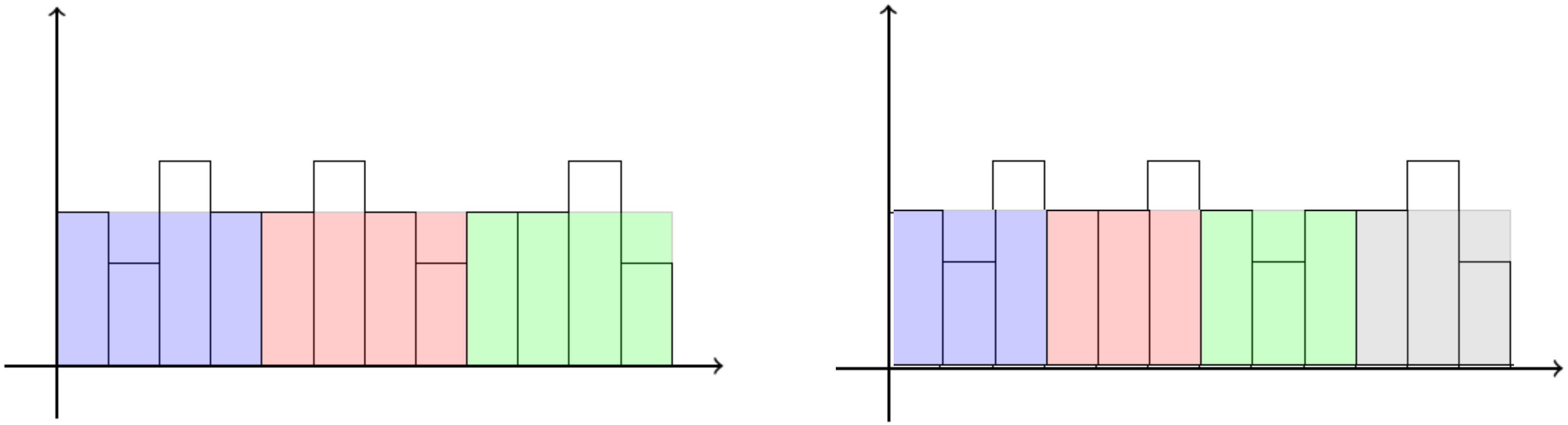


hash = hashfunc(key)
identifier = hash % table_size

In this example:
3 servers to store a DHT of 12 buckets.
1st server is responsible for 3 buckets;
2nd – for 5 buckets,
3rd – for 4 buckets.

Add a server

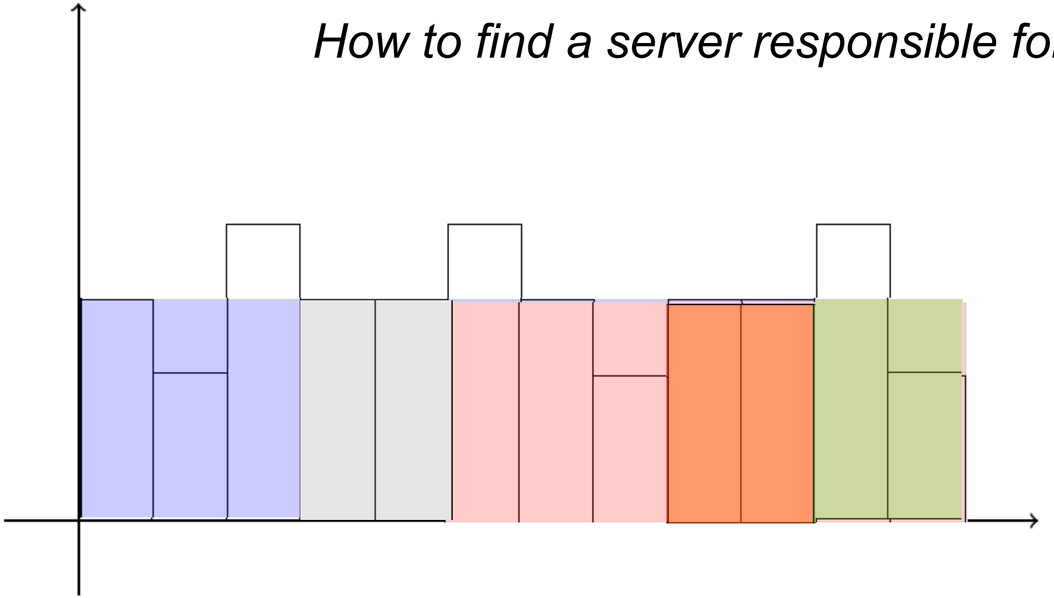
At three o'clock in the morning, do:



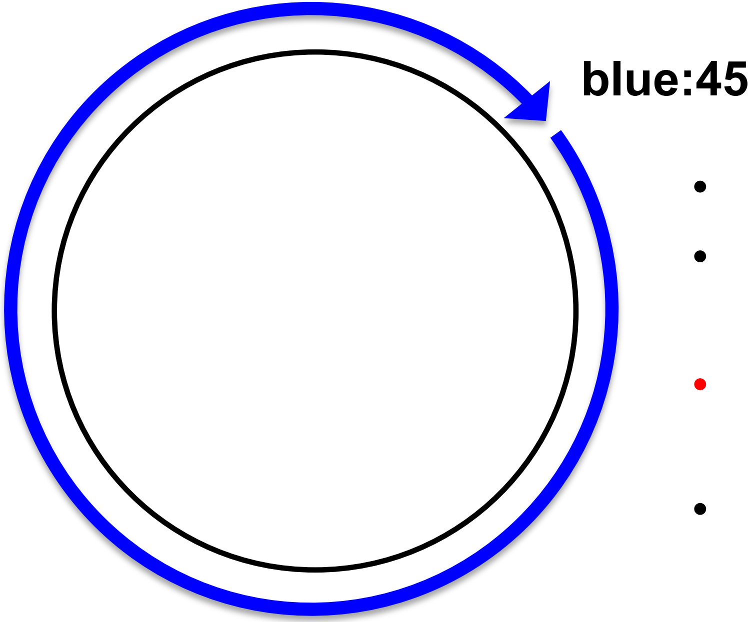
Random distribution

Random distribution of key ranges among servers

How to find a server responsible for a given key?

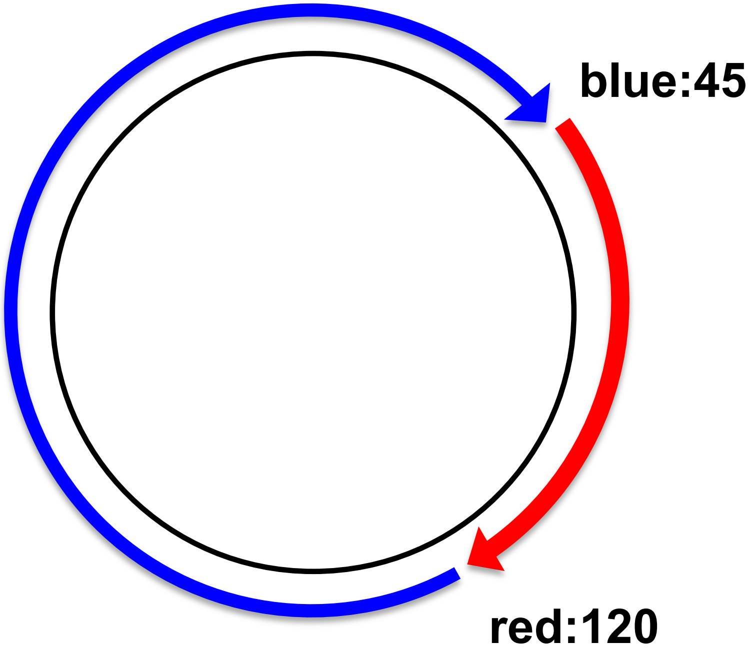


Circular domain



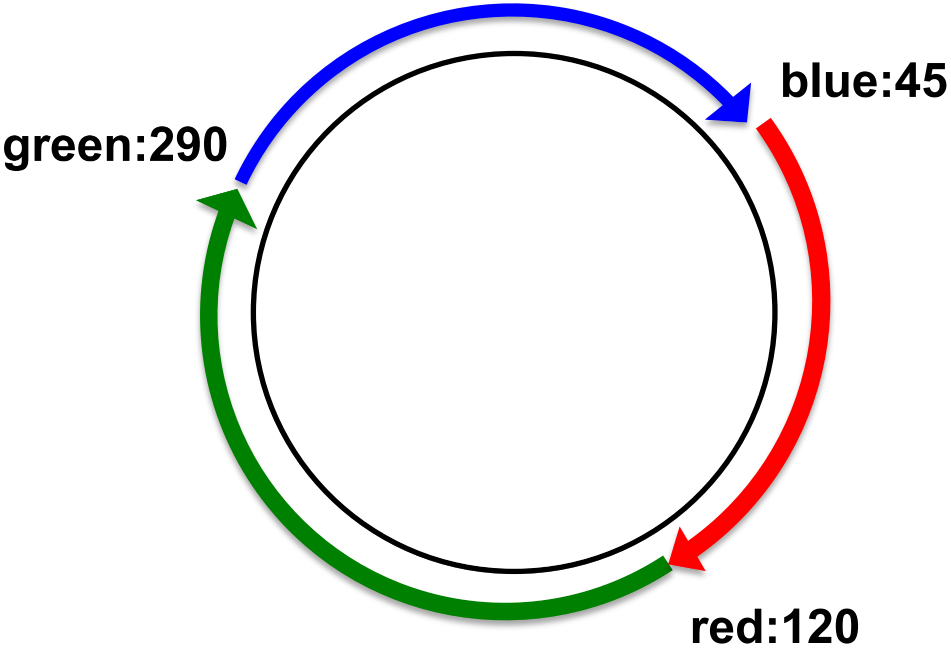
- ID domain: 0,1,2,..., size-1
- clockwise step along the ring
  $i = (i + 1)\%\ size$
- ***responsibility***: from your predecessor to your number
- when inserted: take over responsibility

Circular domain



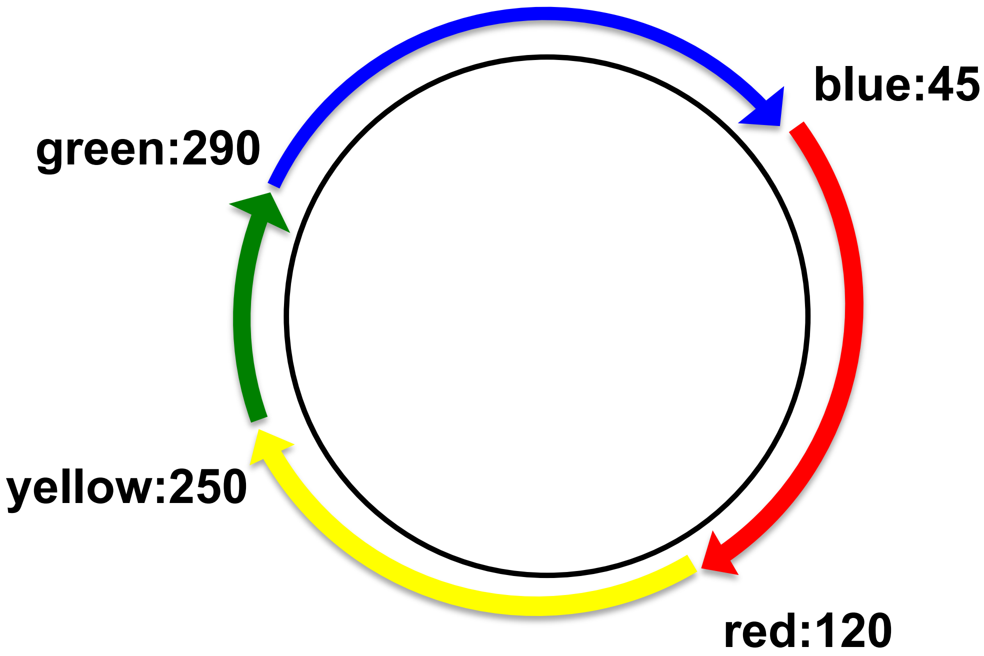
- *responsibility*: from your predecessor to your number
- when inserted: take over responsibility
- e.g., red:120 is responsible for keys in the range (45, 120]

Circular domain



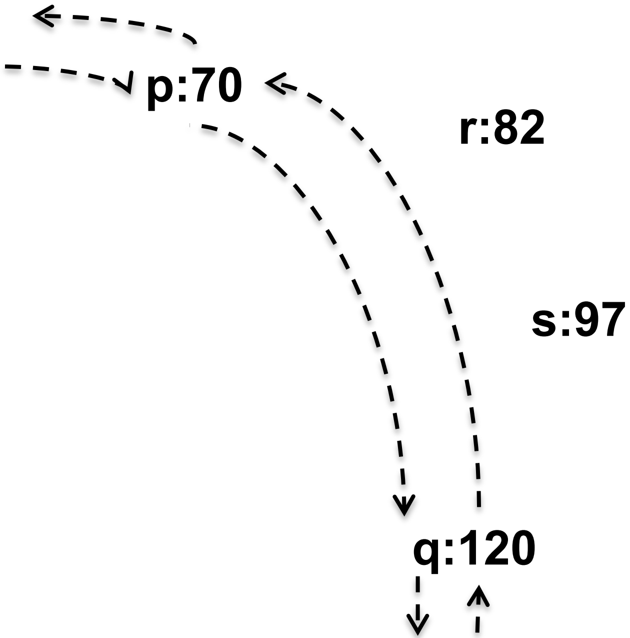
- *responsibility*: from your predecessor to your number
- when inserted: take over responsibility

Circular domain



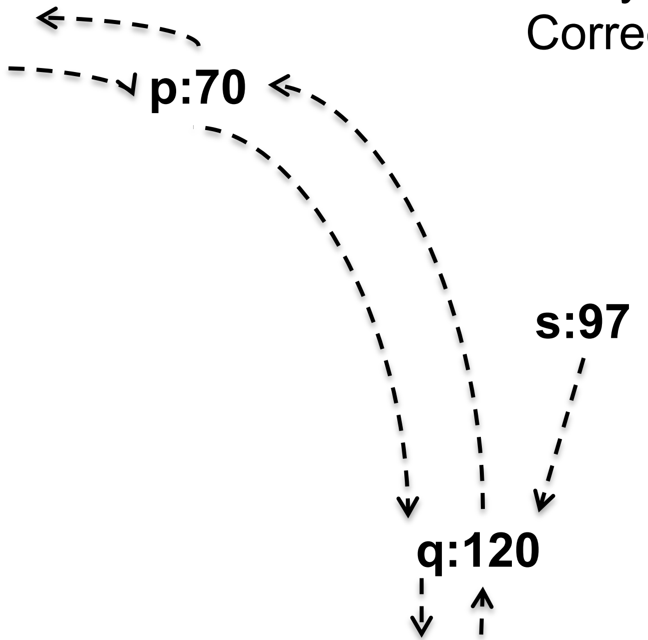
- *responsibility*: from your predecessor to your number
- when inserted: take over responsibility
- talk to the node in front of you

Double linked circle



- predecessor
- successor
- how do we insert a new node
- concurrently

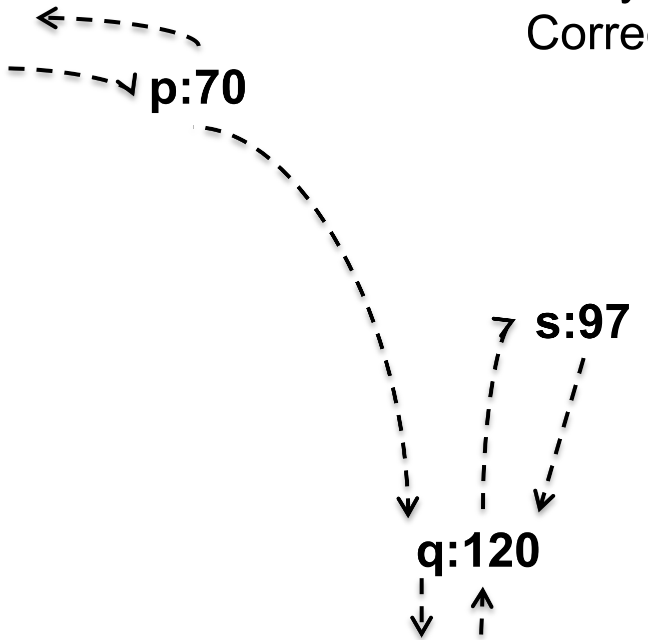
Stabilization



Ask your successor: ***Who is your predecessor?***
Correct a wrong link if any

s:97: - Who is your predecessor?
q:120: - It's p at 70(p).
s:97: - Why don't you point to me!

Stabilization

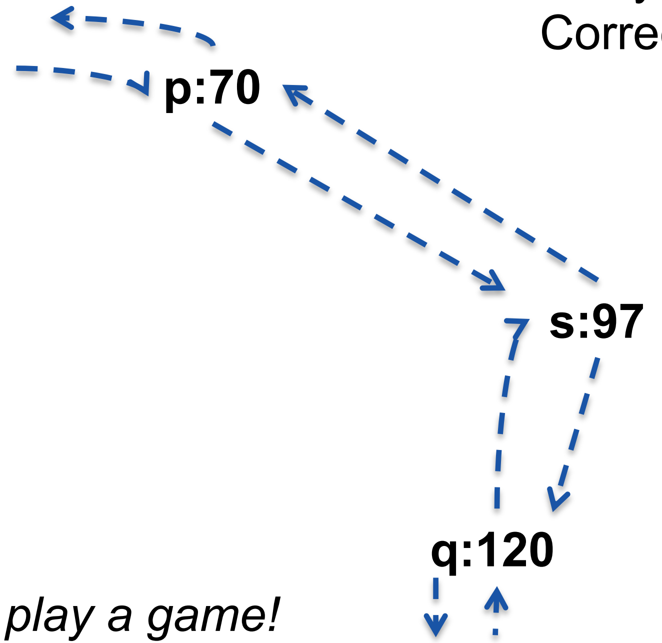


Ask your successor: ***Who is your predecessor?***
Correct a wrong link if any

s:97: - Who is your predecessor?
q:120: - It's p at 70(p).
s:97: - Why don't you point to me!
p:70: - Who is your predecessor?
q:120: - It's s at 97(s).
p:70: - Hmmm, that's a better successor.

Stabilization

Ask your successor: ***Who is your predecessor?***
Correct a wrong link if any



s:97: - Who is your predecessor?
q:120: - It's p at 70(p).
s:97: - Why don't you point to me!
p:70: - Who is your predecessor?
q:120: - It's s at 97(s).
p:70: - Hmmm, that's a better successor.
p:70: - Who is your predecessor?
s:97: - I don't have one.
p:70: - Why don't you point to me!

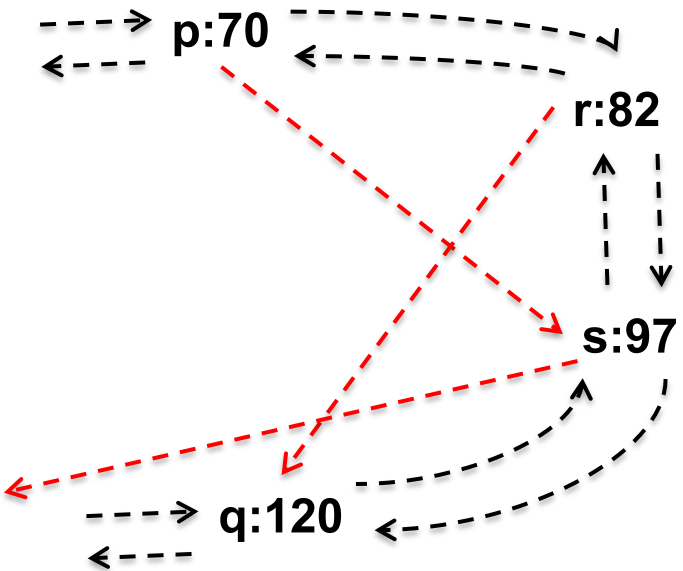
Let's play a game!

Stabilization

Stabilization is run periodically: allow nodes to be inserted concurrently.

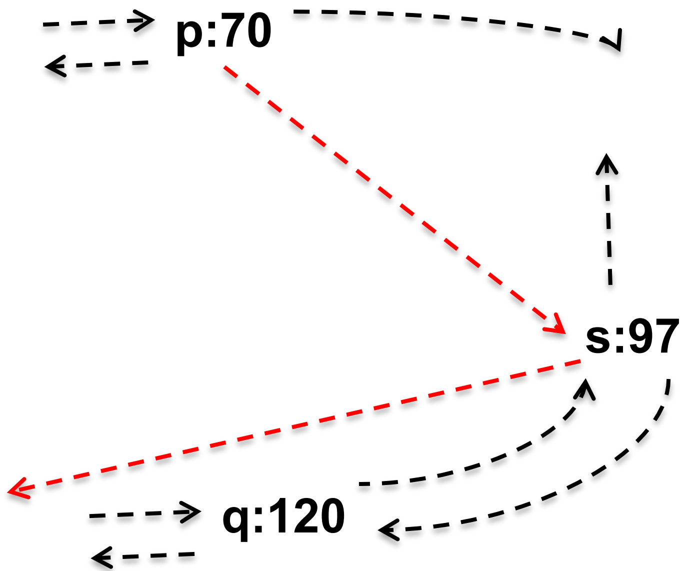
The inserted node will take over responsibility for part of a segment.

Crashing nodes



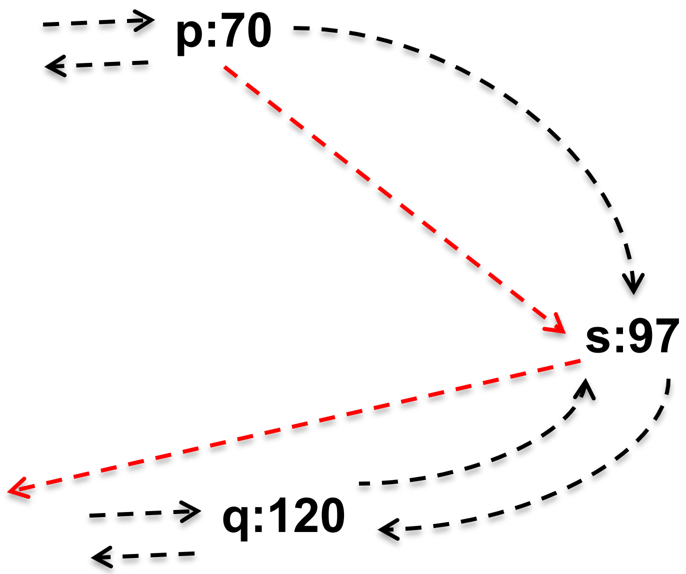
- monitor neighbors
- safety pointer

Crashing nodes



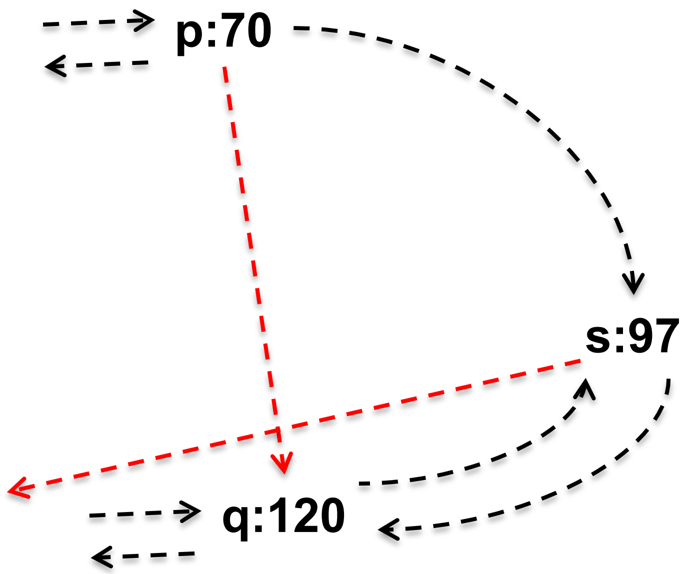
- monitor neighbors
- safety pointer
- detect crash

Crashing nodes



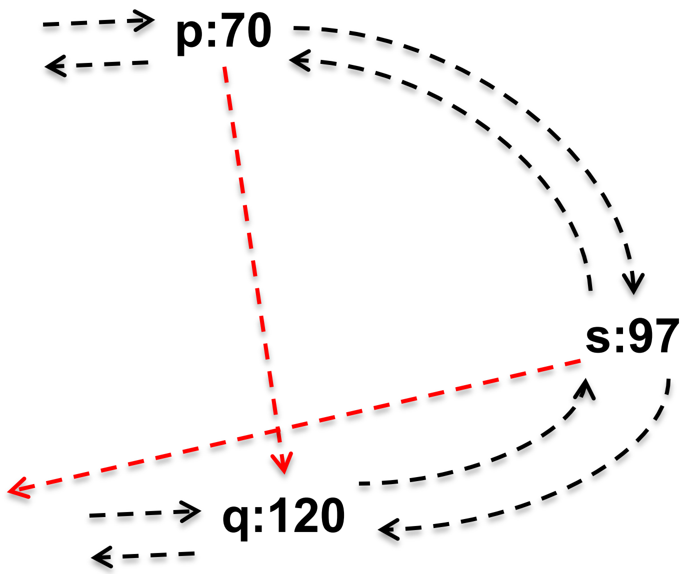
- monitor neighbors
- safety pointer
- detect crash
- update forward pointer

Crashing nodes



- monitor neighbors
- safety pointer
- detect crash
- update forward pointer
- update safety pointer

Crashing nodes



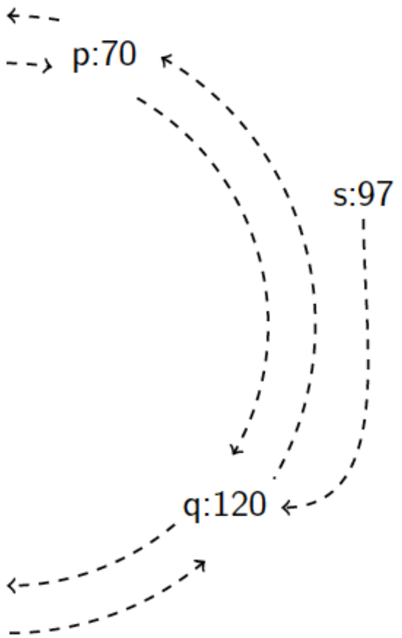
- monitor neighbors
- safety pointer
- detect crash
- update forward pointer
- update safety pointer
- stabilize

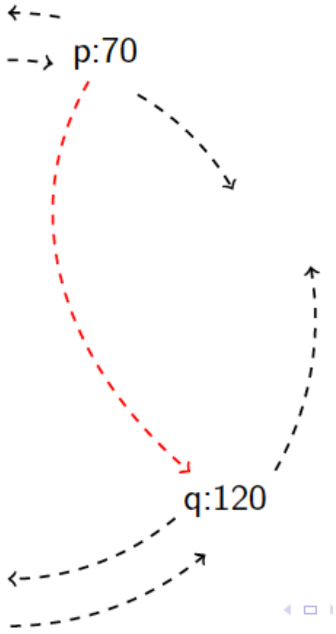
Russian roulette

How many safety pointers do we need?

Replication

Where should we store a replica of our data?





Routing overlay

- The problem of finding an object in our distributed table:
  - Nodes can join and crash
  - A trade-off between routing overhead and updating overhead

In the worst case, we can always forward a request to our successor.

Leaf set

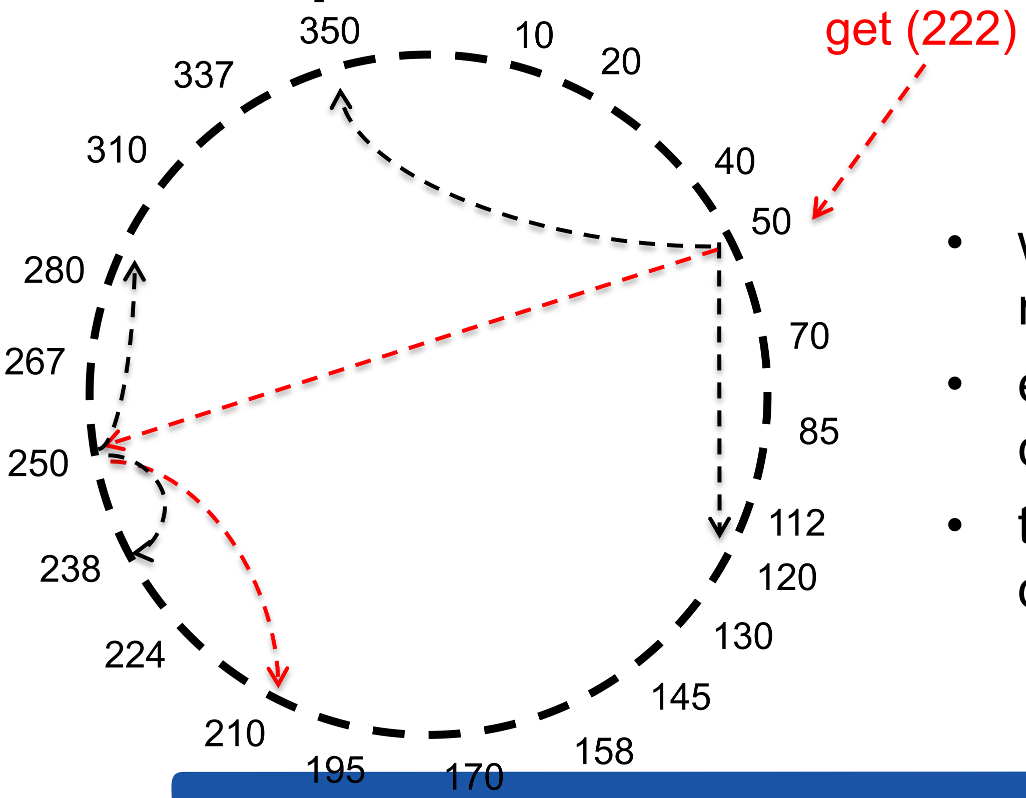
Assume that each node holds a leaf set of its closest ($\pm l$) neighbors (a.k.a. a finger table).

We can jump l nodes in each routing step, but we still have O(n) complexity.

The leaf set is updated in $O(l)$.

The leaf set could be as small as only the immediate neighbors but is often chosen to be a handful.

Improvement



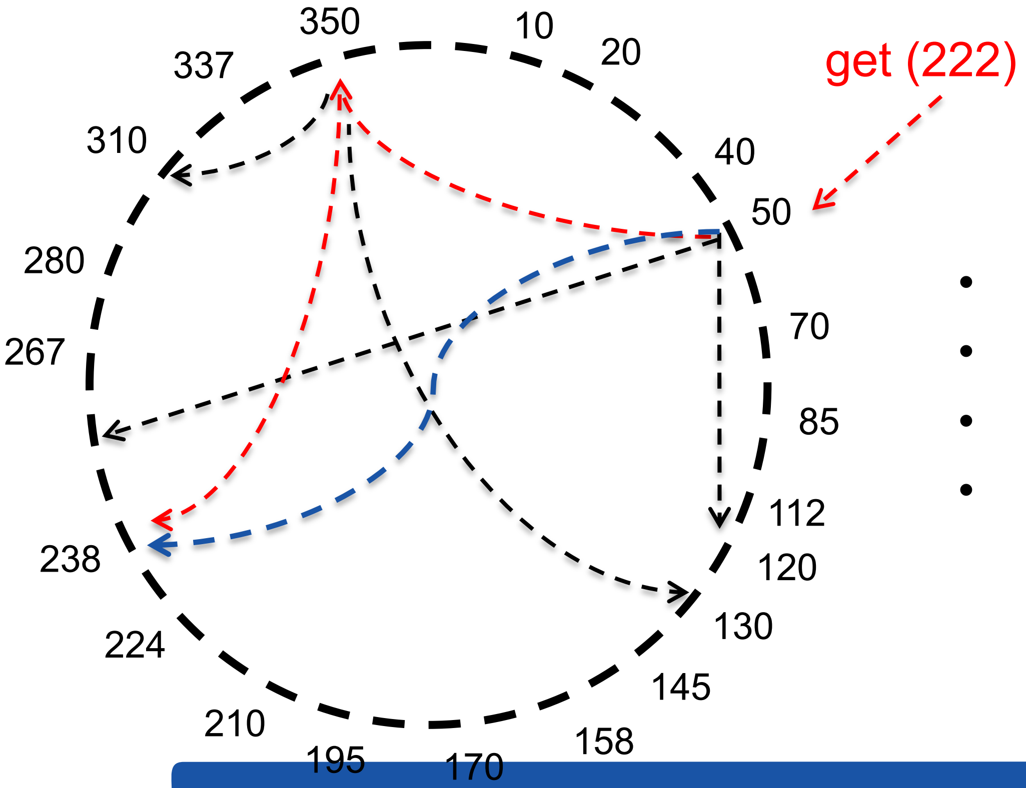
- we're looking for the responsible node of an object
- each router hop brings us closer to the responsible node
- the *leaf set* gives us the final destination

Pastry

In a routing table, each row represents one level of routing.

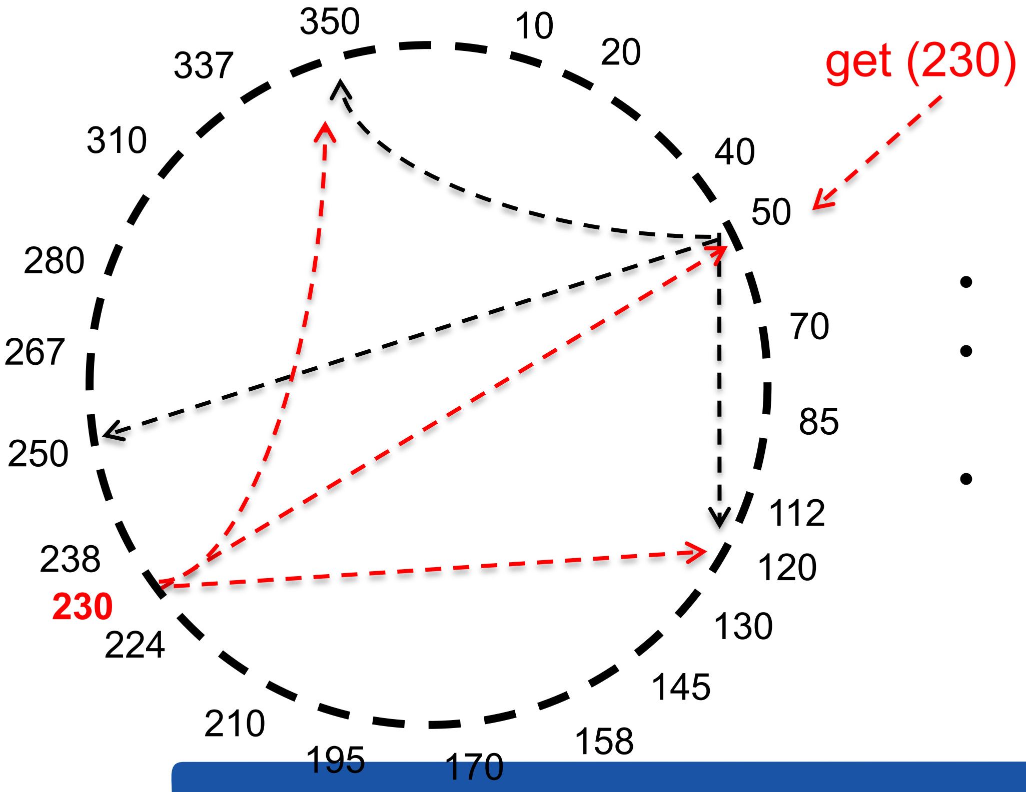
- 32 rows
- 16 entries per row
- any node found in 32 hops
- maximal number of nodes 16^{32} or 2^{128} (more than enough)
- Search is $O(log(n))$, where n is the number of nodes

The price of fast routing



- be lazy
- detect failed nodes when used
- route in the alternative direction
- ask neighbors of alternative node

Network aware routing



- when inserting a new node
- attach to the network-wise closest node
- adopt the routing entries on the way down

Overlay networks

Structured

- a well-defined structure
- takes time to add or delete nodes
- takes time to add objects
- easy to find objects

Unstructured

- a random structure
- easy to add or delete nodes
- easy to add objects
- takes time to find objects

DHT usage

Large scale key-value store.

- fault tolerant system in the high churn rate environment
- high availability, low maintenance

The Pirate Bay

- replaces the tracker with a DHT
- clients connect as part of the DHT
- DHT keeps track of peers that share content

Riak

- large scale key-value store
- inspired by Amazon Dynamo
- implemented in Erlang

Summary DHT

- Why hashing?
- Distribute storage in the ring
- Replication
- Routing