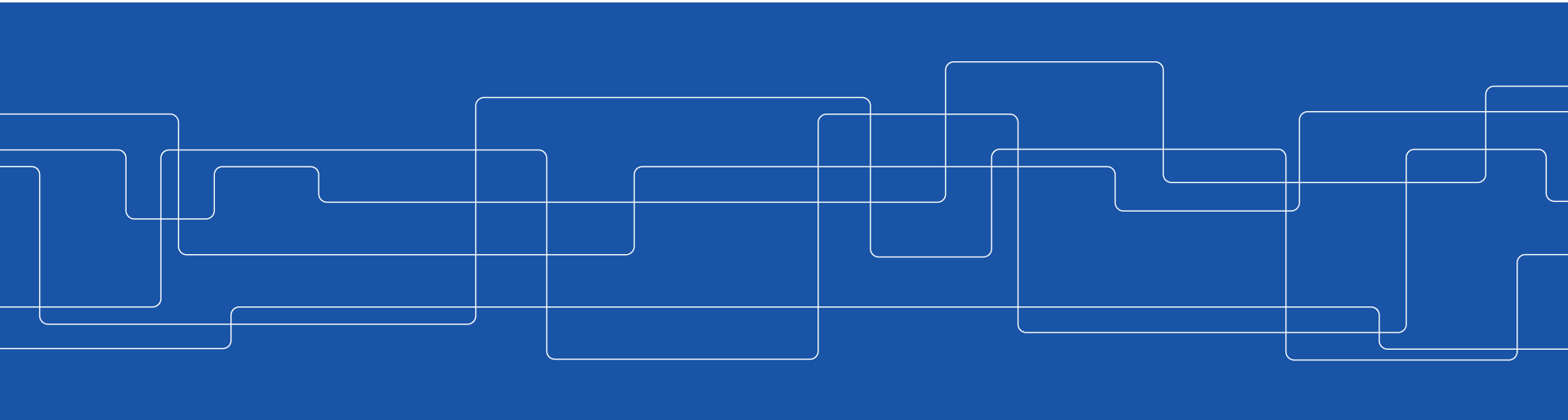




Distributed Hash Tables

Vladimir Vlassov





Distributed Hash Tables

- Large scale data bases
 - 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 a 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

$$\text{identifier} = f(\text{key}, \text{size_of_hash_table})$$

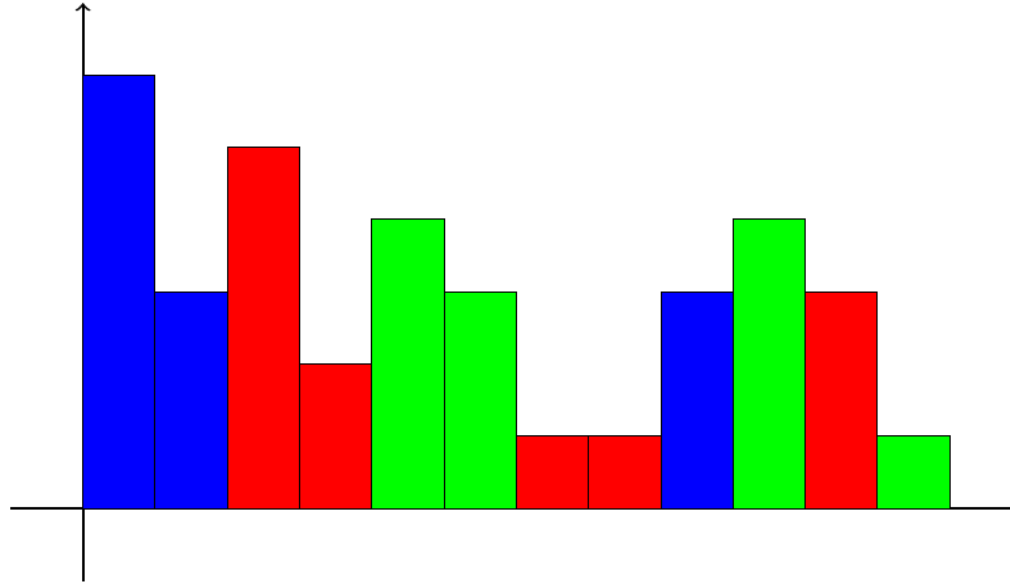
How to select identifiers:

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

why hash?

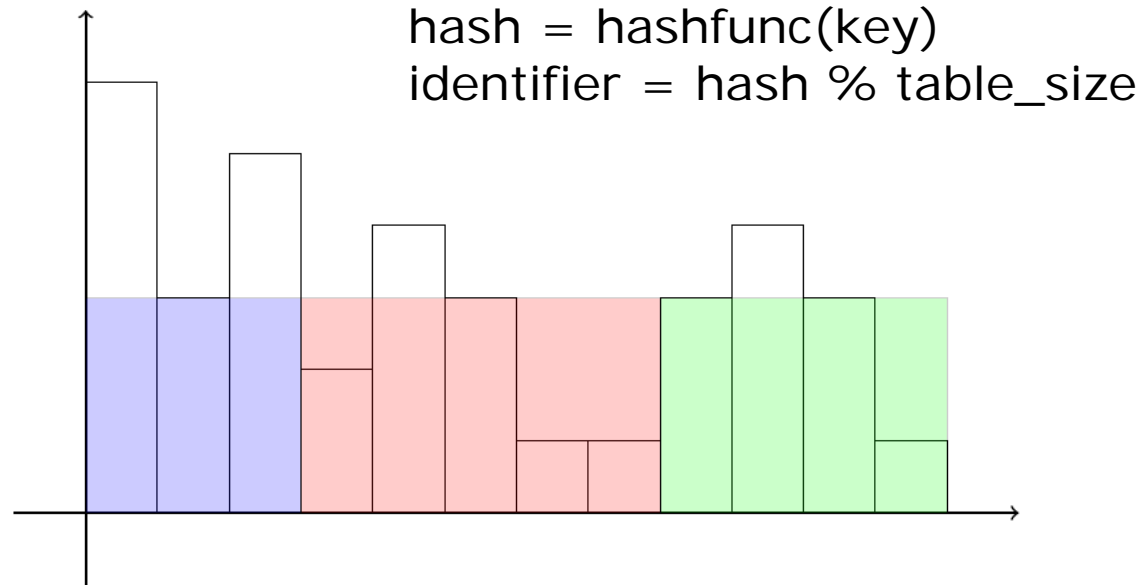
Key distribution – direct map

Direct map of keys to identifiers (buckets) gives a non-uniform (uneven) distribution of keys among buckets



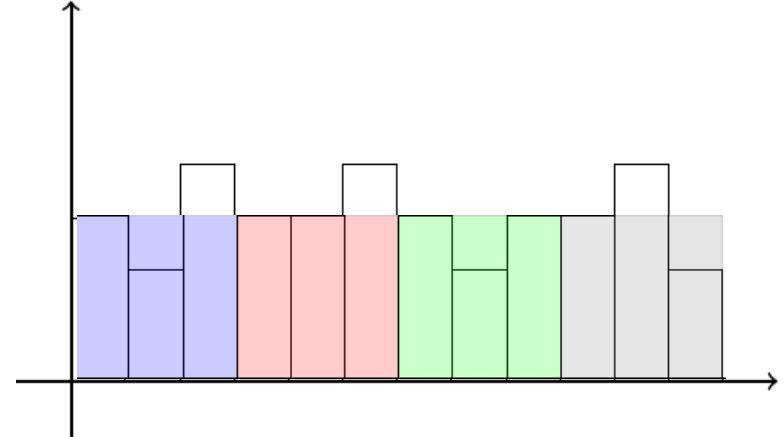
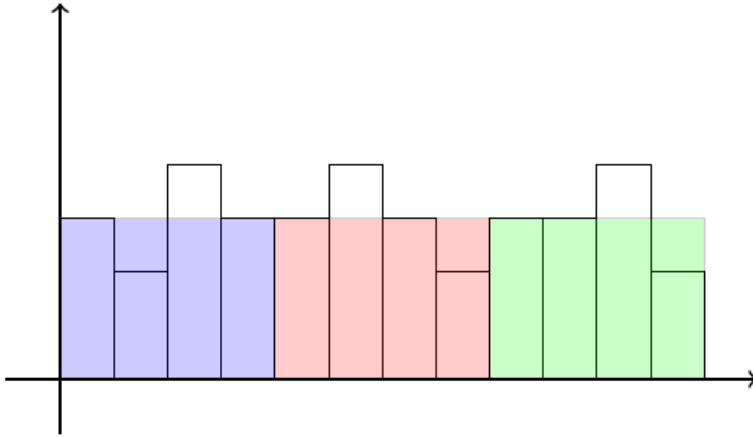
Key distribution – hashing keys

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



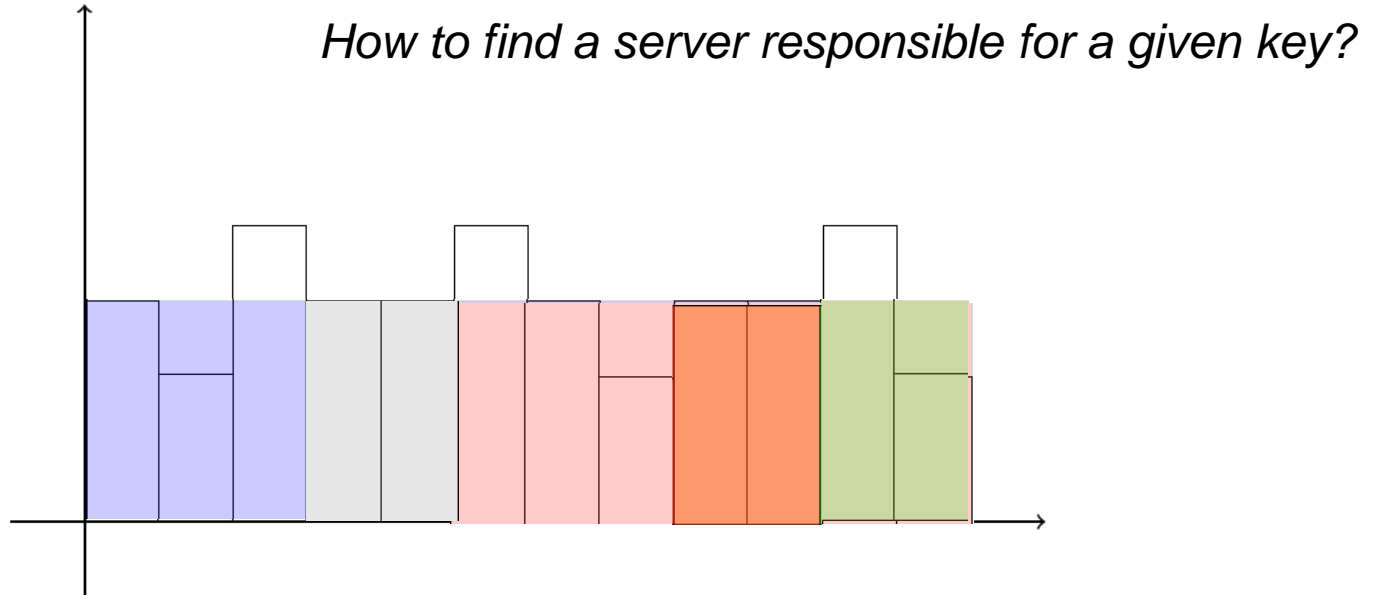
Add a server

at three-o'clock-in-the-morning do:

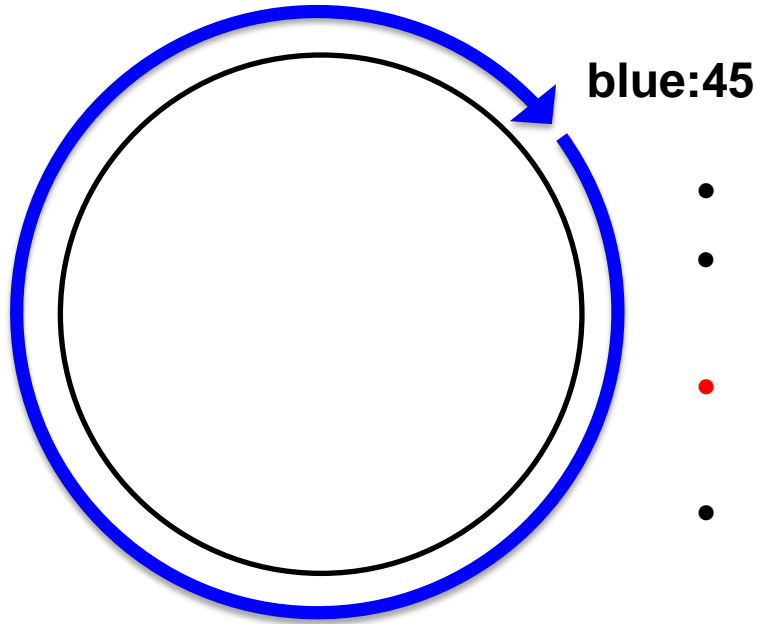


Random distribution

Random distribution of key ranges among servers

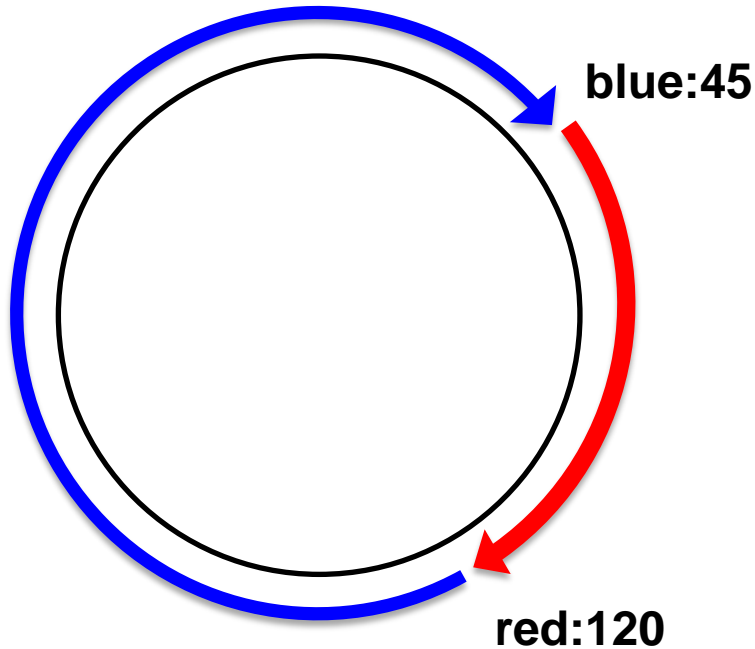


Circular domain



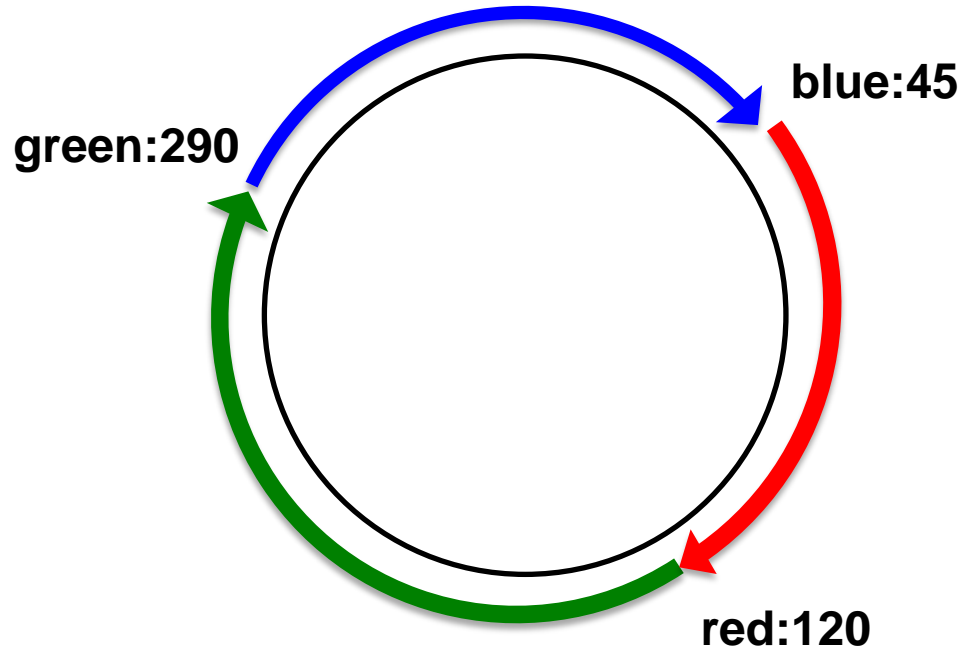
- ID domain: $0, 1, 2, \dots, \text{size}-1$
- clockwise step along the ring
$$i = (i + 1) \% \text{size}$$
- **responsibility**: from your predecessor to your number
- when inserted: take over responsibility

Circular domain



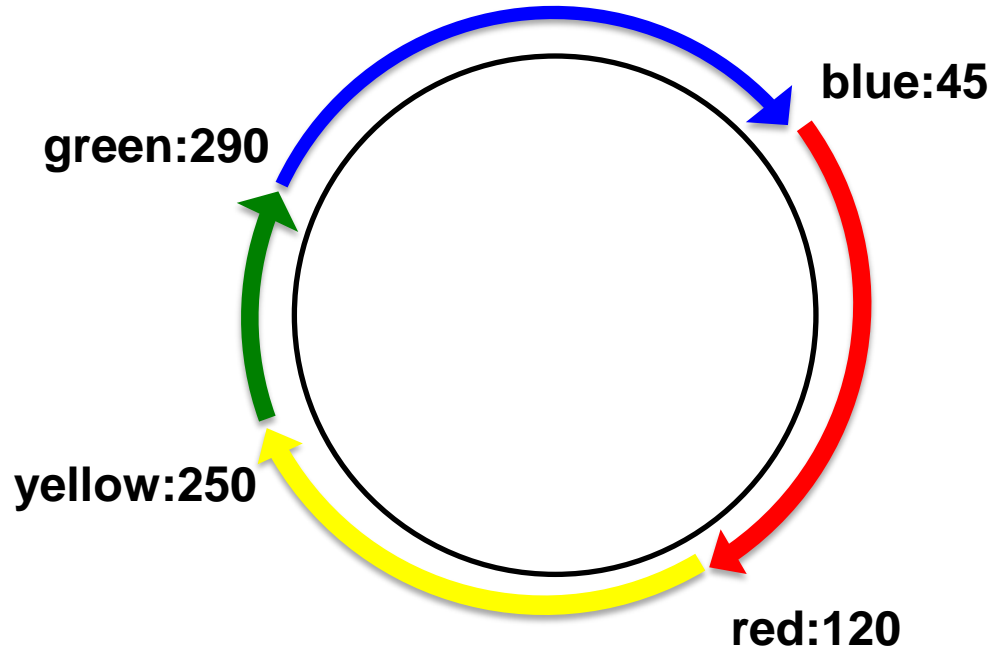
- *responsibility*: from your predecessor to your number
- when inserted: take over responsibility

Circular domain



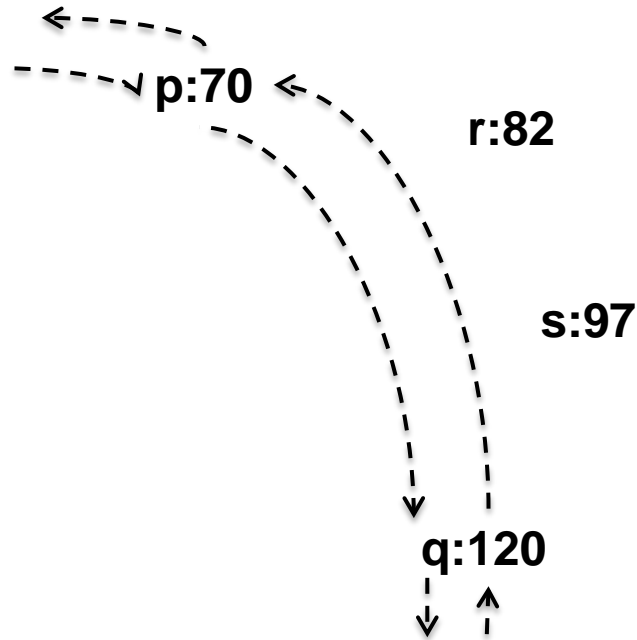
- *responsibility*: from your predecessor to your number
- when inserted: take over responsibility

Circular domain



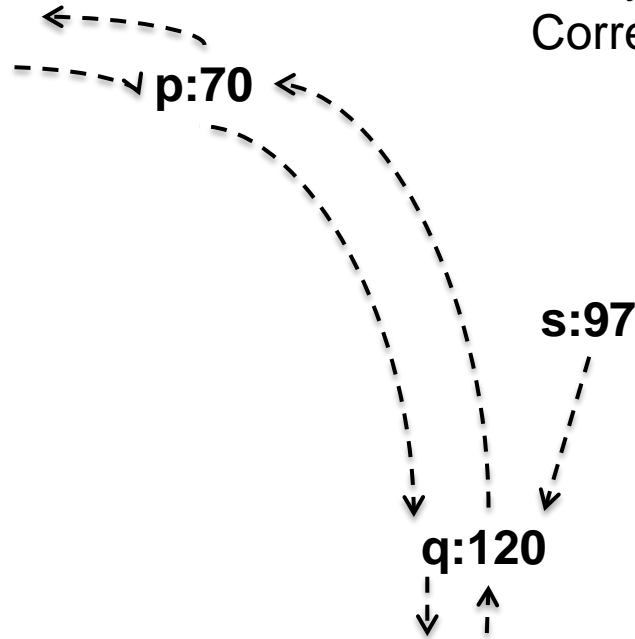
- *responsibility*: from your predecessor to your number
- when inserted: take over responsibility
- talk to the node in front of you

Double linked circle



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

Stabilization



Ask your successor: **Who is your predecessor?**

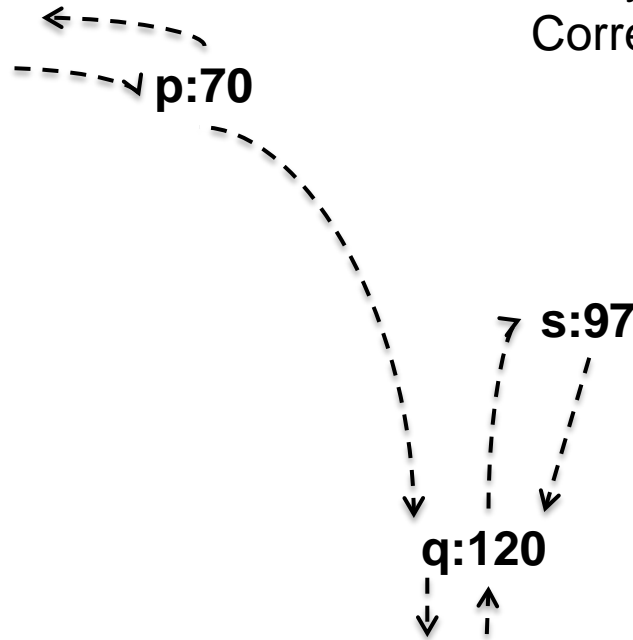
Correct a wrong link if any

s: - Who is your predecessor?

q: - It's p at 70.

s: - Why don't you point to me!

Stabilization



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

s: - Who is your predecessor?

q: - It's p at 70.

s: - Why don't you point to me!

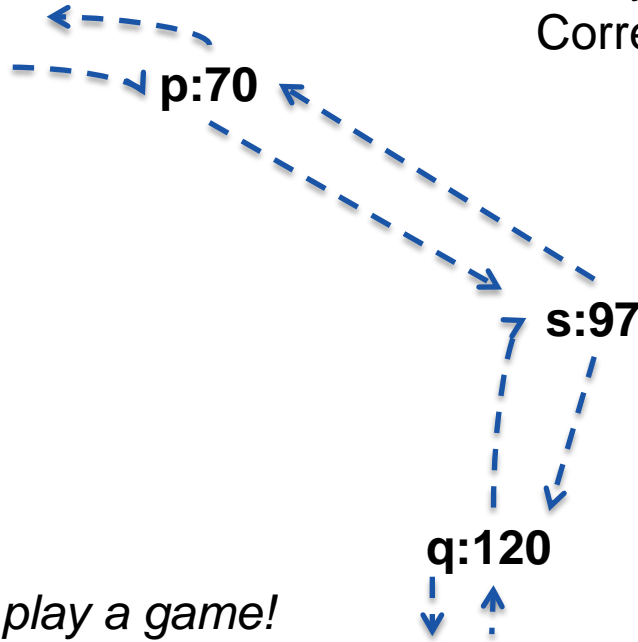
p: - Who is your predecessor?

q: - It's s at 97.

p: - Hmmm, that's a better successor.

Stabilization

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



s: - Who is your predecessor?

q: - It's p at 70.

s: - Why don't you point to me!

p: - Who is your predecessor?

q: - It's s at 97.

p: - Hmmm, that's a better successor.

p: - Who is your predecessor?

s: - I don't have one.

p: - Why don't you point to me!

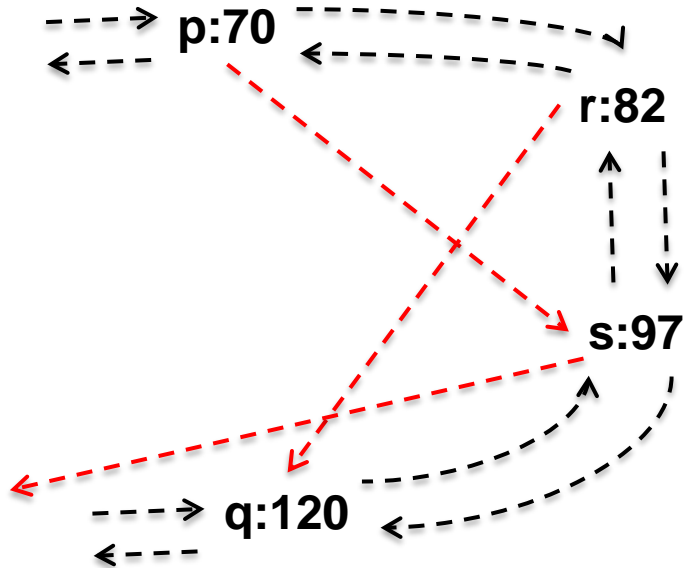


Stabilization

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

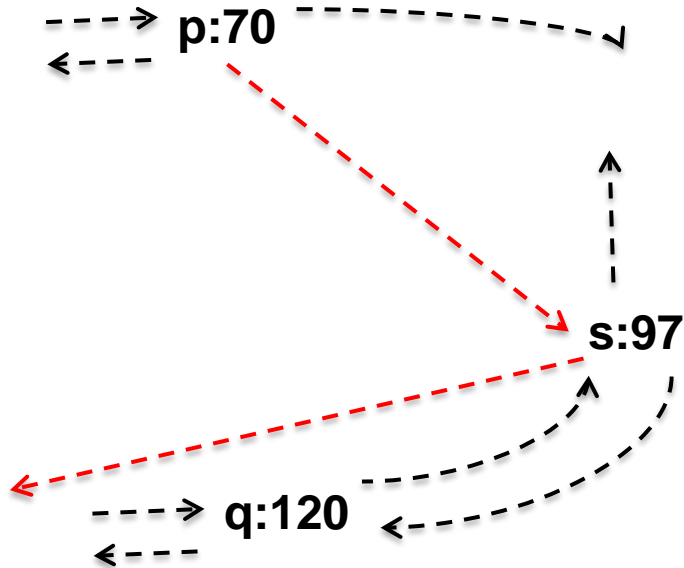
Inserted node will take over responsibility for part of a segment.

Crashing nodes



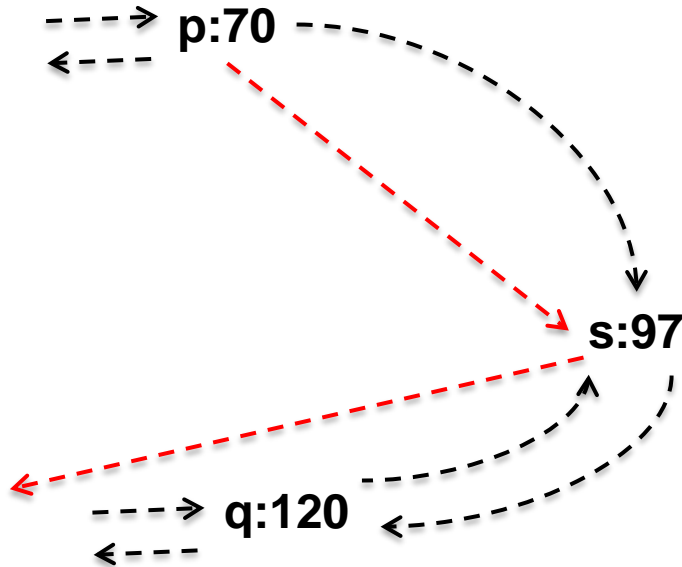
- monitor neighbors
- **safety pointer**

Crashing nodes



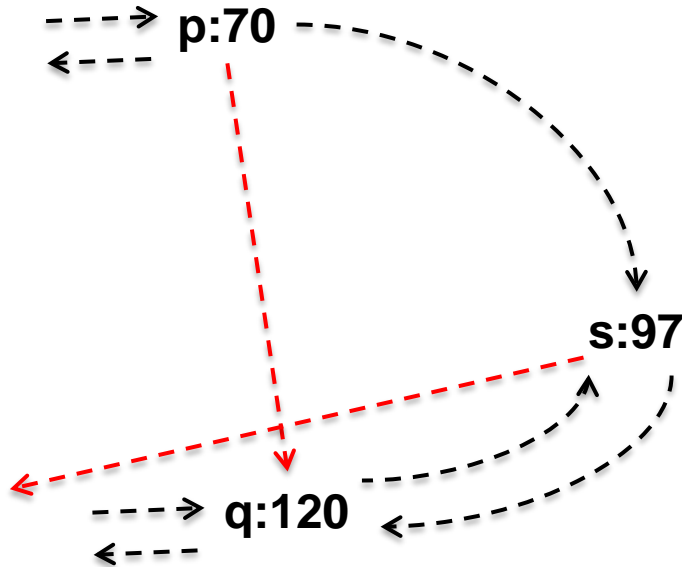
- monitor neighbors
- safety pointer
- detect crash

Crashing nodes



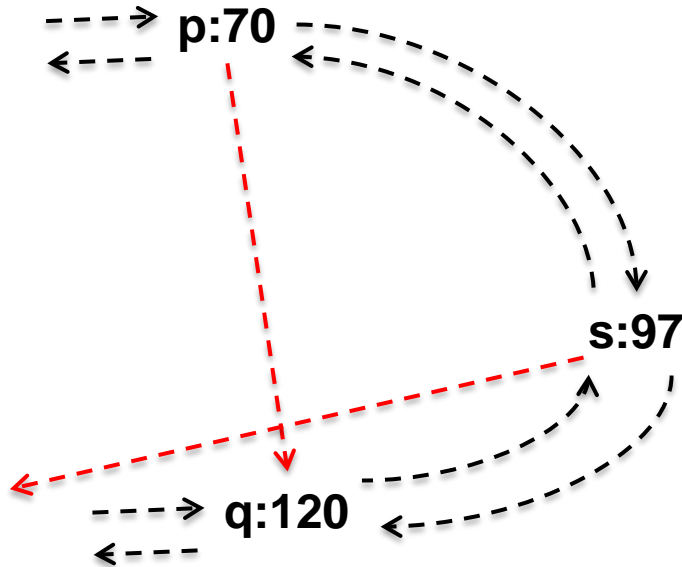
- monitor neighbors
- safety pointer
- detect crash
- **update forward pointer**

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**

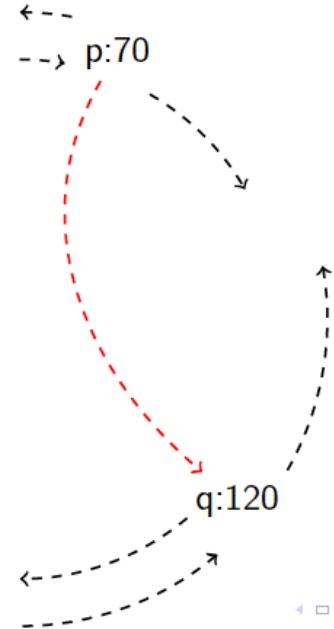
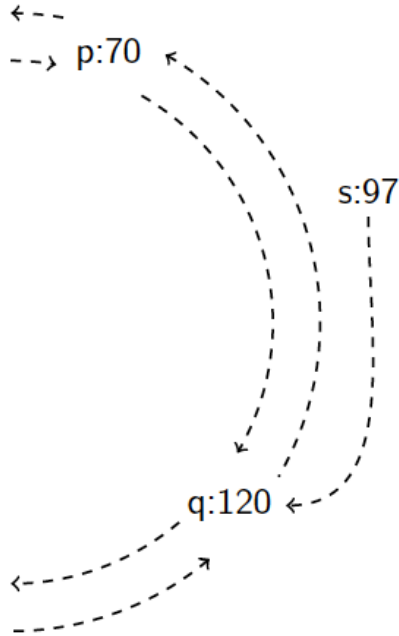


Russian roulette

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
 - trade-off between routing overhead and update overhead

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



Leaf set

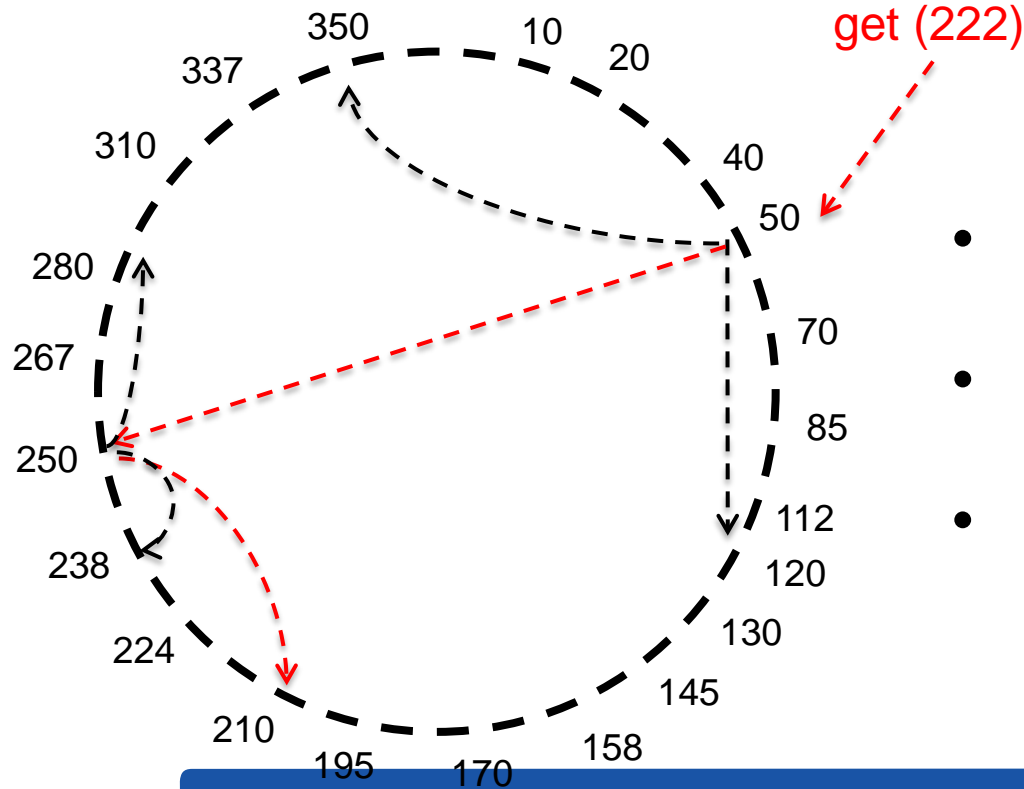
Assume that each node holds a leaf set of its closest ($\pm l$) neighbors (a.k.a. a finder table).

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

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

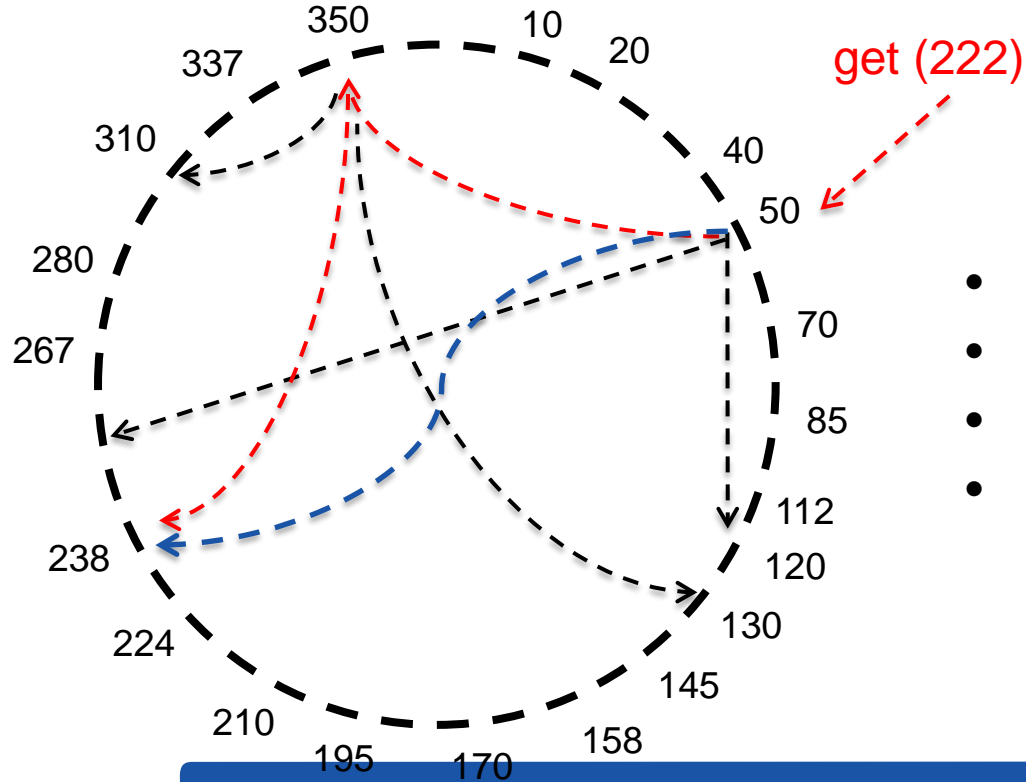


Pastry

A routing table, each row represents one level of routing.

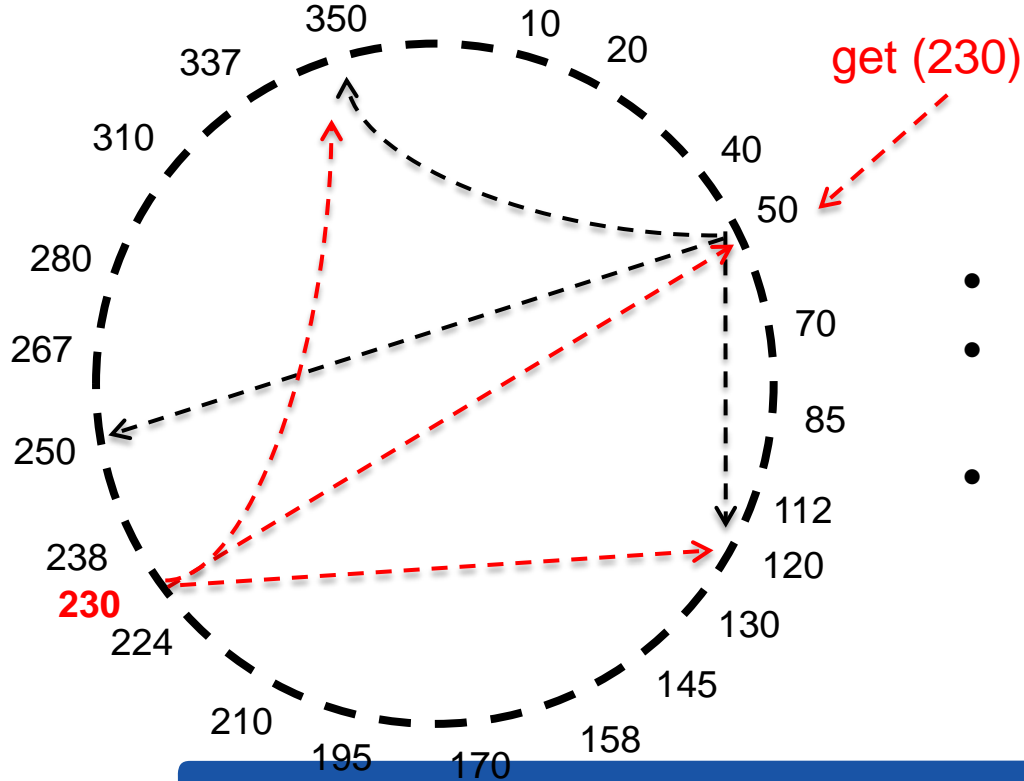
- 32 rows
- 16 entries per row
- any node found in 32 hops
- maximal number of nodes 16^{32} or 2^{128} (more than enough)
- search is $O(\lg(n))$ where n is number of nodes

The price of fast routing



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

Network aware routing



- when inserting 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 high churn rate environment
- high availability low maintenance

The Pirate Bay



The Pirate Bay

- replaces the tracker by a DHT
- clients connects as part in 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 ring
- replication
- routing