Large Scale Distributed Systems

José Orlando Pereira

Departamento de Informática
Universidade do Minho

Searching

- Store and translate arbitrary keys to values (i.e. Map<K,V>)
  - Large number of (k,v) pairs
  - Large number *n* of nodes

- Abstracted to Map<byte[], InetSocketAddress>
  - Key is hashed to a binary string (e.g., SHA-1 with 160 bits) with a uniform random distribution
  - Value is the the address of the node holding (k,v)

Flooding

- Broadcast query to all nodes using an epidemic algorithm
  - Nodes involved in each query: O(n)
  - State in each node for the overlay network: O(log n)

- Not scalable as the number of queries grows
- Idea: Route each query towards the correct node… How?

Naive distributed hashing

- Our key is already randomly distributed
- 1 node = 1 bucket
  - (k mod n) gives the bucket number
  - Nodes involved in each query: O(1)
- Need to map buckets to server addresses
  - Routing state: O(n)

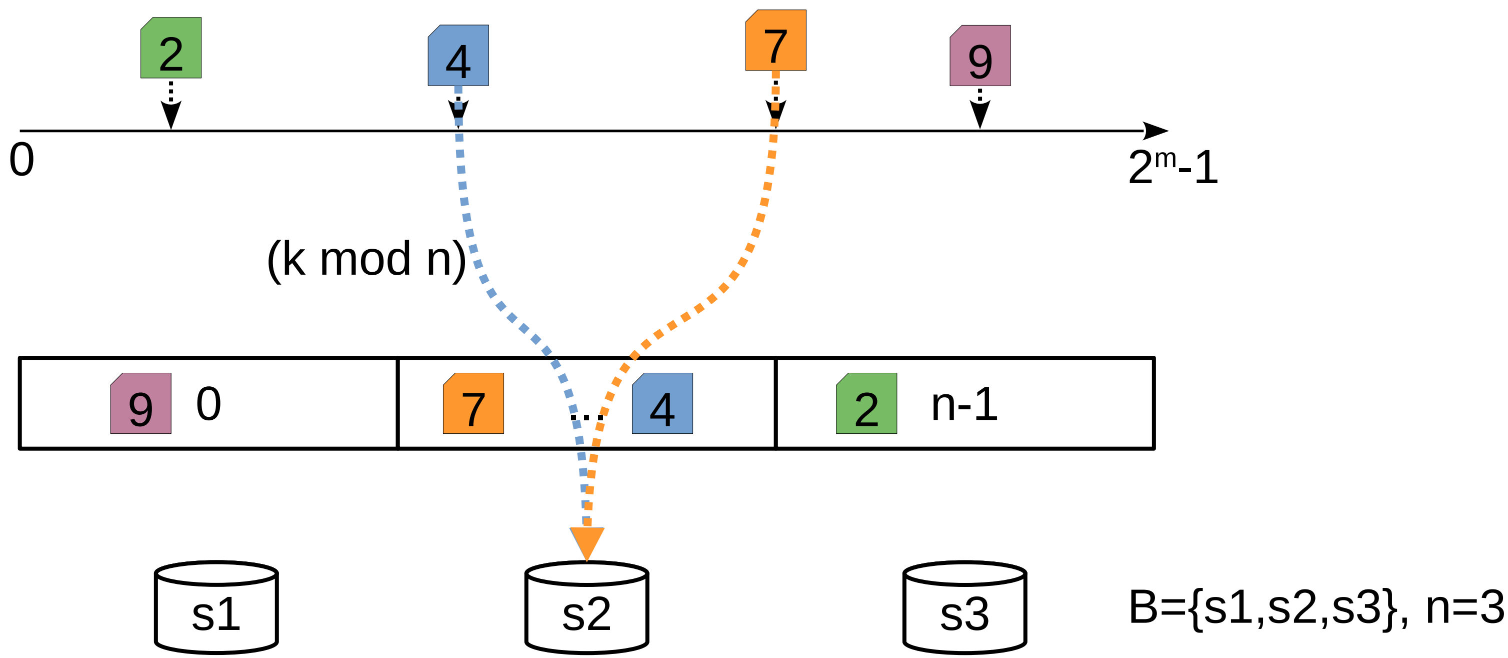
- Node churn: How to update data placement and routing information when nodes enter/leave?

Hashing properties

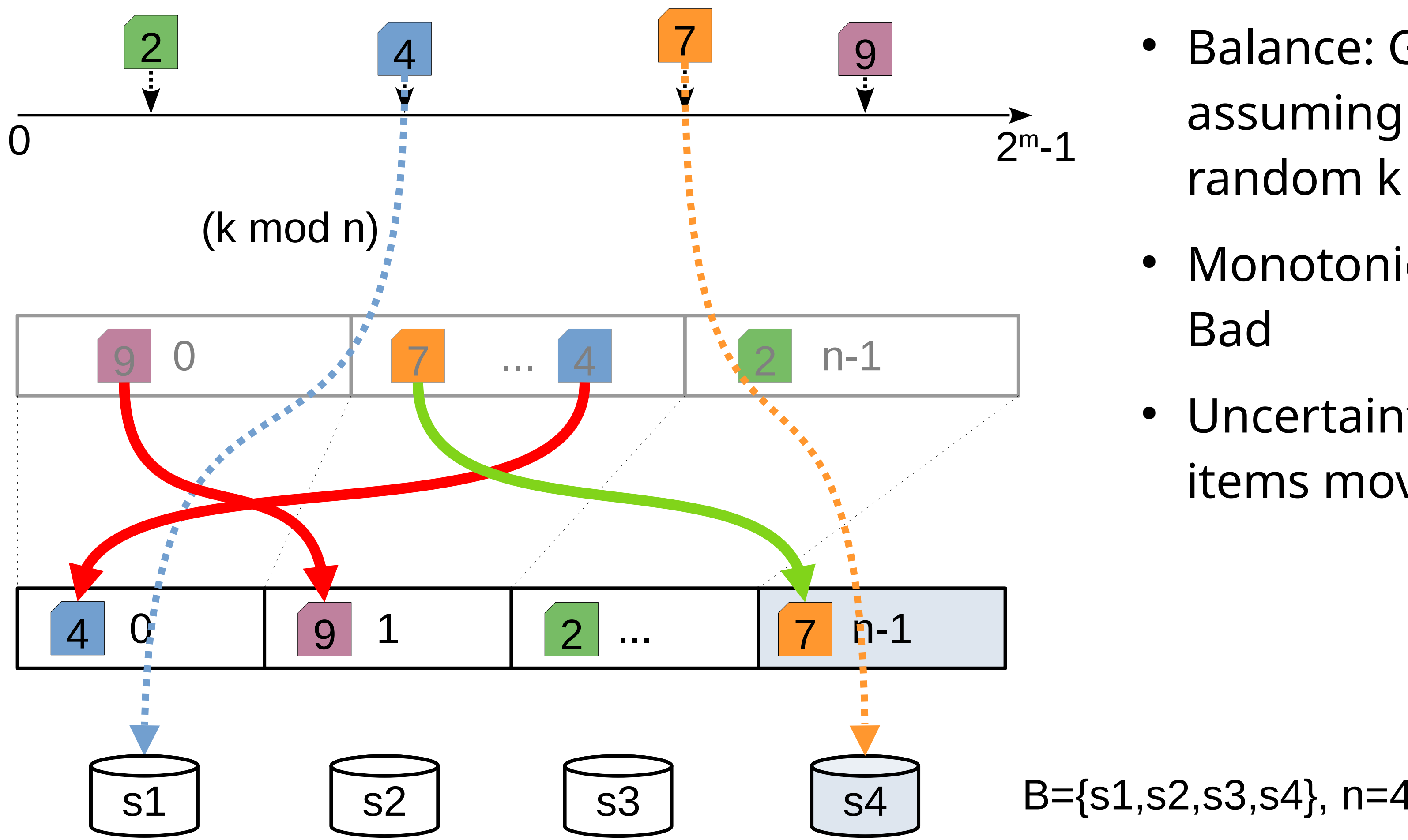
- Balance: Proportion of items in each bucket should be O(1/n)
- Monotonicity: When there is a new bucket, items are moved only to that new bucket (not between old buckets)

- In a distributed system, there is uncertainty about location of keys:
  - Wrong opinions about some key
  - Wrong opinions about some bucket

Naive distributed hashing



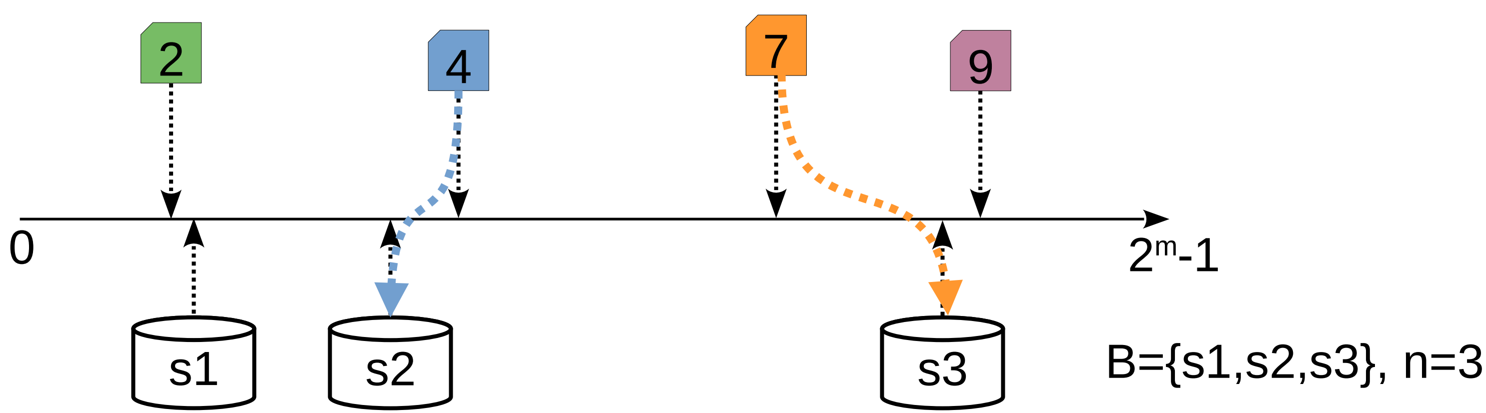
Naive distributed hashing



- Balance: Good, assuming random k
- Monotonicity: Bad
- Uncertainty while items move

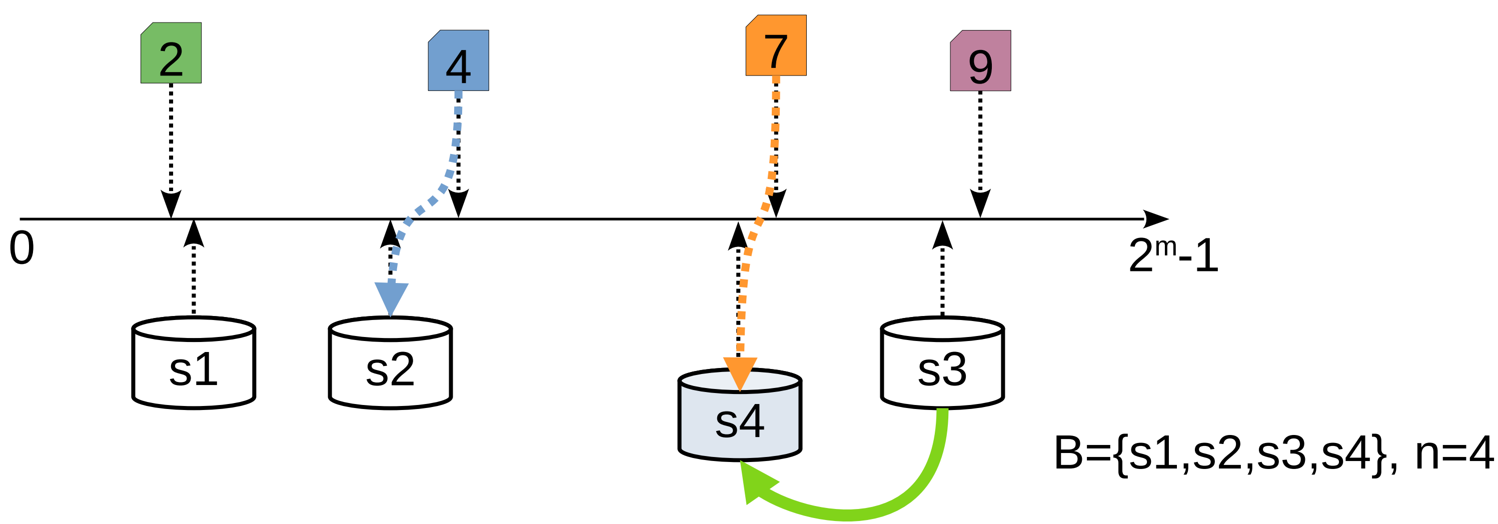
Consistent hashing (key idea)

- Hash keys and bucket ids into the same space and <u>assign by distance</u>
- Balance is not perfect: Distribution and extremes



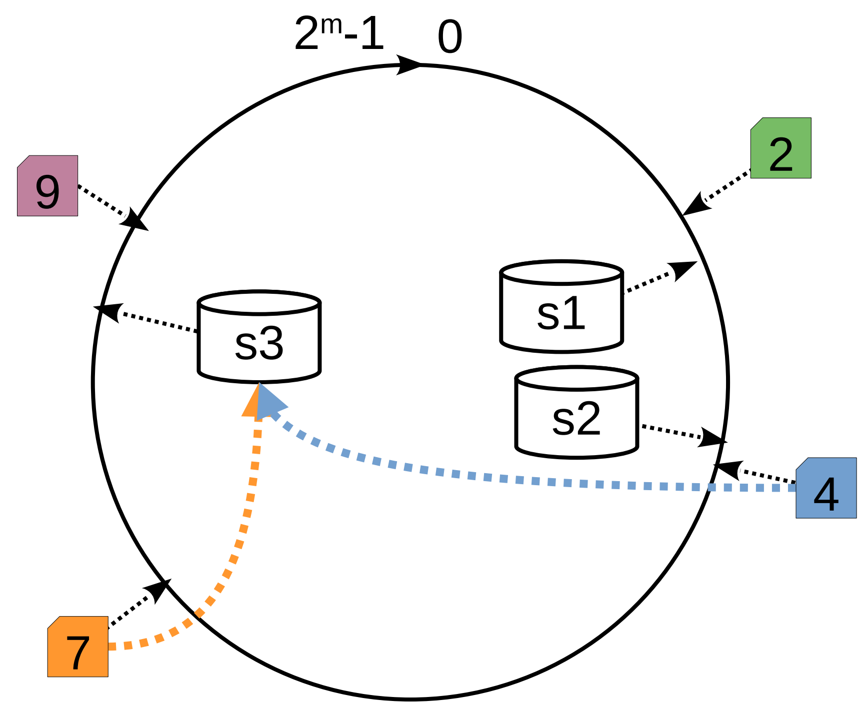
Consistent hashing (key idea)

- Monotonicity: Good! Keys are moved only to the new bucket
- Less uncertainty but all keys move to/from at most two other buckets



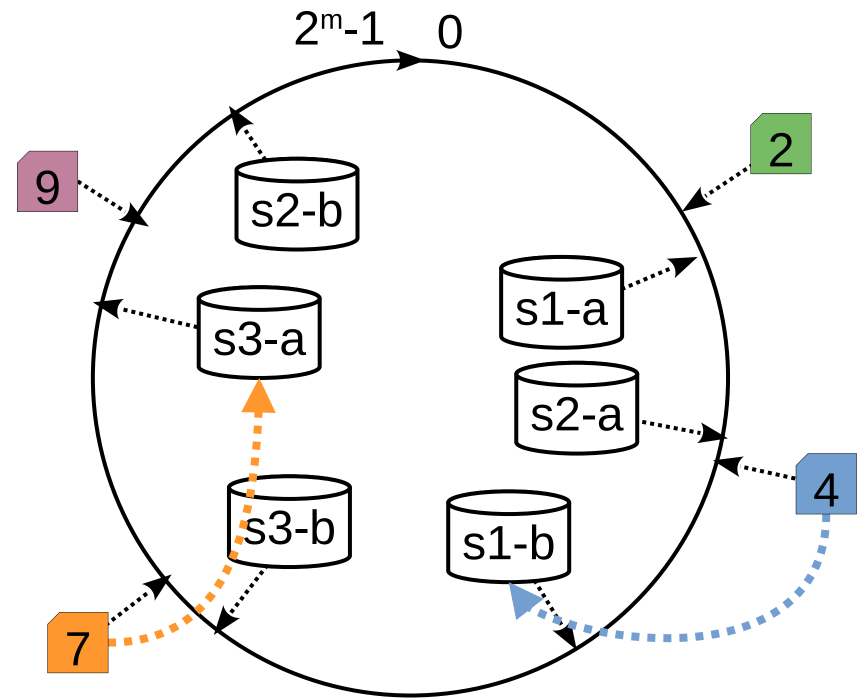
Consistent hashing

- Assume a circular space:
  - mod 2^m
- Clockwise distance
  - Successor bucket
  - Direct comparison, no need for arithmentic



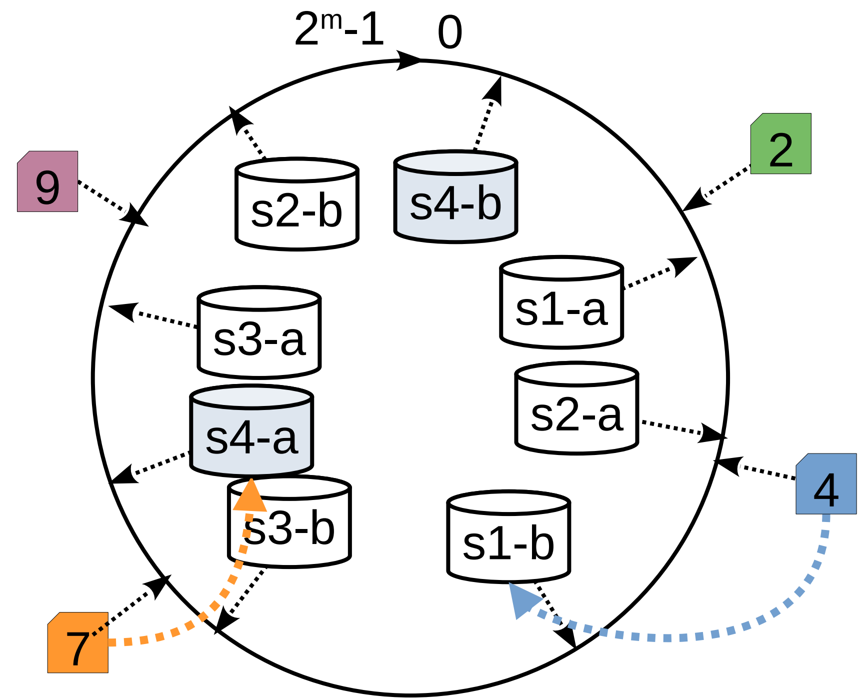
Consistent hashing

- Multiple buckets in each node (virtual nodes)
- Same mean, lower variance



Consistent hashing

- On change: Keys move to/from *k* other nodes



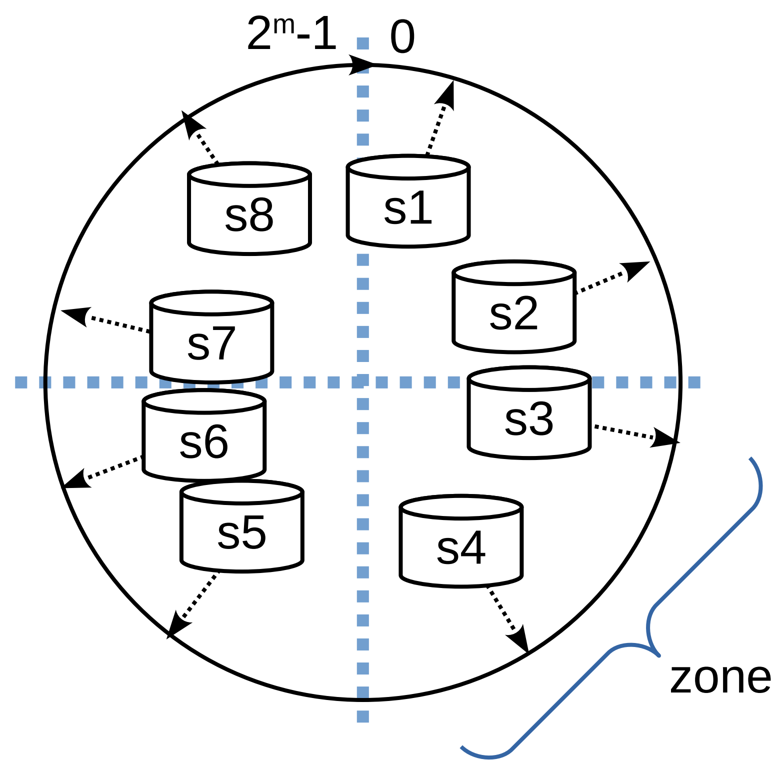
Consistent hashing

- Balance: Good balance with k ~ O(log n) buckets in each node
- Monotonicity: Good! Keys are moved only to the new bucket

- Nodes involved in each query: O(1)
- Routing state: O(n) or O(n log n) with virtual nodes

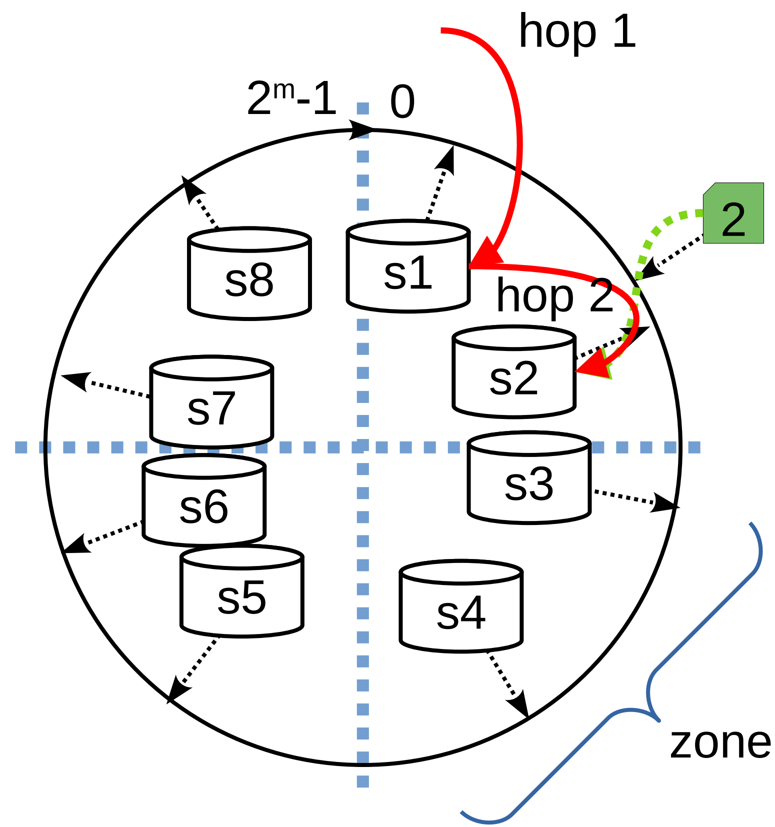
Patitioned routing state (Kelips)

- Split nodes in √n zones
- Each node keeps routing state for:
  - its zone
  - a few contact nodes in each other zone



Patitioned routing state (Kelips)

- Lookup:
  - 1 hop to some contact in zone
  - 1 hop to node
- Use epidemic dissemination to update routing tables within zones



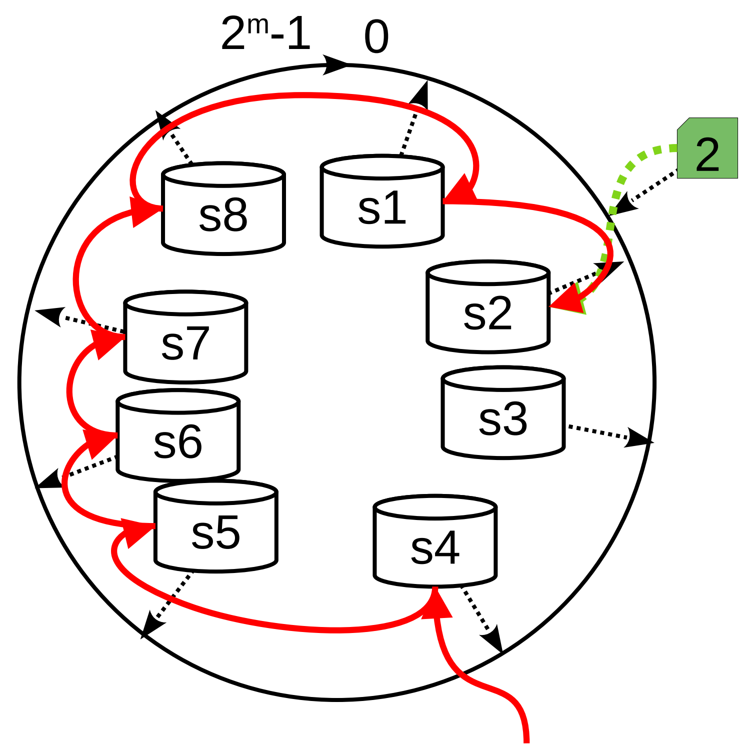
Patitioned routing state (Kelips)

- Balance and Monotonicity unchanged
- Nodes involved in each query: still $O(1)$
- Routing state: down to $O(\sqrt{n})$

- Background traffic to synchronize routing state in each zone
- Uncertainty, due to synchronization and lost contact nodes

Scalability

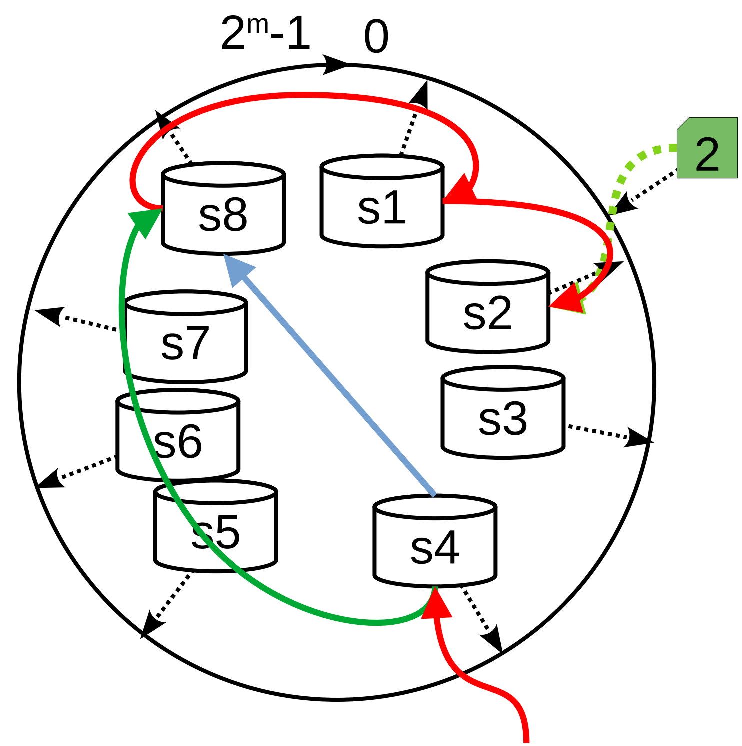
- Can we do O(1) state?
- Yes: Keep a pointer to the successor node
- Lookup is O(n)
  - average n/2 hops
- Fragile...



Scalability

- Idea: Keep a shortcut to "the other side of the ring"

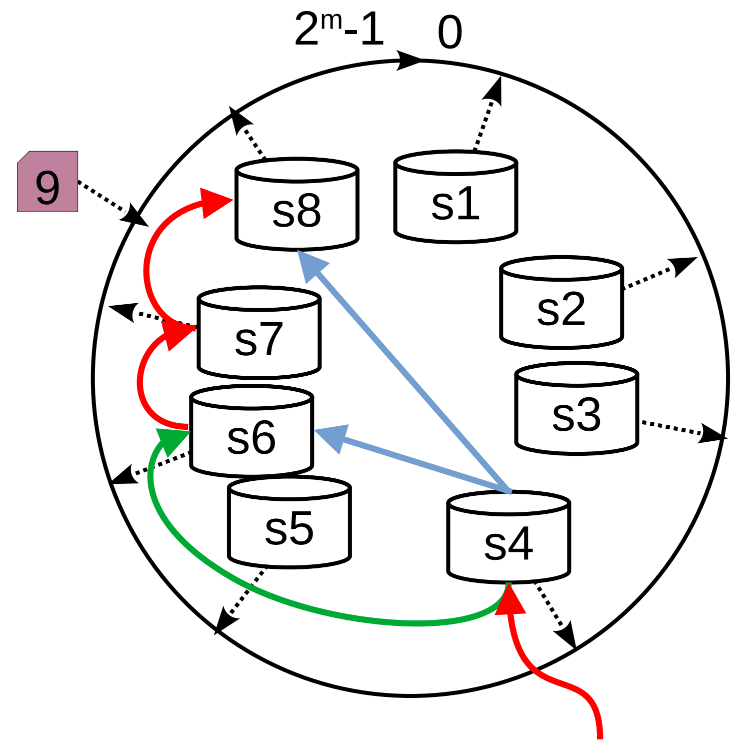
- Lookup is O(n)
  - 2 pointers
  - average n/4 hops



Scalability

- What about items in the first half?
- Split again
- Lookup is O(n)
  - 3 pointers
  - average n/8 hops!

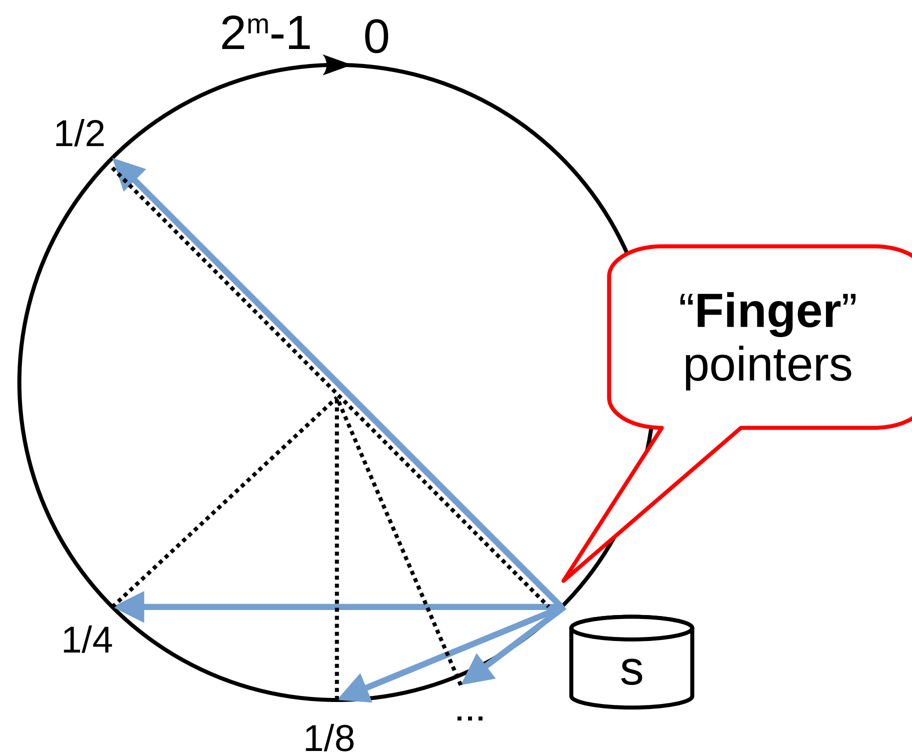
- Can use more than one shortcut...



Scalability

- How many times can we split it?
  - m!
- Routing state: O(m) ~ O(log n)
- Lookup: O(log n)

- How to setup and maintain these pointers?

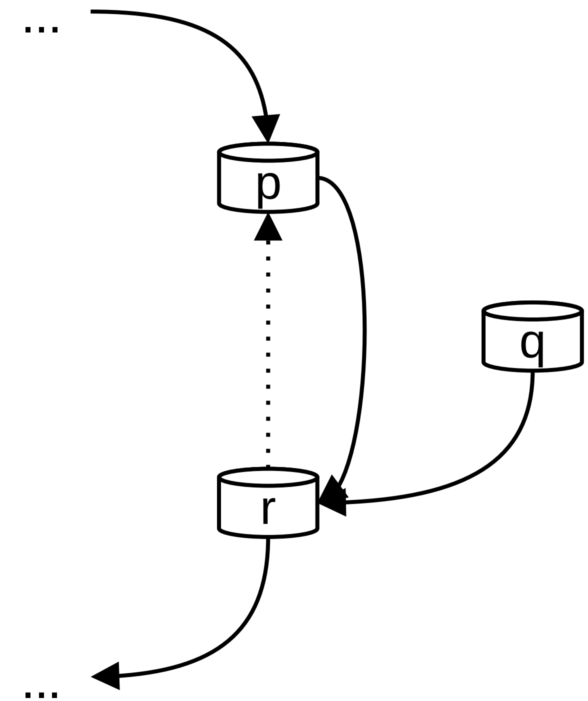


Chord

- Node p keeps finger pointers for:
  - $q = succ(p + 2^i)$, for $0 \leq i < m$
  - $i = 0$ gives the direct successor of p
- Lookup succ(k) at p:
  - if k in local interval, then return p
  - else forward to $q = succ(p + 2^i)$ such that:
    - if exists, largest i with $q \leq k$
    - else with $i = 0$

Chord

- A node q joins the ring by looking up its successor r
- Uses this to set its first pointer (i = 0)

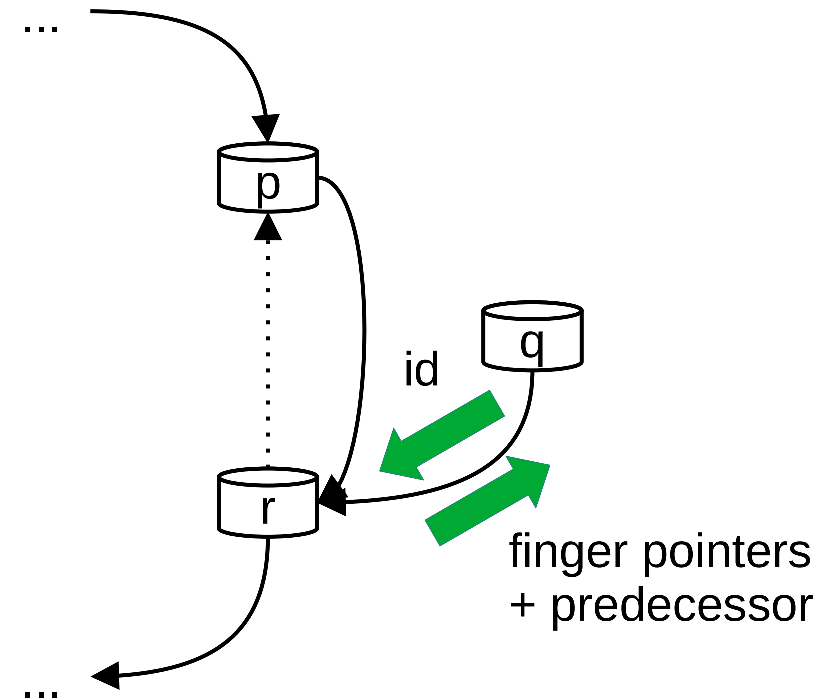


Chord

- The ring is repaired by running two procedures:
  - Stabilize: Runs periodically and exchanges information with the (currently best) successor:
    - Informs successor of a possible new predecessor
    - Obtains or updates finger pointers, possibly discovering a better successor
  - Rectify: Runs when a new predecessor is discovered and selects the candidate with the largest value

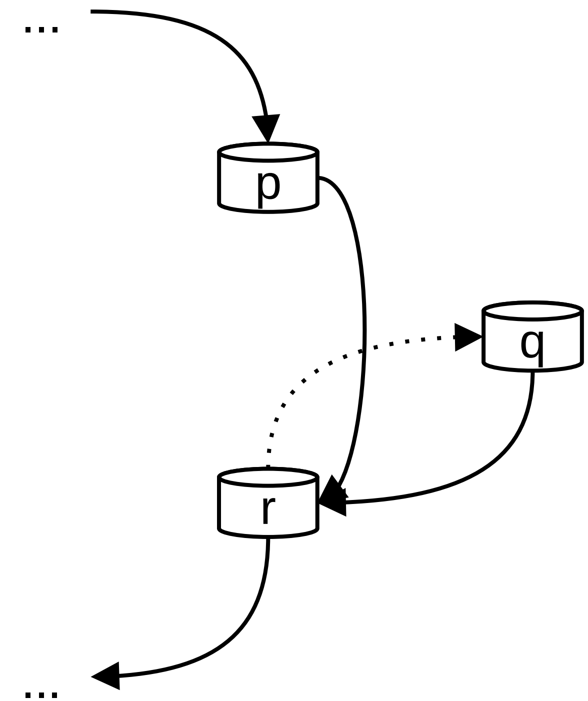
Chord

- Node q <u>stabilizes</u>:
  - Informs r of its id
  - Learns the successor's predecessor p and adopts it as sucessor if p > q
    - Not in this case…
  - Learns r's pointer table and initializes its own



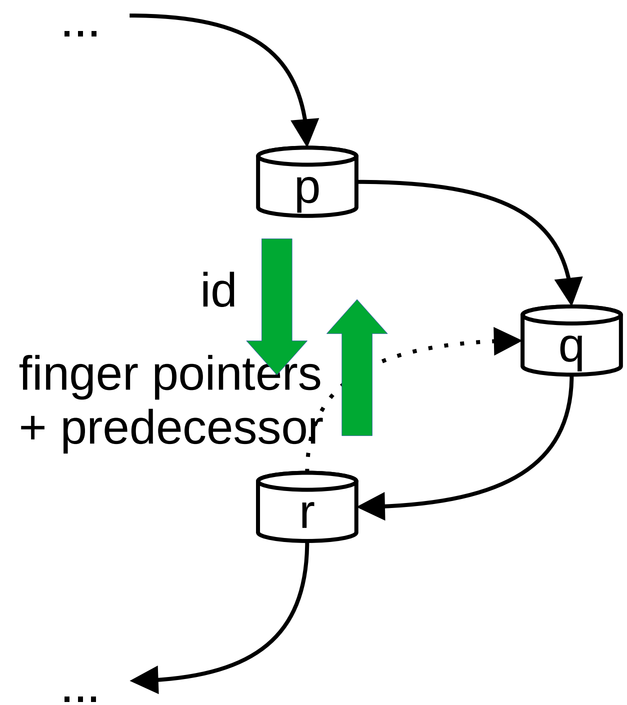
Chord

- Upon learning of a new predecessor, node r rectifies:
  - checks if q > p and sets predecessor pointer to q



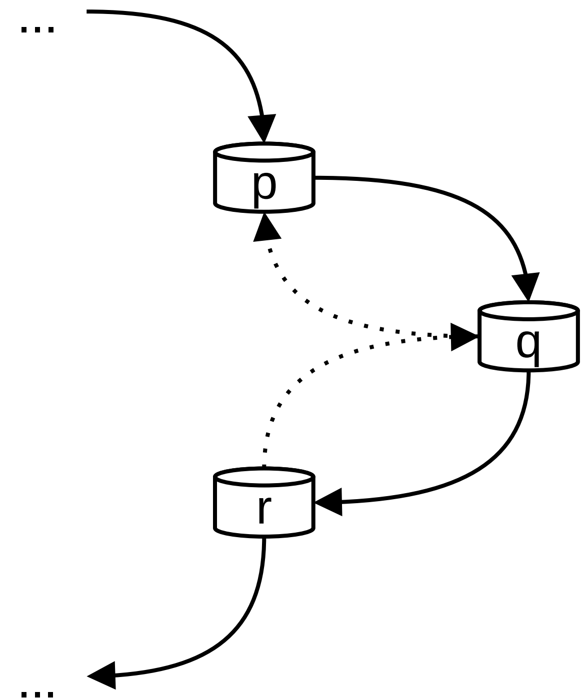
Chord

- Node p stabilizes:
  - Informs r of its id
  - Learns the successor’s predecessor q and adopts it as sucessor if q > p
    - Yes in this case!
  - Learns r’s pointer table and updates its own



Chord

- Upon learning of a new predecessor, node q <u>rectifies</u>:
  - no current predecessor, so it adopts p



Chord

- If q fails or leaves: Periodical stabilization at p will repair the ring

- Why does it work?
  - Stabilize selects the smallest of successor candidates
  - Rectify selects the largest of predecessor candidates
  - It converges to the correct order as long as there are no ties (i.e. nodes with the same id)

Chord

- If state is volatile, it is lost when a node fails or disconnects
  - Caching
- If not, it needs to be replicated to f+1 nodes
  - Each node keeps f+1 predecessor pointers
  - State is replicated forward by owner node
- Note that replication also improves Balance, in the same way as virtual nodes!

Summary

	Balance	**Monotonicity**	**Lookup**	**State**
Naive hashing	Perfect	No	O(1)	O(n)
Consistent hashing	Good(*)	Yes	O(1)	O(n)
Kelips DHT	Good(*)	Yes	O(1)	O(√n)
Sequential cons. hashing	Good(*)	Yes	O(n)	O(1)
Chord DHT	Good(*)	Yes	O(log n)	O(log n)

(*) Very good at the expense of ×(log n) state

- Typical choices:
  - Consistent hashing in the medium / data center scale
  - Chord (and Kademlia) DHTs in the large scale

References

- D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in ACM Symposium on Theory of computing, 1997 (Mainly Section 4) https://www.cs.princeton.edu/courses/archive/fall09/cos518/papers/chash.pdf
- I. Stoica et al., "Chord: a scalable peer-to-peer lookup protocol for internet applications," IEEE/ACM Trans. Netw., vol. 11, no. 1, pp. 17–32, 2003 http://dx.doi.org/10.1109/tnet.2002.808407

More…

- I. Gupta, K. Birman, P. Linga, A. Demers, and R. van Renesse, “Kelips: Building an Efficient and Stable P2P DHT through Increased Memory and Background Overhead,” Peer-to-Peer Systems II. pp. 160–169, 2003 http://dx.doi.org/10.1007/978-3-540-45172-3_15
- P. Zave, “How to Make Chord Correct,” arXiv [cs.DC], Feb. 23, 2015 http://arxiv.org/abs/1502.06461
- P. Maymounkov and D. Mazières, “Kademlia: A Peer-to-Peer Information System Based on the XOR Metric,” in Peer-to-Peer Systems, 2002 http://dx.doi.org/10.1007/3-540-45748-8_5