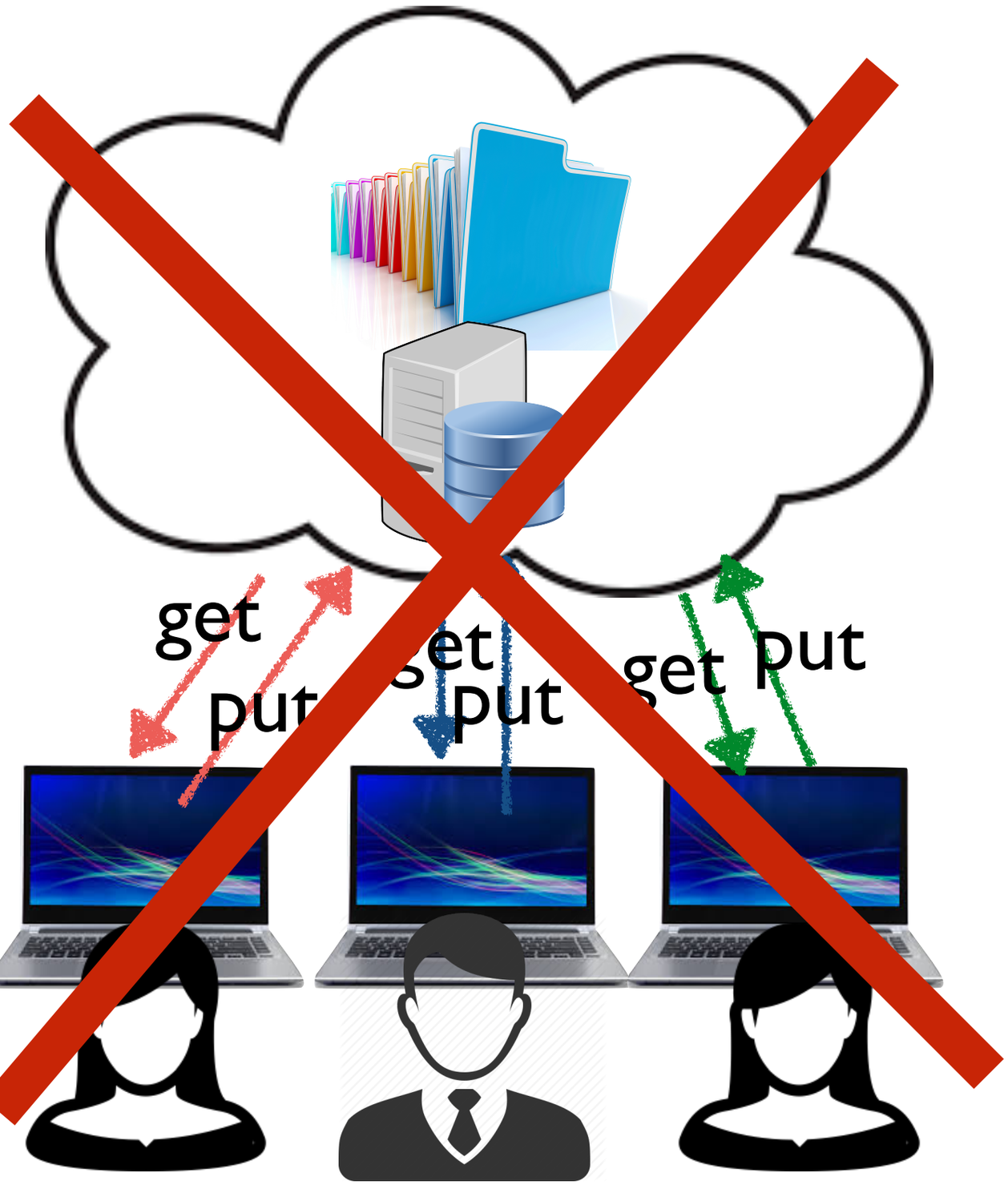
Distributed Systems

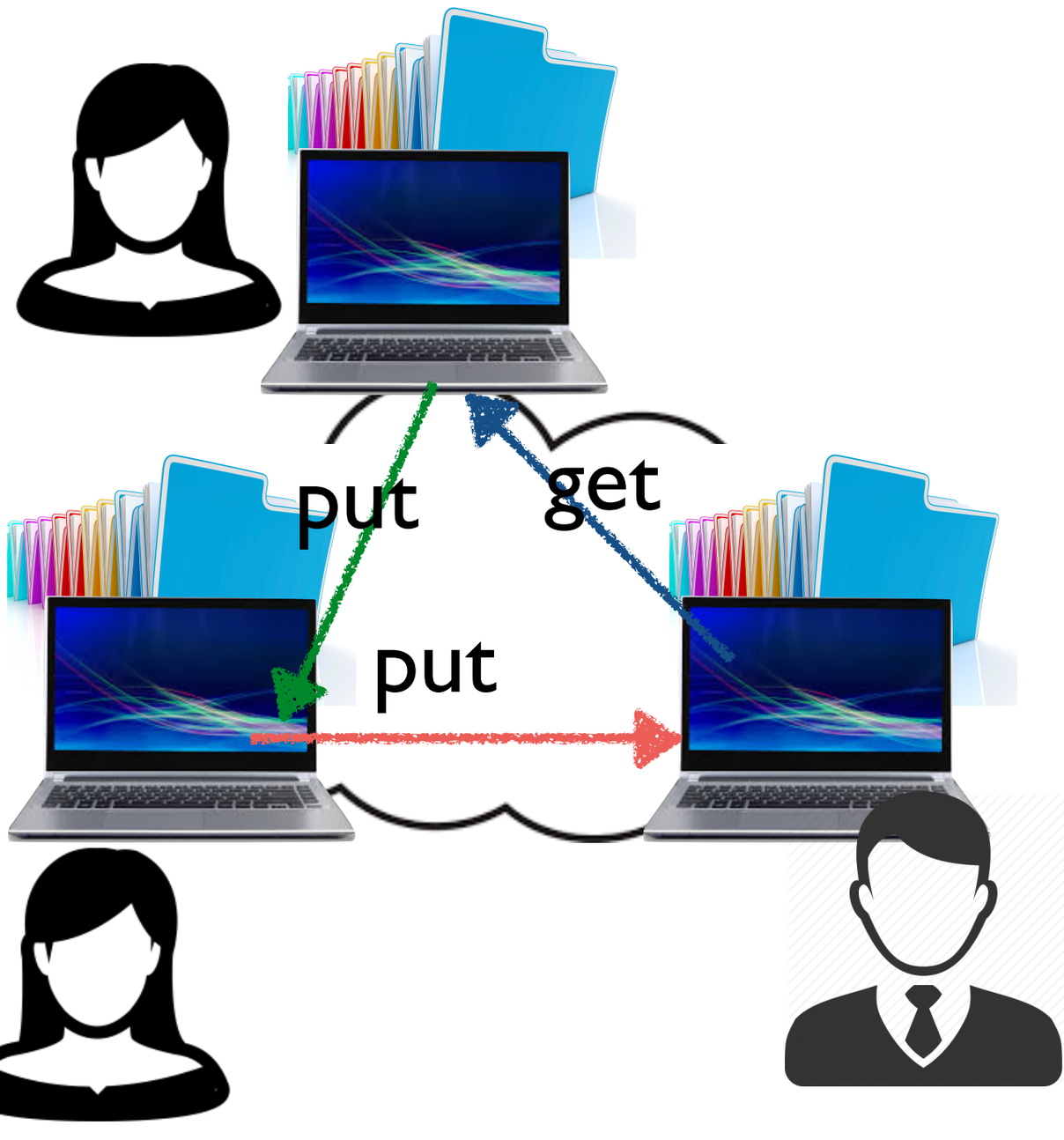
Spring Semester 2020

Lecture 27: P2P Systems (Chord)

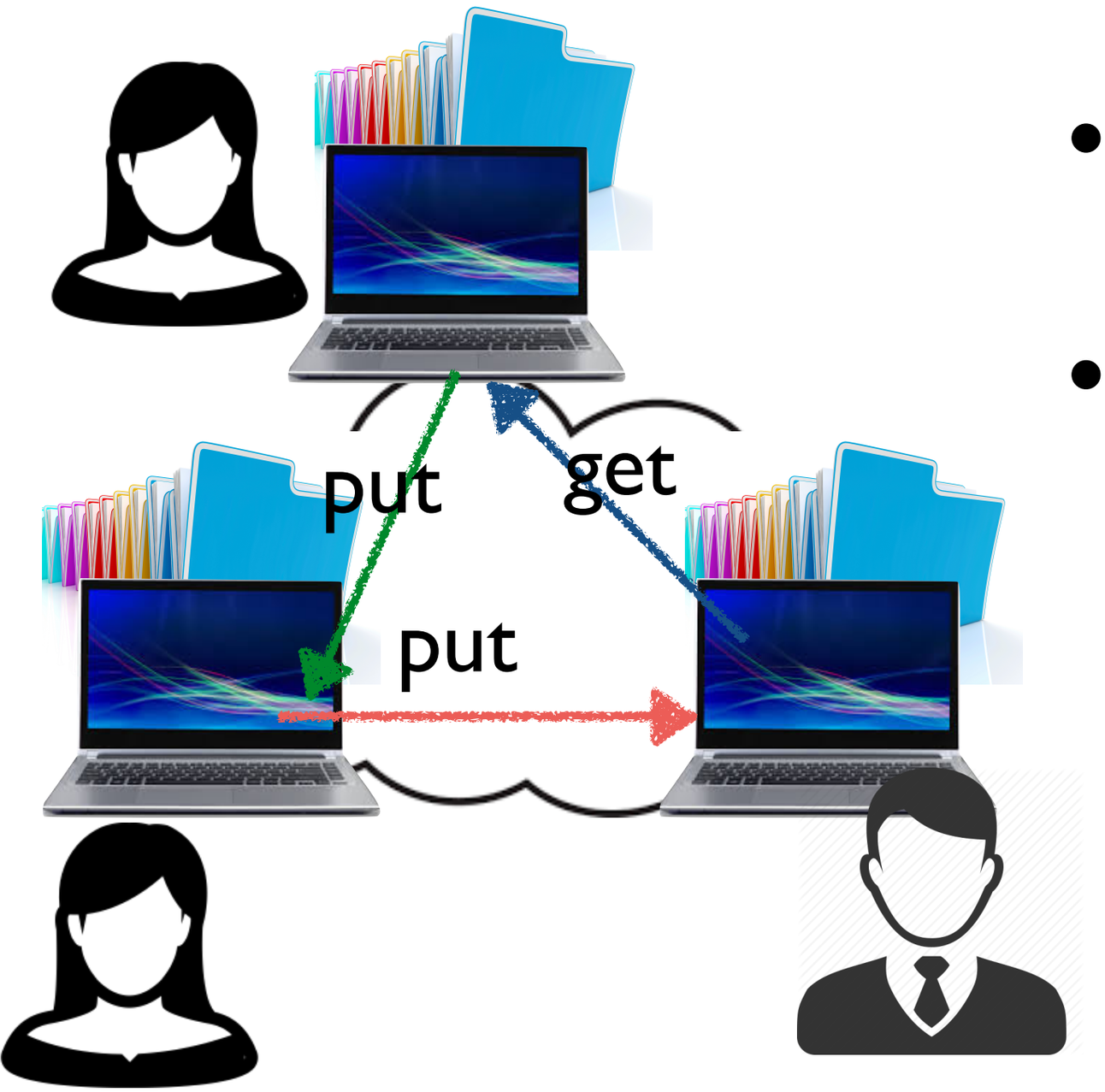
John Liagouris
liagos@bu.edu

Peer-to-Peer



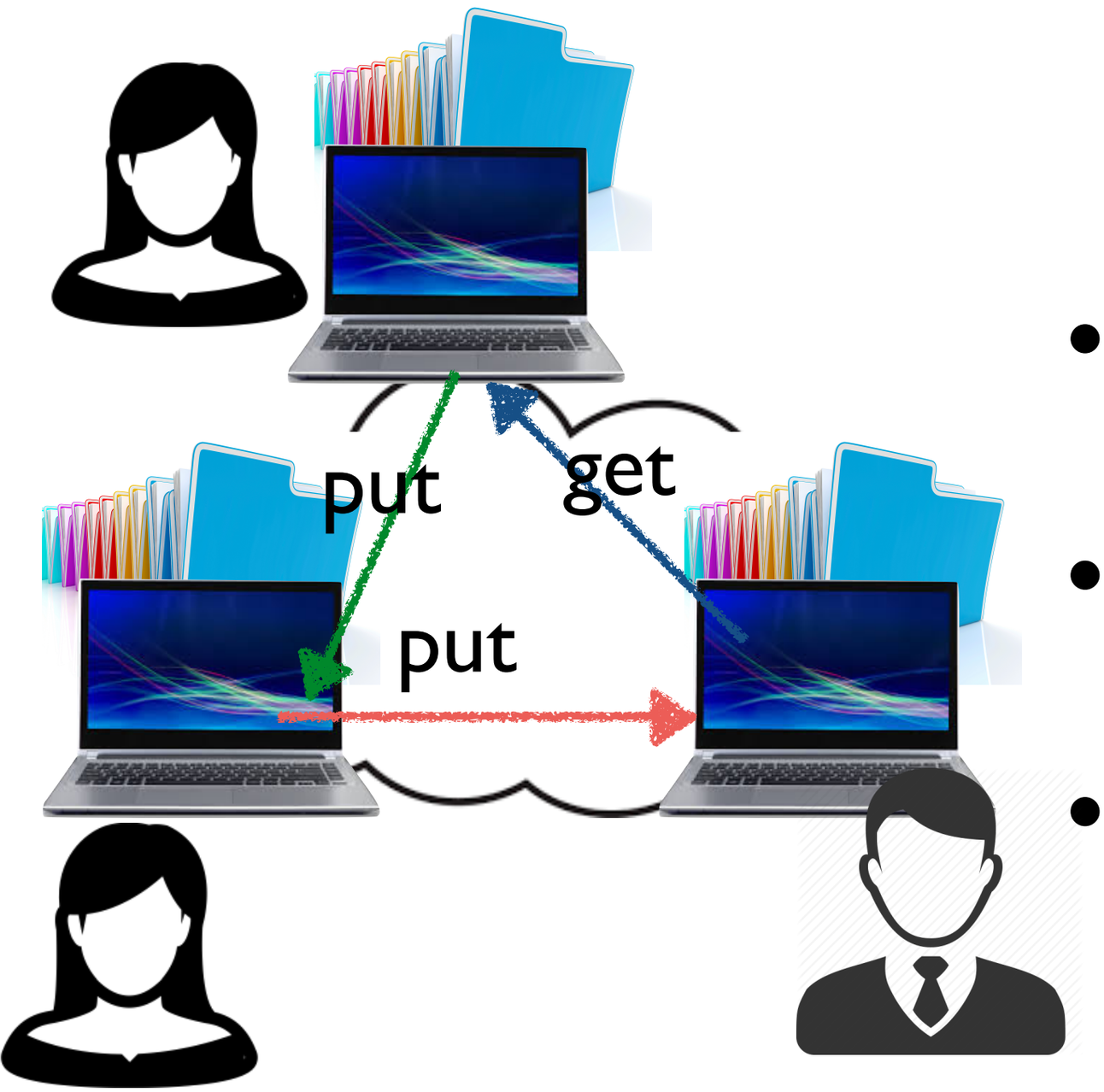


Why might P2P be a win?



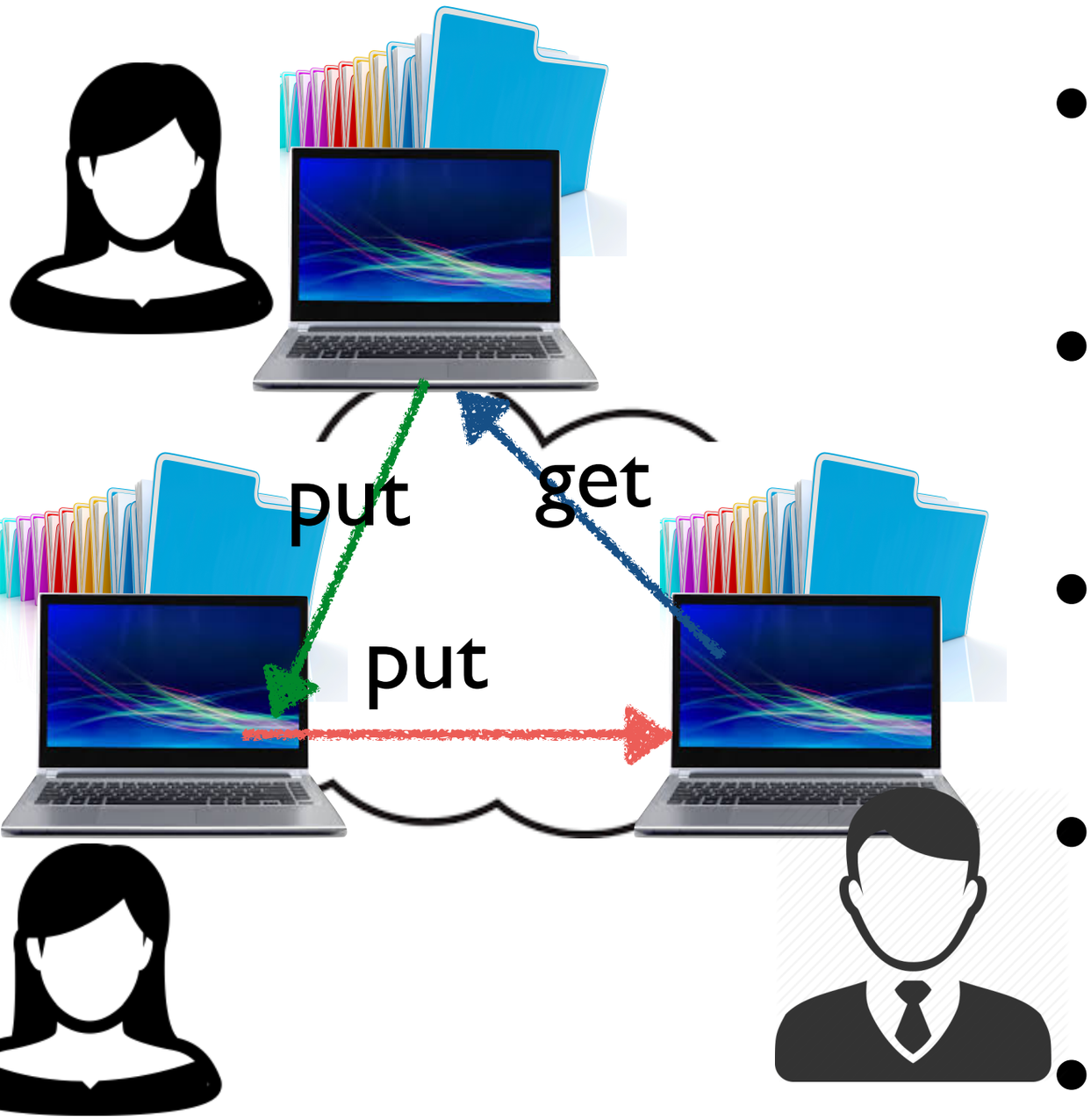
- Spreads network/caching costs over users
- Absence of server may mean:
  - Easier to deploy
  - Less chance of overload
  - Single failure won't wreck the whole system
  - harder to attack

Why not everything P2P?



- User computers not as reliable than managed servers
- If open, can be attacked via evil participants
- Can be hard to find data items over millions of users

In practice P2P has some successful niches



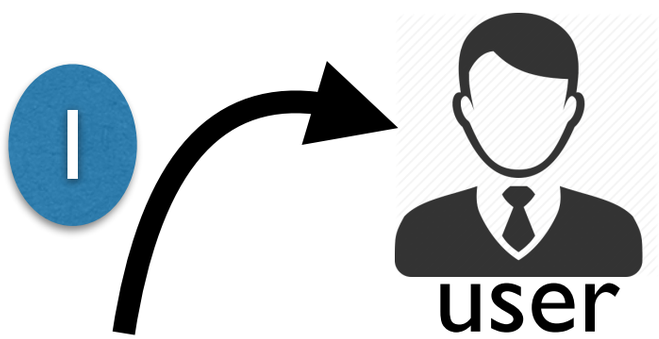
- Client-client video/music, where serving costs are high
- Chat (user to user anyway; privacy and control)
- Popular data but owning organization has no money
- No natural single owner or controller (Bitcoin)
- Illegal file sharing ("Metallica vs Napster Inc.")

Classic BitTorrent

http://ubuntu.com/ubuntu.iso.torrent

- Torrent file contains content hash and IP address of tracker
- Tracker know clients/ locations that already has a copy of the file
- Transfers happens as massively parallel sending of chucks from other clients
- Huge download bandwidth without expensive server/link

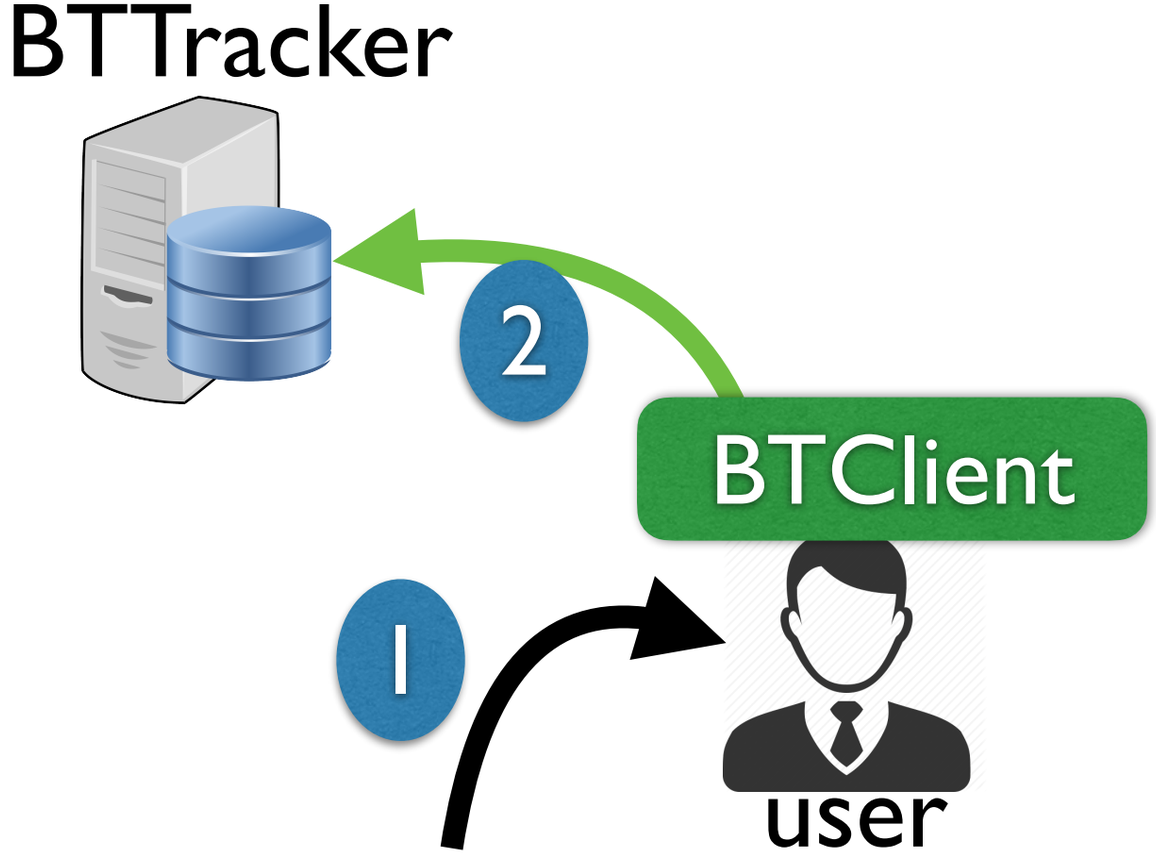
Classic BitTorrent



http://ubuntu.com/ubuntu.iso.torrent

- Torrent file contains content hash and IP address of tracker
- Tracker know clients/ locations that already has a copy of the file
- Transfers happens as massively parallel sending of chucks from other clients
- Huge download bandwidth without expensive server/link

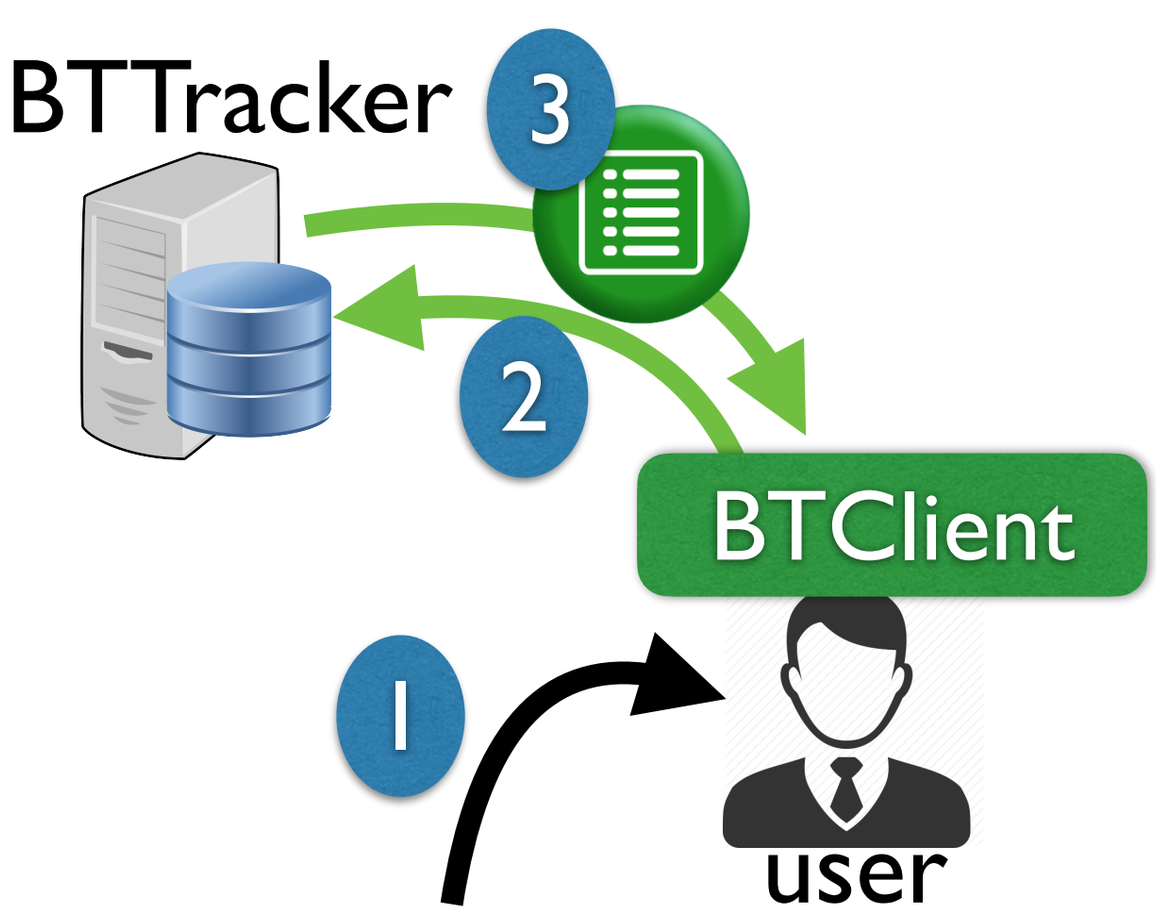
Classic BitTorrent



http://ubuntu.com/ubuntu.iso.torrent

- Torrent file contains content hash and IP address of tracker
- Tracker know clients/ locations that already has a copy of the file
- Transfers happens as massively parallel sending of chucks from other clients
- Huge download bandwidth without expensive server/link

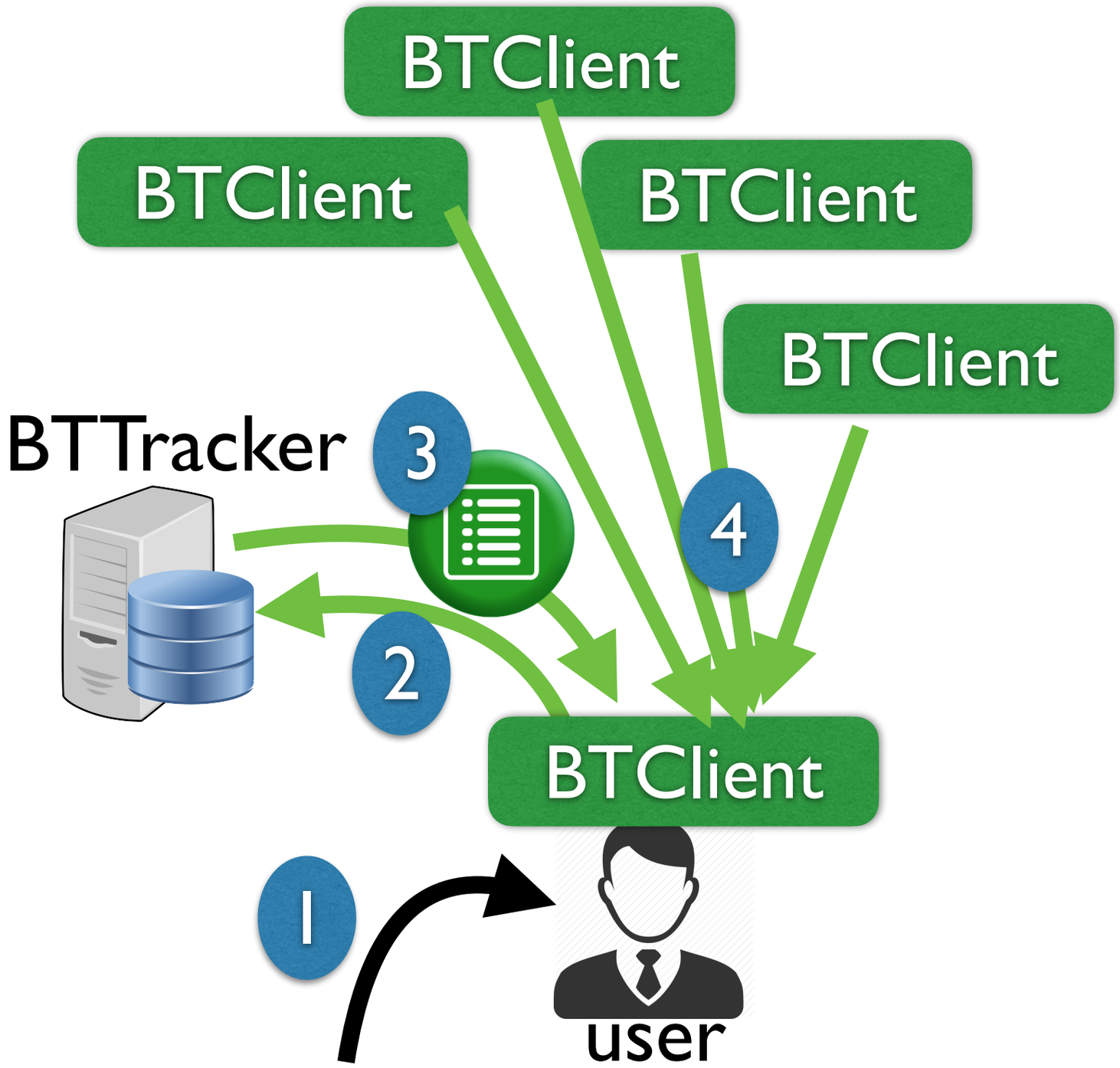
Classic BitTorrent



http://ubuntu.com/ubuntu.iso.torrent

- Torrent file contains content hash and IP address of tracker
- Tracker know clients/ locations that already has a copy of the file
- Transfers happens as massively parallel sending of chucks from other clients
- Huge download bandwidth without expensive server/link

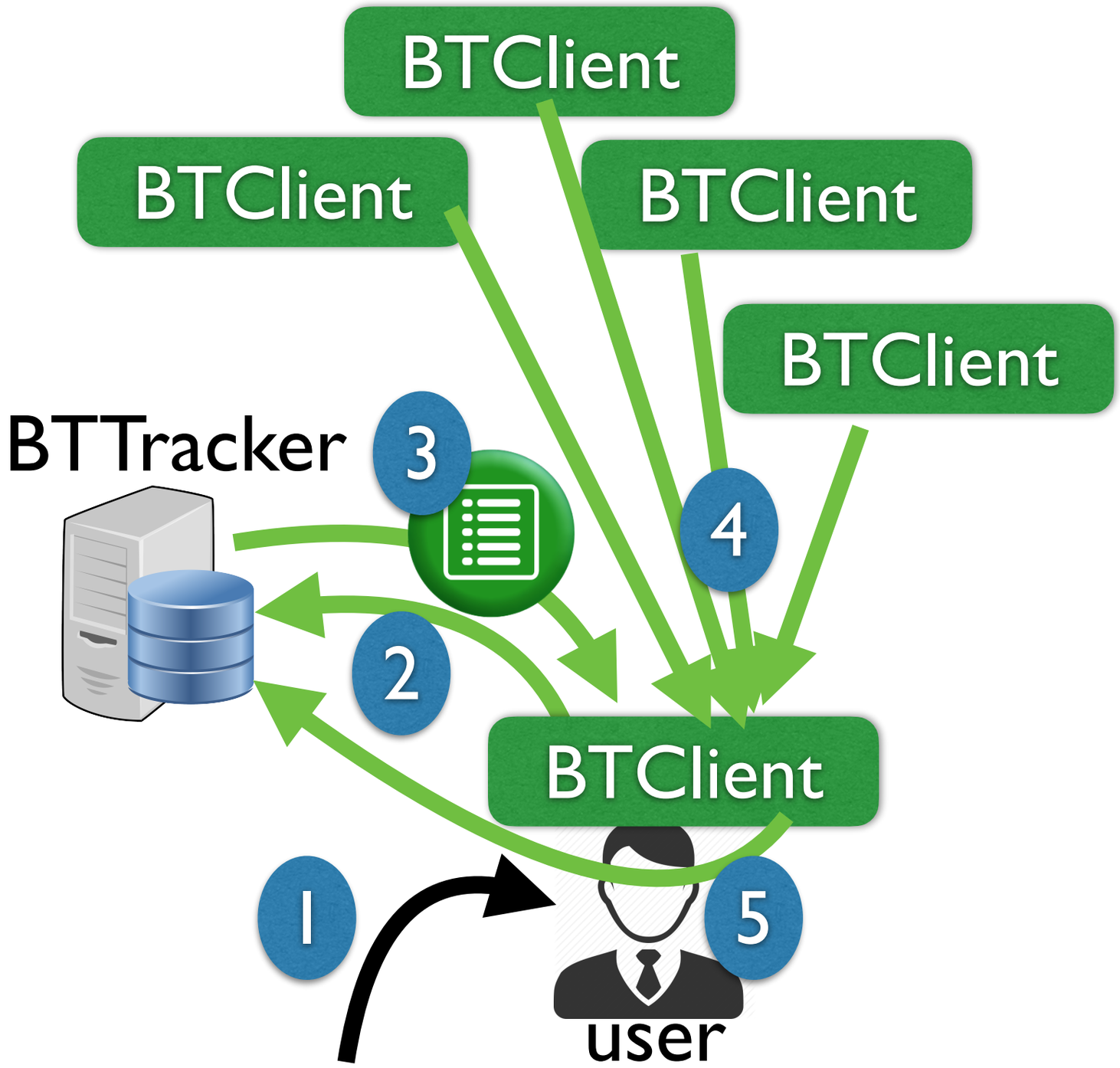
Classic BitTorrent



- Torrent file contains content hash and IP address of tracker
- Tracker know clients/ locations that already has a copy of the file
- Transfers happens as massively parallel sending of chuncks from other clients
- Huge download bandwidth without expensive server/link

http://ubuntu.com/ubuntu.iso.torrent

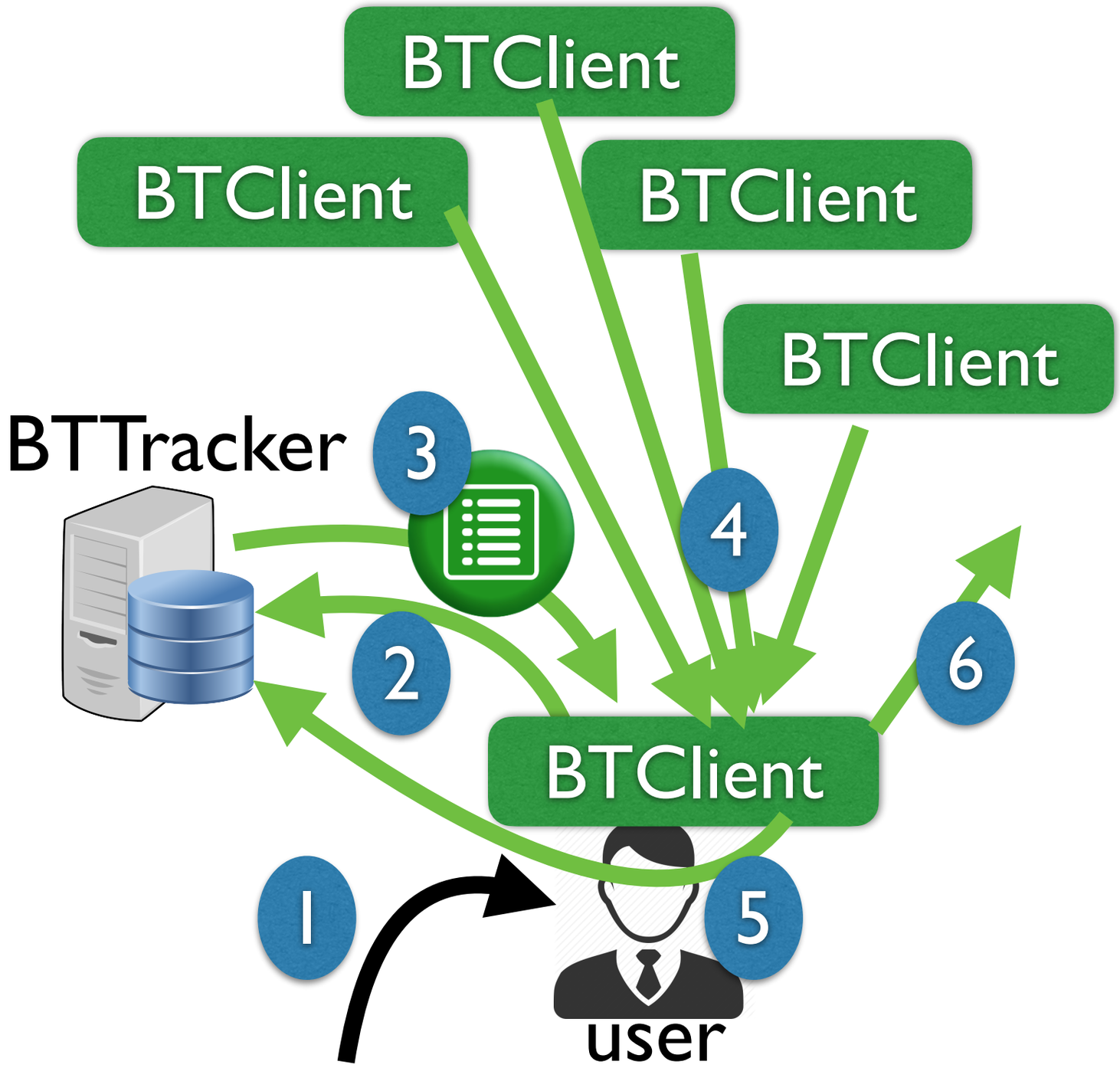
Classic BitTorrent



- Torrent file contains content hash and IP address of tracker
- Tracker know clients/ locations that already has a copy of the file
- Transfers happens as massively parallel sending of chucks from other clients
- Huge download bandwidth without expensive server/link

http://ubuntu.com/ubuntu.iso.torrent

Classic BitTorrent

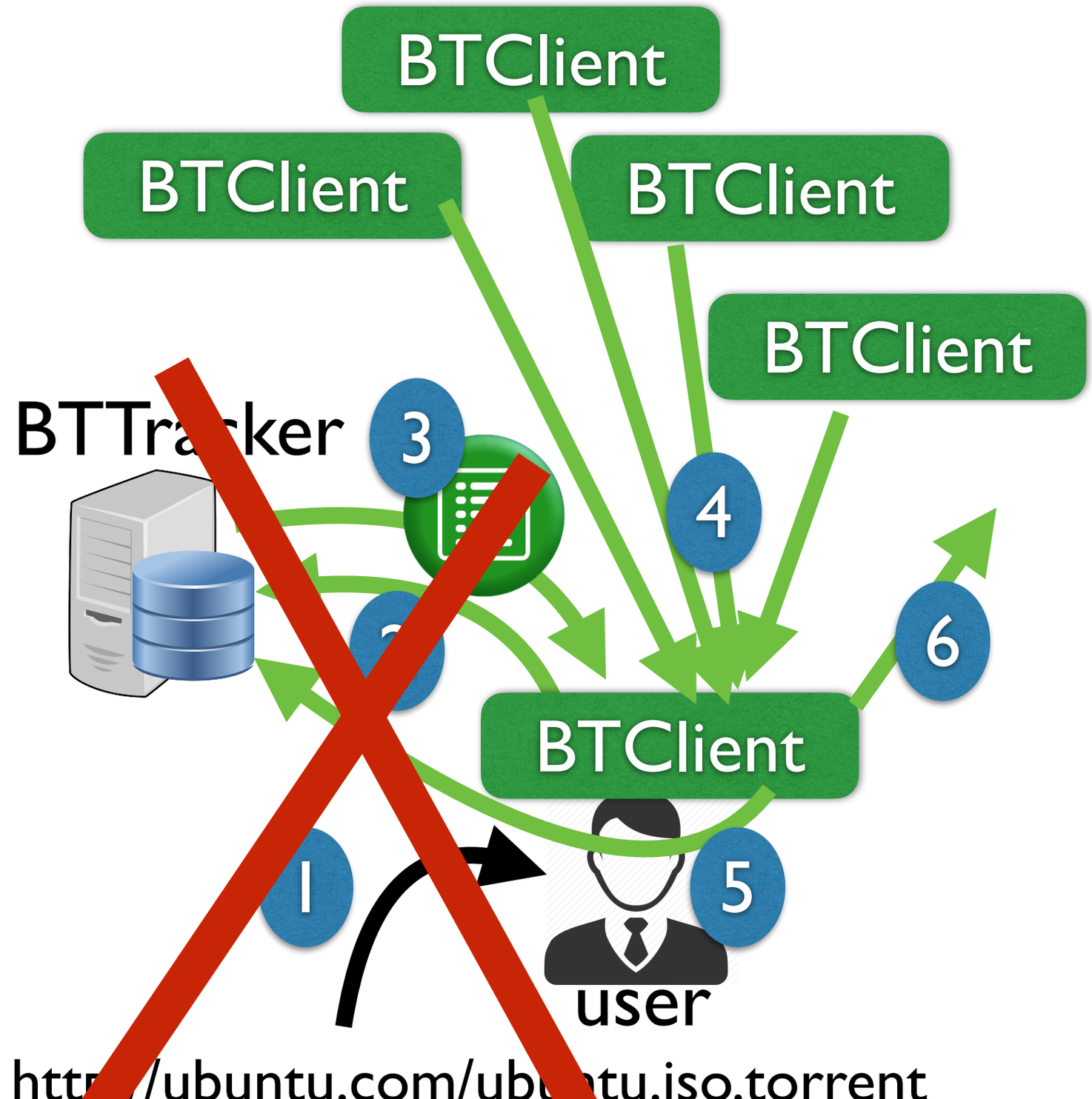


- Torrent file contains content hash and IP address of tracker
- Tracker know clients/ locations that already has a copy of the file
- Transfers happens as massively parallel sending of chucks from other clients
- Huge download bandwidth without expensive server/link

http://ubuntu.com/ubuntu.iso.torrent

'*Eliminate*' Tracker using DHT

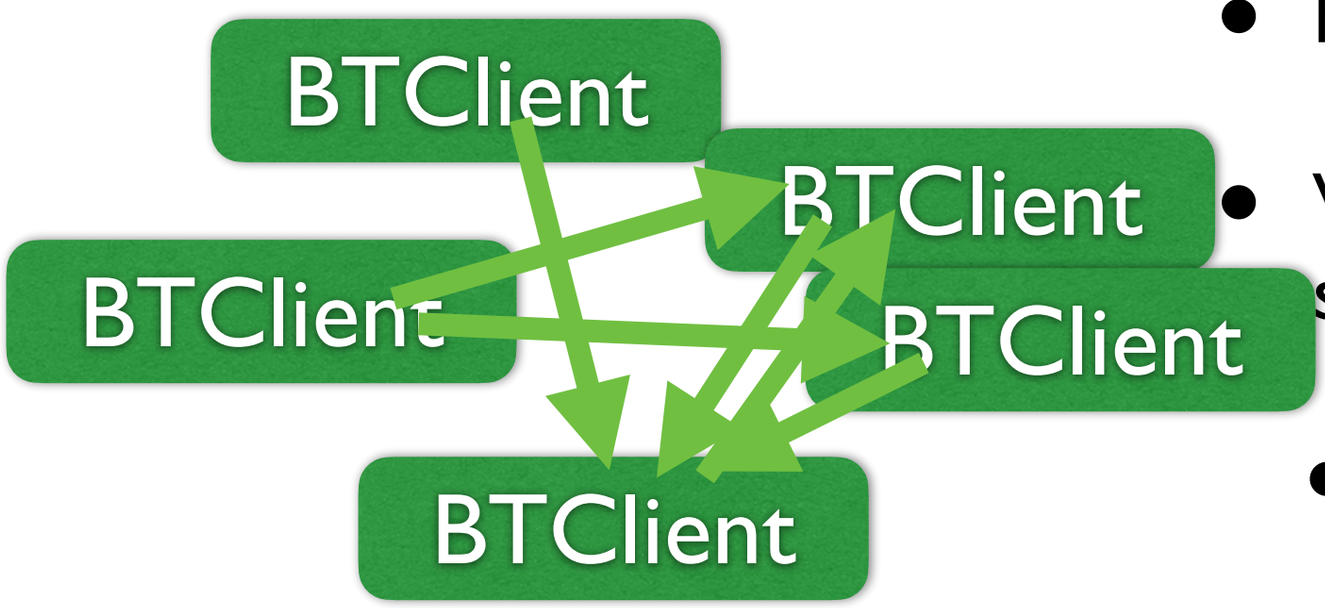
(topic of today's readings)



- BT clients cooperatively implement a giant key/value store

'*Eliminate*' Tracker using DHT
(topic of today's readings)

put(infohash,self)

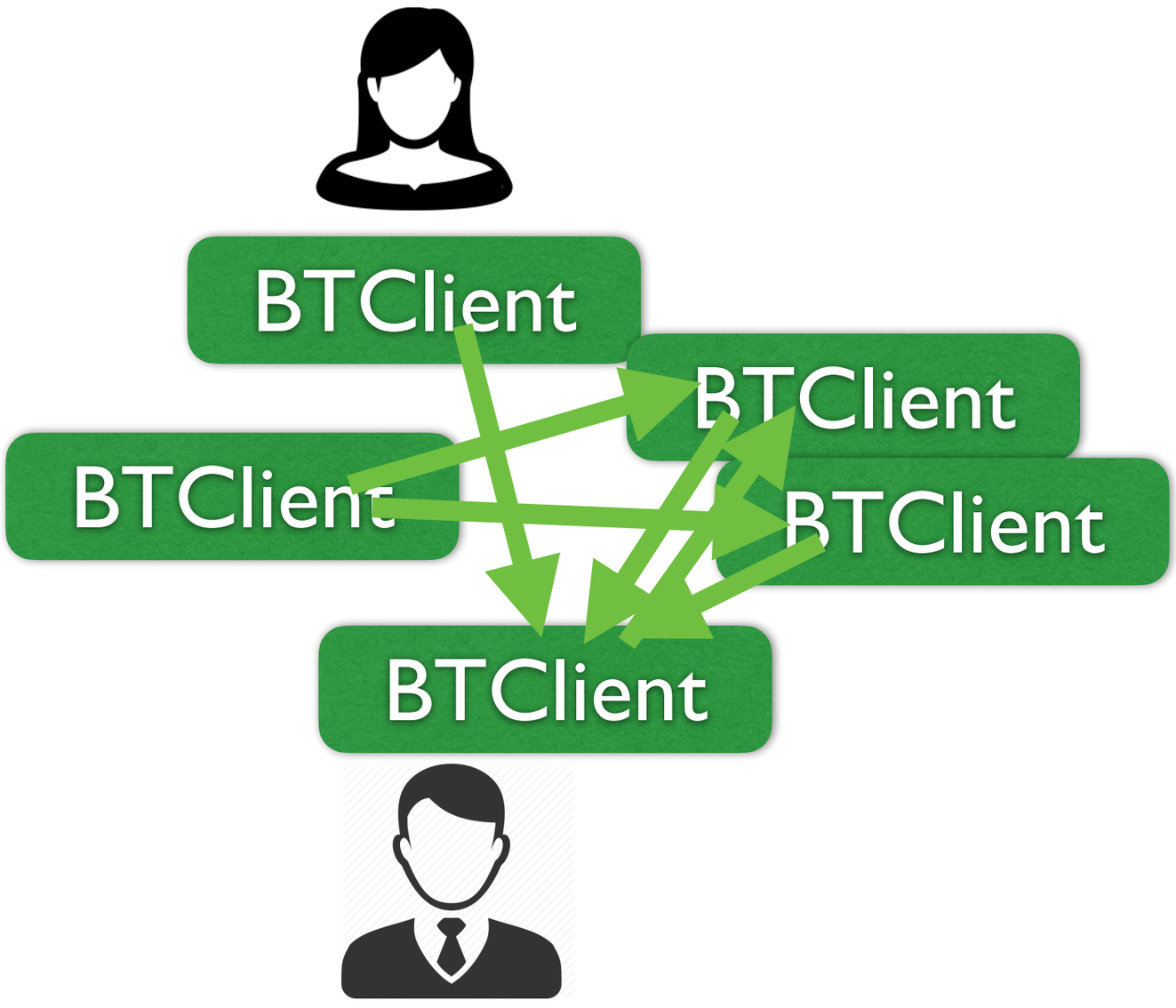


get(infohash)
put(infohash,self)

- BT clients cooperatively implement a giant key/value store — Distributed Hash Table
- Key is file content hash "infohash"
- Value is IP address of client willing to server the file
  - Kademlia can store multiple values for a key
- Get to find clients and Put to register as willing to serve
- clients join DHT to help implement it

Why DHT good for BT?

put(infohash,self)

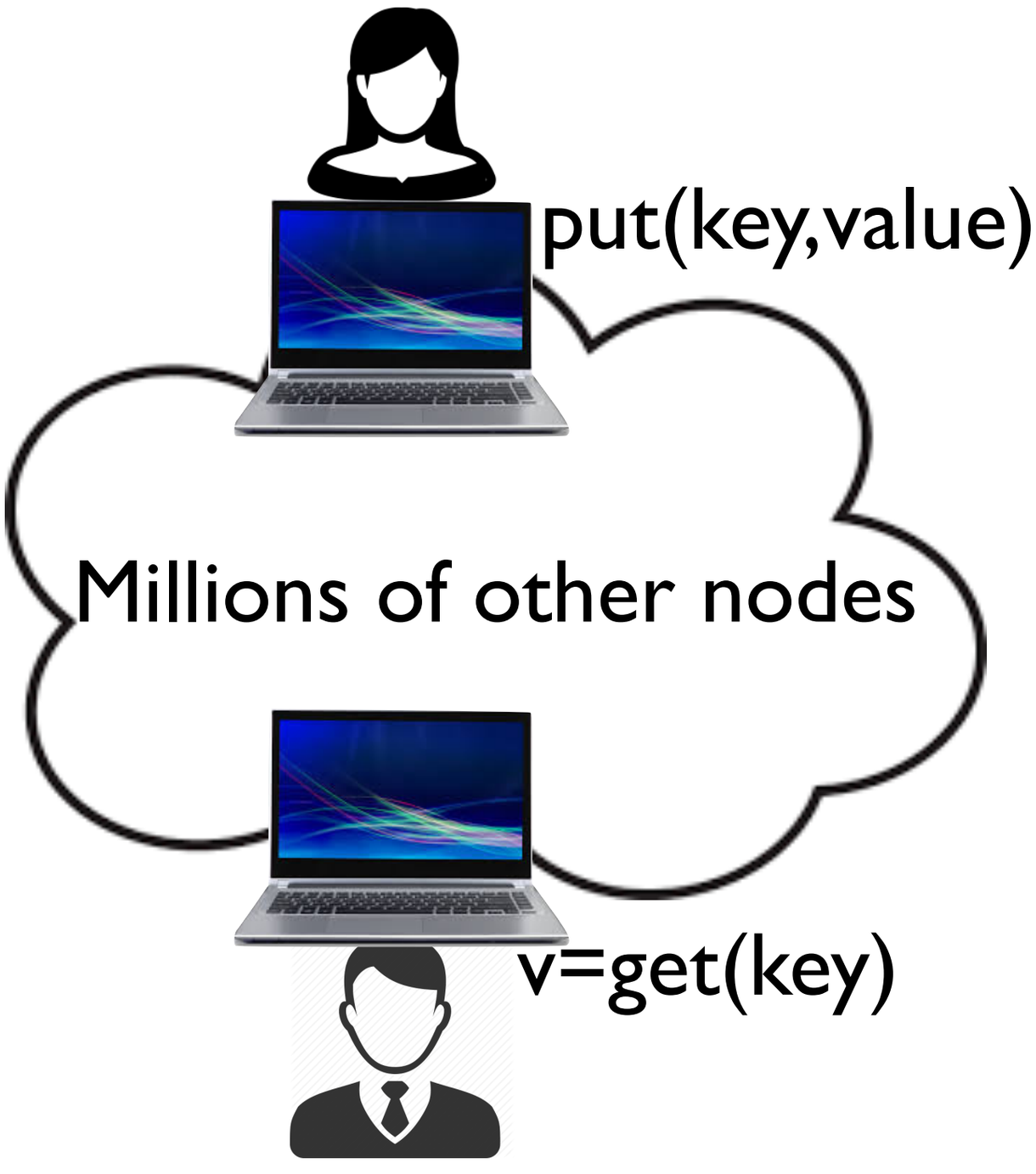


get(infohash)
put(infohash,self)

- Like having one single giant tracker, less fragmented that many trackers
  - clients more likely to find each other
- Maybe classic tracker too exposed to legal attacks
- In reality not clear that BT depends heavily on DHT mostly backup for classic trackers — seems like there is friction to change and misunderstanding

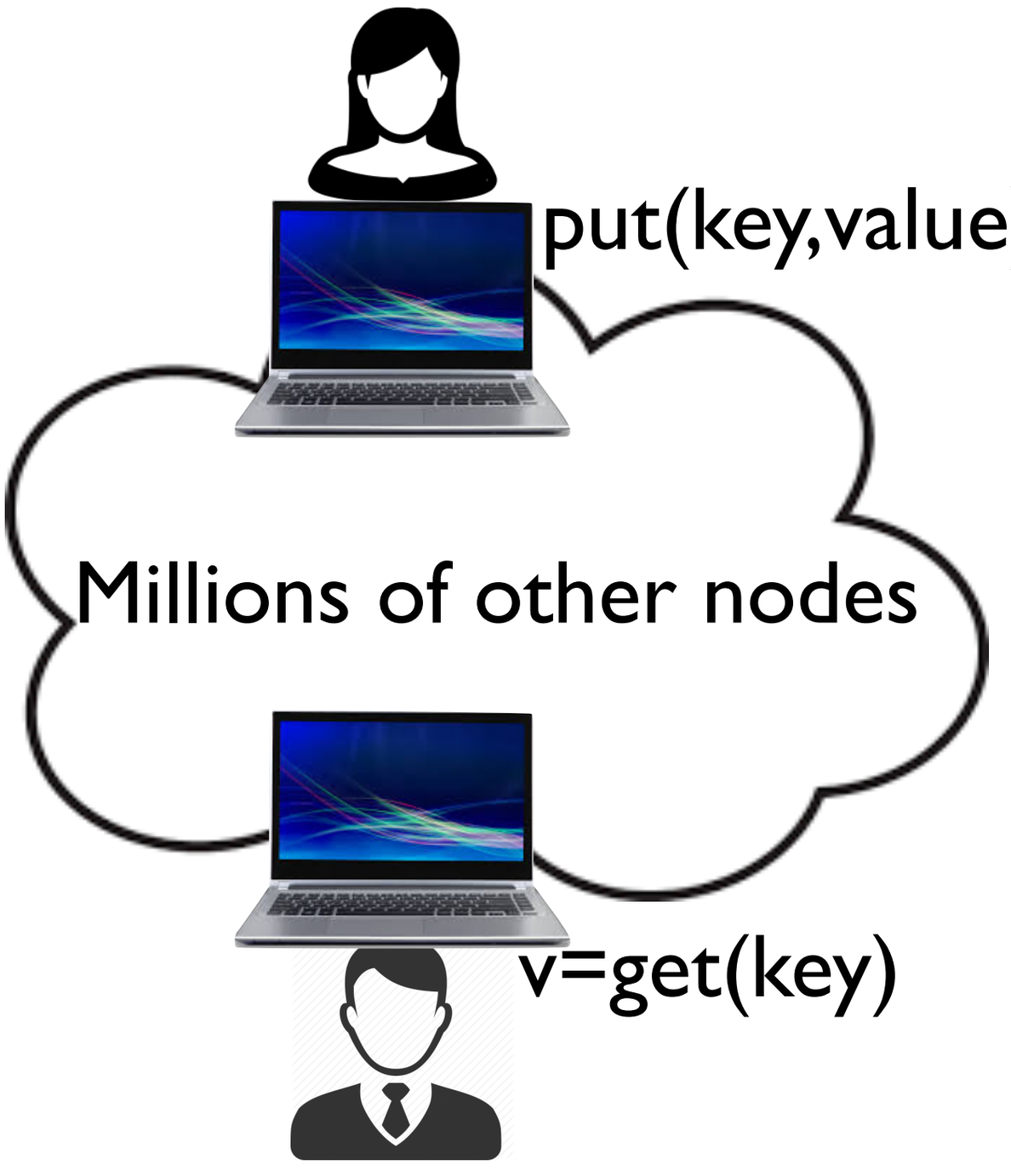
How do DHTs work?

Scalable DHT lookup



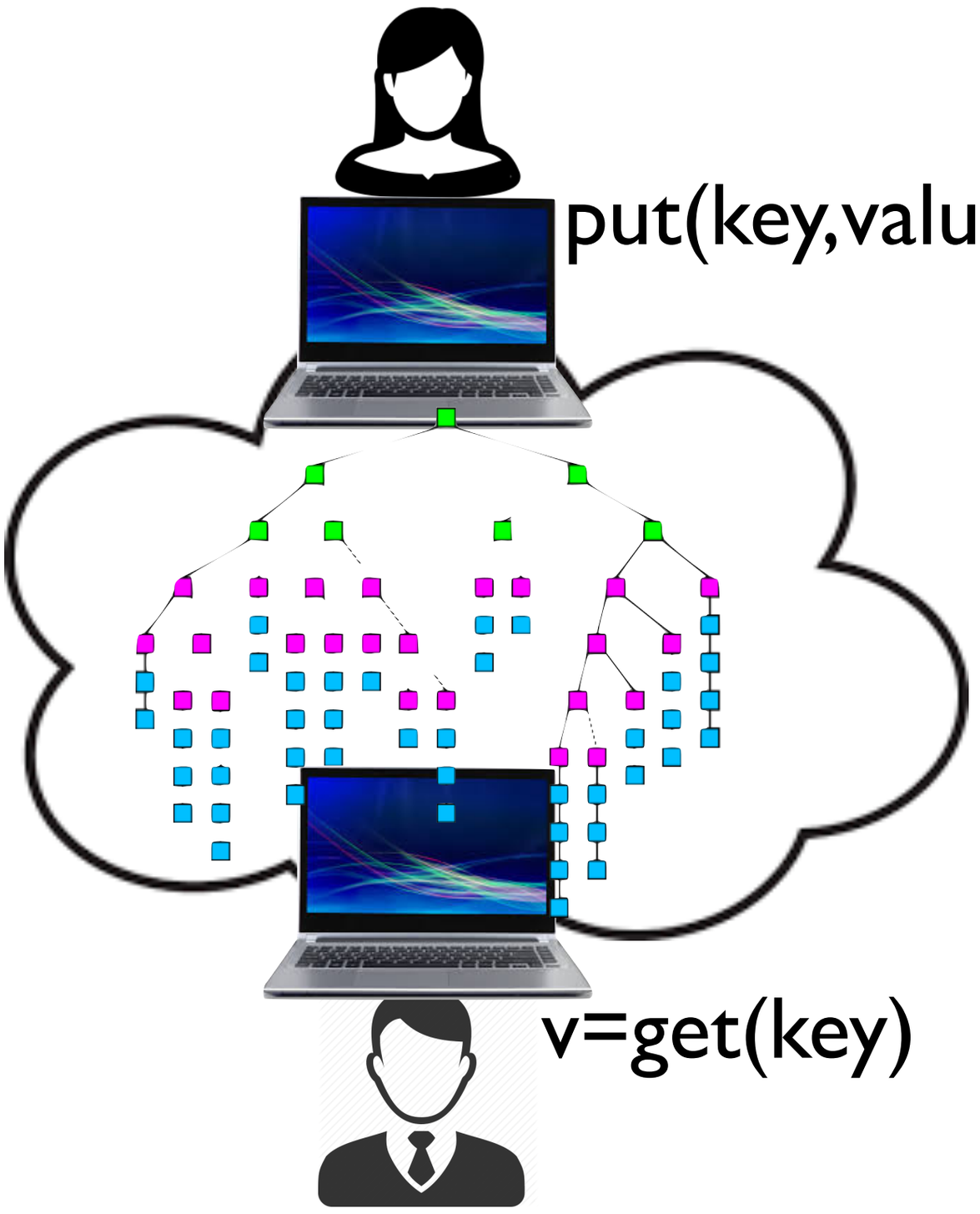
- Key/Value store spread over millions of nodes
  - Interface:
    - put(key,value)
    - get(key) -> value
- loose consistency; likely that get(k) sees put(k), but no guarantee
- loose guarantees about keeping data alive

Why is it Hard?



- **Millions** of participating nodes
- Could broadcast/flood request — but too many messages
- Every node could know about ever other node
  - Hashing is easy then
    - But keeping a million-node table up to date is hard
- Want modest state, and modest number of messages

Basic Idea

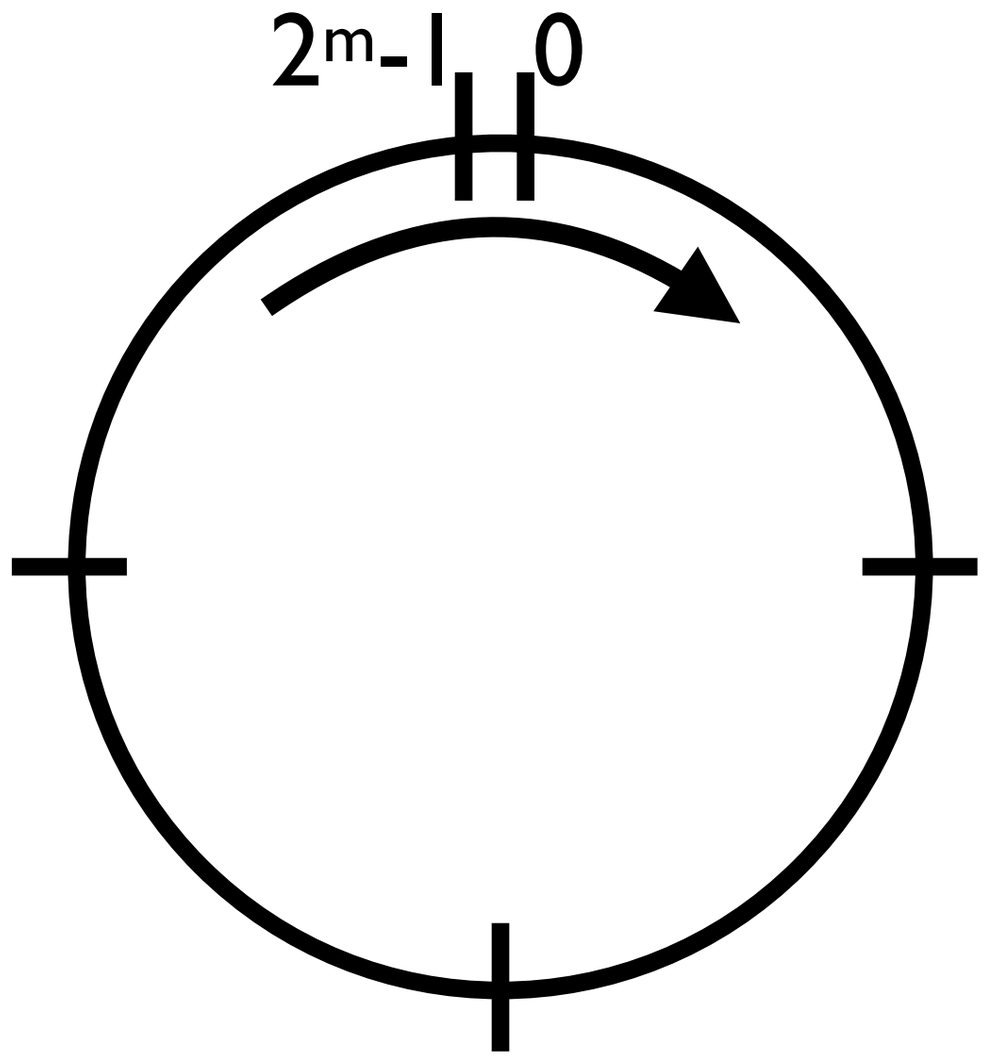


- Impose a data structure (e.g. tree) over the nodes
  - Each node has references to only a few other nodes
  - Lookups traverse the data structure -- “routing" — eg. hop from node to node
- DHT should route get() to same node as previous put()

Example: The "Chord" peer-to-peer lookup system, Stoica et.al; 2001

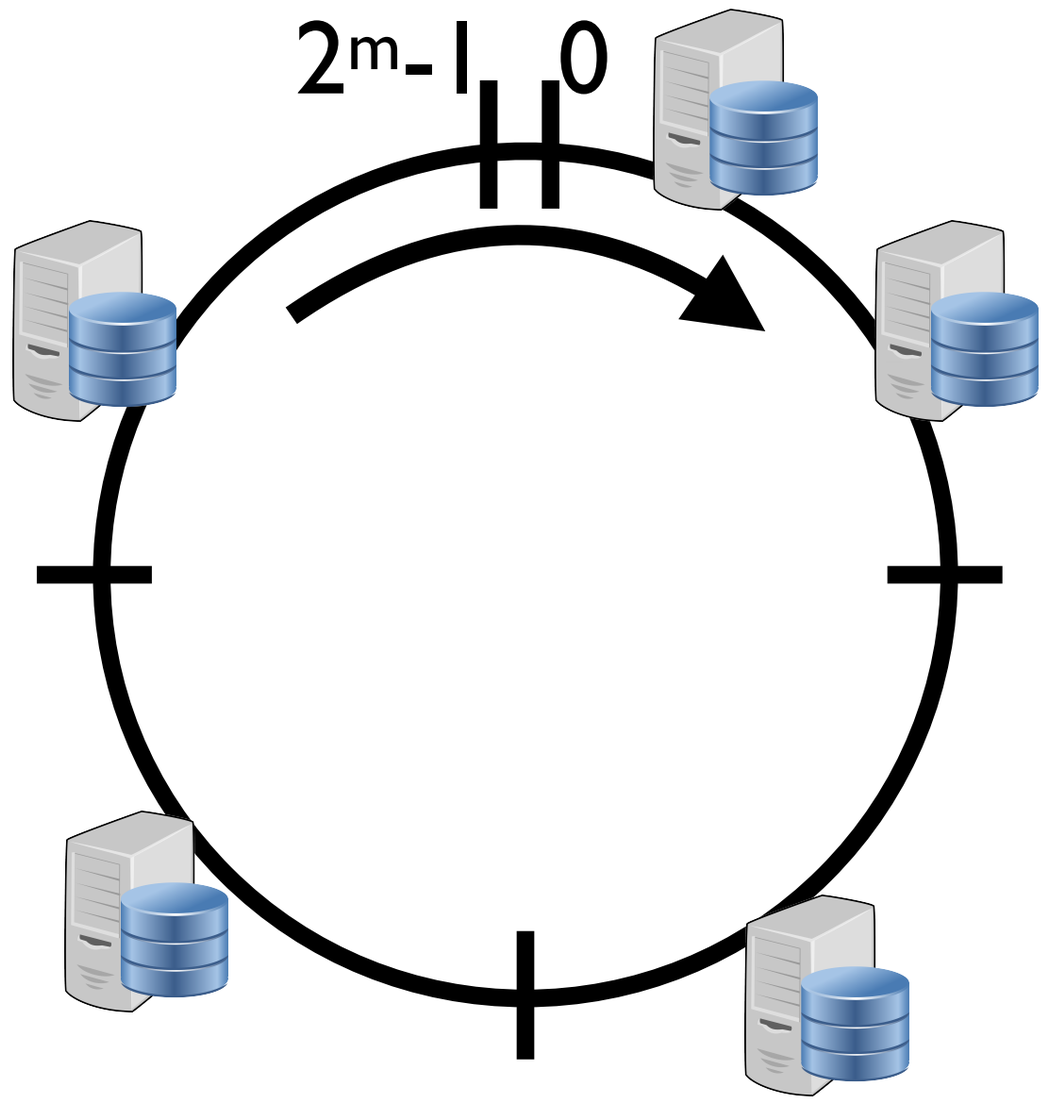
The Ring

- Chord's ID-space topology
- Ring: All IDs are 160-bit numbers, viewed in a ring.



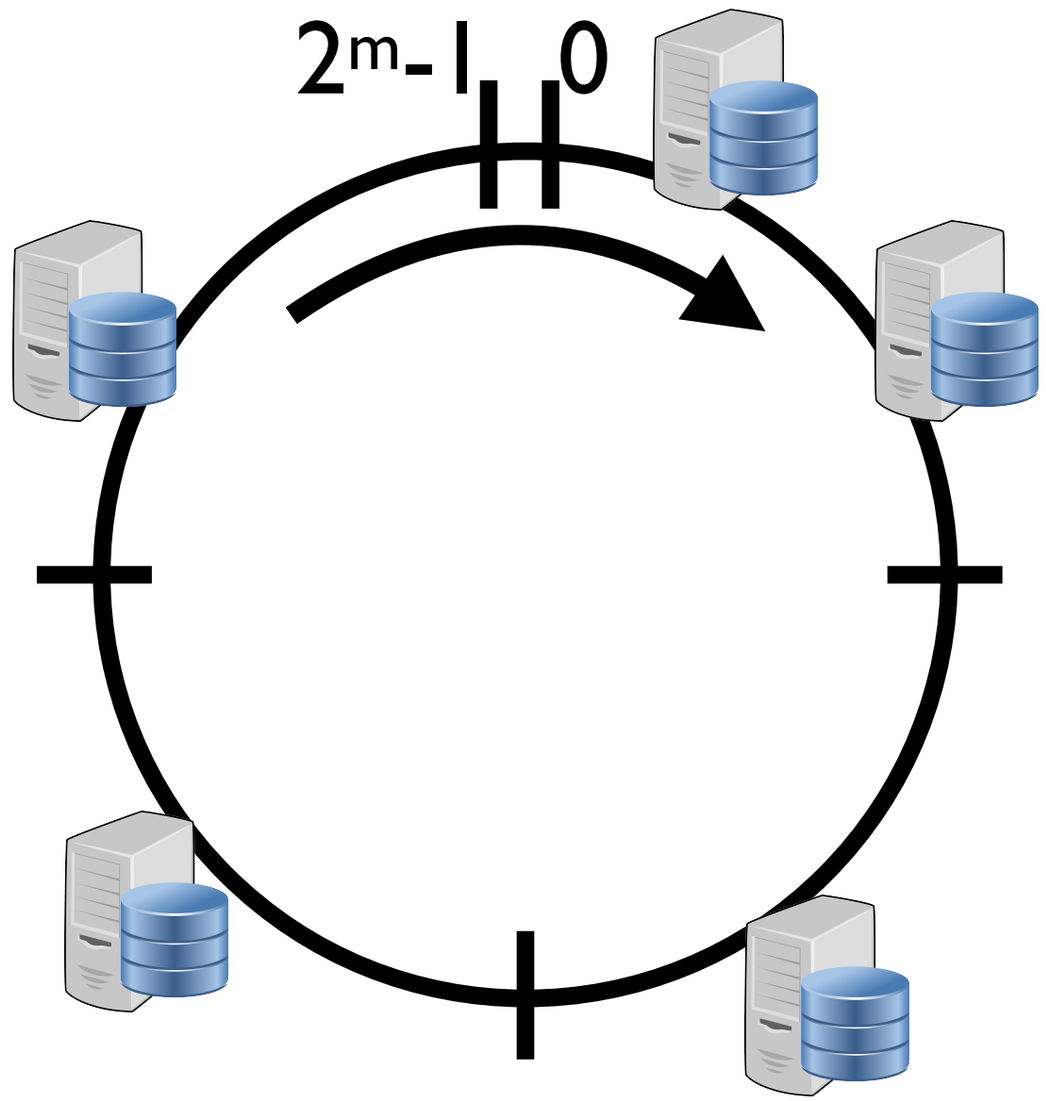
Nodes

- Chord's ID-space topology
  - Ring: All IDs are 160-bit numbers, viewed in a ring.
  - Each node has an ID, randomly chosen



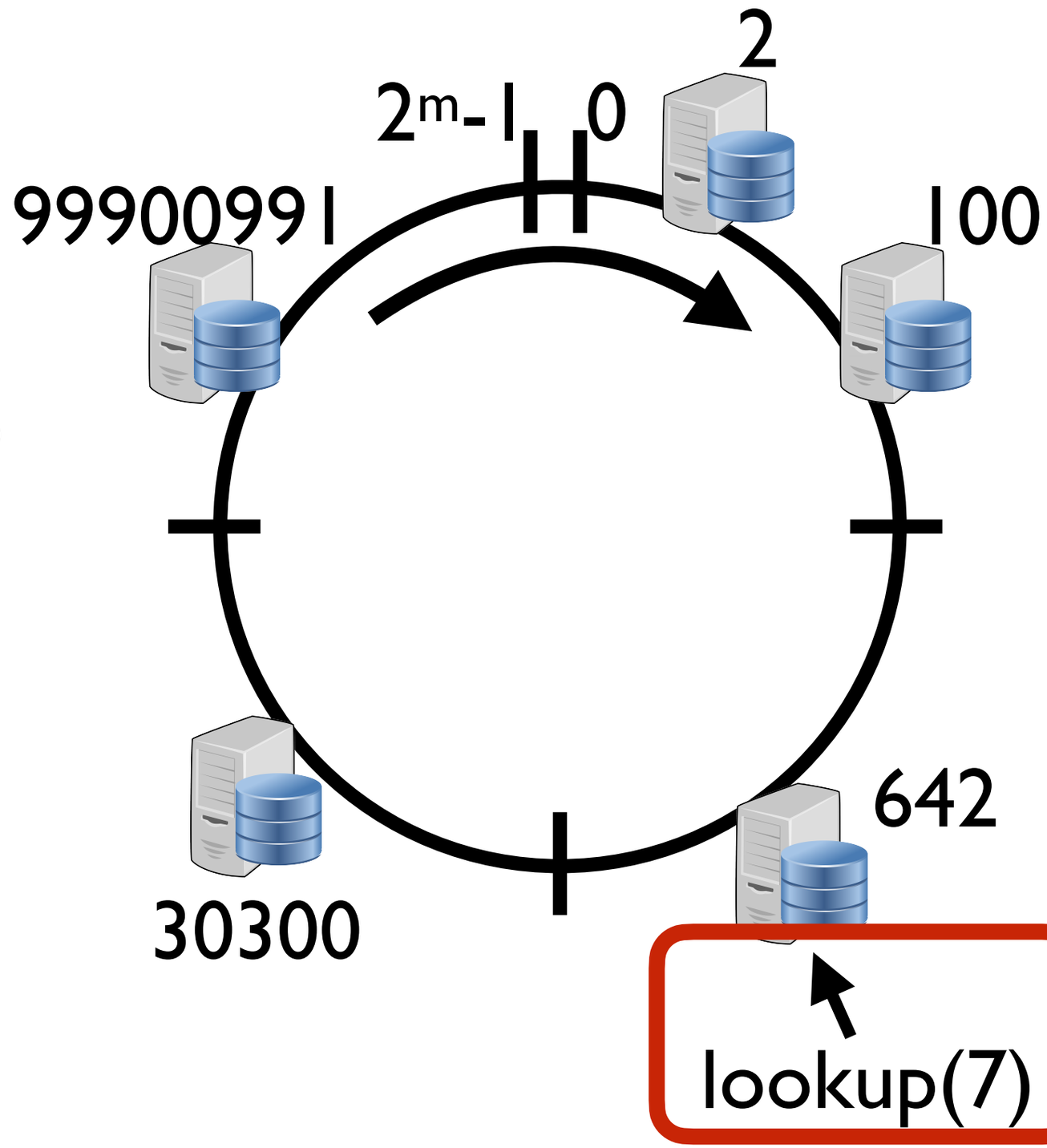
Keys

- Key stored on first node whose ID is equal to or greater than key ID.
  - Closeness is defined as the "clockwise distance"
- If node and key IDs are uniform, we get reasonable load balance.
- So keys IDs should be hashes (e.g. bittorrent infohash)



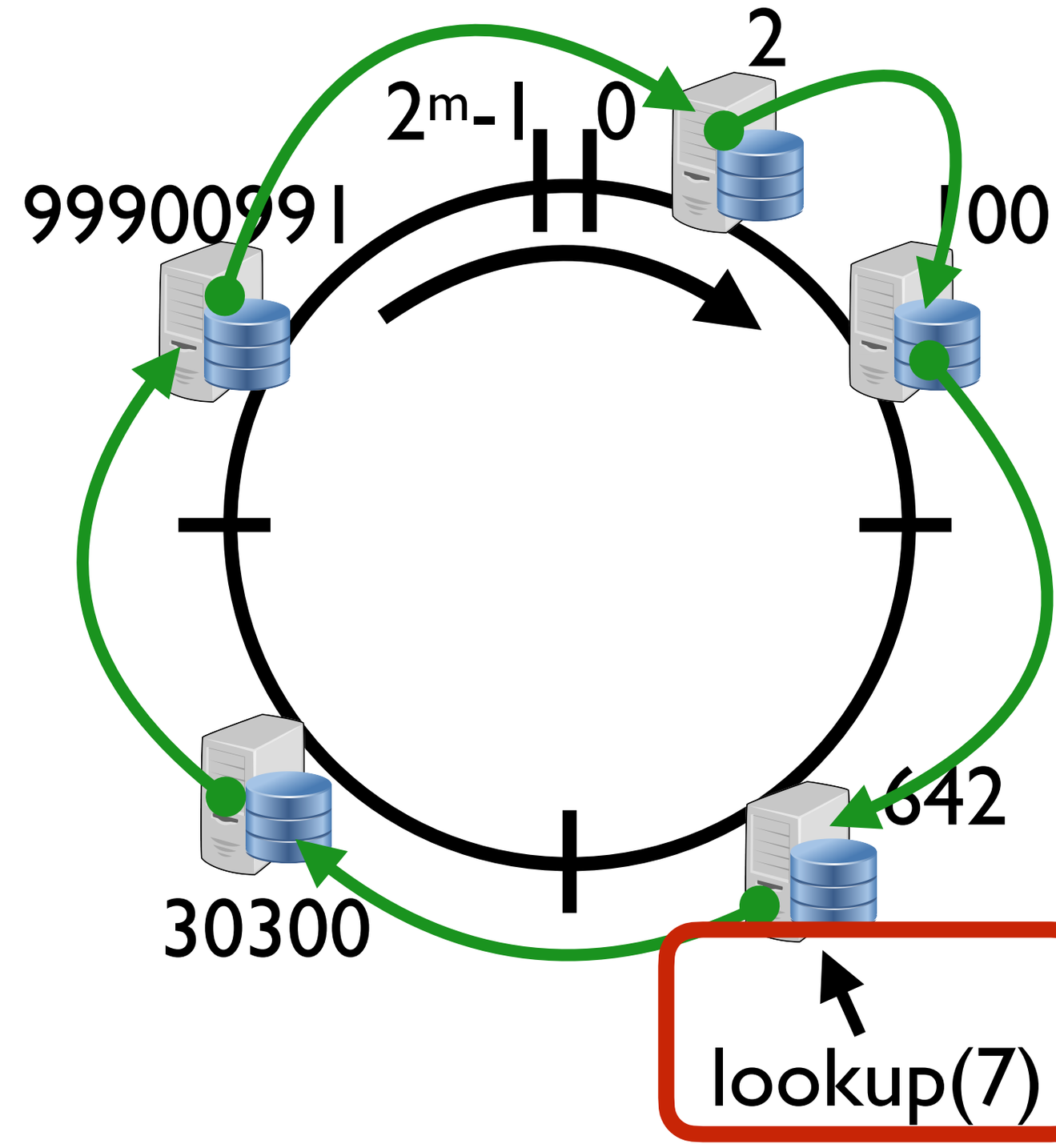
Basic Routing — Slow

- Query is at some node.
  - Node needs to forward the query to a node "closer" to key.
- If we keep moving query closer, eventually we'll win.



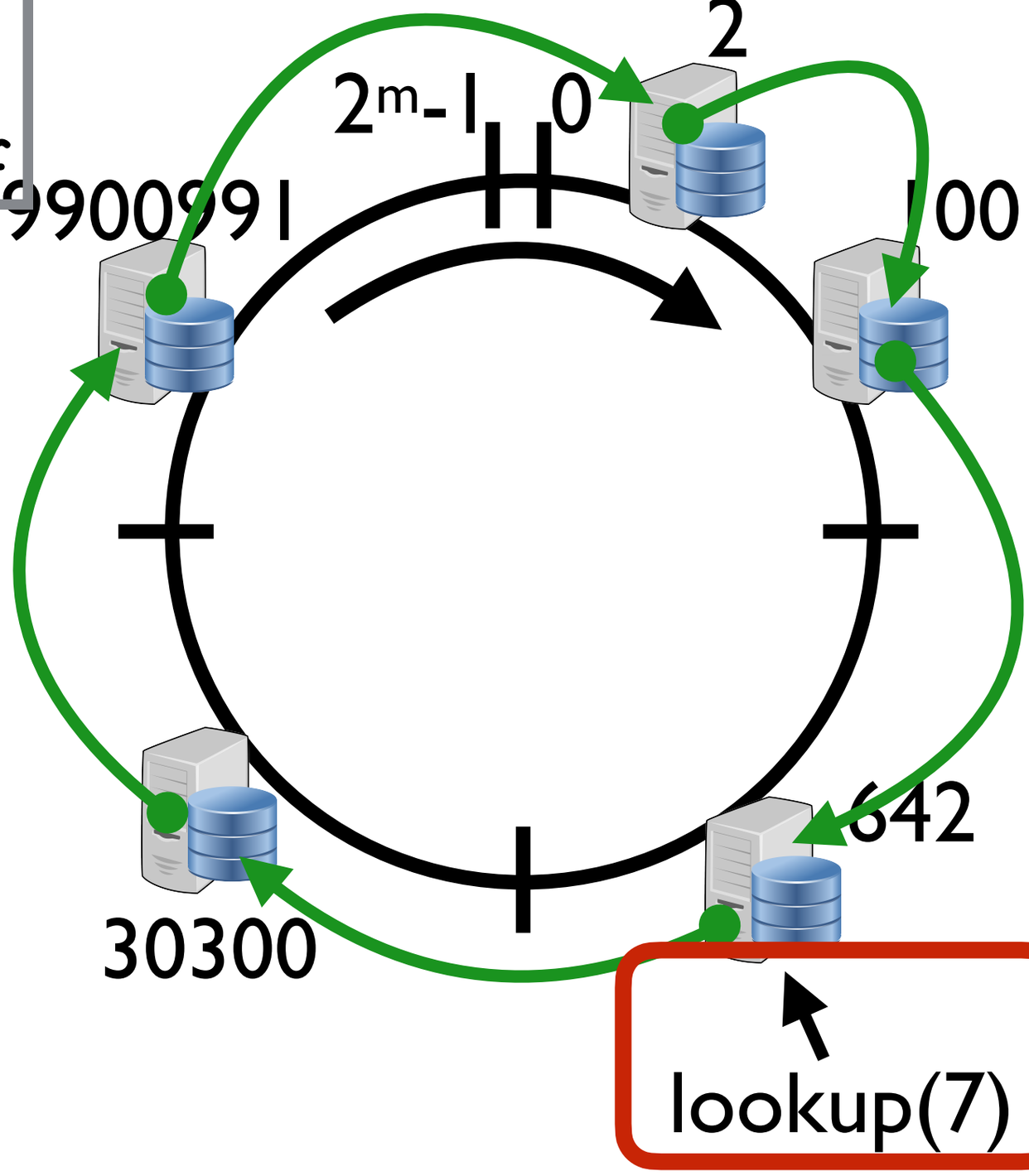
Basic Routing — Slow

- Each node knows its "successor" on the ring.



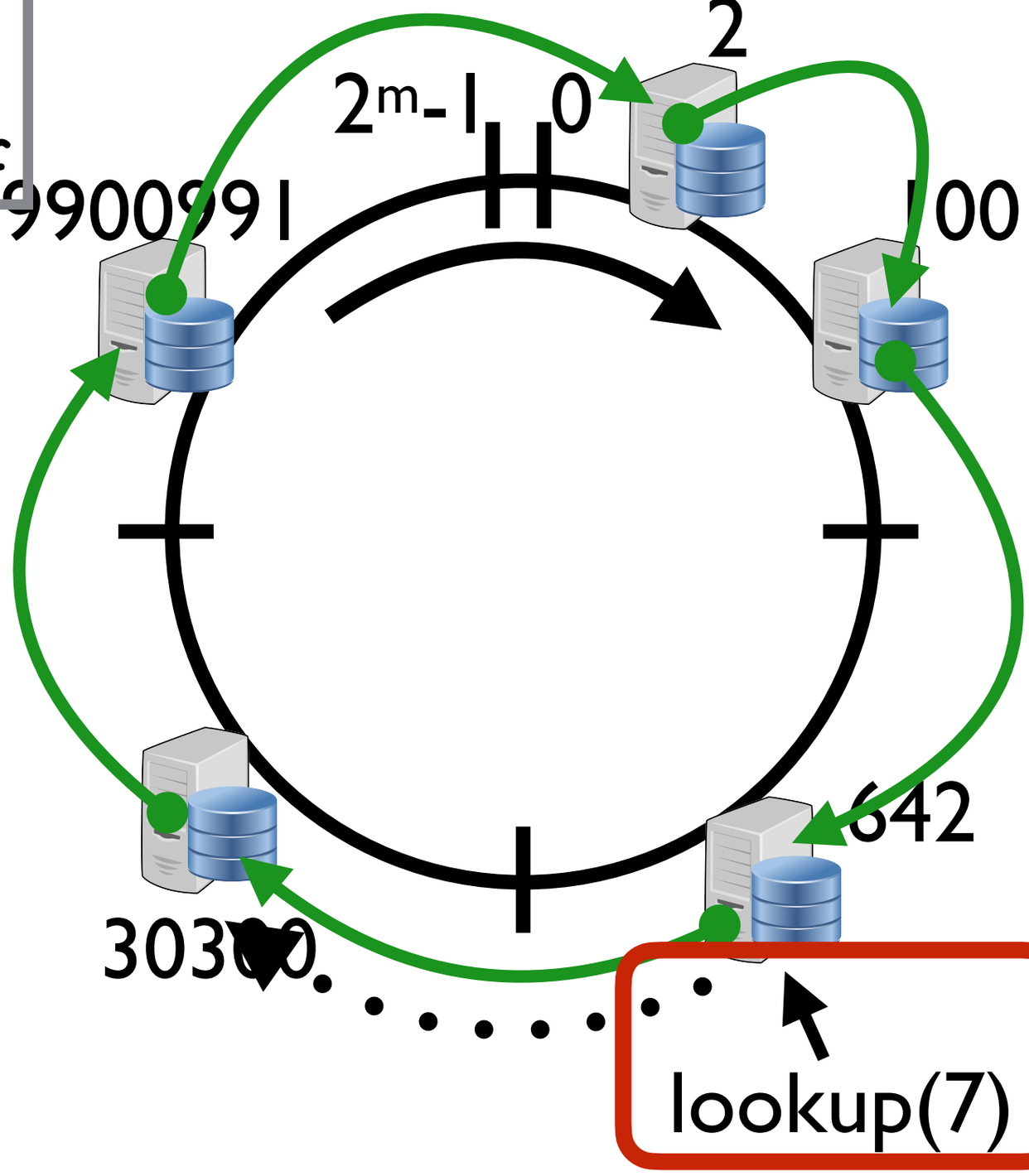
Basic Routing — Slow

```
n.lookup(k):
   if n < k <= n.successor
      return n.successor
   else
    forward to n.successor
```



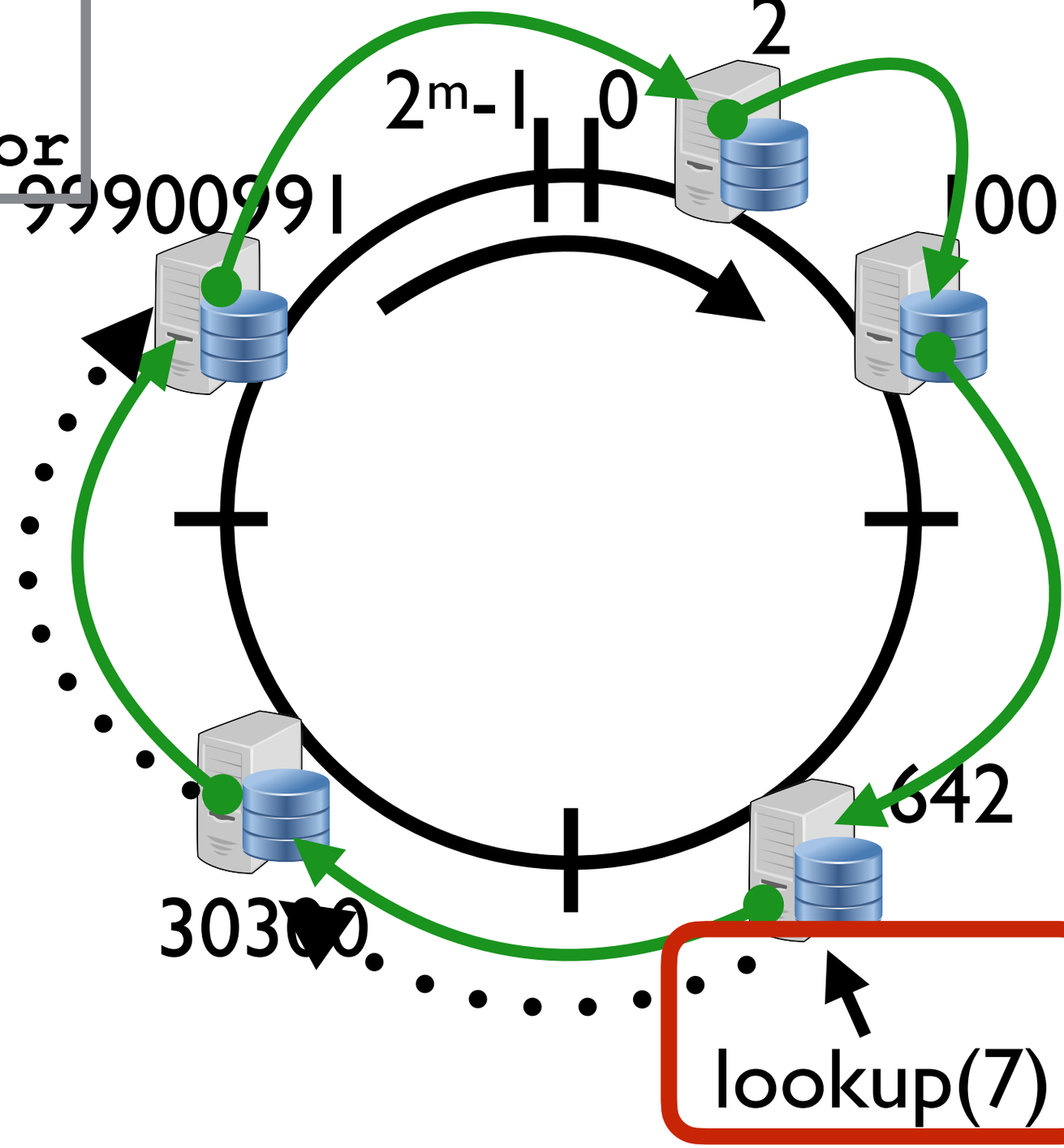
Basic Routing — Slow

```
n.lookup(k):
   if n < k <= n.successor
      return n.successor
   else
     forward to n.successor
```



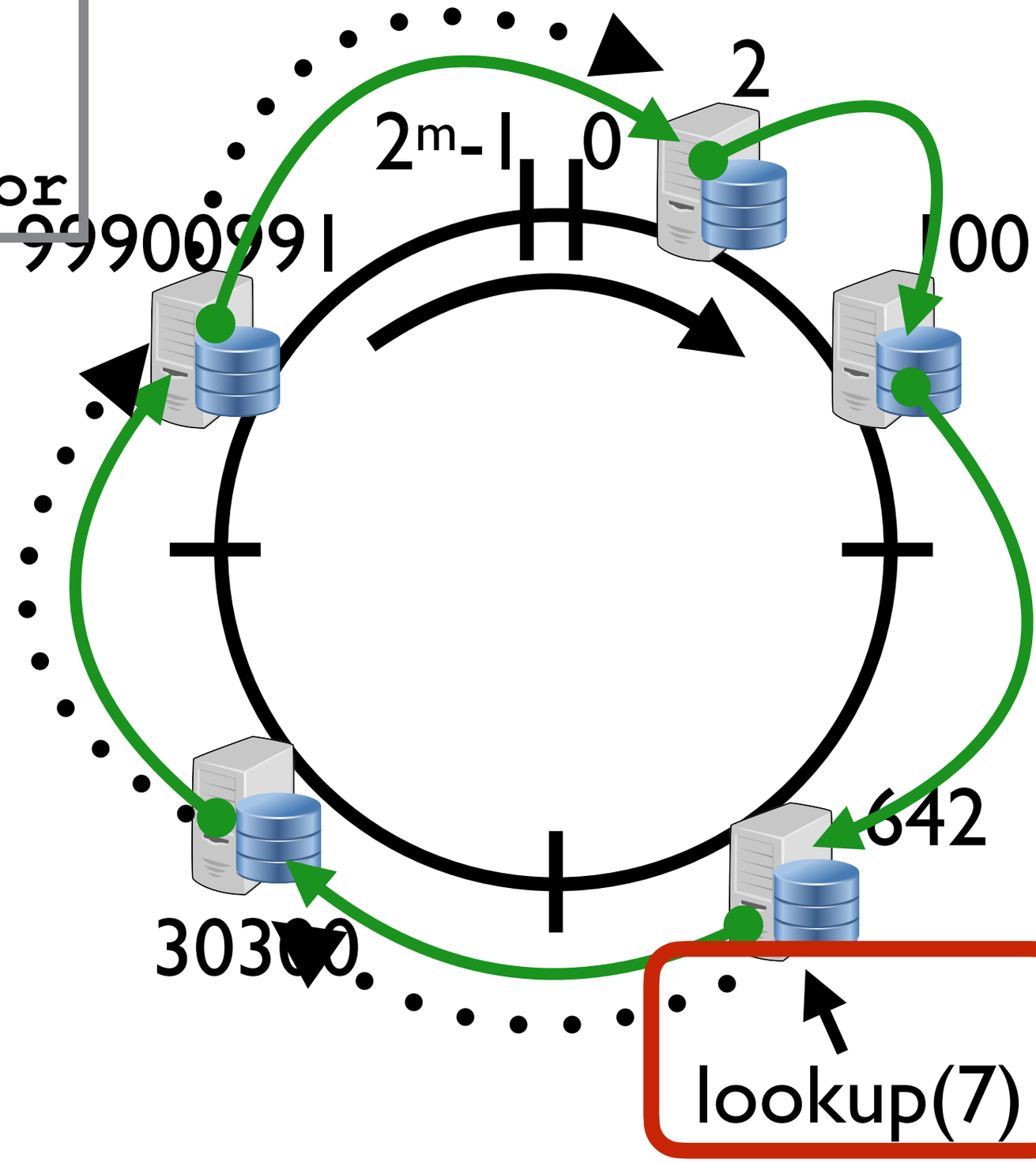
Basic Routing — Slow

```
n.lookup(k):
   if n < k <= n.successor
      return n.successor
   else
    forward to n.successor
```



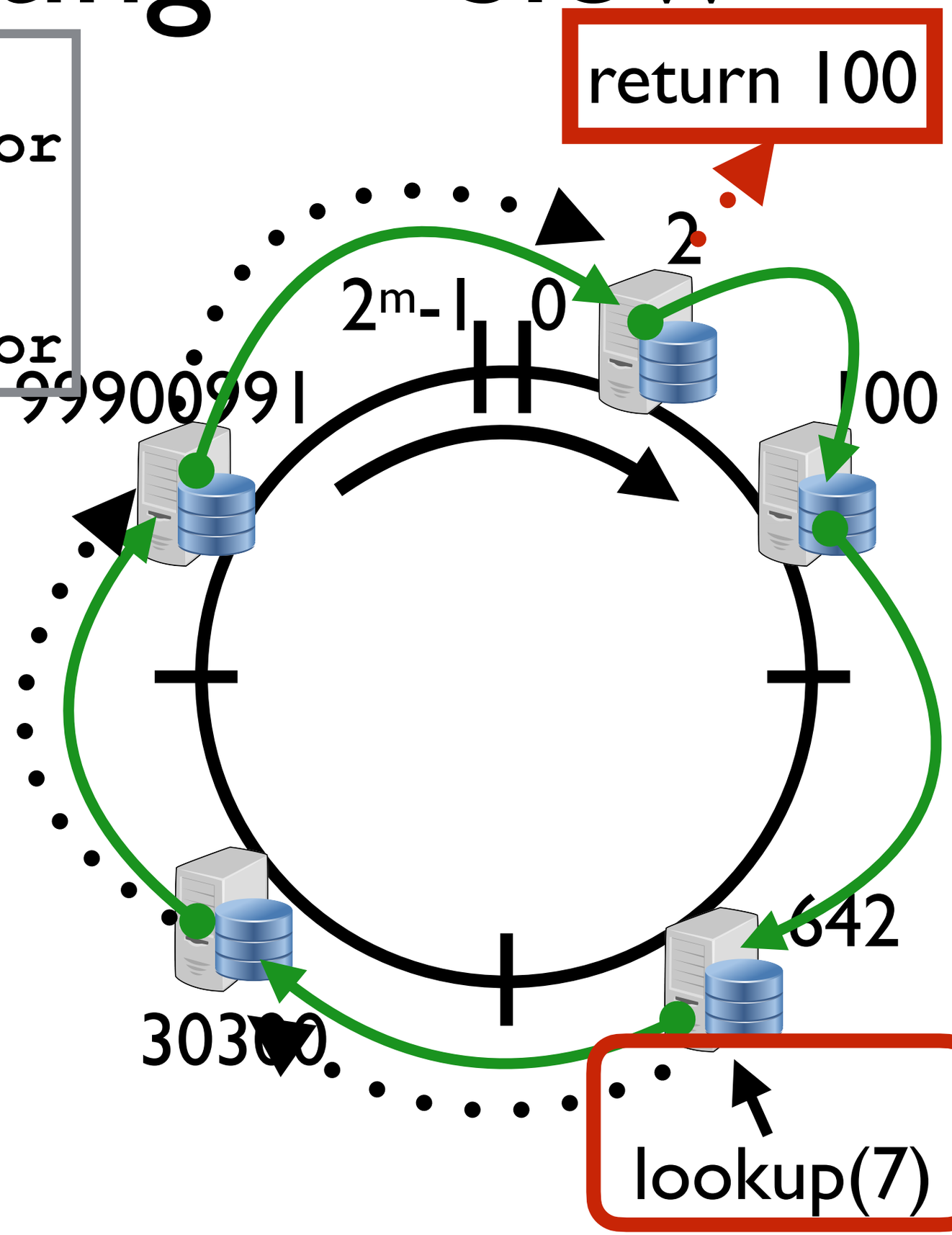
Basic Routing — Slow

```
n.lookup(k):
   if n < k <= n.successor
      return n.successor
   else
    forward to n.successor
```



Basic Routing — Slow

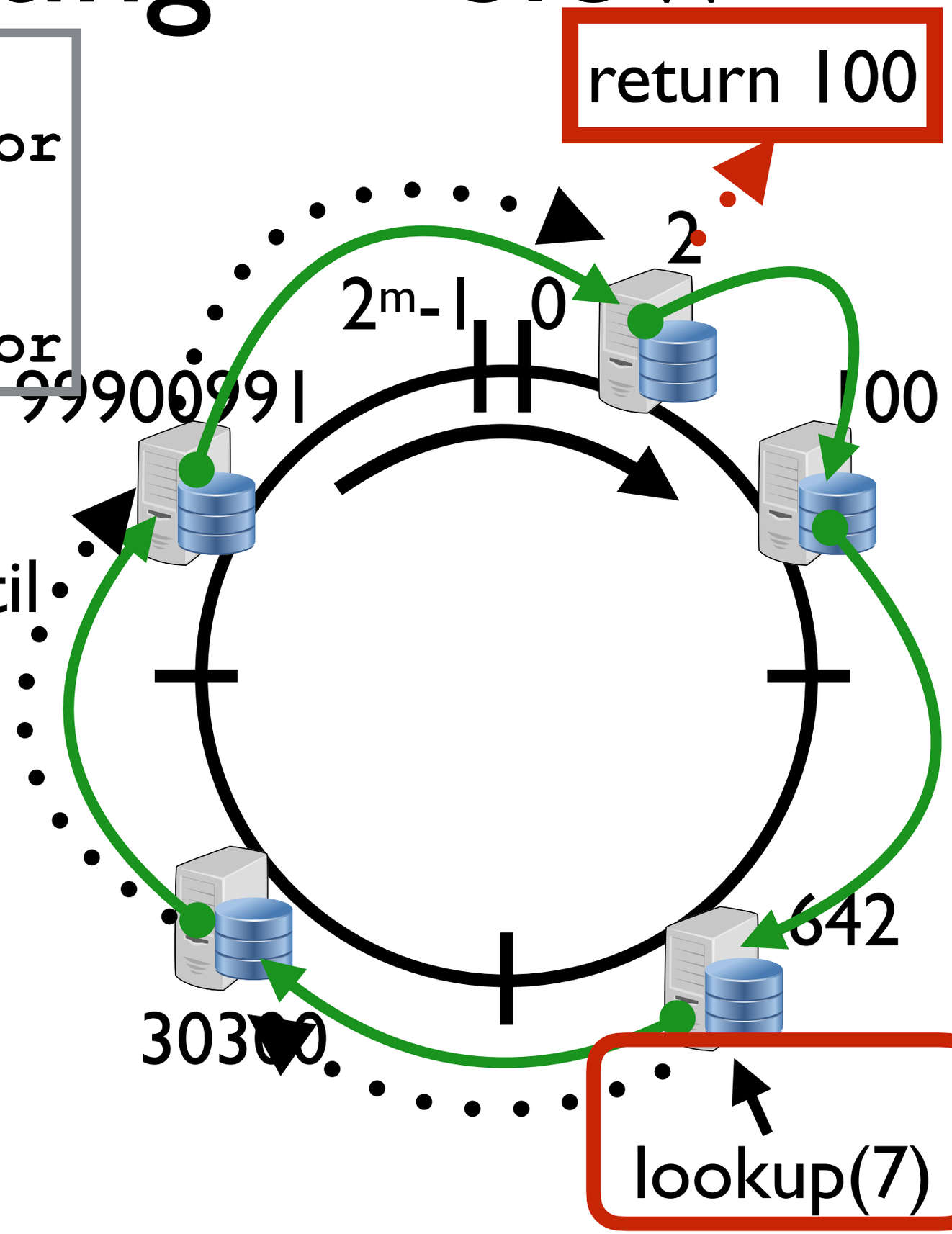
```
n.lookup(k):
   if n < k <= n.successor
      return n.successor
   else
    forward to n.successor
```



Basic Routing — Slow

```
n.lookup(k):
   if n < k <= n.successor
      return n.successor
   else
    forward to n.successor
```

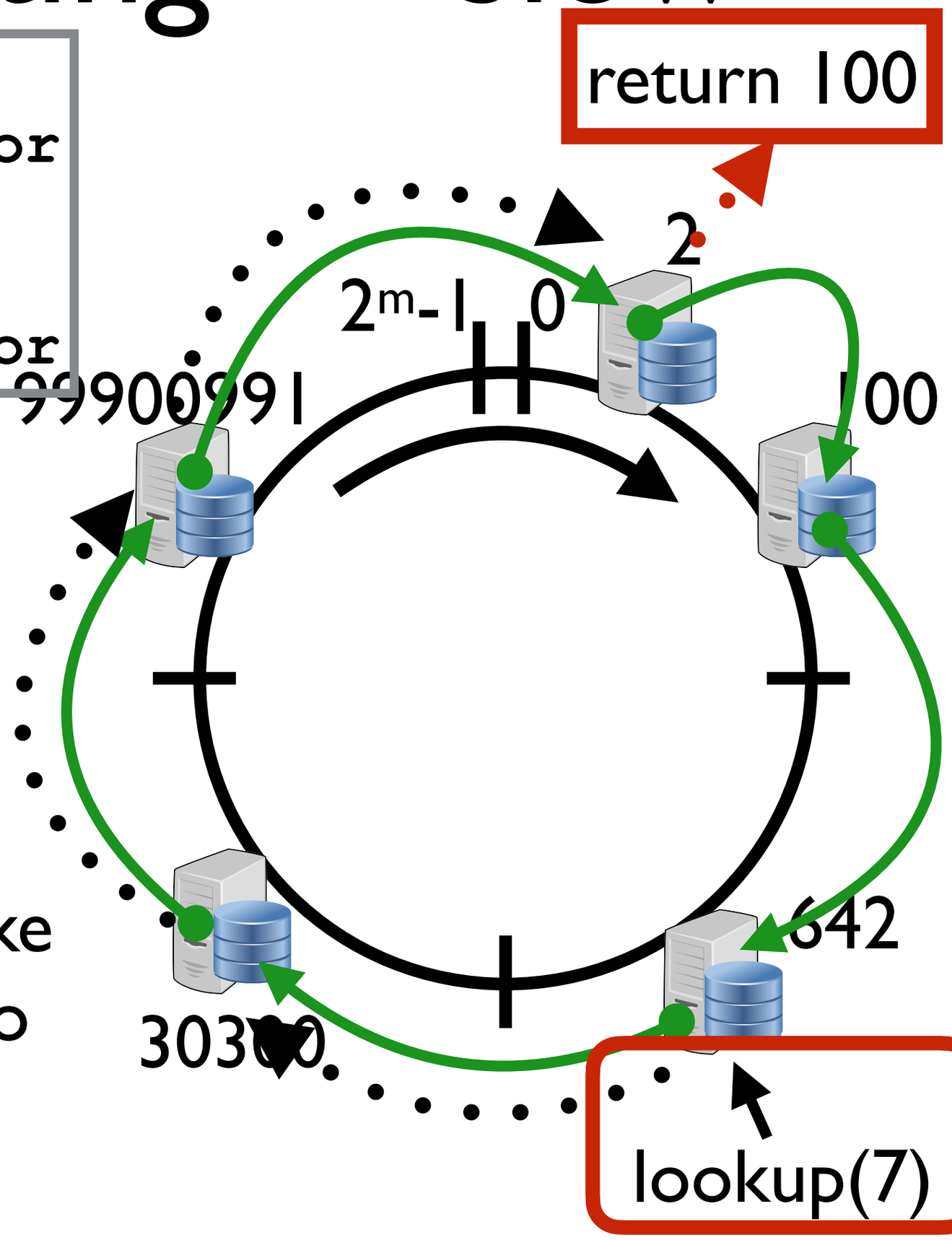
- forward query in clockwise direction until done — n.successor must be correct!
- otherwise we may skip over the responsible node and get(k) won't see data of put(k)



Basic Routing — Slow

```
n.lookup(k):
   if n < k <= n.successor
      return n.successor
   else
    forward to n.successor
```

- Forwarding through successor is slow
  - Data structure is a linked list: O(n)
- Can we make it more like a binary search? Need to be able to halve the distance at each step.

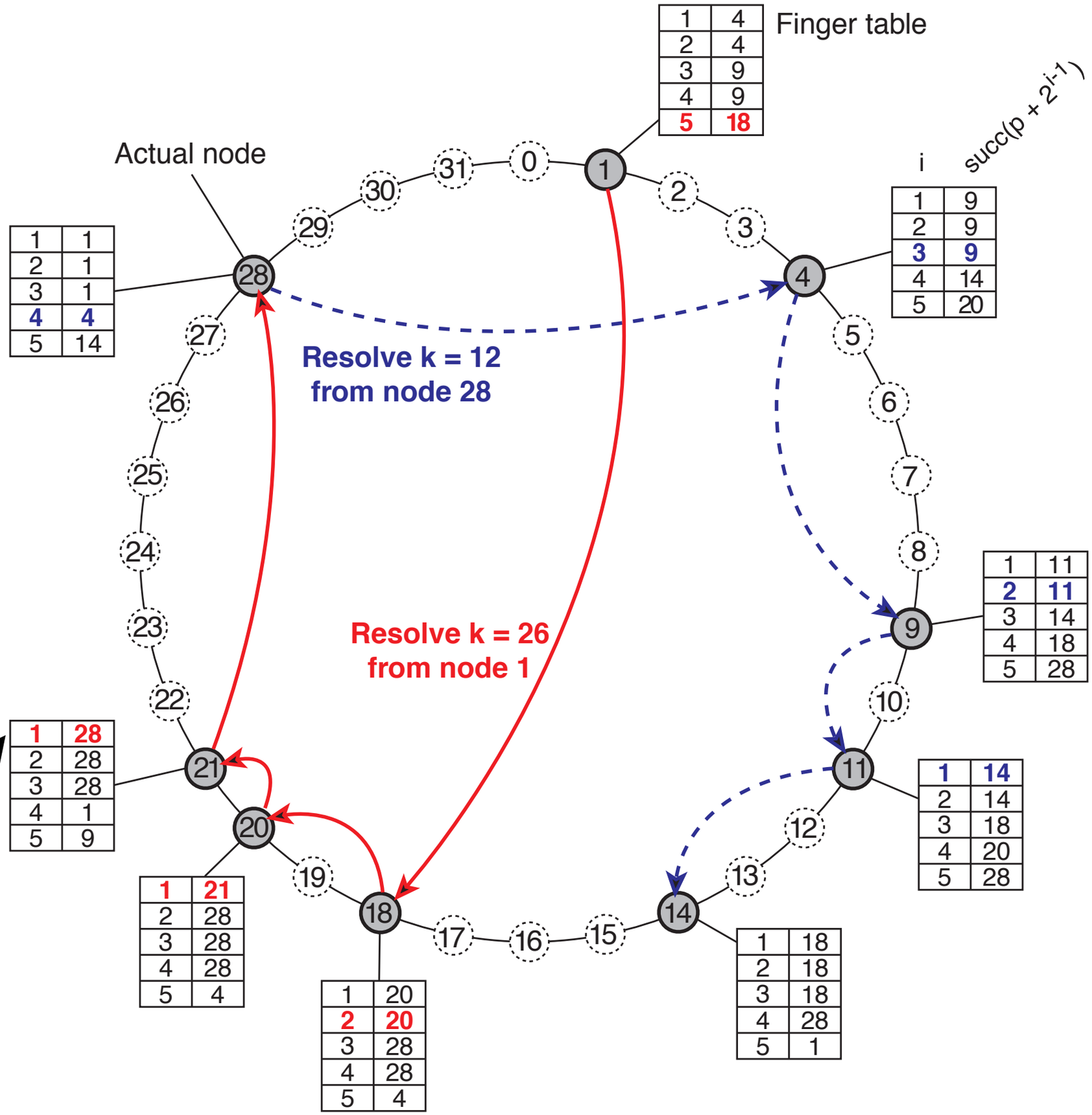


log(n) "finger table" routing

Keep track of nodes exponentially further away

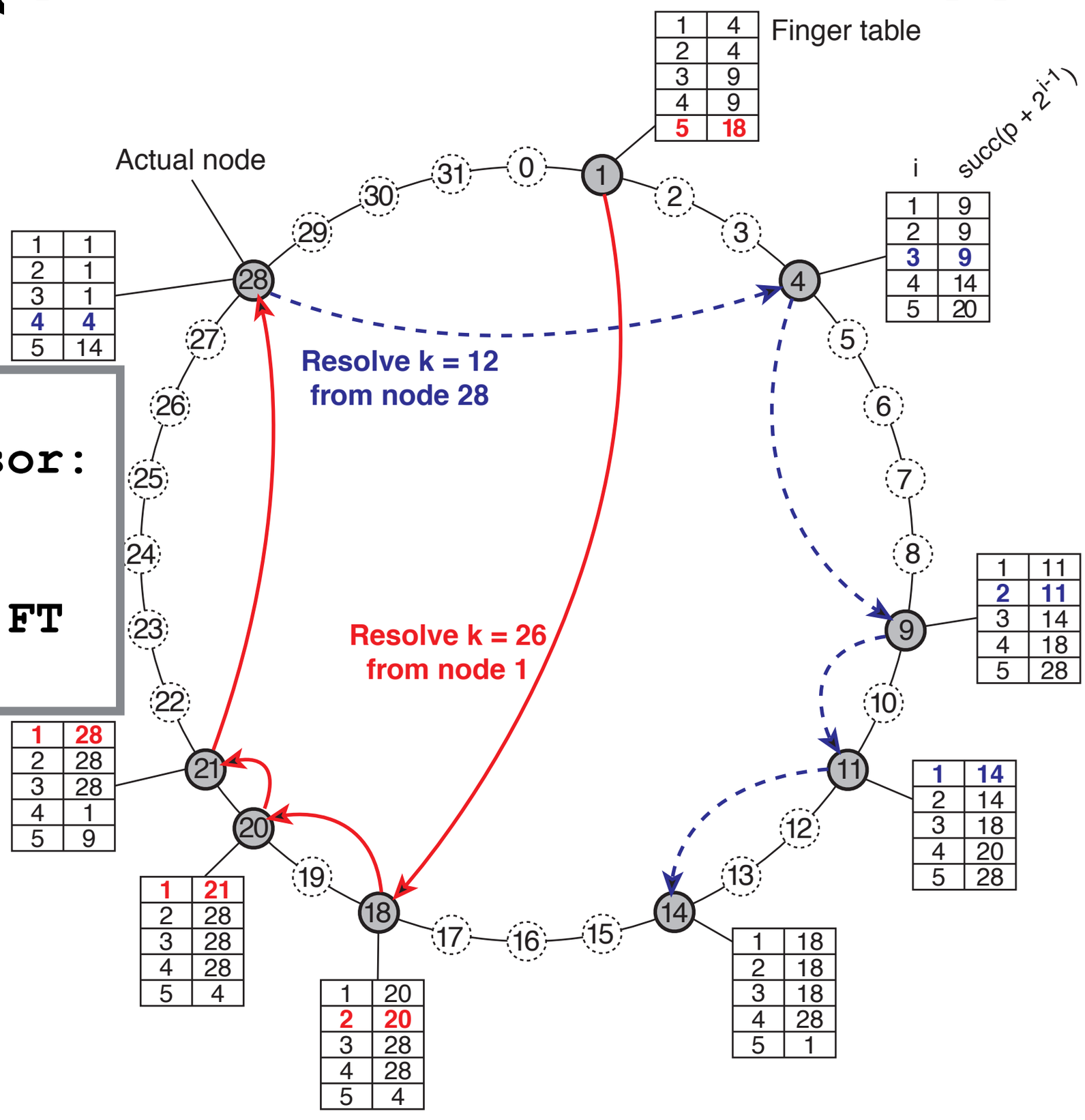
i-th row in finger table: f[i] contains successor of the point: $n + 2^{i-1}$

First finger of node 21:
$21 + 2^{1-1} = 22$
(not shown in the table)
Successor node of first finger: 28



log(n) "finger table" routing

```
n.lookup(k):
 if n < k <= n.successor:
   return n.successor
 else:
   n' = cp_node(k) in FT
   forward to n'
```



log(n) "finger table" routing

```
n.lookup(k):
 if n < k <= n.successor:
    return n.successor
 else:
    n' = cp_node(k) in FT
    forward to n'
```

i	target	succ(target)
1	$n+2^0=n+1$	4
2	$n+2^1=n+2$	4
3	$n+2^2=n+4$	9
4	$n+2^3=n+8$	9
5	$n+2^4=n+16$	18

succ(n+2^(i-1))

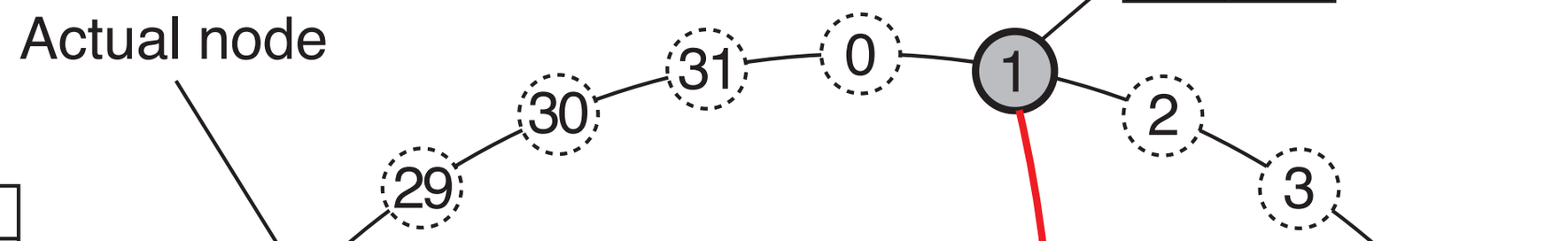
i

1	4
2	4
3	9
4	9
5	**18**

Finger ta

Actual node

1	1



log(n) "finger table" routing

```
n.lookup(k):
 if n < k <= n.successor:
    return n.successor
 else:
    n' = cp_node(k) in FT
    forward to n'
```

i	target	succ(target)
1	$n+2^0=n+1$	4
2	$n+2^1=n+2$	4
3	$n+2^2=n+4$	9
4	$n+2^3=n+8$	9
5	$n+2^4=n+16$	18

Why do lookups now take log(n) hops?

log(n) "finger table" routing

```
n.lookup(k):
 if n < k <= n.successor:
   return n.successor
 else:
   n' = cp_node(k) in FT
   forward to n'
```

i	target	succ(target)
1	$n+2^0=n+1$	4
2	$n+2^1=n+2$	4
3	$n+2^2=n+4$	9
4	$n+2^3=n+8$	9
5	$n+2^4=n+16$	18

Why do lookups now take log(n) hops?

One of the fingers must take you roughly half-way to your destination

log(n) "finger table" routing

```
n.lookup(k):
 if n < k <= n.successor:
   return n.successor
 else:
   n' = cp_node(k) in FT
   forward to n'
```

i	target	succ(target)
1	$n+2^0=n+1$	4
2	$n+2^1=n+2$	4
3	$n+2^2=n+4$	9
4	$n+2^3=n+8$	9
5	$n+2^4=n+16$	18

There's a binary lookup tree rooted at every node

Threaded through other nodes' finger tables

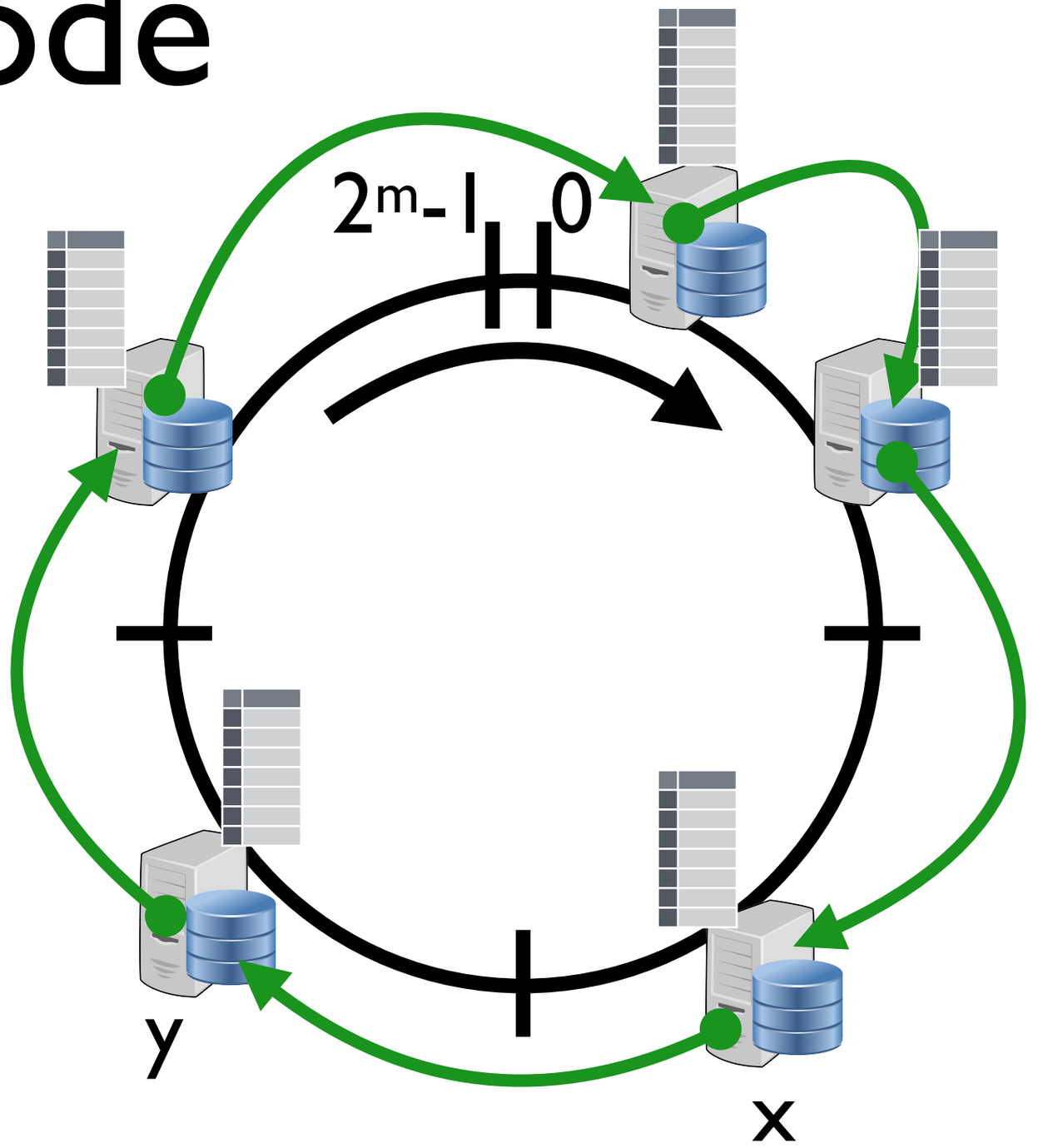
This is *better* than simply arranging the nodes in a single tree

Every node acts as a root, so there's no root hotspot

But a lot more state in total

How to init FT of new Node

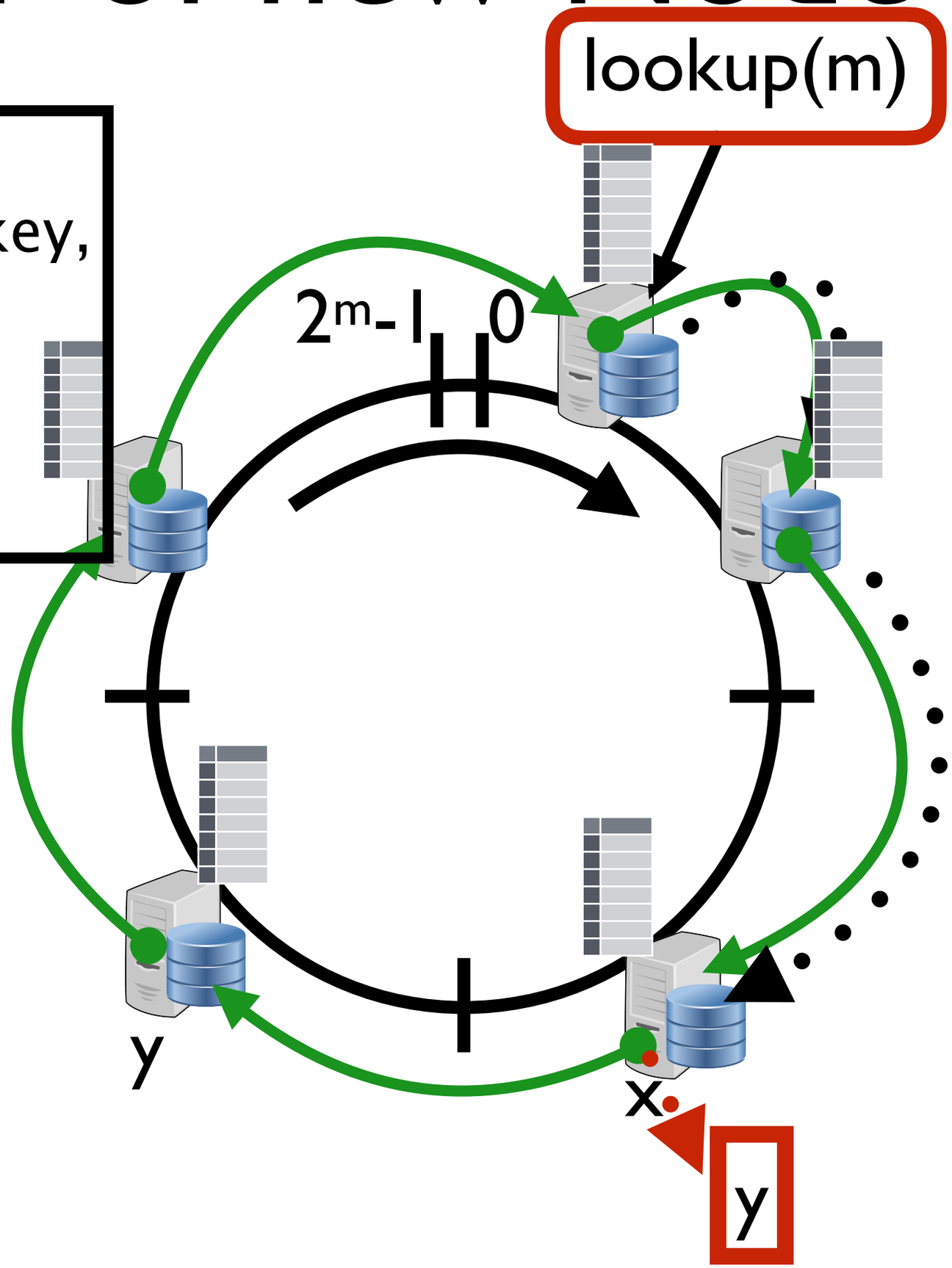
- Assume system starts out with correct routing tables.
  - Use routing tables to help the new node find information.
  - Add new node in a way that maintains correctness



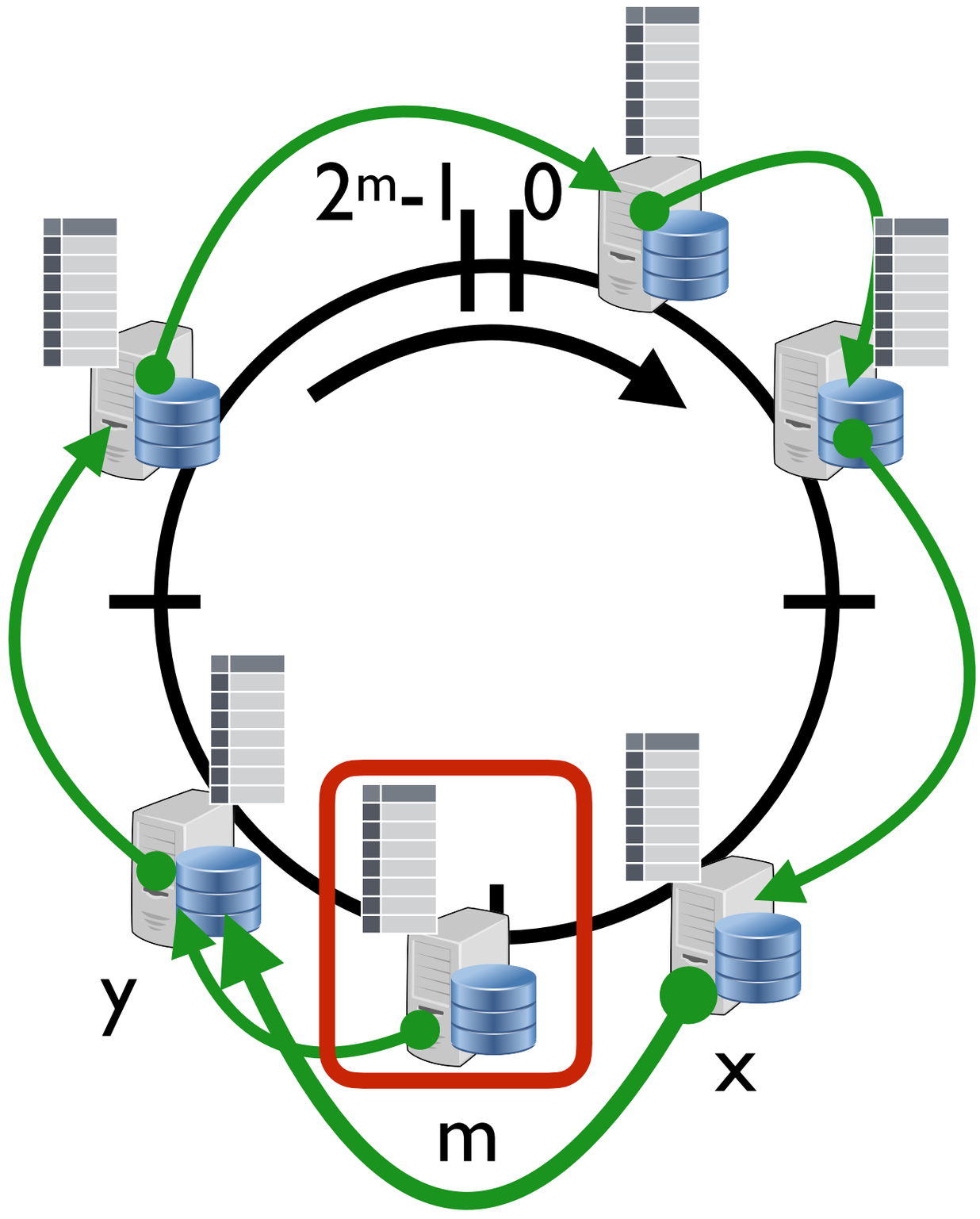
How to init FT of new Node

New node m:
Sends a lookup for its own key, to any existing node.
This yields m.successor
m asks its successor for its entire finger table.

At this point the new node can forward queries correctly
Tweaks its own finger table in background
By looking up each m + 2^{i-1}



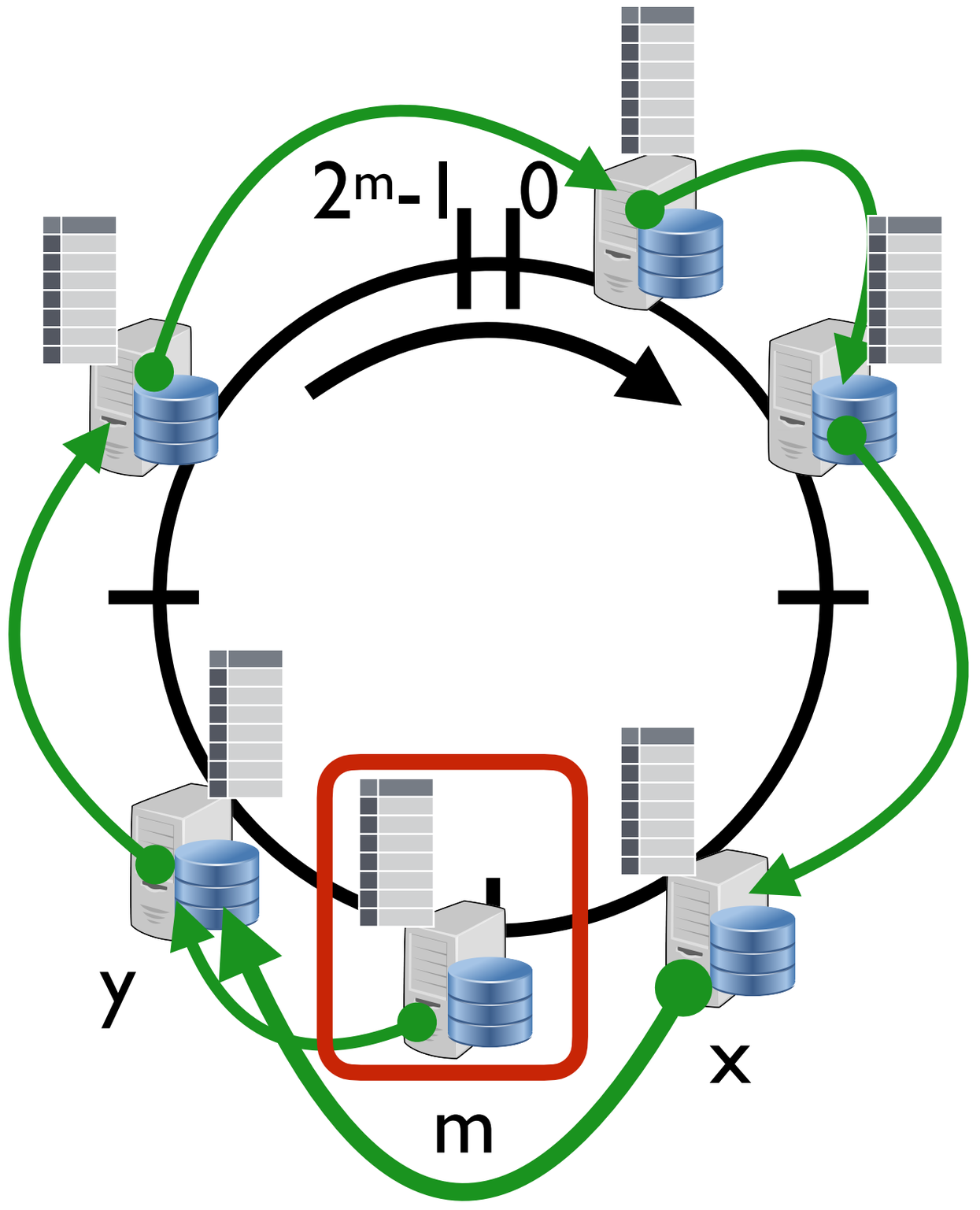
Will Routing work after?



Will Routing work after?

If m doesn't do anything, lookup will go to where it would have gone before m joined.

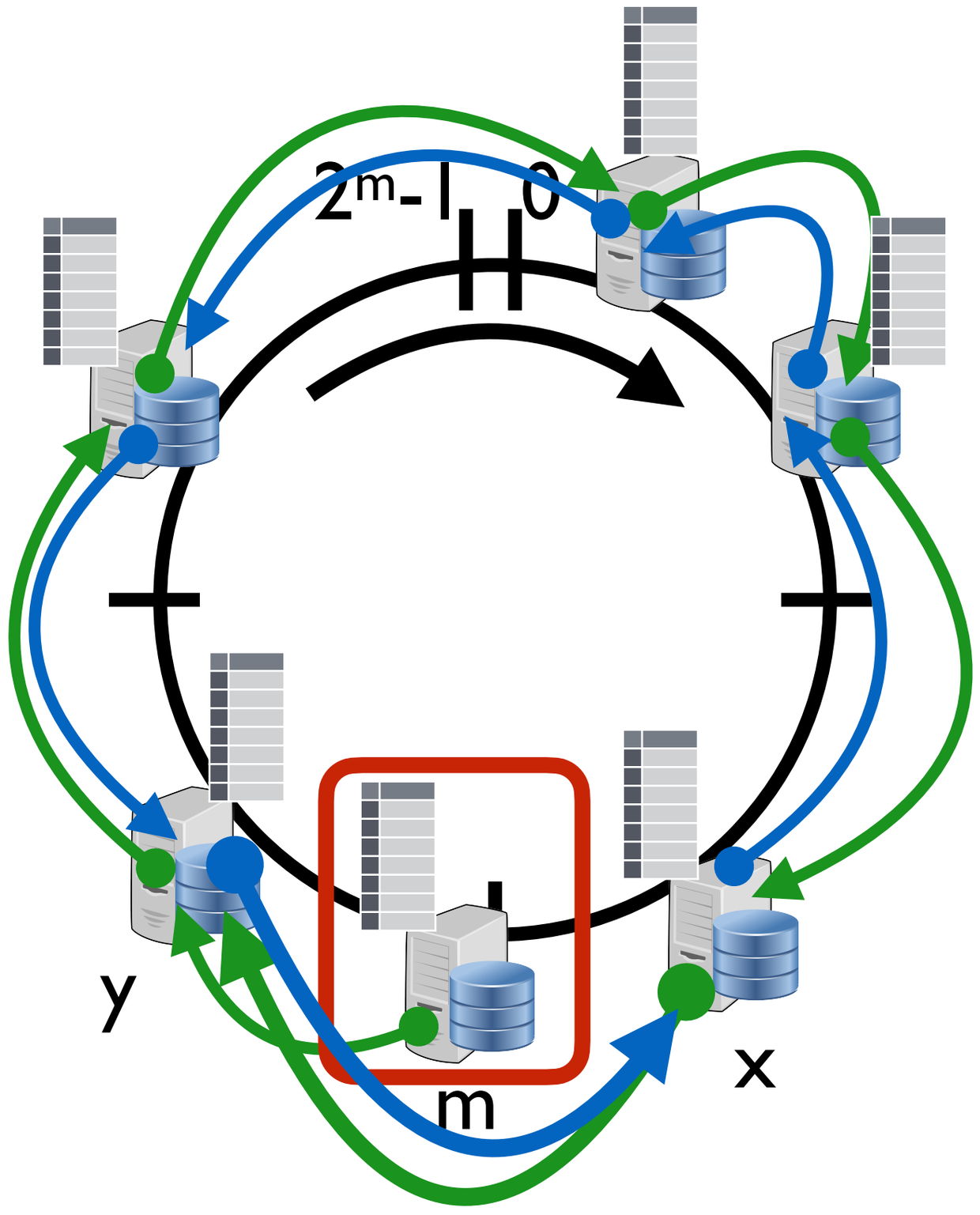
eg. To m's predecessor. Which will return its n.successor -- which is not m.



Will Routing work after?

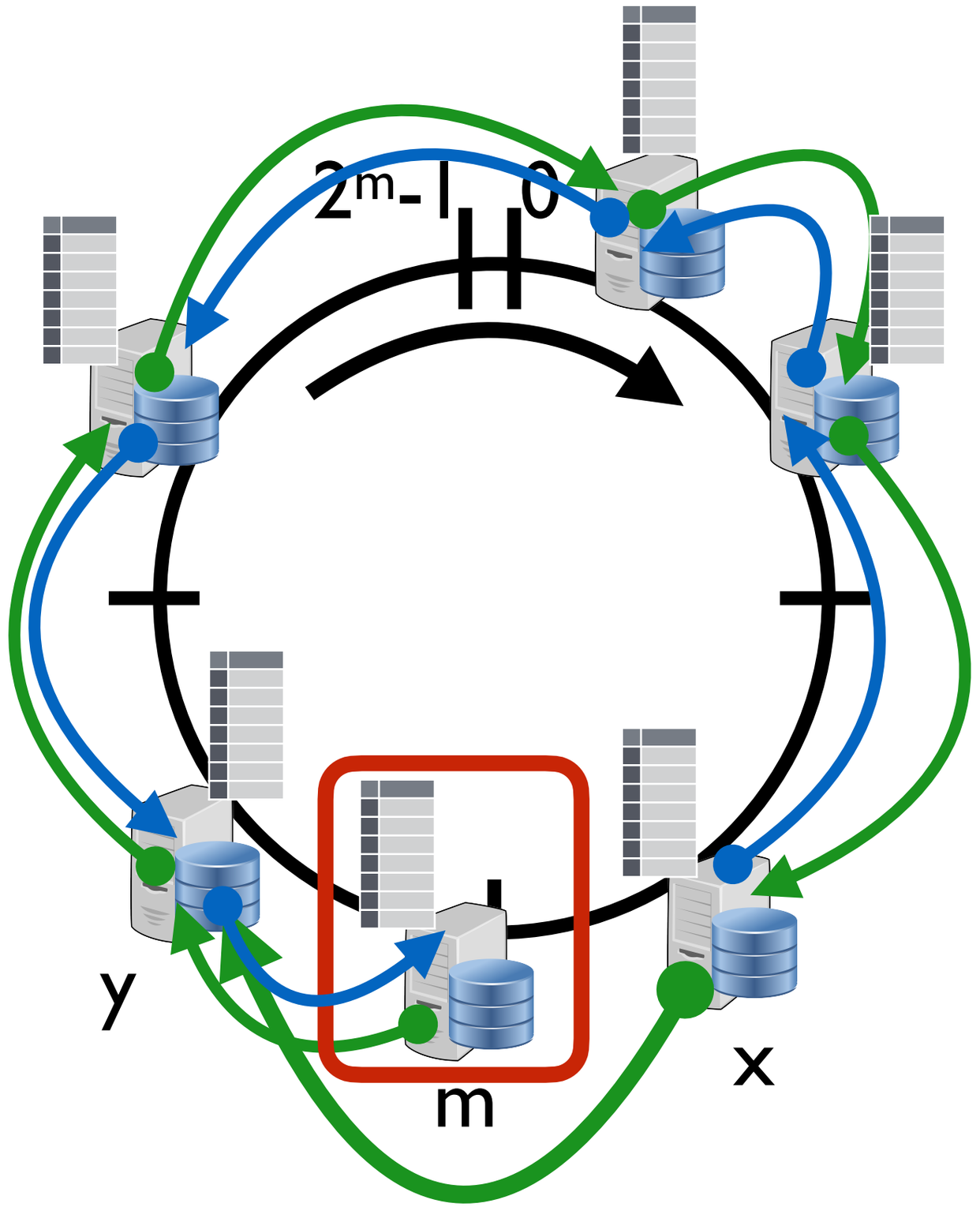
- For correctness, m's predecessor needs to set successor to m.

- Each node keeps track of its current predecessor.



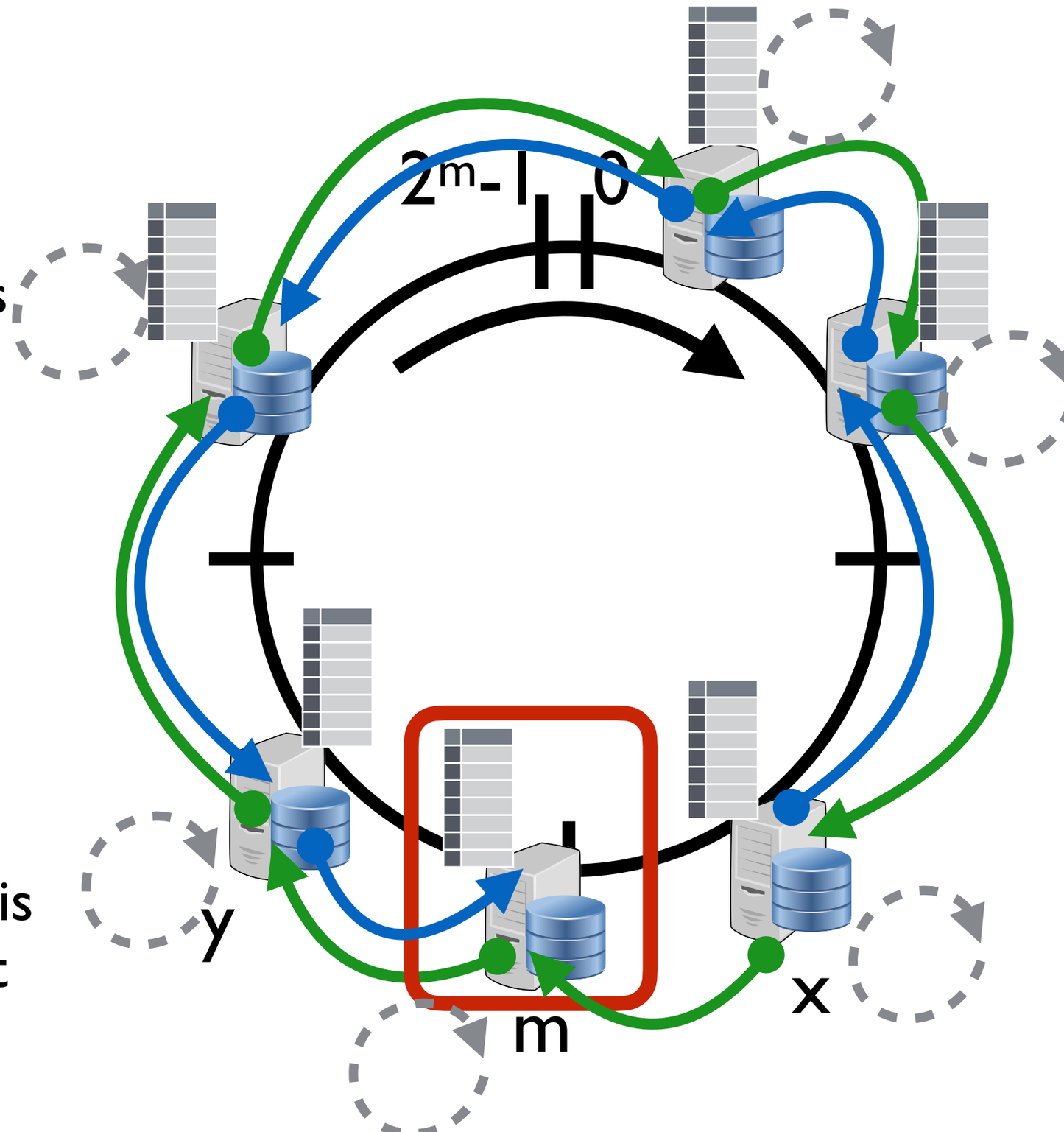
Will Routing work after?

- For correctness, m's predecessor needs to set successor to m.
  - Each node keeps track of its current predecessor.
  - When m joins, tells its successor that its predecessor has changed.



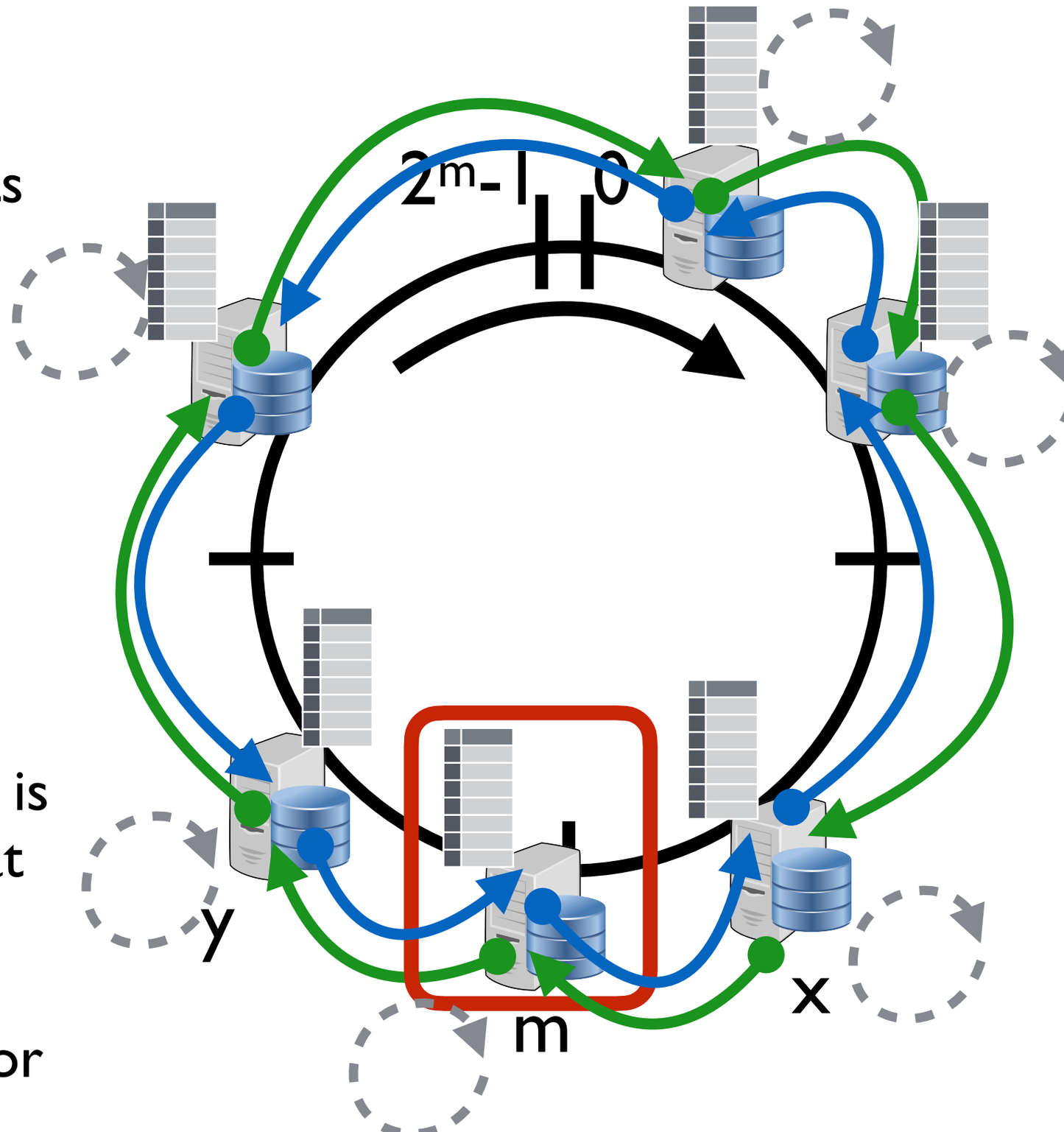
Will Routing work after?

- For correctness, m's predecessor needs to set successor to m.
  - Each node keeps track of its current predecessor.
  - When m joins, tells its successor that its predecessor has changed.
  - Periodically ask your successor who its predecessor is: If that node is closer to you, switch to that node as your successor



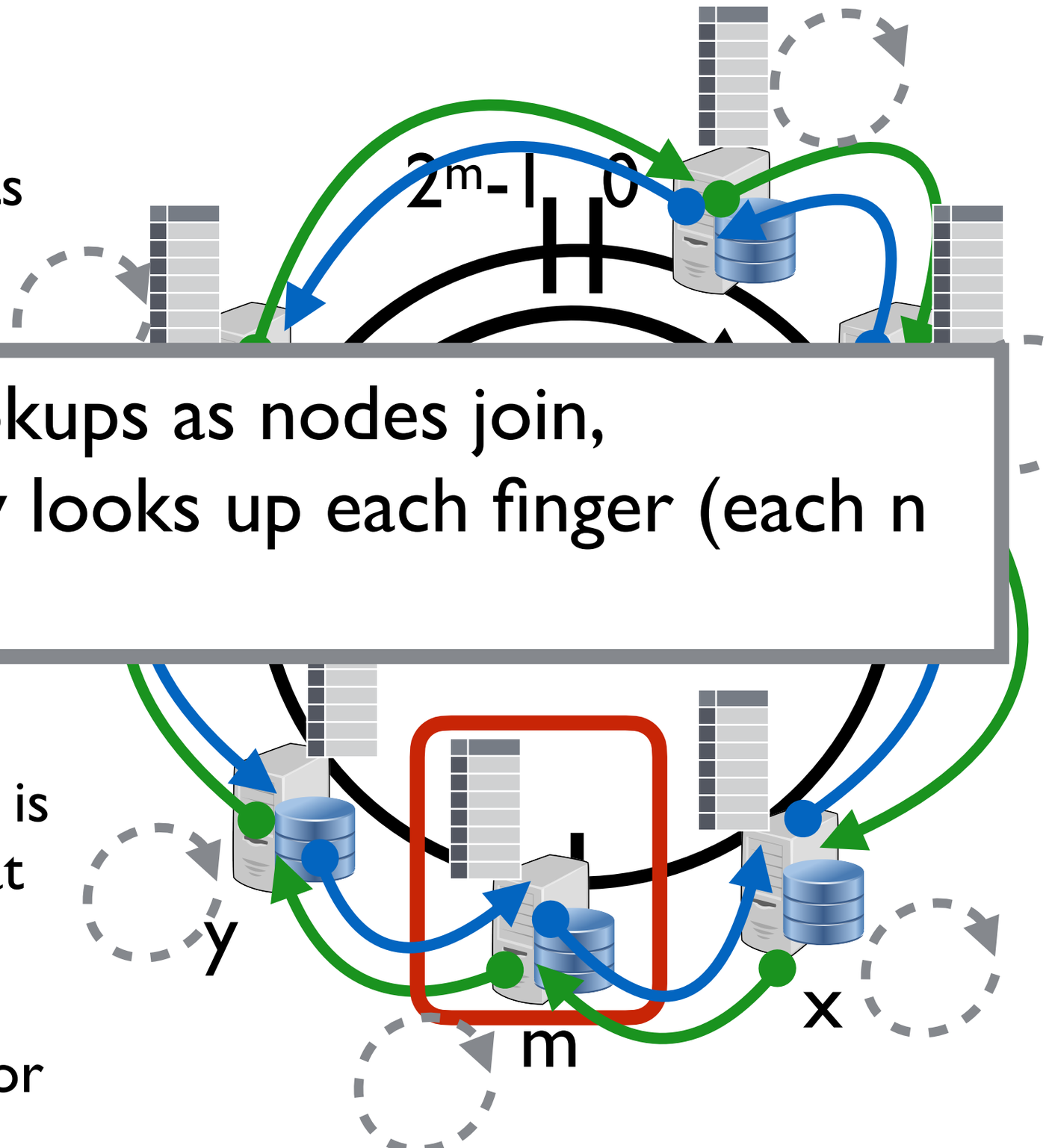
Will Routing work after?

- For correctness, m's predecessor needs to set successor to m.
  - Each node keeps track of its current predecessor.
  - When m joins, tells its successor that its predecessor has changed.
- Periodically ask your successor who its predecessor is: If that node is closer to you, switch to that node as your successor. notify new successor that you might be its predecessor



Will Routing work after?

- For correctness, m's predecessor needs to set successor to m.
  - Each node keeps track of its current predecessor.
  - Periodically ask your successor who its predecessor is: If that node is closer to you, switch to that node as your successor. notify new successor that you might be its predecessor

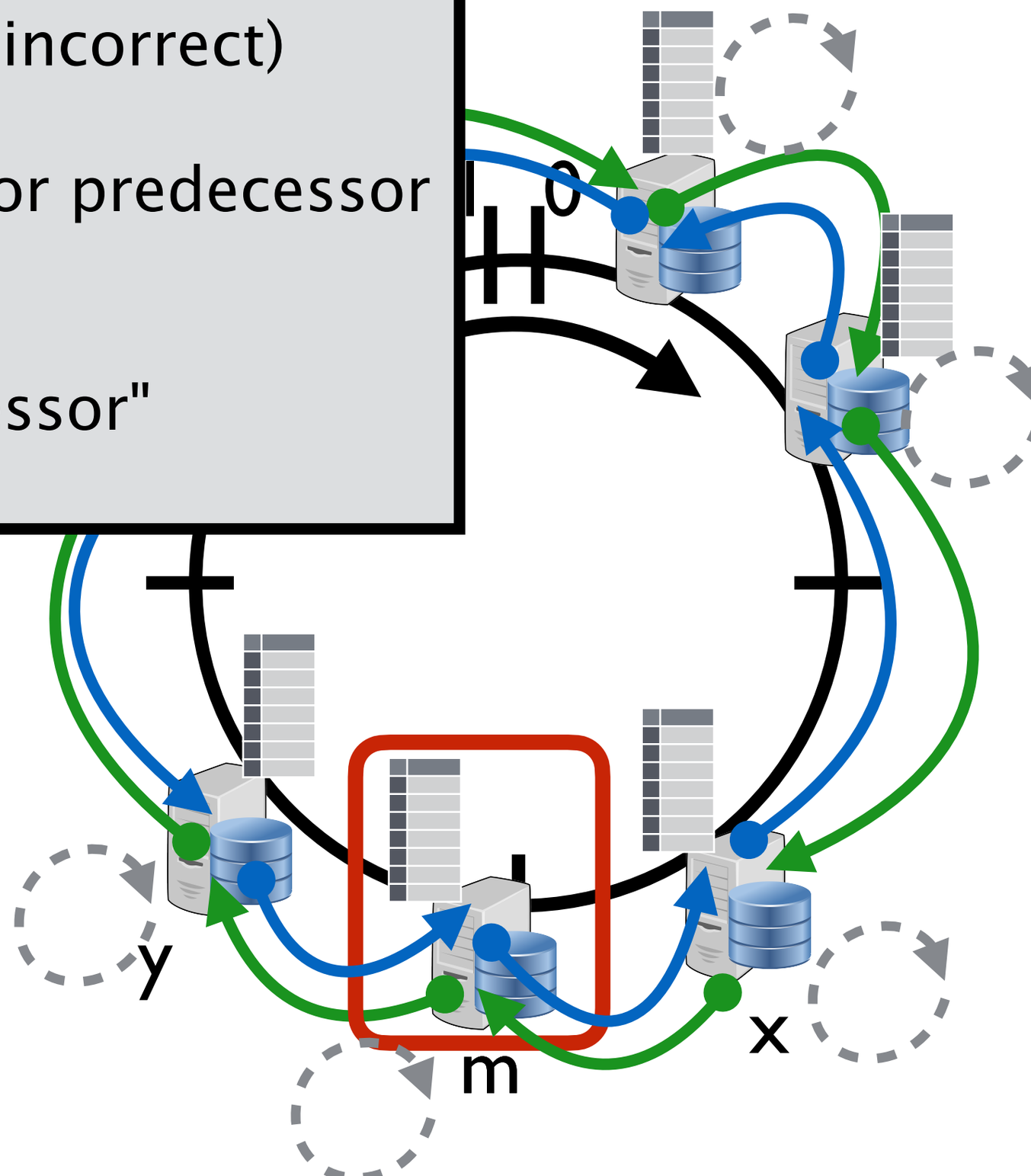


To maintain log(n) lookups as nodes join, Every one periodically looks up each finger (each n + 2^i)

Does Routing now work?

So if we have x m y
- x.successor will be y (now incorrect)
- y.predecessor will be m
- x will ask its x.successor for predecessor
  - x learns about m
  - sets x.successor to m
  - tells m "x is your predecessor"
  - called "stabilization"

Correct successors are sufficient for correct lookups!



Does Routing now work?

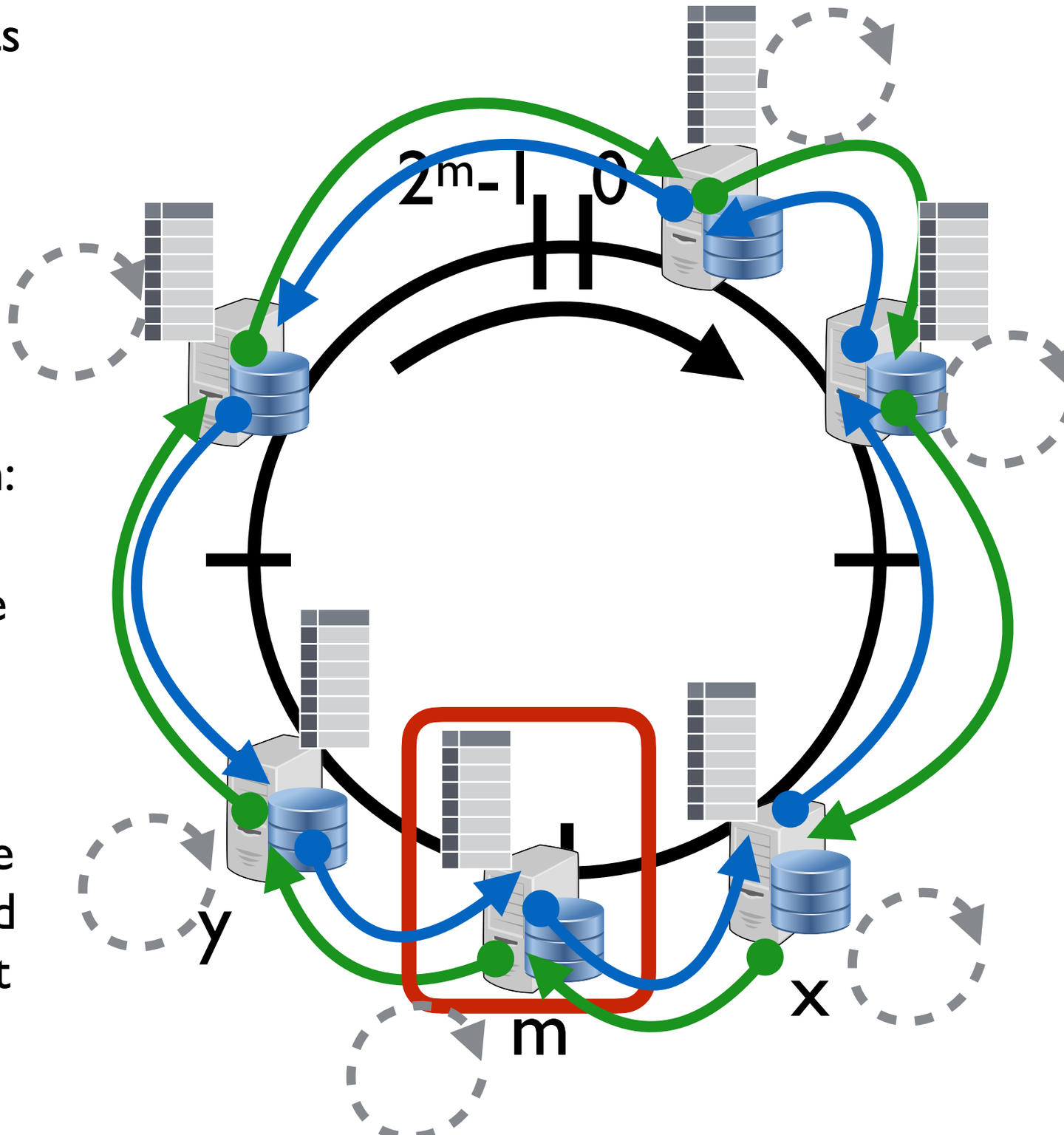
- Whenever a node updates its successor pointer it also updates its 1st FT entry

- If a lookup is forwarded based on 1st entry, then no miss

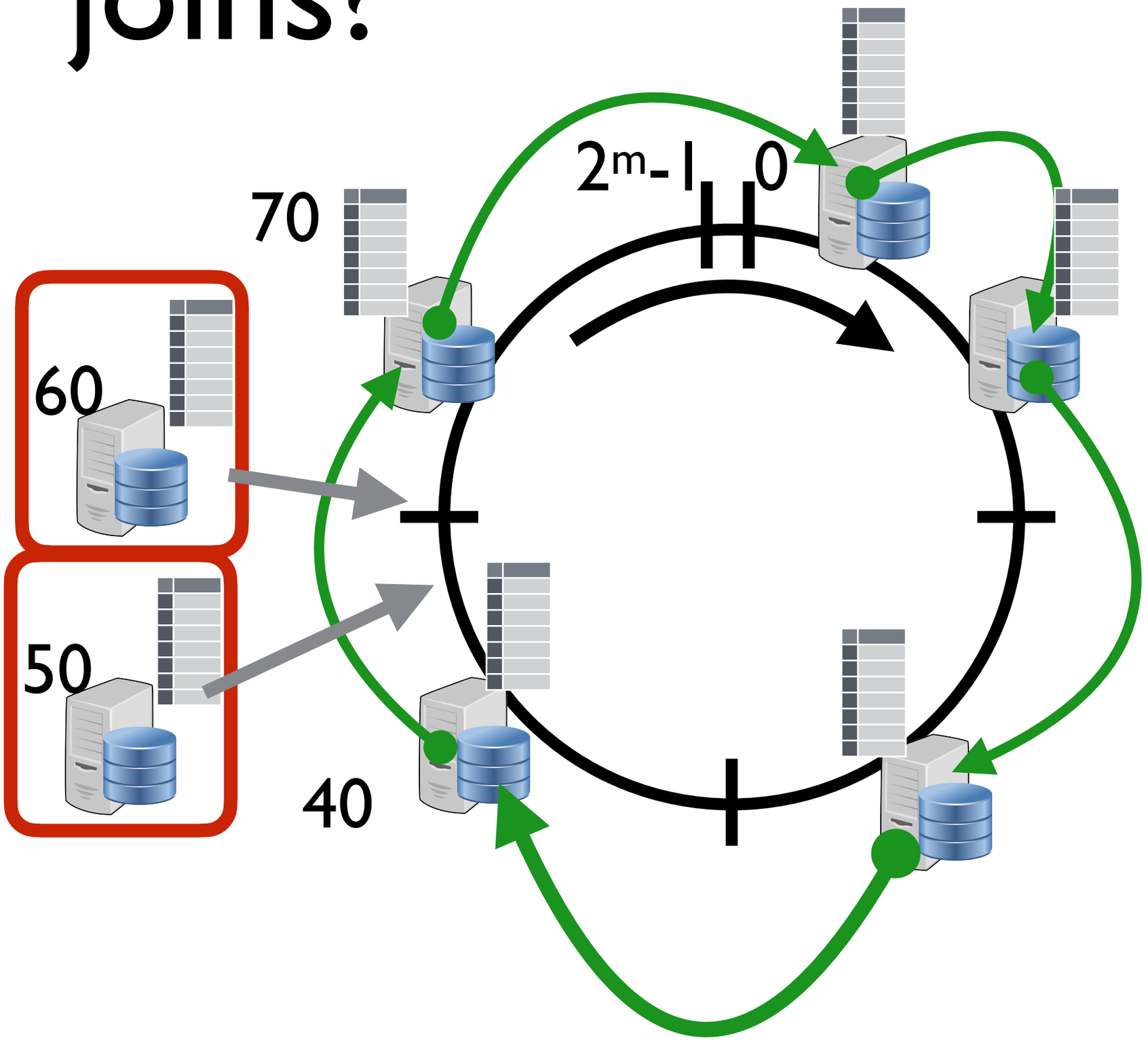
- If another node forwards a lookup based on an outdated entry, let i-th:

  - If there is a new node, let m, the request will be forwarded to a node that precedes m

  - If a node is removed and the the i-th entry points to the removed node, then the RPC will timeout and the node will use the previous entry i-1



What about concurrent joins?

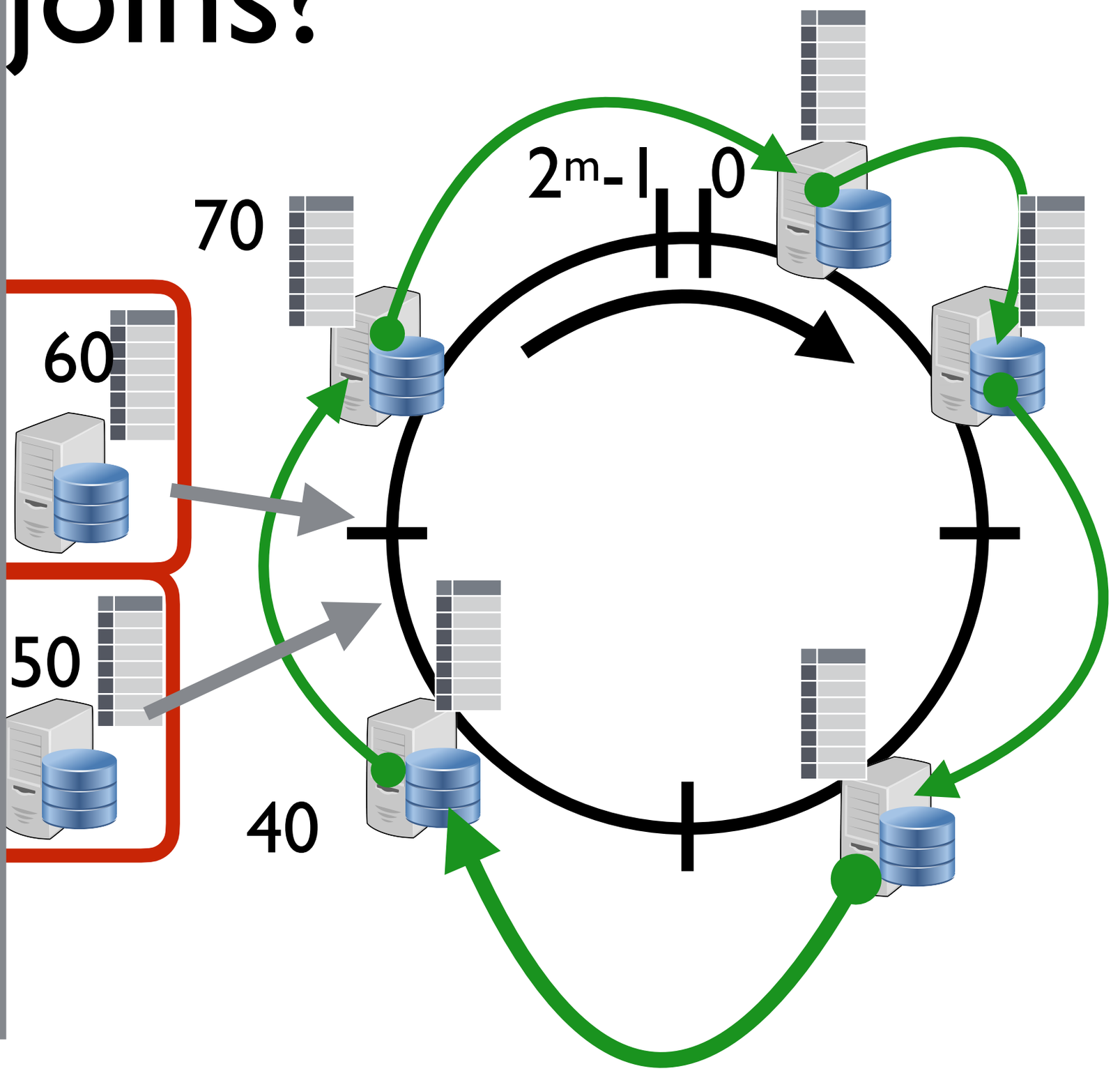


What about concurrent joins?

At first 40, 50, and 60 think their successor is 70!

Which means lookups for e.g. 45 will yield 70, not 50

However, after one stabilization, 40 and 50 will learn about 60 then 40 will learn about 50



Retrospective

- DHTs seem very promising for finding data in large p2p systems
- Decentralization seems good for load, fault tolerance
- But: the security problems are difficult
- But: churn is a serious problem, particularly if log(n) is big
- So DHTs have not had the impact that many hoped for