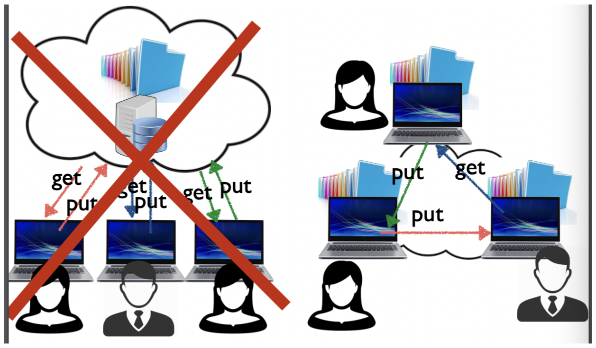
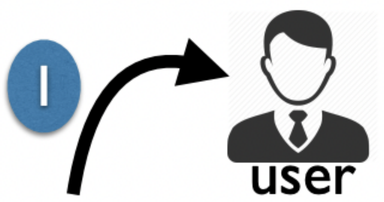
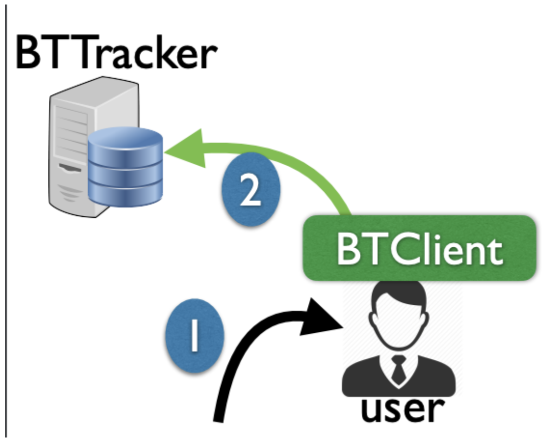
CAS CS 350
Lec 24

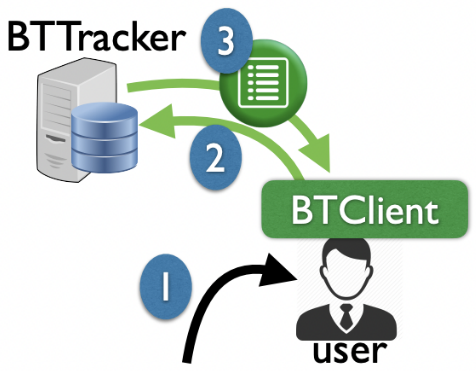
Peer to Peer Systems

1. Peer to Peer
    a. Distributed system where computers talk directly to each other without the need of server in between (no coordinator, master, driver)
    b. All nodes (systems) are equal
    c. They follow the same protocol and have same responsibilities
    d. 

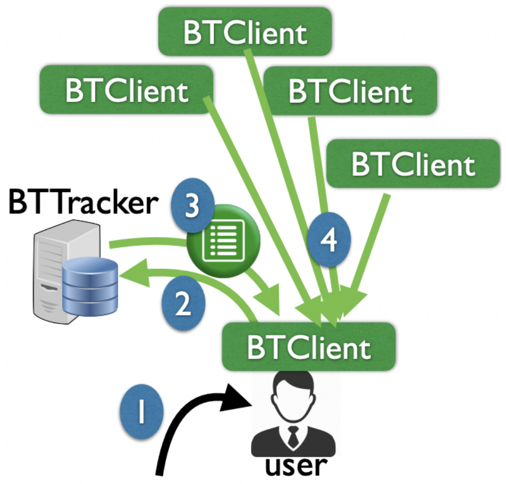
2. Why might P2P be a win?
    a. Decentralized
    b. Better load balancing
    c. Spreads network/caching costs over users
    d. Absence of server may mean
        i. Easier to deploy
        ii. Less chance of overload
        iii. Single failure won't wreck the whole system (resilience to failures)
        iv. Harder to attack
3. Why not everything P2P?
    a. Hard to guarantee consistency (existence of leader simplifies the system)
        i. Linearizability → clients cannot see stale data
    b. User computers not as reliable than managed servers
    c. If open, can be attacked via email participants
    d. Can be hard to find data items over millions of users (files we want to download may be offline), difficult to locate the files effectively
4. In practice P2P has some successful niches
    a. Client-client video/music where serving costs are high
    b. Chat (user to user anyway; privacy and control)
    c. Popular data but owning organization has no money
    d. No natural single owner or controller (Bitcoin)
    e. Illegal file sharing ("Metallica vs Napster Inc.")

5. Classic BitTorrent (users can download files from other users without using servers)
    a. Torrent file contains content hash and IP address of tracker
    b. Tracker know clients/locations that already has a copy of the file
    c. Transfers happens as massively parallel sending of chucks from other clients
    d. Huge download bandwidth without expensive server/link
    e. 

    f. Users get back a list of download (user clicks a link and gets a torrent file → has the hash value of request torrent, and ID of the tracker)
    g. 

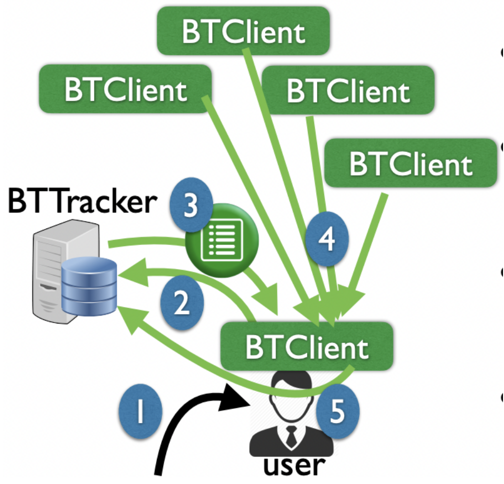
    h. Contact tracker
    i. 

    j. Returns torrent



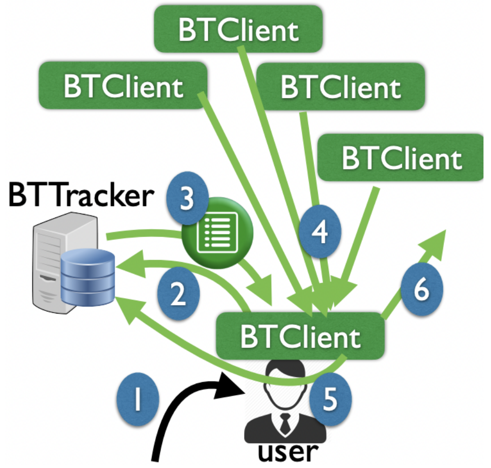
k.

l. Contact the servers directly



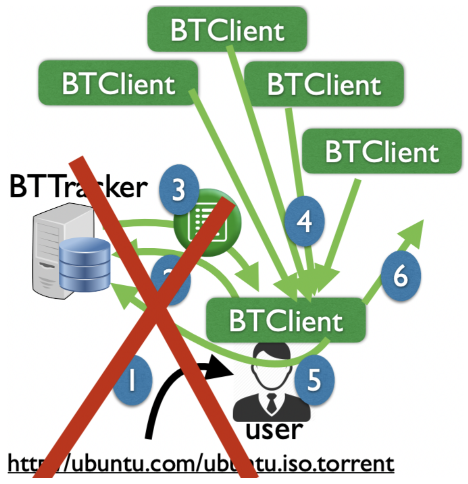
m.

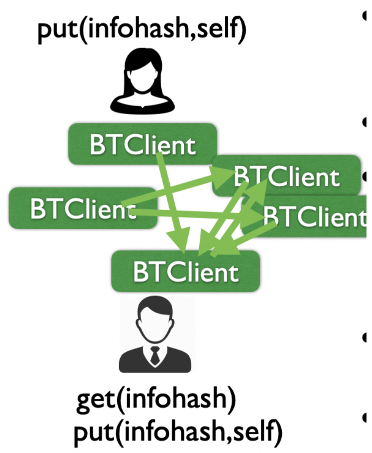
n. Servers return to the user and user downloads them in parallel



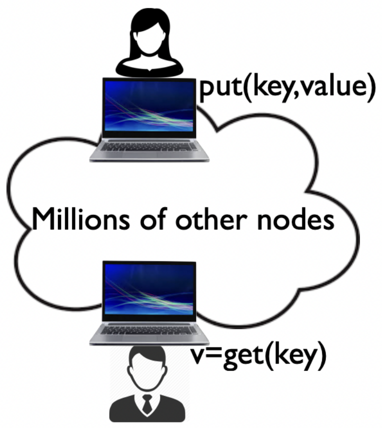
o.

p. User can send it to other users since it has the torrent

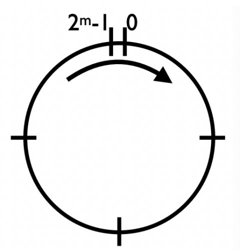
6. Eliminate Tracker using DHT
    a. 

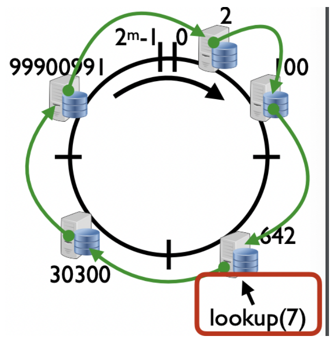
    b. BT clients cooperatively implement a giant key/value store
    c. 

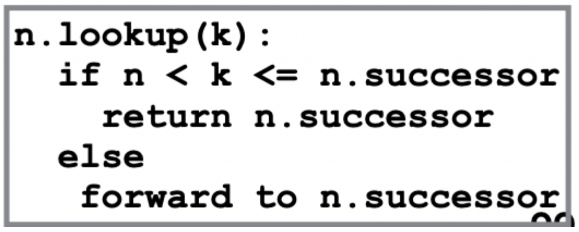
    d. BT clients cooperatively implement a giant key/value store - distributed hash table
    e. Key is file content hash "infohash"
    f. Value is IP address of client willing to server the file
        i. Kademlia can store multiple values for a key
    g. Get to find clients and put to register as willing to serve
    h. Clients join DHT to help implement it
    i. System can load balance client requests/ more resilient to attack (fault tolerance, availability)
7. Why DHT good for BT?
    a. Like having one single giant tracker, less fragmented that many trackers
        i. Clients more likely to find each other
    b. Maybe classic tracker too exposed to legal attacks
    c. In reality not clear that BT depends heavily on DHT mostly backup for classic trackers

8. Scalable DHT lookup
    a. 

    b. key/value store spread over millions of nodes
    c. Interface
        i. put(key,value)
        ii. get(key) → value
    d. Loose consistency (clients can see stale data); likely that get(k) sees put(k), but no guarantee
    e. Loose guarantees about keeping data alive
9. Why is it hard?
    a. Finding the keys is difficult (since no central server exists)
    b. Millions of participating nodes
    c. Could broadcast/flood request – but too many messages
    d. Every node could know about every other node
    e. Hashing is easy then
        i. But keeping a million-node table up to date is hard
    f. Want modest state, and modest number of messages
10. Basic Idea
    a. Impose a data structure (e.g. tree) over the nodes
        i. Each node has references to only a few other nodes
        ii. Lookups traverse the data structure – "routing" – eg. hop from node to node
    b. DHT should route get() to same node as previous put()
    c. Solves the space problem (only stores its children in the system)
    d. All requests go through root node(if it fails, it is not good)
11. The "chord" peer-to-peer lookup system
    a. More flexible organization and efficient communication protocol
    b. DHT implementation

12. The ring
   a. 

   b. Idea of consistent hashing (range of hashed value is mapped or represented as ring
   c. Chord's ID-space topology
      i. Ring: all IDs are 160-bit numbers, viewed in a ring
      ii. Each node has an ID, randomly chosen
13. Keys
   a. Key stored on first node whose ID is equal to or greater than key ID
      i. Closeness is defined as the "clockwise distance"
   b. If node and key IDs are uniform, we get reasonable load balance
   c. So keys IDs should be hashes (e.g. bittorrent infohash)
14. Basic Routing – slow
   a. Query is at some node
      i. Node needs to forward the query to a node "closer" to key
      ii. If we keep moving query closer, eventually we'll win
   b. Each node knows its "successor" on the ring
   c. 

   d. 

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

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

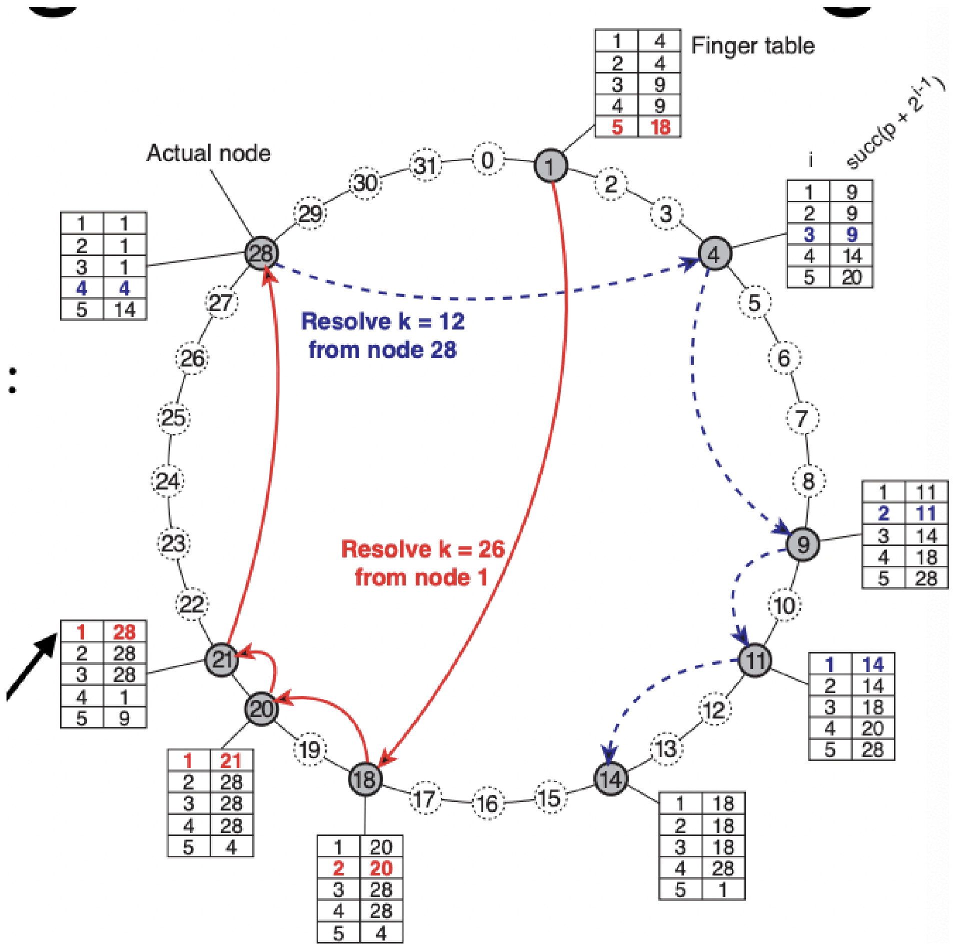
g. Client contacts a node on the ring (decentralized, no single entry)
   i. ask for ID of node that stores particular key
   ii. if it knows the server that stores the key, it contacts that client
   iii. If not, it asks the successor of the node and forwards the request to its successor and the process is repeated (until the owner of the key is found)

h. Node is found by the predecessor

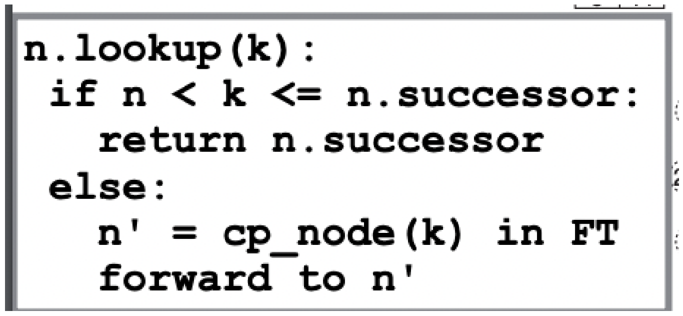
i. It is slow (worst case, traverse all nodes in the ring, which is expensive)
   i. Data structure is a linked list: O(n)

j. Can we do better?
   i. Binary search (sort the ids and keep them in order and do binary search) → in logarithmic time

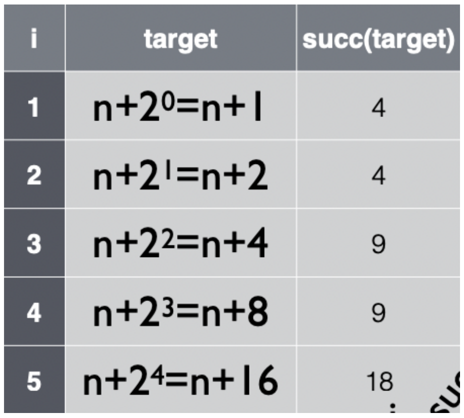
15. log(n) “finger table” routing
   a. Keep track of nodes exponentially further away
   b. I-th row in finger table: f[i] contains successor of the point: n + 2^(i-1)


Finger table
Actual node
i
succ(p + 2^(i-1))
Resolve k = 12 from node 28
Resolve k = 26 from node 1
Node 1: 1 4, 2 4, 3 9, 4 9, 5 18
Node 4: 1 9, 2 9, 3 9, 4 14, 5 20
Node 9: 1 11, 2 11, 3 14, 4 18, 5 28
Node 11: 1 14, 2 14, 3 18, 4 20, 5 28
Node 14: 1 18, 2 18, 3 18, 4 28, 5 1
Node 18: 1 20, 2 20, 3 28, 4 28, 5 4
Node 20: 1 21, 2 28, 3 28, 4 28, 5 4
Node 21: 1 28, 2 28, 3 28, 4 1, 5 9
Node 28: 1 1, 2 1, 3 1, 4 4, 5 14

   c.
   d. ex) first finger of node 21:
      i. 21 + 2^(1-1) = 22
      ii. Successor node of first finger

e. 

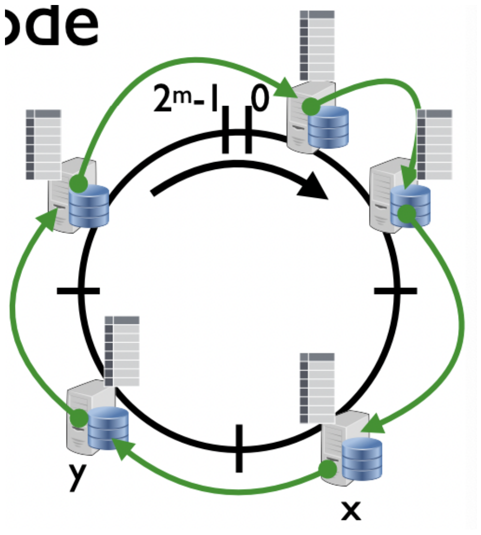
```
n.lookup(k):
 if n < k <= n.successor:
   return n.successor
 else:
   n' = cp_node(k) in FT
   forward to n'
```

f. Each node in this picture represents a finger entry
g. The finger entry contains id of the successor node of that node
    i. 22, 23, 25 → successor node of 28
h. When it receives a lookup request
    i. First check if the next node is responsible for the key
    ii. If so, it replies
    iii. However, if not, node n does not request to the successor node
    iv. Instead, it looks at the local finger table (scans the finger table and find the maximum entry that is before the key that the client requested)
i. Want to find the owner of key 10
    i. 

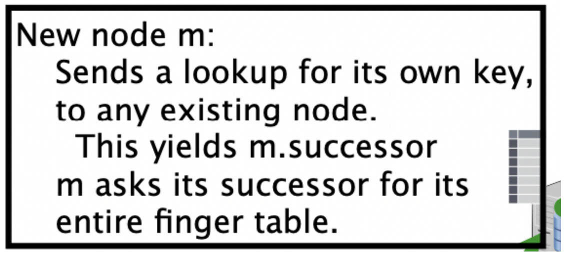
i	target	succ(target)
1	$n+2^0=n+1$	4
2	$n+2^1=n+2$	4
3	$n+2^2=n+4$	9
4	$n+2^3=n+8$	9
5	$n+2^4=n+16$	18

    ii. Scan the finger table
    iii. Send request to the successor node that is less than 10 (closest) → node 4 (Skip the node in between)
j. Why do lookups now take log(n) hops?
    i. One of the fingers must take you roughly half-way to your destination
k. Better to have multiple entry points
    i. No single failure, better load balancing, finger table takes more space (but size of the finger table is logarithmic to the range of the hash function)
l. There's binary lookup tree rooted at every node
m. Threaded through other nodes' finger tables
n. This is "better" than simply arranging the nodes in a single tree
    i. Every node acts as a root, so there's no root hotspot
    ii. But a lot more state in total

16. How to init FT of new node

a. 

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

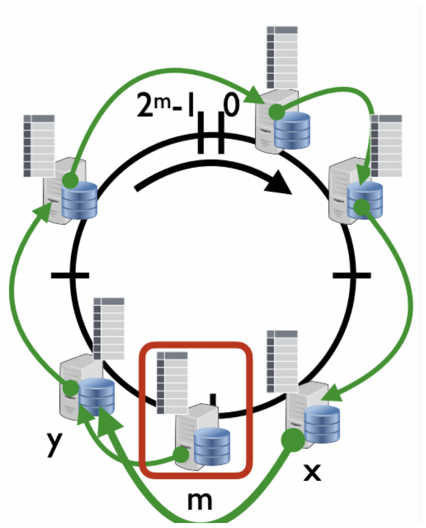
c. 

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.

d. At this point, the new node can forward queries correctly
e. Tweaks its own finger table in background by looking up each m + 2^(i-1)

17. Will Routing work after?

a. If m doesn't do anything, lookup will go to where it would have gone before m joined: to m's predecessor, which will return its n.successor –, which is not m

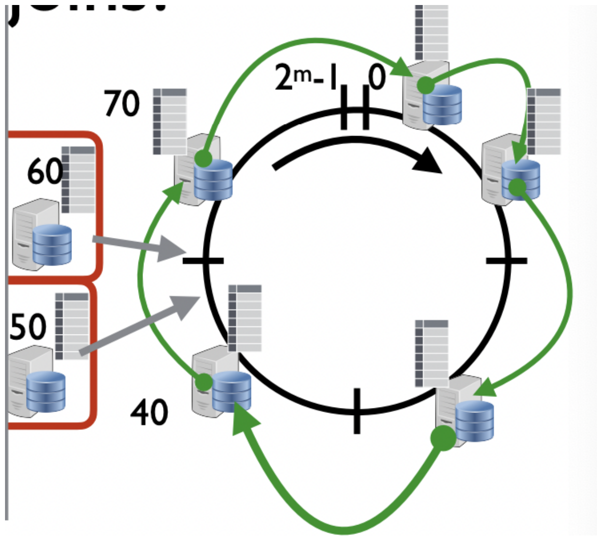
b. 

c. For correctness, m’s predecessor needs to se successor to m
    i. Each node keeps track of its current predecessor
    ii. When m joins, tells its successor that its predecessor has changed
    iii. 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

d. When a new node is added, the existing nodes will help the new node to construct finger table by exchanging messages

e. New node contacts any of the existing nodes and does lookup request as key of my ID
    i. Return a new id → this node should be the successor
    ii. Node m must tell its neighbors its existence and construct its finger table
    iii. M notifies its successor (node y) that its predecessor has changed from x to m (m and y know about each other), x still knows y as its predecessor
    iv. At some point, each node will ask its successor, who its predecessor is
        1. If the successor replies differently, it means that the ring is changed (update the information)
        2. Node m has to update its predecessor as x (done next time) → node x will contact node m and update begins (x, y, m know about each other)
        3. Finger tables are still outdated (since other nodes do not know existence of node m except y and x)
        4. Finger tables are updated asynchronously by each node (done periodically in the background by running protocol to update the protocol)
        5. Completely decentralized

f. Even though finger tables are not all updated, all nodes have correct successor/predecessor points
    i. Sufficient for correct lookups
        1. Whenever a node updates its successor pointer it also updates its 1st FT entry
        2. If a lookup is forwarded based on 1st entry, then no miss
        3. If another node forwards a lookup based on an outdated entry, let i-th:
            a. If there is a new node, let m, the request will be forwarded to a node that precedes m
            b. If a node is removed and the i-th entry points to the removed node, then the RPC will timeout and the node will use the previous entry i-1

18. What about concurrent joins?
    a. 

    b. At first 40, 50, and 60 think their successor is 70
    c. Which means lookups for e.g. 45 will yield 70, not 50
    d. However, after one stabilization, 40 and 50 will learn about 60
    e. Then 40 will learn about 50
    f. Step by step
19. Retrospective
    a. DHTS seem very promising for finding data in large p2p systems
    b. Decentralization seems good for load, fault tolerance
    c. But: the security problems are difficult
    d. But: churn is a serious problem, particularly if log(n) is big
    e. So DHTs have not had the impact that many hoped for