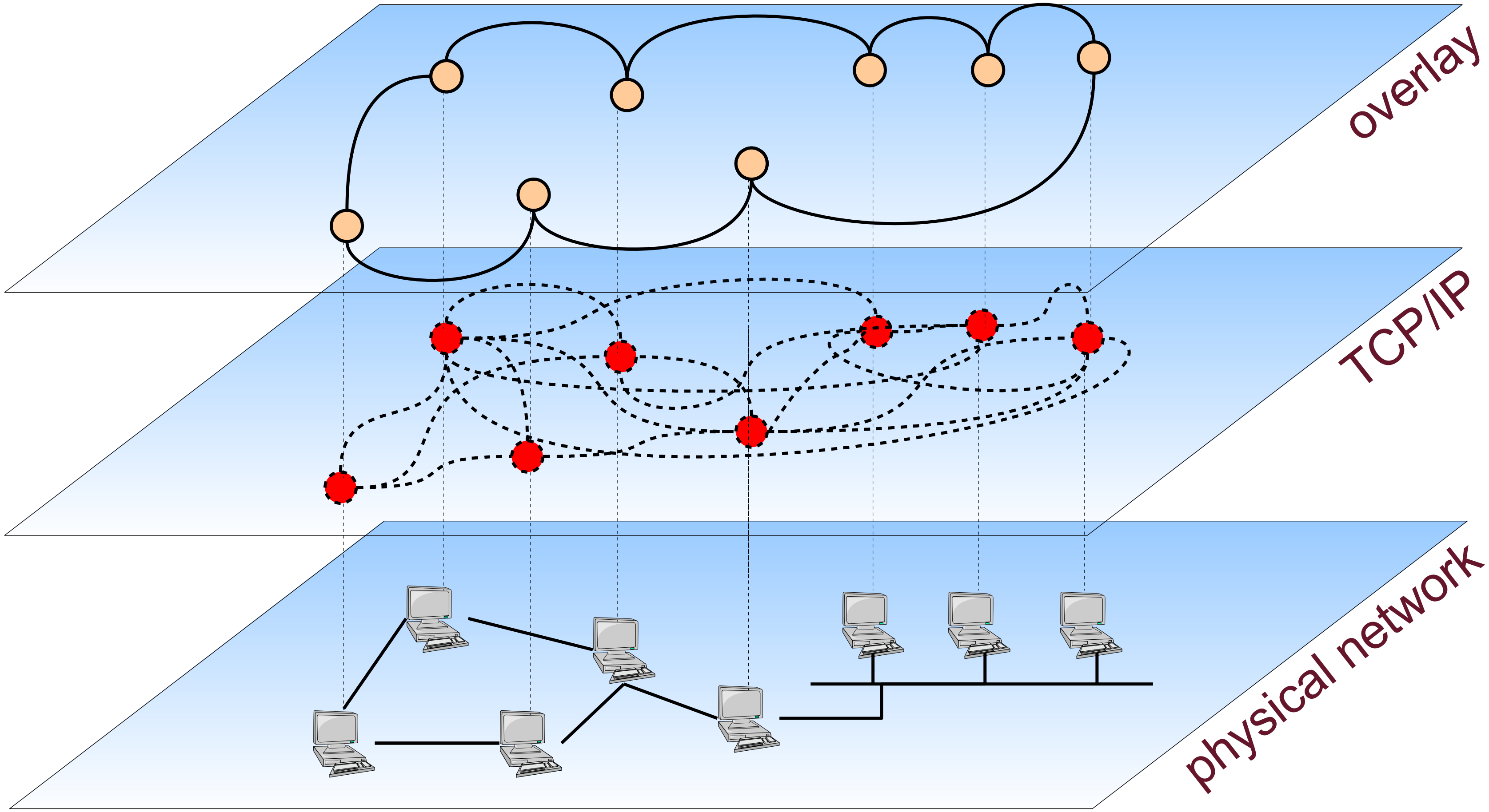
Distributed Systems: Distributed Hash Tables

Gian Pietro Picco
Dipartimento di Ingegneria e Scienza dell'Informazione
University of Trento, Italy
gianpietro.picco@unitn.it
http://disi.unitn.it/~picco

Distributed Hash Tables

- A hash table is a data structure that efficiently associates a ***key*** to an ***item***
  - `put(item, id)`
  - `item = get(id)`
  - A key is generated by applying a ***hash function*** to an item (e.g., SHA-1)
    - The mathematical properties of the function guarantee that it is very unlikely (although possible) that different items are hashed to the same key
- In a ***distributed hash table (DHT)*** the key and items are distributed across network hosts
  - "items" are the identifiers of resources
    - e.g., files, users
  - "keys" are associated to resources ***and*** to hosts
  - the name space is flat: only "opaque" identifiers, no structure in the names to be searched (unlike DNS)

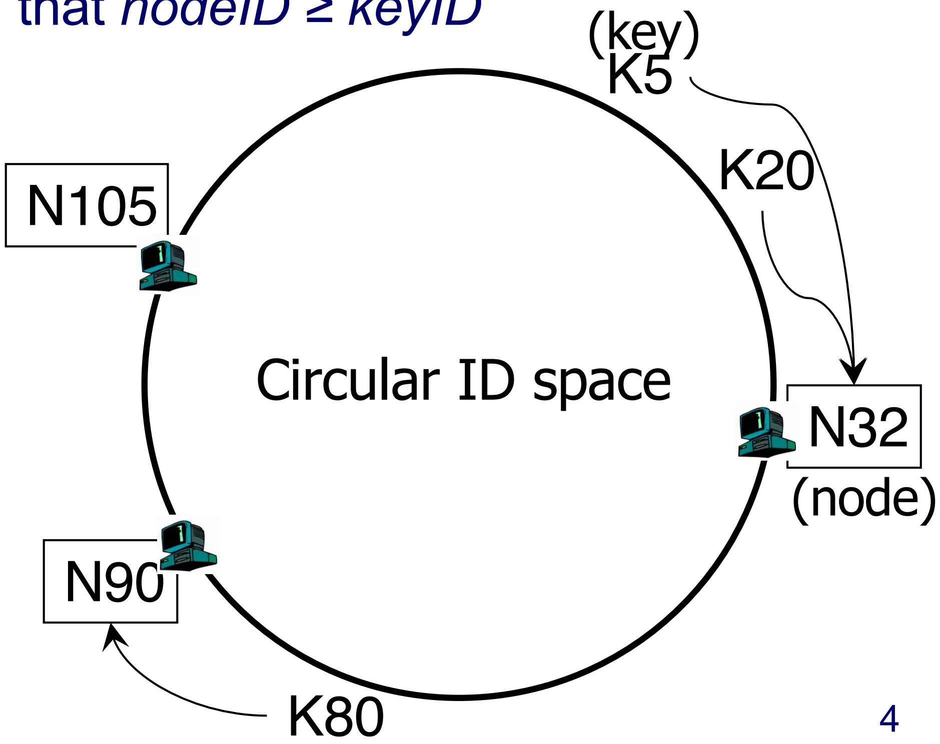
Overlay Networks



- Tremendous design flexibility in providing application-specific topologies and protocols

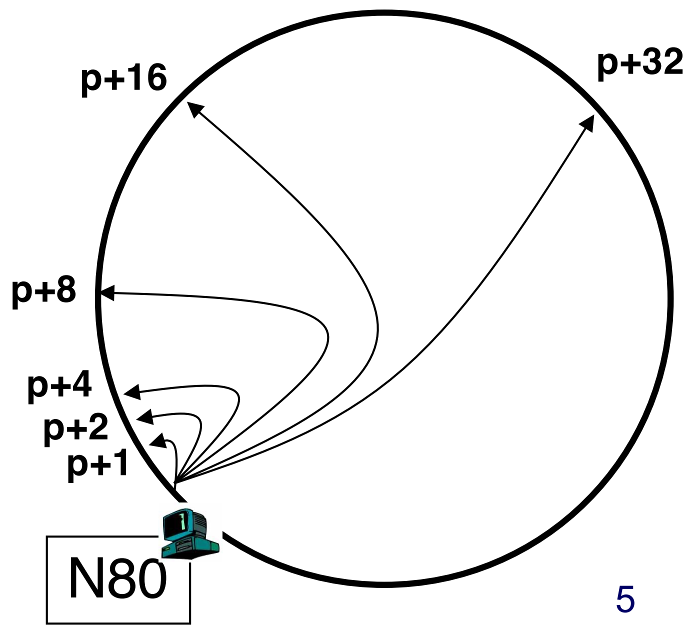
DHT Example: Chord

- Idea: associate to each node and item a unique ID in a uni-dimensional space (a ring)
  - e.g., in the range $[0 \ldots 2^m]$, *m* is typically 160 bits
  - the ID is the hash of
    - the content of the item (e.g., a file)
    - the IP address, in the case of a node
  - A key *k* is ***mapped*** to its ***successor*** *succ(k)*, the node with next higher ID
    - i.e. the smallest node such that *nodeID* ≥ *keyID*
- Properties:
  - routing table size is O(log N), where N is #nodes
  - a file is found in O(log N) hops

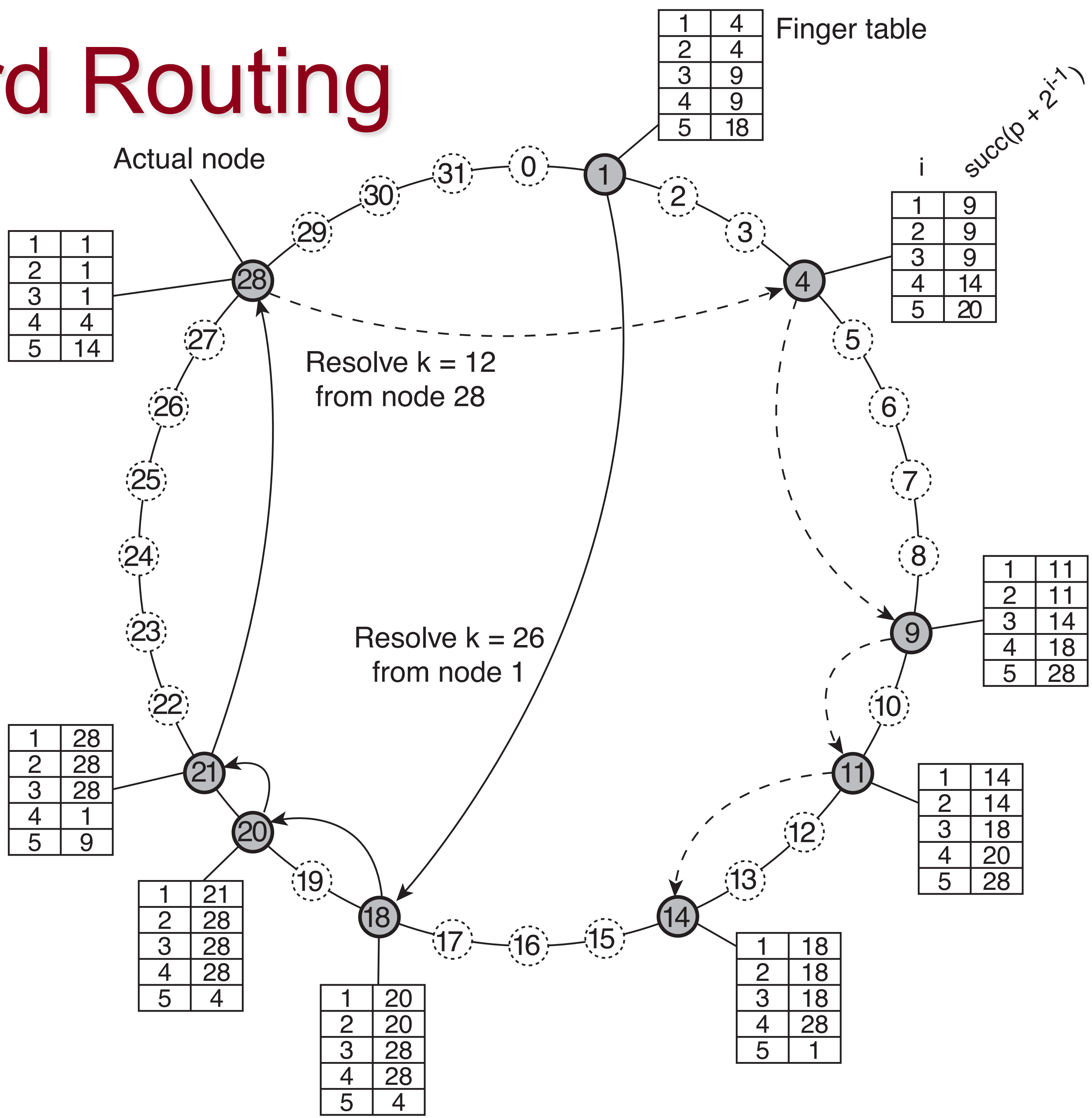


Finger tables

- The difficult problem in DHTs is to efficiently search for the node hosting an item, given the item's key
  - lookup(k) returns succ(k)
  - Trivial, inefficient solution: linear search
    - each node p keeps track of succ(p+1) and pred(p)
    - when p receives a lookup(k), it serves it if pred(p) < k ≤ p, otherwise it routes to succ(p+1) or pred(p)
- In Chord, the problem is solved by using routing tables ("finger tables") that contain "shortcuts" to exponentially bigger portions of the ID space
  - $FT_p(i) = succ(p+2^{i-1})$
    - Entry i in the finger table FT_p of node p is the first node that succeeds p by at least 2^{i-1}
    - Modulo m, restarts after 0
  - A lookup(k) is routed to the node q with index j in the FT such that $FT(j) \leq k < FT(j+1)$



Chord Routing



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

Dealing with Churn

- Churn is the rate at which nodes join and leave the system
  - This may be voluntary (e.g., user disconnects) or induced by (communication or host) failures
- If a node p wants to join, it requests a lookup for succ(p+1); then it contacts this node and its predecessor to insert itself in the ring
  - Similar for leaving
- Problem: keep the finger tables up to date
  - Each node runs a background procedure that contacts succ(q+1) and asks to return pred(succ(q+1))
    - $FT_q[1]$ must refer to succ(q+1)
    - If the latter returns q, everything is consistent
    - If the successor returns p such that $q < p \leq succ(q+1)$ a node p has joined; q updates its routing table to set $FT_q[1]=p$
  - Similar procedures are run in the background to verify the consistency of the other finger tables, as well as pred(q)