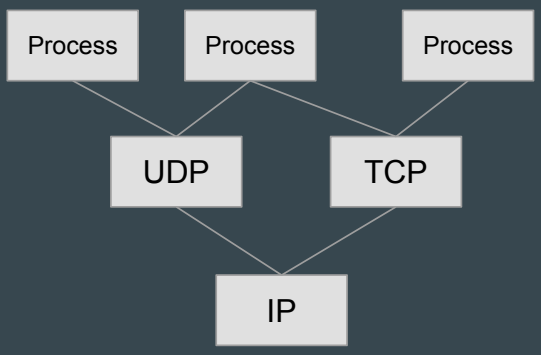
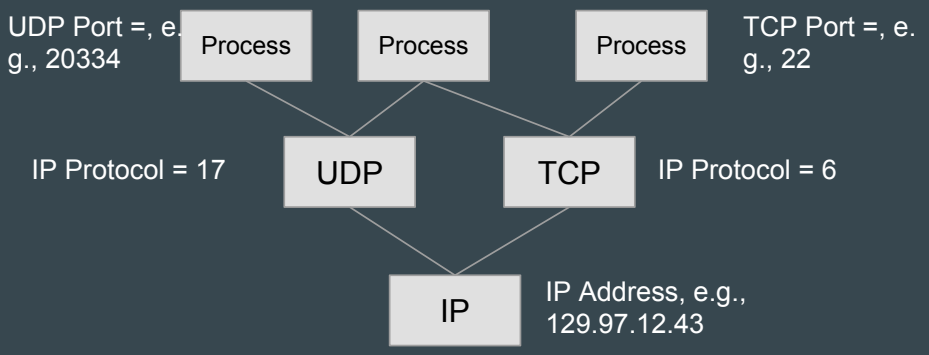
Focus on UDP & TCP over IP

- UDP & TCP are transport-layer protocols
- Over IP, which is a network-layer protocol



- UDP & TCP multiplexed over IP.
- Multiple processes multiplexed over each of UDP & TCP.

Multiplexing



The “5-tuple”

- On an (the) Internet, a connection or association is identified uniquely by the 5-tuple:
  - ⟨source-ip-address, source-port, destination-ip-address, destination-port, protocol⟩
  - E.g., ⟨129.197.2.13, 20334, 216.58.199.14, 80, 6⟩
  - E.g., ⟨129.197.2.13, 50000, 216.58.199.14, 80, 6⟩

1 5 Tuple

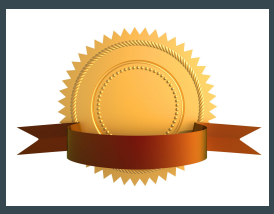
We can understand the flow of the packet through your system by looking at the 5-tuple.

- source ip address
- source port
- destination ip address
- destination port
- protocol

The software view

- OS/library for UDP/TCP
- Applications can be written on top of UDP/TCP
- Client (initiator) - server (responder) paradigm.

The socket API



- POSIX standard API for UDP/TCP applications
- (Does not necessarily mean that it's a great API.)
- Essential calls:

  - *socket()*
  - *listen()*
  - *accept()*
  - *send()*
  - *sendto()*
  - *bind()*
  - *connect()*
  - *receive()*
  - *sendto()*
  - *recvfrom()*

2 Socket API

Here a the 9 basic socket commands.

Here is a UDP example: You have multiple identities association with ip so you can just chose one or use all of them.

Checkout `man 7 ip` for more information.

```
struct in_addr // this is a struct for internet addresses, it is just a uint32

int sockfd = -1
if(sockfd == socket(AF_INET, SOCK_DGRAM, 0) < 0) { // see man socket to learn how this
    function works
   //shit done borked
   return -1
}

struct sockaddr_in server
server.sin_port = 0 // some ports have specific privileges and such, 0 allows the OS to
    pick the port
server.sin_port_family = AF_INET
if(bind(sockfd, (struct aockaddr_in *)&server, sizeof(struct socaddr_in)) < 0) { // see
    man bind
   //what did you do wrong
   return -1
}

if(getsockname(sockfd, (struct sockaddr) &server, sizeof(struct socaddr_in)) < 0){ //man
    getsockname to get the socket address
   //seriously how did you fuck this up
   return -1
}

print(inet_ntoa(server.sin_addr))
print(ntohs(server, sin_port))

//make sure you put that buffer length thing properly (see buffer overflow attacks, hee
    hee)
//this is blocking which is why you have to check the reclen, might actually need a loop
    here
if(reclen = recfrom(sockfd, buffer, bufferlength-1, 0, (struct socketaddr *) &client,
    sizeof(struct sockaddr_in)) < 0) { // man recvfrom
   //layyyyyme
   return -1;
}

print(inet_ntoa(client.sin_addr))
print(ntohs(client, sin_port))

//dont forget to close your shit
close(sockfd)
```

```
//To get the address you want, in a separte program
getifaddrs
//avoid memory leaks
freeifaddrs

struct ifaddrs *ifa //this is a linked list
if(getifaddrs(&ifa) < 0){
   return -1
}

for(struct ifaddrs *1 ifa; i != NULL; i = i->ifa_next) {
   if(i->ifa_addr == NULL) continue

   //add address to list of available addresses that the user can use
}
```

We need to watch for big/little endian cause OSs are separate from networks. We have a call in in_addr that will fix its shit for us.

CHECK YOUR RETURN VALUES FUCKER!!! This shit sucks to debug. LARA I'M LOOKING AT YOU, IF YOU FUCK THIS UP I WILL KILL YOU.

You might need to compile with a flag -D_BSD_SOURCE

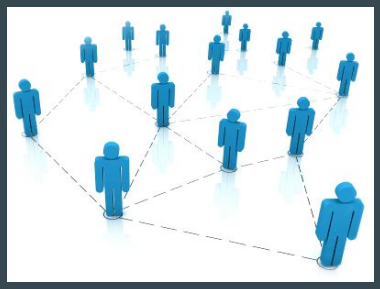
TCP works very similar to the above UDP example except that it is connection oriented. So instead of just checking how much data was received you must listen, accept, and receive. After you accept it locks the server from accepting more connections. All the same checks and such must happen.

Helpful commands:

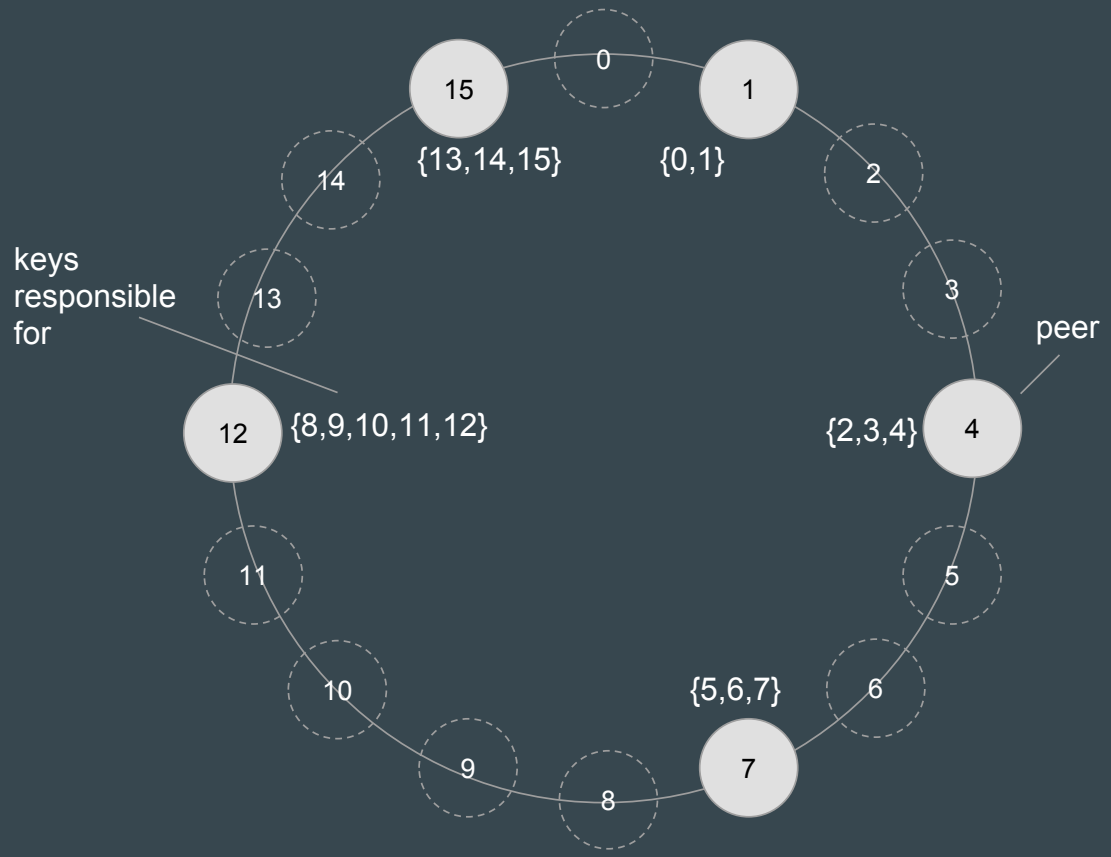
- man all the pages
- netstat

The Chord DHT

- *m*-bit *key* unique for every peer and piece of content.
- Define: *succ(k)* for any key *k* is peer that exists with smallest *id* ≥ *k*.
- Content with key *k* hosted by *succ(k)*.



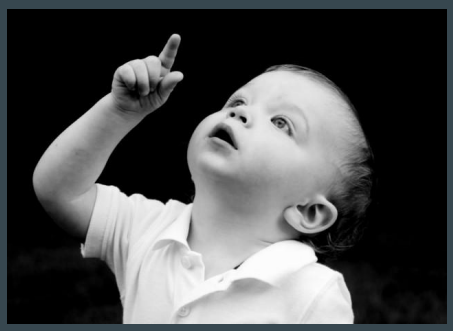
- *m = 4*


0
15
1
{13,14,15}
{0,1}
14
2
keys
responsible
for
13
3
peer
12
{8,9,10,11,12}
{2,3,4}
4
11
5
10
{5,6,7}
6
9
7
8

We can look at this as a circle. The number of entites that this id space supports is 2^M. This represents a peer to peer network, each node that we see is a peer on the network. We try to use cord to let people look up peers when they only know the id of one in the network but not the one they are looking for. We care about minimizing the number of hops between peers to get to our requested one. Everytime you hop we decrease a packet's time to live value by one. If it drops to 0 the packet is dropped. We do this so that a bug doesn't lead to packets living forever. The diameter of this graph is the longest distance of the shortest path. We assume that a packet will follow the route that is most optimal (routing protocols try to make the routing tables use only these shortest paths). We look at the internet's diameter as described above, this value must fit within the time to live value of a packet (which is confined to 8 bits).

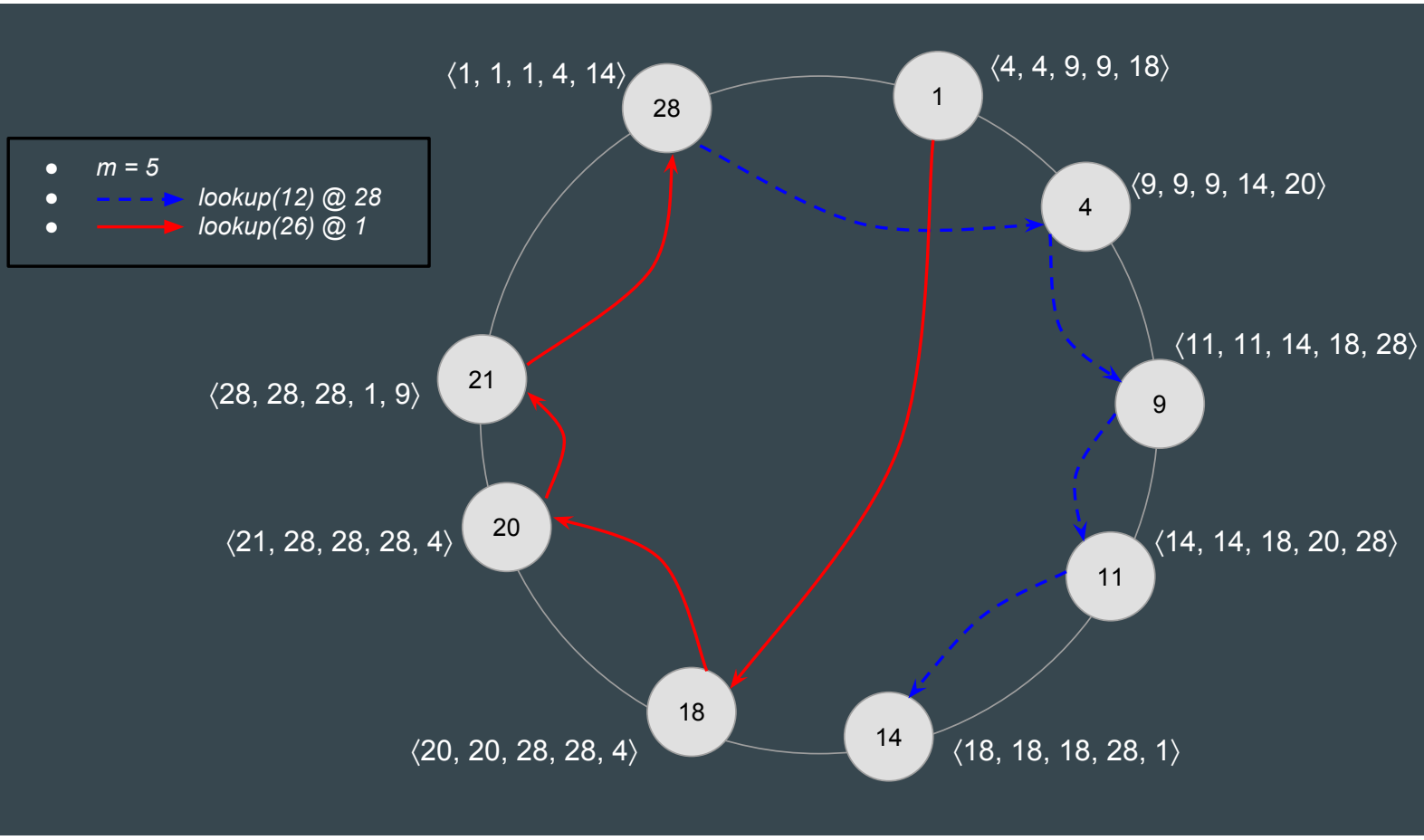
lookup(k)

- We may want to *lookup(k)* at any peer.
- Query is routed to *succ(k)*. How?



The Chord Approach

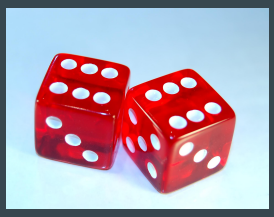
- Maintain a “finger table” (routing table) at Peer *p*, $FT_p[]$.
- *m* entries
- $FT_p[i] = succ(p + 2^{i-1})$, for all $i \in [1,m]$


m = 5
lookup(12) @ 28
lookup(26) @ 1
28
⟨1, 1, 1, 4, 14⟩
1
⟨4, 4, 9, 9, 18⟩
4
⟨9, 9, 9, 14, 20⟩
9
⟨11, 11, 14, 18, 28⟩
11
⟨14, 14, 18, 20, 28⟩
14
⟨18, 18, 18, 28, 1⟩
18
⟨20, 20, 28, 28, 4⟩
20
⟨21, 28, 28, 28, 4⟩
21
⟨28, 28, 28, 1, 9⟩

We want to use this chord diagram to get the fastest and most space efficient algorithm. We apply that formula to a couple of values (up to m) counting up and figuring out the successor if we were too look for that at each. The worst case space space complexity of this is m^2 (so is the average space). If we have an outside client look up peer 12 at peer 28 (ie they only had the address of peer 28 but wanted peer 12). We look at the entries to the table with the greatest number less than 12, at 28 this is 4 so we go to 4. At 4 we do the same thing and go to so on until we get to peer 14 where there is no value less than 12 so this is the correct value. We see in the other example that the last node is still less than the value we are looking for so we hop to the first value in the table of 28. Only a peer of value greater than 12 can host 12 because of how we defined successors (they must have id greater). The successor rule can be broken if there is no other key that works, else it can loop around.

Claims

- We go around the circle at most once
  - Termination guaranteed
- Expected number of "hops" is $O(\log n)$, where n = # peers.
  - Each hop covers half the arc-length.
  - Hand-wave: peers are equidistant under randomness assumption.

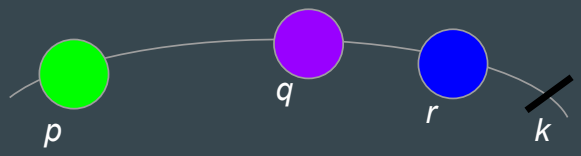


n is the number of peers. This means that the expected number of hops is $O(\log n$. We know that each hop covers half the arc-length (we can see that in a finger table shits grows exponentially making it cover more). When we add a peer we just chose a random number (doing it over again if a collision occurs).

Each hop covers > half the arc-length…

Claim:

Suppose we do *lookup*(k) @ peer p. Assume that r is the peer immediately before *succ* (k). Suppose that the next hop at p is q. Then: q - p > (r - p)/2.



Proof

Suppose $q = FT_p[j]$.

Then, $q \geq p + 2^{j-1}$. And, $r < p + 2^j$.

when we do a loop up at q we get that its at the jth entry in the table. So then $q \geq p + 2^{j-1}$ by using our formula for building the table and the definiton of successor. If we say r is the penultimate hop so we know that $r < p + 2^j$ if r was greater than that we would have jumped farther.