Network communication

David Hovemeyer

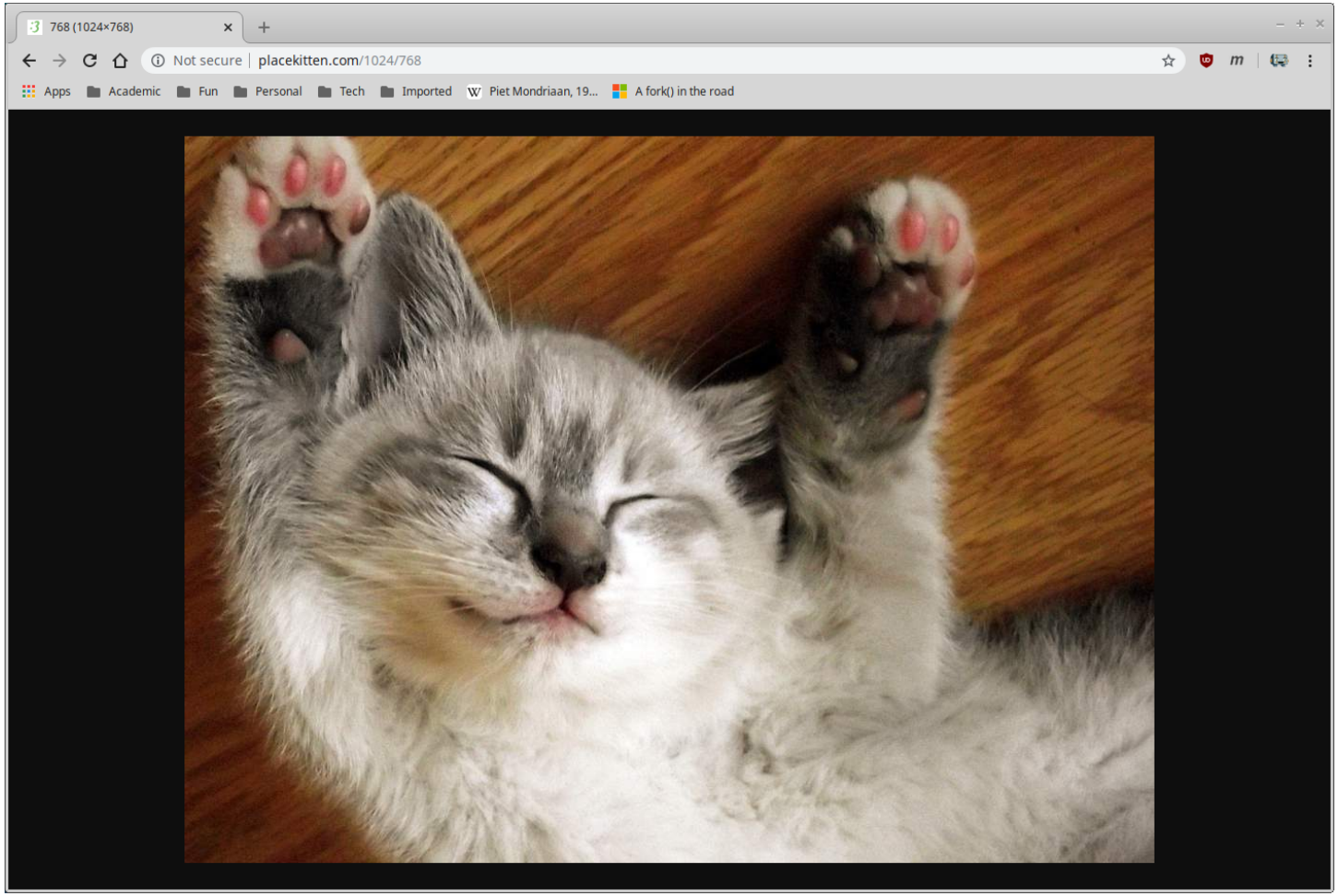
15 November 2019

Using a web browser

Type a URL into a web browser:

http://placekitten.com/1024/768

The internet of cats



Nice! (But how did that actually work?)

Networks

Networks

Network: allow communication between computers

Access remote data

Share information

Hard to overstate importance of networking: *everything* can communicate over the Internet now (laptops, phones, cars, refrigerators, ...)

Network interface

To connect to a network, a computing device needs a *network interface*

- Wired: ethernet, Infiniband (for high-performance applications)
- Wireless: 802.11 (wifi), cellular modem

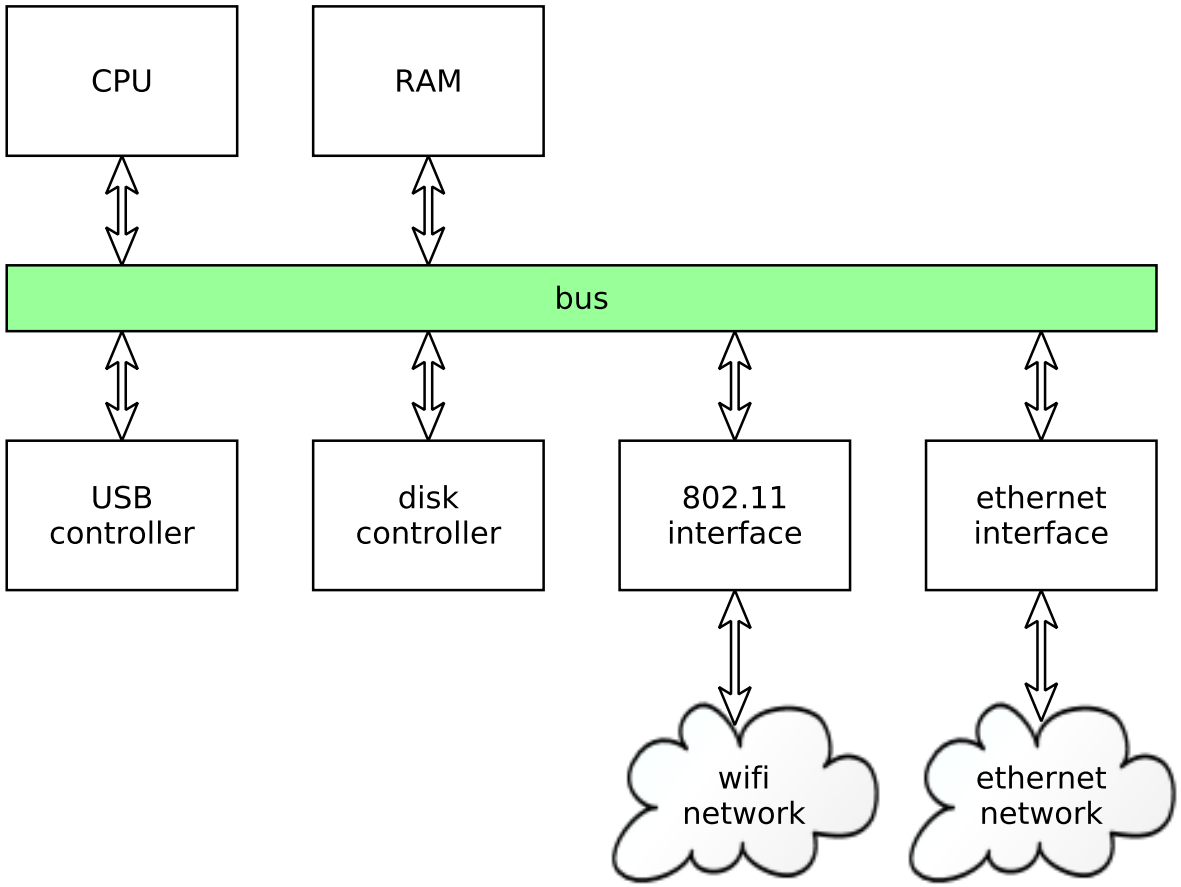
To the computing device (the ‘‘host’’), the network interface is just a peripheral device

- Much like a disk controller, USB controller, etc.

OS can request to send data out to the network

Network interface device notifies host CPU when data arrives from the network (possibly by raising a hardware interrupt)

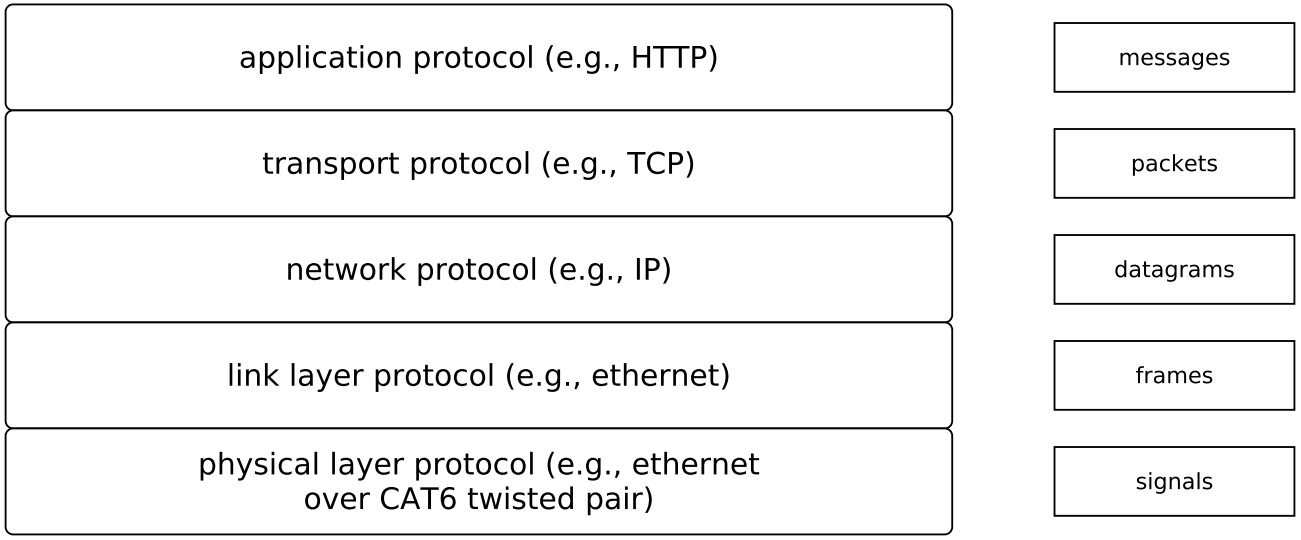
Network interface example



Protocol stack

In addition to network interface hardware, a *protocol stack* is needed to allow network applications to communicate over the attached network interface(s)

‘‘Protocol stack’’: so called because network protocols are layered



Issues

Some important issues to consider:

- How are differing network technologies interfaced to each other?
- How are devices and systems identified on the network?
- How is data routed to the correct destination?
- What APIs do network applications use to communicate?

We'll cover all of these (at least briefly)

Network security

Ideal of networking is to provide access to information and computing resources from anywhere

But...connecting a computing device to the network potentially exposes it to malicious actors

Issue: controlling access

- Permit only authorized agents access to data and services

When implementing and deploying networked systems and applications, we must think *very* carefully about

- what the security requirements are, and
- whether the system meets them

TCP/IP

TCP/IP

TCP/IP: a suite of *internetworking* protocols

- ‘‘internetworking’’ = connecting lots of physical networks together, including when they use different technologies or protocols

Two versions: IPv4 and IPv6

- IPv4: 32 bit addresses (not enough of these!), widely deployed
- IPv6: 128 bit addresses, not as widely deployed (but significant adoption in mobile networks)

Ubiquitous: if you’re using a network, you’re using TCP/IP

Scale of global TCP/IP internet is immense (*billions* of communicating devices)

IP

IP = Internet Protocol

This is the underlying *network protocol* in the TCP/IP protocol suite

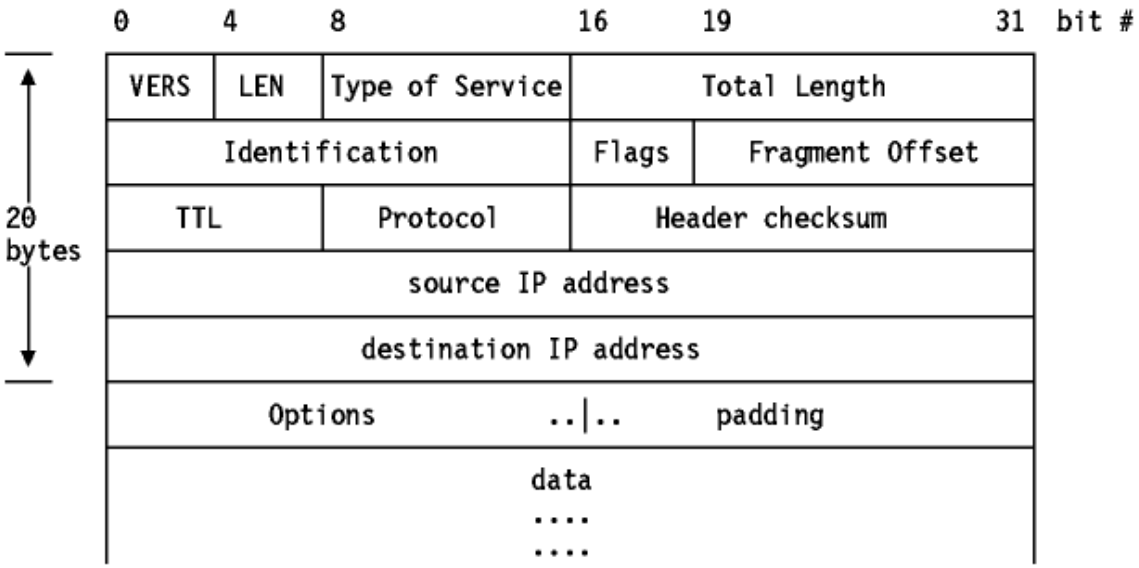
Ultimately, all data is sent and received using *IP datagrams*:
fixed-size packets of data sent and received using IP addresses
to indicate the source and destination

Transport protocols (such as TCP and UDP) are layered on top of IP

- E.g., a TCP connection consists of IP datagrams containing TCP data

IP is an *unreliable* protocol: when a datagram is sent, it might not reach the recipient (we'll see why in a bit)

An IP datagram



[Image source: http://www.danzig.us/tcp-ip-lab/ibm-tutorial/3376c23.html]

Details:

- Consists of *header* followed by *data*
- May be fragmented and reassembled
- Protocol field indicates which transport protocol is being used

TCP

TCP: Transmission Control Protocol

A *connection* protocol layed on IP (value in Protocol field is 6)

TCP allows the creation of virtual connections between peer systems on network

A connection is a bidirectional data stream (each peer can send data to the other)

Data is guaranteed to be delivered in the order sent

Connection can be closed (analogy: hanging up when phone call ends)

TCP is a *reliable* protocol: if any data is lost en route, it is automatically resent

- Much cleverness is required to make this work!

UDP

UDP: User Datagram Protocol

A *datagram* protocol layed on IP (value in Protocol fields is 17)

Not connection-oriented: data could be received in any order, no fixed duration of conversation (more analogous to sending a letter than talking on the phone)

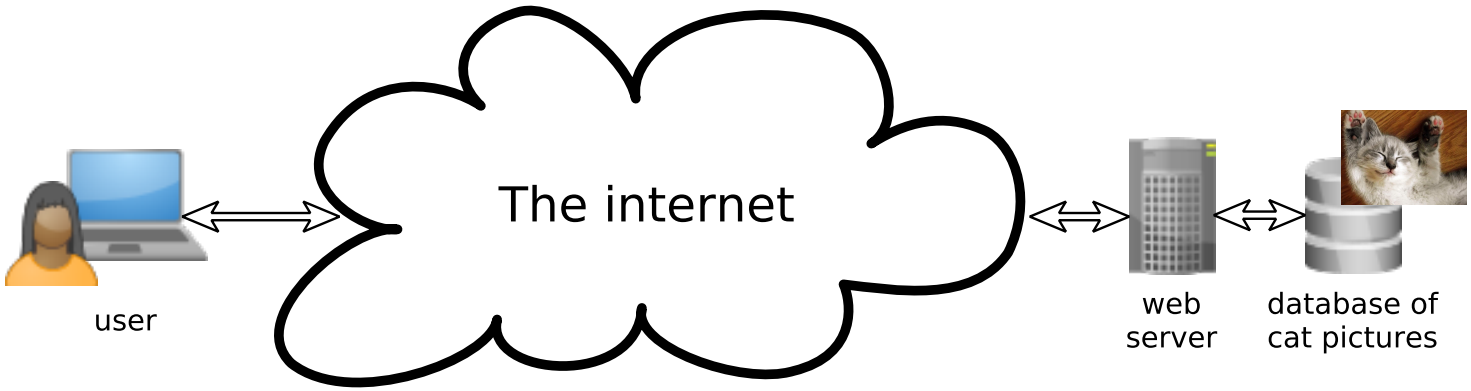
Unreliable: data sent might not be received

Used in applications where minimizing latency is important and data loss can be tolerated

Routing: idealized

Routing: How does data get to its destination?

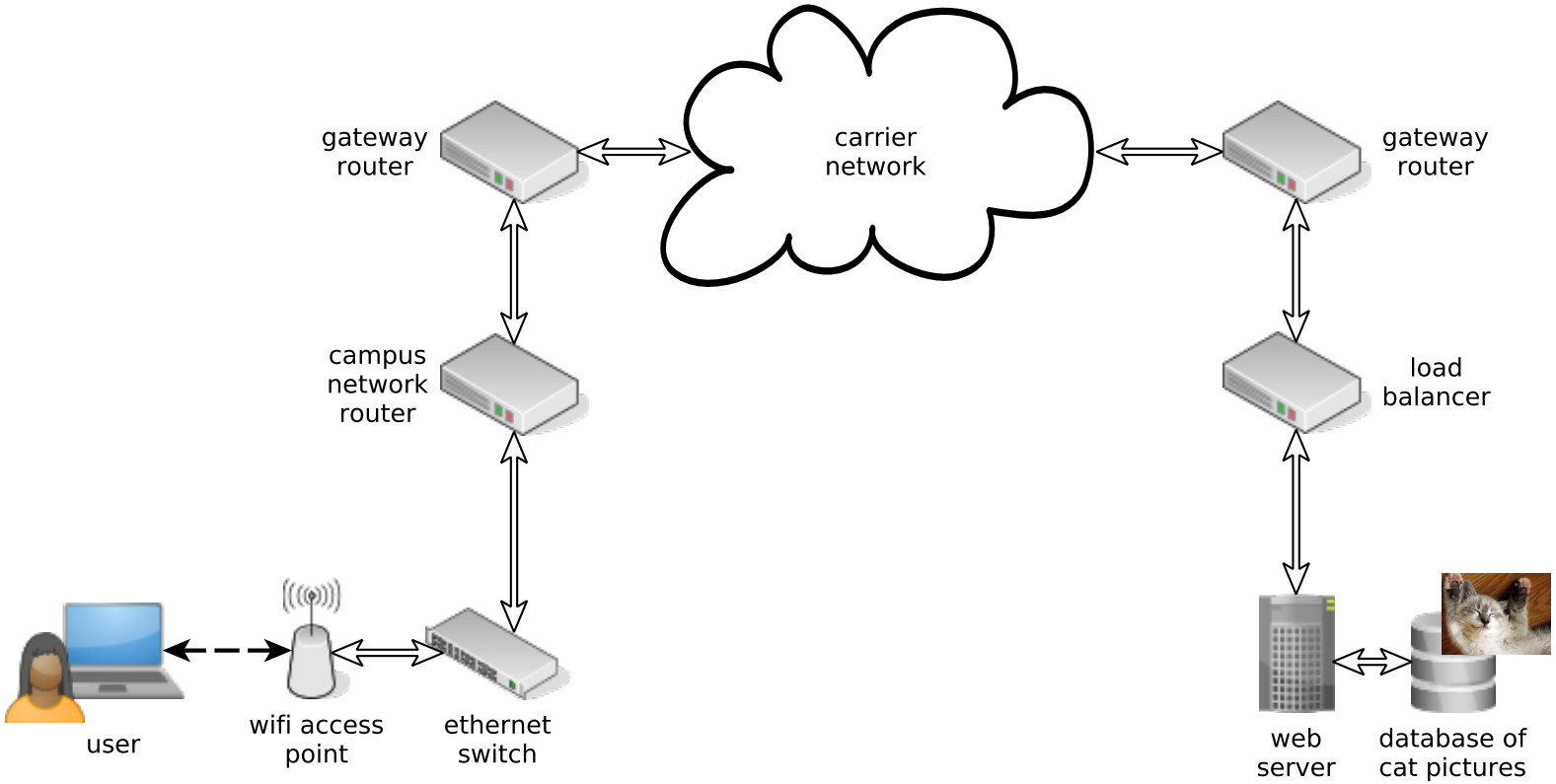
Idealized view:



Routing: the reality

Routing: How does data get to its destination?

Slightly more realistic view:



Addressing

Two kinds of address:

- Network address: address of a network interface within the overall internet (e.g.: IPv4 address)
- Hardware address: a hardware-level address of a network interface (e.g.: ethernet MAC address)

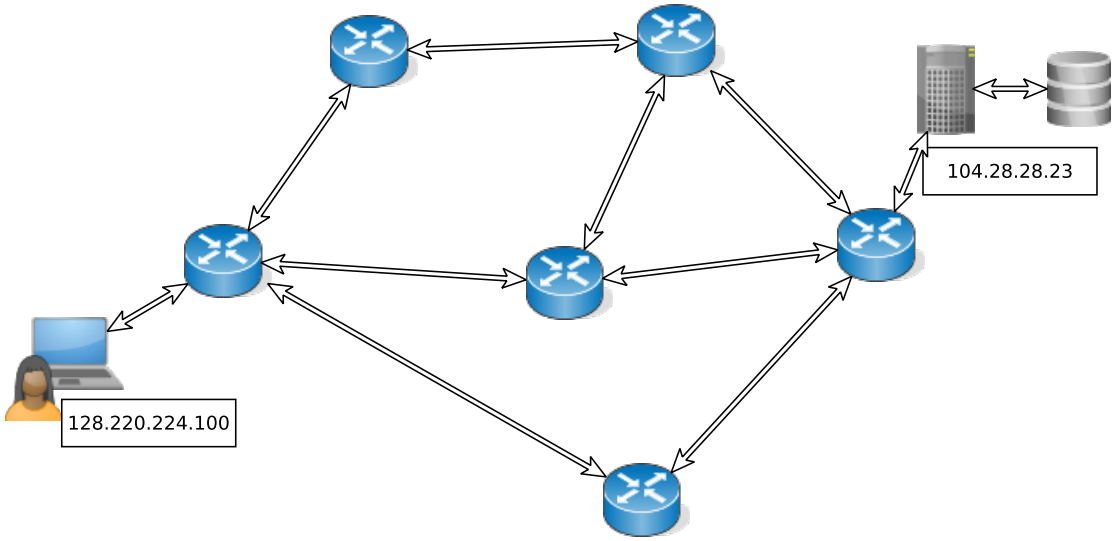
Network address is used to make routing decisions at the scale of the overall internet

- Network address conveys information about the network on which the interface can be found
- A *router* makes routing decisions based on a network address

Hardware address is used to deliver a data packet to a destination within the local network

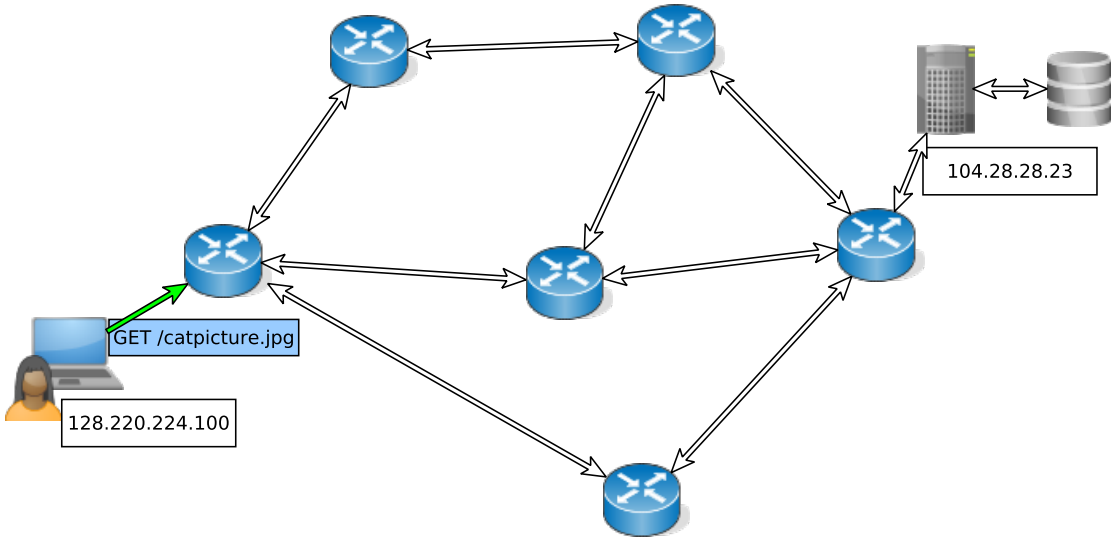
- A *switch* makes routing decisions based on a hardware address

Routing



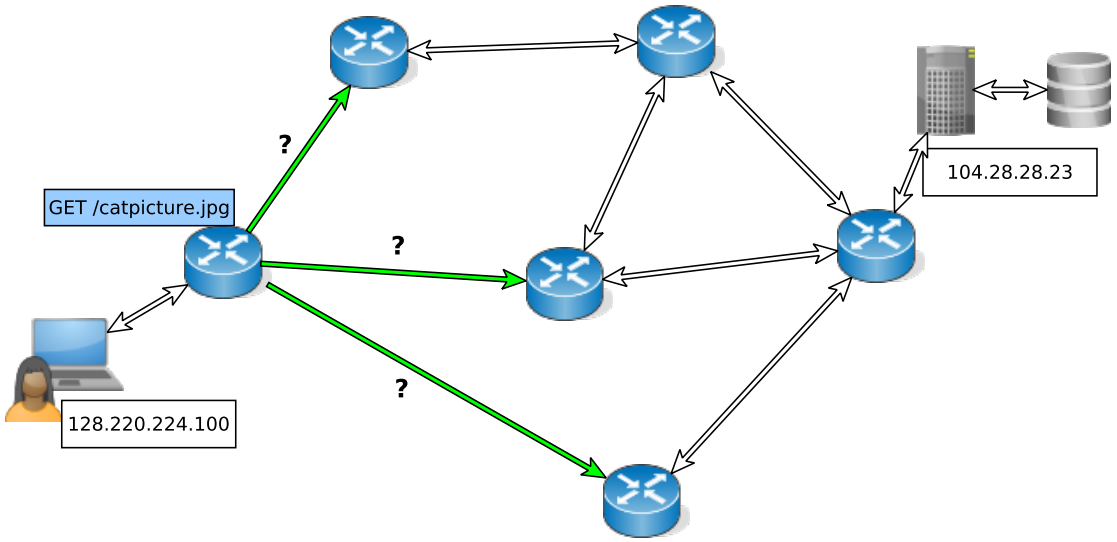
Network with client, server, and intermediate routers

Routing



Client sends request to server: packet sent on default route (user's computer has only one network interface)

Routing

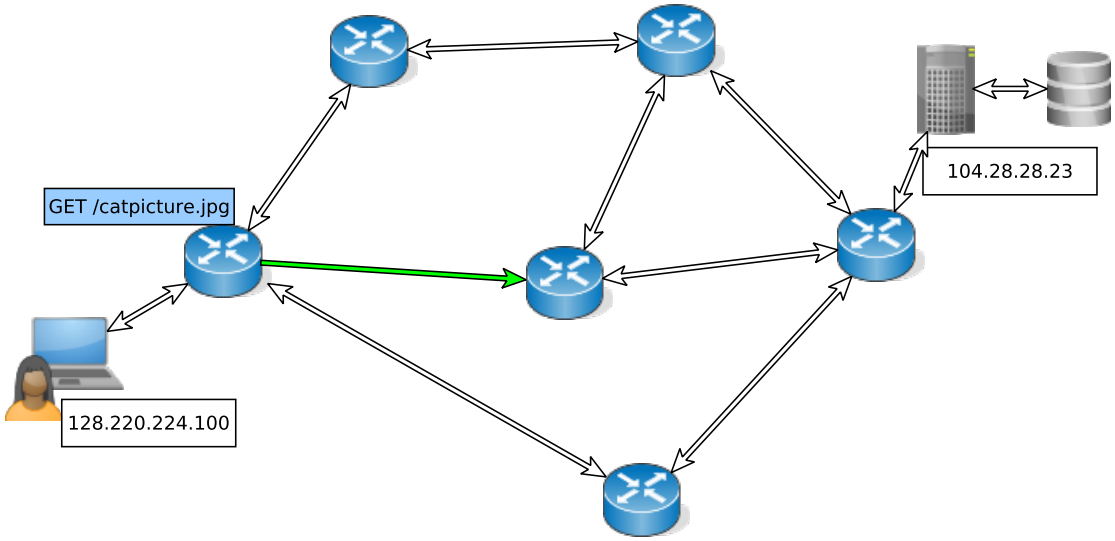


Router has a choice of outgoing links on which to send the packet

Each router has a *routing table* specifying which link to use based on matching the network part of the destination address

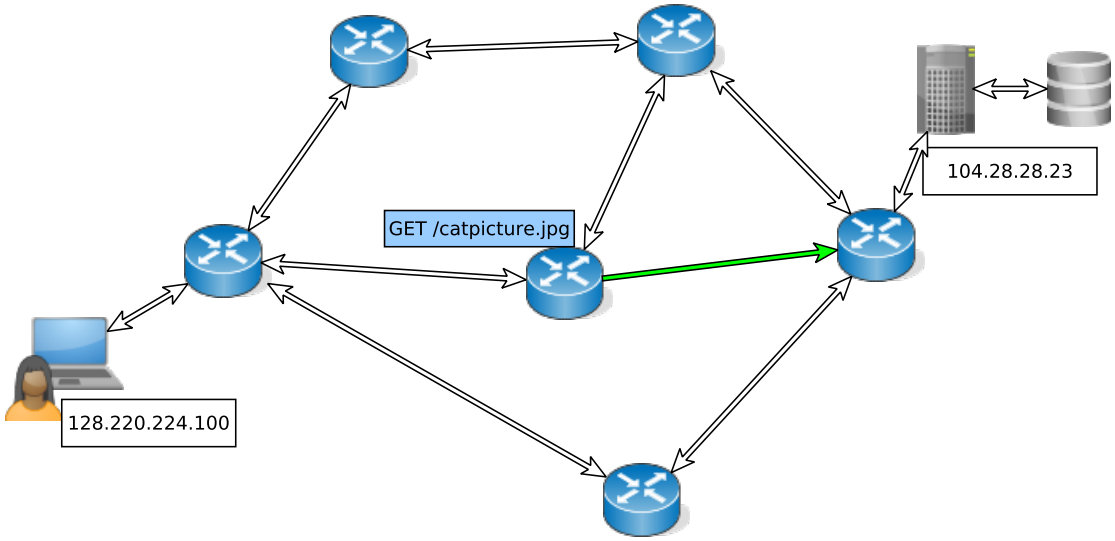
Routing algorithms: try to deliver packets efficiently, and avoid routing loops

Routing



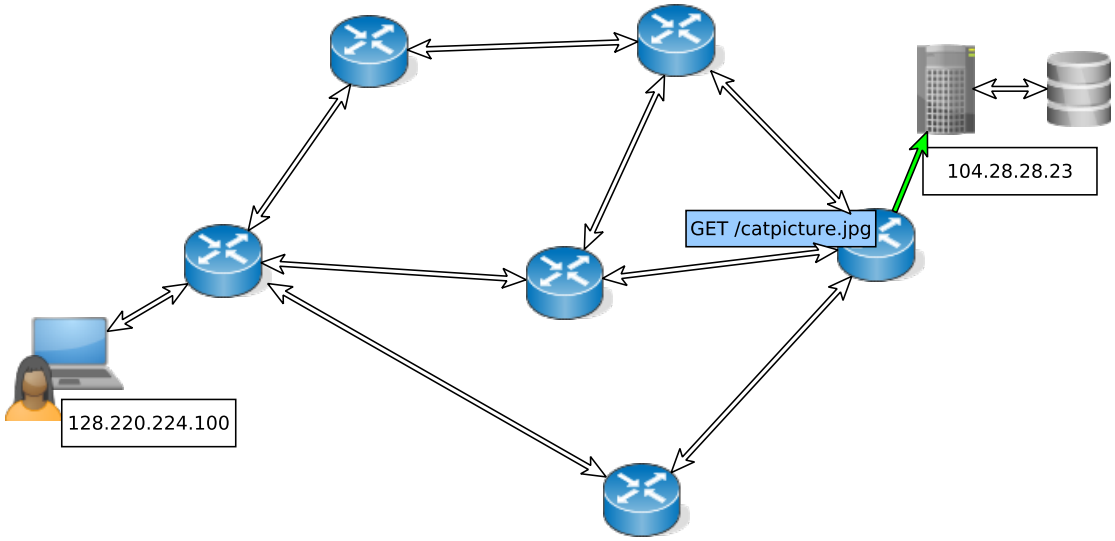
Choose outgoing link based on routing table

Routing



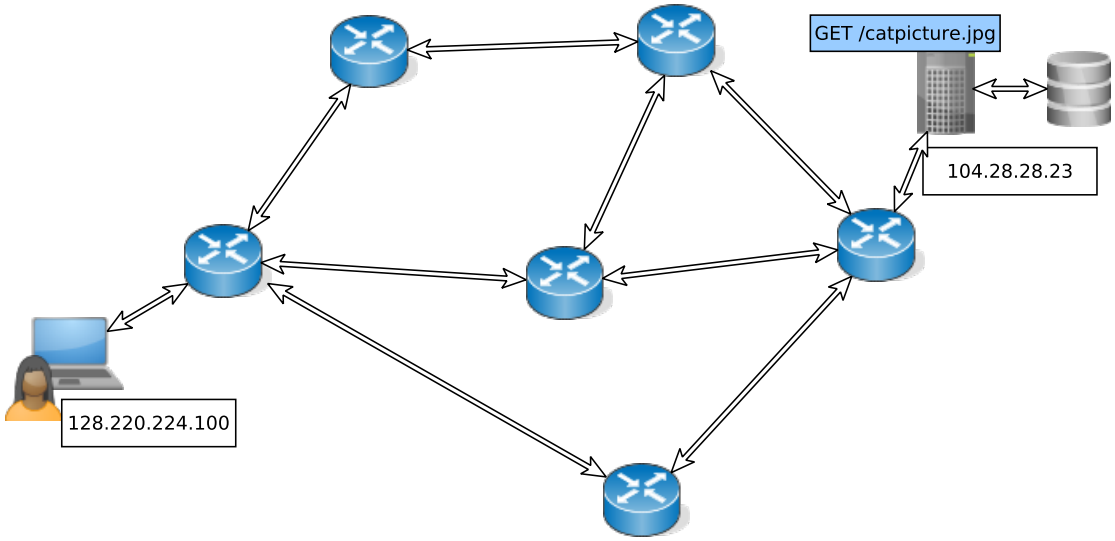
Next hop

Routing



Final hop

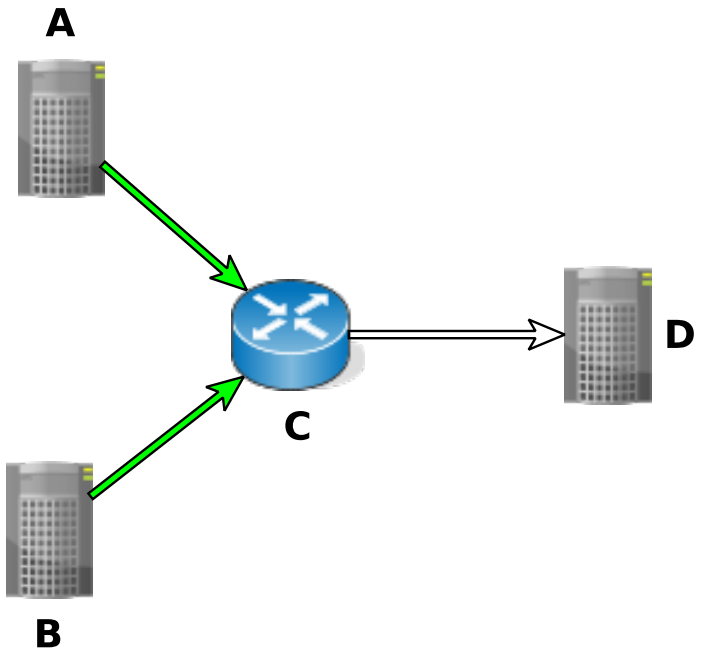
Routing



Packet delivered to server

Server's response will be delivered back to client in a similar manner

Why IP is unreliable

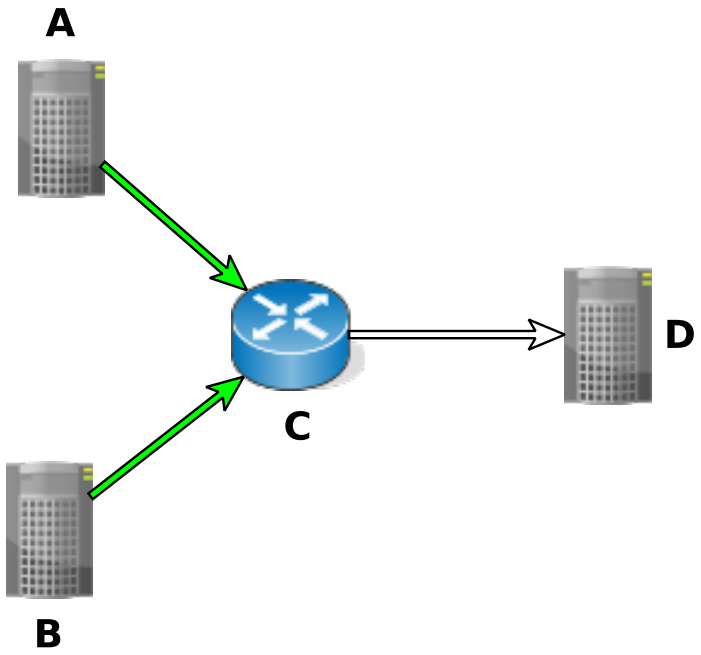


Scenario: A and B both try to send a packet to D at the same time

Outgoing link C→D can only carry one of the two packets

What to do?

Why IP is unreliable

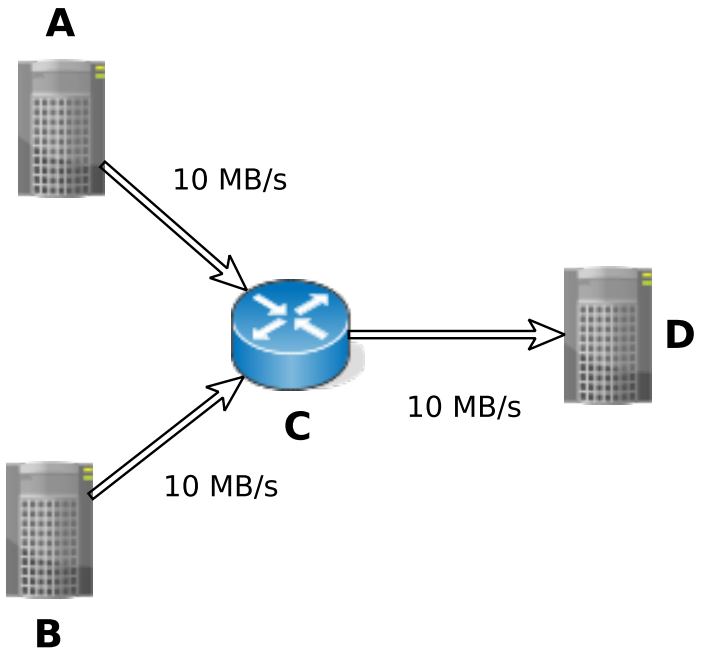


Solution: *queuing*

Router C has a *queue* of unsent packets to be forwarded to D

Either A's packet or B's packet will need to wait in the queue

Why IP is unreliable



Problem: outgoing link C→D cannot handle aggregate data rate of incoming data from A→C and B→C

But, C's queue of packets waiting to be sent to D is finite! (An unbounded queue would imply unbounded delay, not good)

Solution: C discards packets to D when its queue is full

Dropped packets

Dropped packets are a necessary consequence of finite capacity links and finite queues

Reliable protocols such as TCP require acknowledgment of data sent

No acknowledgment $\rightarrow$ assume packet dropped, retransmit

Unix sockets

Unix sockets

Unix sockets: API to allow programs to communicate over networks

Designed to work with many underlying protocols

Socket = ‘‘communications endpoint’’, appears to process as a file descriptor

Several important kinds of sockets:

- *Server socket*: used by server to accept connections from clients (not used for actual exchange of data)

- *Client socket*: used to exchange data between client and server systems

Socket system calls

Important socket system calls:

socket: create an unconnected socket

bind: associate a socket with a network interface identified by a network address

listen: make a socket a server socket (to allow incoming connections)

accept: wait for an incoming connection

Socket addresses

Socket API designed to work with many underlying network technologies

struct sockaddr: ‘‘supertype’’ for all network addresses

- A ‘‘type’’ field is at beginning of struct to distinguish variants
- E.g. if type field contains AF_INET, it’s an IP address

struct sockaddr_in: ‘‘subtype’’ for IP addresses

Create server socket

```
int create_server_socket(int port) {
  struct sockaddr_in serveraddr = {0};
  int ssock_fd = socket(AF_INET, SOCK_STREAM, 0);
  if (ssock_fd < 0)
    fatal("socket failed");

  serveraddr.sin_family = AF_INET;
  serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
  serveraddr.sin_port = htons((unsigned short)port);
  if (bind(ssock_fd, (struct sockaddr *) &serveraddr,
           sizeof(serveraddr)) < 0)
    fatal("bind failed");

  if (listen(ssock_fd, 5) < 0) fatal("listen failed");

  return ssock_fd;
}
```

Wait for incoming connection

```
int accept_connection(int ssock_fd, struct sockaddr_in clientaddr) {
  unsigned clientlen = sizeof(clientaddr);
  int childfd = accept(ssock_fd,
                       (struct sockaddr *) &clientaddr,
                       &clientlen);
  if (childfd < 0)
    fatal("accept failed");
  return childfd;
}
```

Server loop

```
int main(int argc, char **argv) {
  char buf[256];
  int port = atoi(argv[1]);
  int ssock_fd = create_server_socket(port);

  while (1) {
    struct sockaddr_in clientaddr;
    int clientfd = accept_connection(ssock_fd, &clientaddr);
    ssize_t rc = read(clientfd, buf, sizeof(buf));
    if (rc > 0) {
      write(clientfd, buf, rc);
    }
    close(clientfd);
  }
}
```

Testing the server

Run the server:

```
$ gcc -Wall -o server server.c
$ ./server 30000
```

Test using telnet program:

```
$ telnet localhost 30000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
hey there!
hey there!
Connection closed by foreign host.
```

Implementation issues

- Reading from socket can return fewer bytes than requested (generally need to call read in a loop)

- Network connections can be broken (need to check result of read and write, error often indicates that the connection no longer exists)

Hostnames

DNS: Domain Name Service

Assign meaningful names (such as ugradx.cs.jhu.edu) to network addresses (such as 128.220.224.100)

getaddrinfo: look up network address for hostname

csapp.h/csapp.c

The textbook *Computer Systems: A Programmer's Perspective* includes a library of convenient functions for writing network applications

Open_listenfd: open a server socket given port name as string

open_clientfd: simplified interface for connecting to a server by specifying host name (or address) and port

rio_ functions: Robust I/O routines, handle looping for short reads/writes and interruptions from signals automatically

- Example: rio_readn: read n bytes from a file descriptor

Using these routines can significantly reduce the complexity of implementing network applications in C and C++

Application protocols

Application protocols

Application protocol: determines how data is exchanged by instances of an application program

- Usually: a server and a client
- Another possibility: peer to peer (P2P) applications

Example: HTTP, HyperText Transport Protocol

- Used by web browsers and web servers

Application protocols in 1 minute

Synchronous: The connected peers take turns talking

- Asynchronous protocols: possible, but significantly more complicated to implement

Client/server protocol: client sends request, server sends response

- Repeat as necessary

Message format: both peers must be able to determine where each message starts and ends

- Also, each peer must be able to determine the meaning of each received message

Text-based protocols are common because they are easy to debug and reason about

HTTP

A synchronous client/server protocol used by web browsers, web servers, web clients, and web services

- HTTP 1.1: https://tools.ietf.org/html/rfc2616

Client sends request to server, server sends back a response

- Each client request specifies a *verb* (GET, POST, PUT, etc.) and the name of a *resource*

Requests and responses may have a *body* containing data

- The body's *content type* specifies what kind of data the body contains

HTTP request example

Command: curl -v http://placekitten.com/1024/768 -o kitten.jpg

Request sent by curl program:

```
GET /1024/768 HTTP/1.1
Host: placekitten.com
User-Agent: curl/7.58.0
Accept: */*
```

Request is sent via a TCP connection to port 80

HTTP response example

Response sent by placekitten.com:

```
HTTP/1.1 200 OK
Date: Wed, 13 Nov 2019 12:33:20 GMT
Content-Type: image/jpeg
Transfer-Encoding: chunked
Connection: keep-alive
Set-Cookie: __cfduid=de2a22cdd3ed939398e0a56f41ce0e4a31573648400; expi
Access-Control-Allow-Origin: *
Cache-Control: public, max-age=86400
Expires: Thu, 31 Dec 2020 20:00:00 GMT
CF-Cache-Status: HIT
Age: 51062
Server: cloudflare
CF-RAY: 5350c608682a957e-IAD
```

Headers were followed by a body containing 40,473 bytes of binary data

Kitten