CSCI-1680
Layering and Encapsulation

Rodrigo Fonseca

Based partly on lecture notes by David Mazières, Phil Levis, John Jannotti

Today

- **Review**
  - Switching, Multiplexing

- **Layering and Encapsulation**
- **Intro to IP, TCP, UDP**

- **Extra material: sockets primer**

Circuit Switching

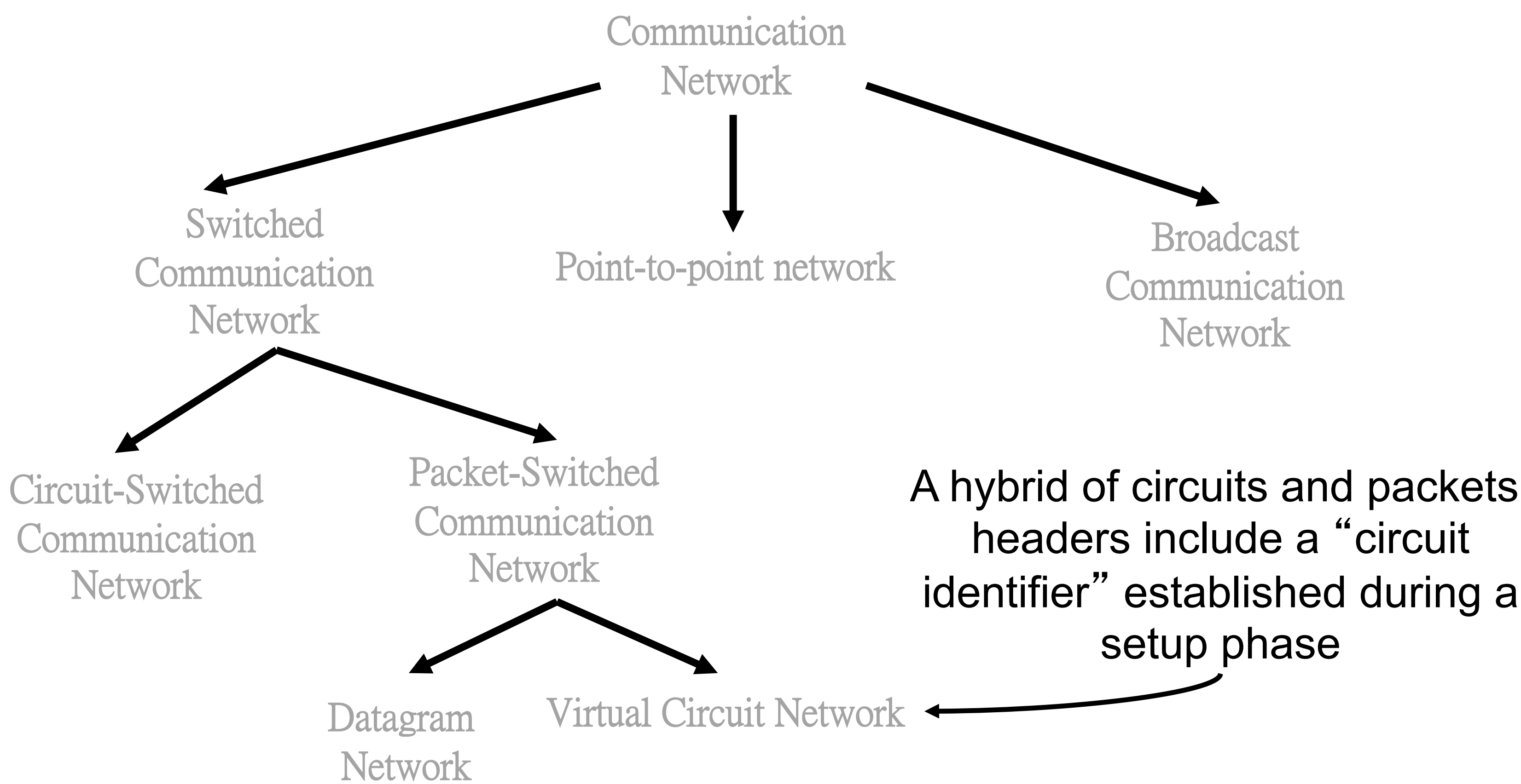
- **Guaranteed allocation**
  - Time division / Frequency division multiplexing
- **Low space overhead**
- **Easy to reason about**

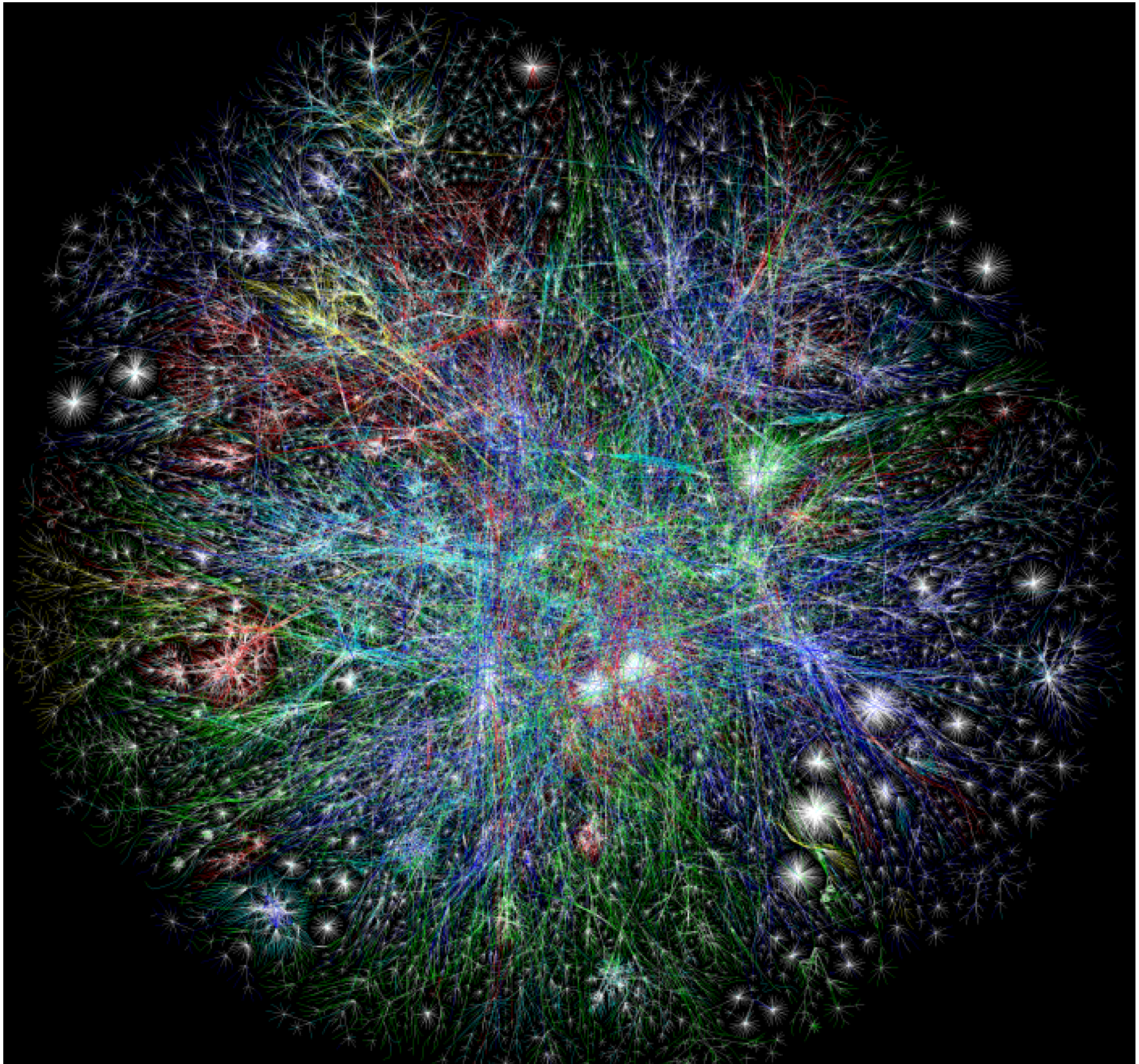
- **Failures: must re-establish connection**
  - For any failures along path
- **Overload: all or nothing**
  - No graceful degradation
- **Waste: allocate for peak, waste for less than peak**
- **Set up time**

Packet Switching

- **Break information in small chunks: *packets***
- **Each packet forwarded independently**
  - Must add metadata to each packet
- **Allows statistical multiplexing**
  - High utilization
  - Very flexible
  - Fairness not automatic
  - Highly variable queueing delays
  - Different paths for each packet

A Taxonomy of networks





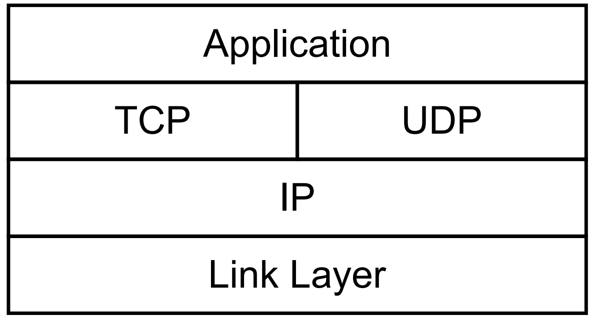
Traceroute map of the Internet, ~5 million edges, circa 2003. opte.org

Managing Complexity

- *Very* **large number of computers**
- **Incredible variety of technologies**
  - Each with very different constraints
- **No single administrative entity**
- **Evolving demands, protocols, applications**
  - Each with very different requirements!

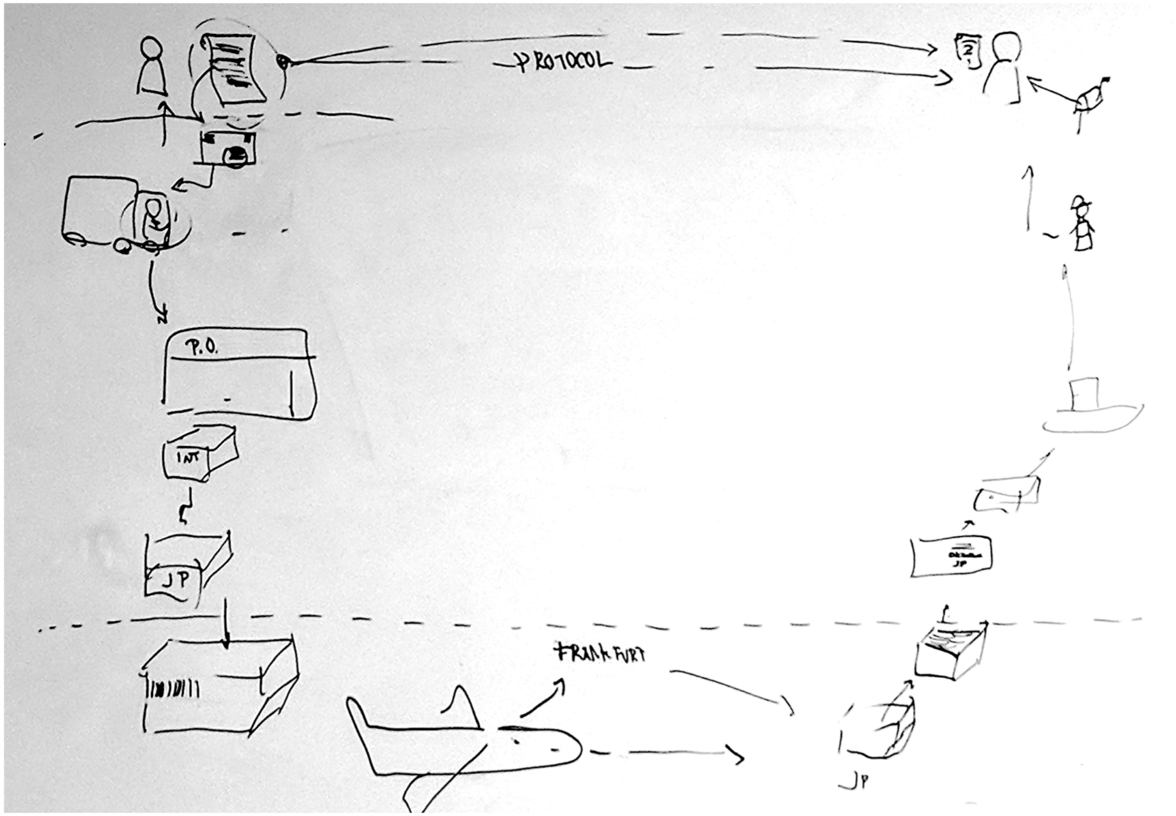
- **How do we make sense of all this?**

Layering



- **Separation of concerns**
  - Break problem into separate parts
  - Solve each one independently
  - Tie together through common interfaces: abstraction
  - Encapsulate data from the layer above inside data from the layer below
  - Allow independent evolution

Analogy to Delivering a Letter

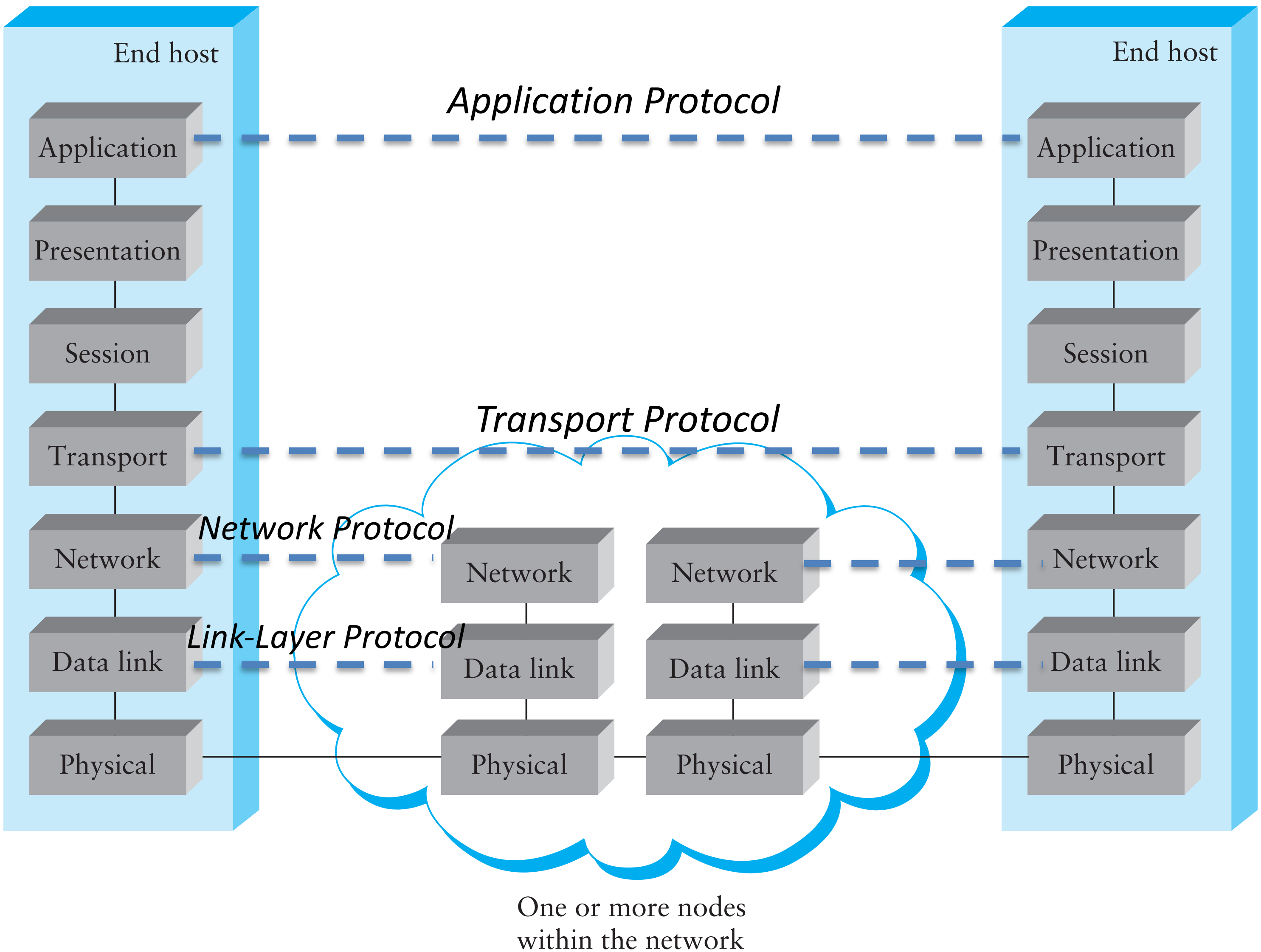


We did this drawing in class, for delivering a letter from you to a friend in Japan. There are many layers, addressing schemes, well-defined interfaces, independent Evolution, abstraction. A protocol is the communication between entities in the same Level (e.g., you and your friend). Layers use the services of lower layers to provide services to upper layers. (Of course, this is much more fun when we are drawing live, hope this starts to get the idea across).

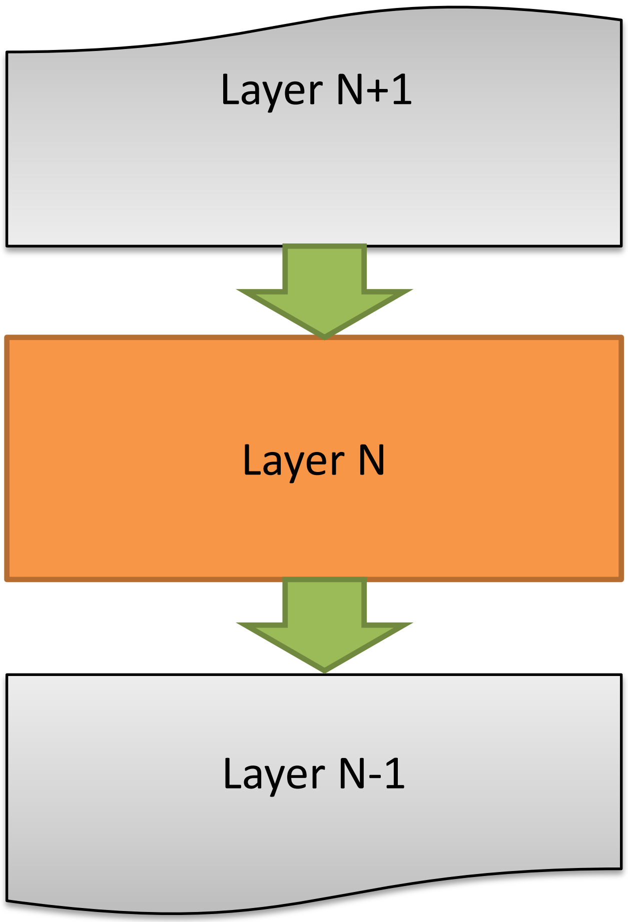
Layers

- **Application – what the users sees, *e.g.*, HTTP**
- **Presentation – crypto, conversion between representations**
- **Session – can tie together multiple streams (*e.g.*, audio & video)**
- **Transport – demultiplexes, provides reliability, flow and congestion control**
- **Network – sends *packets*, using *routing***
- **Data Link – sends *frames*, handles media access**
- **Physical – sends individual bits**

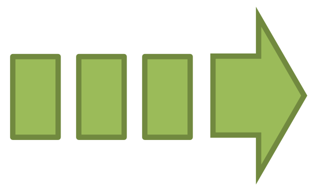
OSI Reference Model



Layers, Services, Protocols



Service: abstraction provided to layer above
API: concrete way of using the service



Protocol: rules for communication within same layer

Layer N uses the services provided by N-1 to implement its protocol and provide its own services

Layers, Services, Protocols

Application	Service: user-facing application. Application-defined messages
Transport	Service: multiplexing applications Reliable byte stream to other node (TCP), Unreliable datagram (UDP)
Network	Service: move packets to any other node in the network IP: Unreliable, best-effort service model
Link	Service: move frames to other node across link. May add reliability, medium access control
Physical	Service: move bits to other node across link

Protocols

- **What do you need to communicate?**
  - Definition of message formats
  - Definition of the semantics of messages
  - Definition of valid sequences of messages
    - Including valid timings
- **Also, who do you talk to? …**

Naming/Addressing

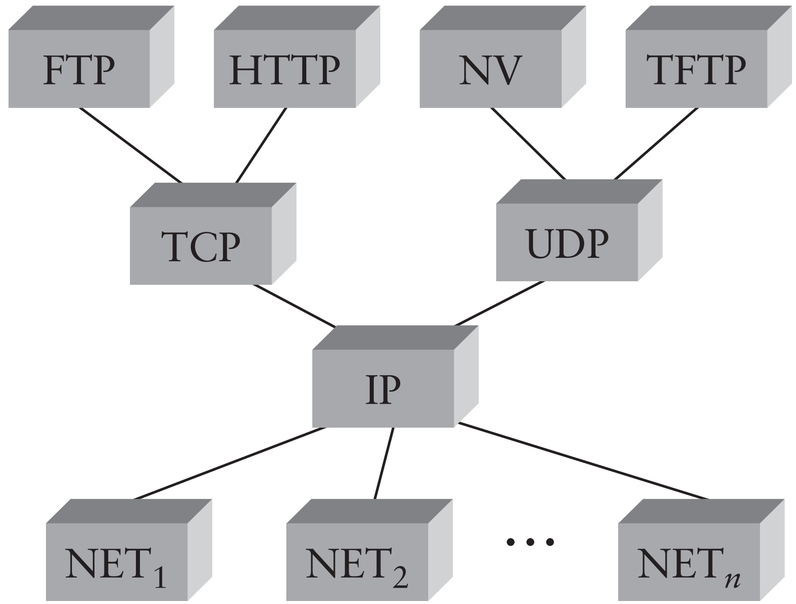
- **Each node typically has a unique* name**
  - When that name also tells you how to get to the node, it is called an *address*
- **Each layer can have its own naming/addressing**
- ***Routing* is the process of finding a path to the destination**
  - In packet switched networks, each packet must have a destination address
  - For circuit switched, use address to set up circuit
- **Special addresses can exist for broadcast/multicast/anycast**

** within the relevant scope*

Challenge

- **Decide on how to factor the problem**
  - What services at which layer?
  - What to leave out?
  - More on this later (End-to-end principle)
- **For example:**
  - IP offers pretty crappy service, even on top of reliable links… why?
  - TCP: offers reliable, in-order, no-duplicates service. Why would you want UDP?

IP as the Narrow Waist

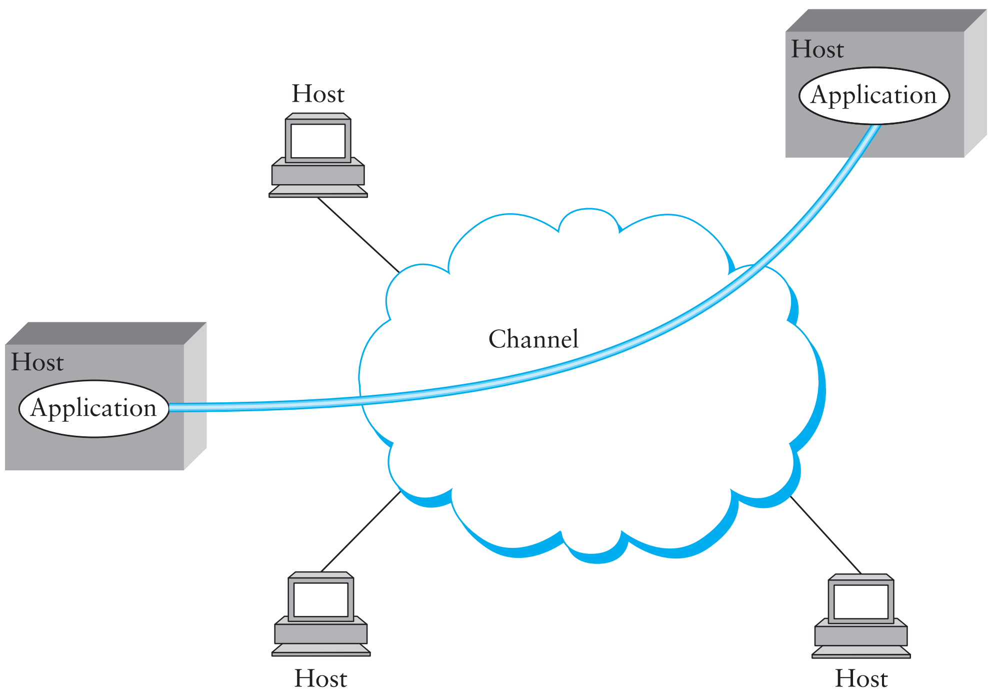


- **Many applications protocols on top of UDP & TCP**
- **IP works over many types of networks**
- **This is the "Hourglass" architecture of the Internet.**
  - If every network supports IP, applications run over many different networks (*e.g.*, cellular network)

Network Layer: Internet Protocol (IP)

- **Used by most computer networks today**
  - Runs *over* a variety of physical networks, can connect Ethernet, wireless, modem lines, etc.
- **Every host has a unique 4-byte IP address (IPv4)**
  - *E.g.*, www.cs.brown.edu →128.148.32.110
  - The *network* knows how to route a packet to any address
- **Need more to build something like the Web**
  - Need naming (DNS)
  - Interface for browser and server software
  - Need demultiplexing within a host: which packets are for web browser, Skype, or the mail program?

Inter-process Communication



- Talking from host to host is great, but we want abstraction of inter-process communication
- Solution: *encapsulate* another protocol within IP

Transport: UDP and TCP

- **UDP and TCP most popular protocols on IP**
  - Both use 16-bit *port* number & 32-bit IP address
  - Applications *bind* a port & receive traffic on that port
- **UDP – User (unreliable) Datagram Protocol**
  - Exposes packet-switched nature of Internet
  - Adds multiplexing on top of IP
  - Sent packets may be dropped, reordered, even duplicated (but there is corruption protection)
- **TCP – Transmission Control Protocol**
  - Provides illusion of reliable 'pipe' or 'stream' between two processes anywhere on the network
  - Handles congestion and flow control

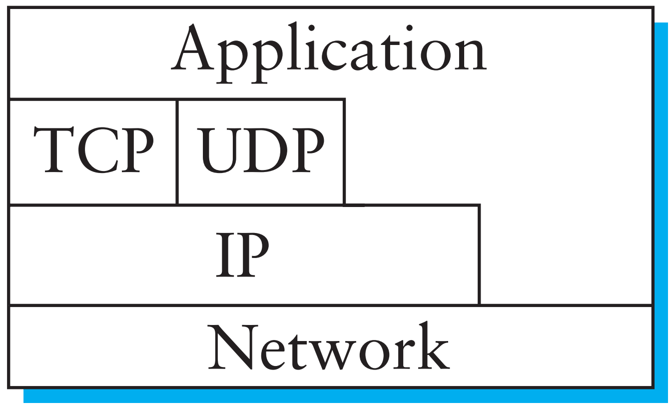
Uses of TCP

- **Most applications use TCP**
  – Easier to program (reliability is convenient)
  – Automatically avoids congestion (don't need to worry about taking down the network
- **Servers typically listen on well-know ports:**
  – SSH: 22
  – SMTP (email): 25
  – Finger: 79
  – HTTP (web): 80

Transport: UDP and TCP

- **UDP and TCP most popular protocols on IP**
  - Both use 16-bit *port* number & 32-bit IP address
  - Applications *bind* a port & receive traffic on that port
- **UDP – User (unreliable) Datagram Protocol**
  - Exposes packet-switched nature of Internet
  - Adds multiplexing on top of IP
  - Sent packets may be dropped, reordered, even duplicated (but there is corruption protection)
- **TCP – Transmission Control Protocol**
  - Provides illusion of reliable 'pipe' or 'stream' between two processes anywhere on the network
  - Handles congestion and flow control

Internet Layering

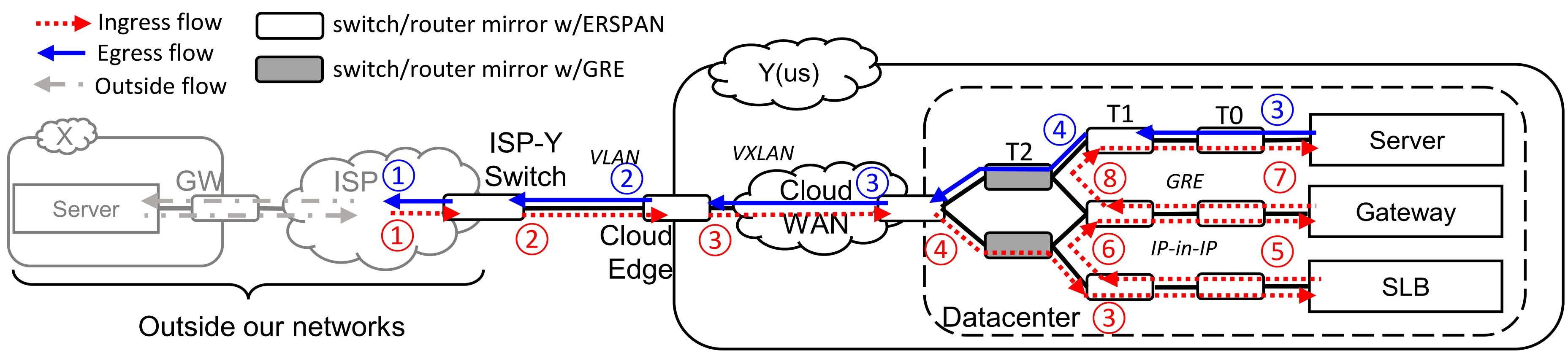


- **Strict layering not *required***
  - TCP/UDP “cheat” to detect certain errors in IP-level information like address
  - Overall, allows evolution, experimentation

One more thing…

- **Layering defines interfaces well**
  - What if I get an Ethernet frame, and send it as the payload of an IP packet across the world?
- **Layering can be recursive**
  - Each layer agnostic to payload!
- **Many examples**
  - Tunnels: e.g.,
    VXLAN is ETH over UDP (over IP over ETH again…)
  - Our IP assignment: IP on top of UDP "links"

Example



Number	Header Format										
	Headers Added after Mirroring			Mirrored Headers							
①	ETHERNET	IPV4	ERSPAN	ETHERNET						IPV4	TCP
②	ETHERNET	IPV4	ERSPAN	ETHERNET					802.1Q	IPV4	TCP
③	ETHERNET	IPV4	ERSPAN	ETHERNET		IPV4	UDP	VXLAN	ETHERNET	IPV4	TCP
④	ETHERNET	IPV4	GRE			IPV4	UDP	VXLAN	ETHERNET	IPV4	TCP
⑤	ETHERNET	IPV4	ERSPAN	ETHERNET	IPV4	IPV4	UDP	VXLAN	ETHERNET	IPV4	TCP
⑥	ETHERNET	IPV4	GRE		IPV4	IPV4	UDP	VXLAN	ETHERNET	IPV4	TCP
⑦	ETHERNET	IPV4	ERSPAN	ETHERNET		IPV4	GRE		ETHERNET	IPV4	TCP
⑧	ETHERNET	IPV4	GRE			IPV4	GRE		ETHERNET	IPV4	TCP

* This is just an example, do not worry about the details, or the specific protocols!

From: Yu et al., A General, Easy to Program and Scalable Framework for Analyzing In-network Packet Traces, NSDI 2019

- **We didn't cover these in class, but these concepts about the socket API are useful for, and exercised by, the Snowcast assignment!**

Using TCP/IP

- **How can applications use the network?**
- ***Sockets* API.**
  - Originally from BSD, widely implemented (*BSD, Linux, Mac OS X, Windows, …)
  - Important do know and do once
  - Higher-level APIs build on them
- **After basic setup, much like files**

Sockets: Communication Between Machines

- **Network sockets are file descriptors too**
- **Datagram sockets: unreliable message delivery**
  - With IP, gives you UDP
  - Send atomic messages, which may be reordered or lost
  - Special system calls to read/write: `send`/`recv`
- **Stream sockets: bi-directional pipes**
  - With IP, gives you TCP
  - Bytes written on one end read on another
  - Reads may not return full amount requested, must re-read

System calls for using TCP

<u>**Client**</u>

<u>**Server**</u>

`socket` – **make socket**

`bind` – **assign address, port**

`listen` – **listen for clients**

`socket` – **make socket**

`bind*` – **assign address**

`connect` – **connect to listening socket**

`accept` – **accept connection**

- **This call to bind is optional, connect can choose address & port.**

Socket Naming

- **Recall how TCP & UDP name communication endpoints**
  - IP address specifies host (128.148.32.110)
  - 16-bit port number demultiplexes within host
  - Well-known services listen on standard ports (*e.g.* ssh – 22, http – 80, mail – 25, see /etc/services for list)
  - Clients connect from arbitrary ports to well known ports
- **A connection is named by 5 components**
  - Protocol, local IP, local port, remote IP, remote port
  - TCP requires connected sockets, but not UDP

Dealing with Address Types

- **All values in network byte order (Big Endian)**
  - `htonl()`, `htons()`: host to network, 32 and 16 bits
  - `ntohl()`, `ntohs()`: network to host, 32 and 16 bits
  - Remember to always convert!
- **All address types begin with family**
  - `sa_family` in `sockaddr` tells you actual type
- **Not all addresses are the same size**
  - e.g., `struct sockaddr_in6` is typically 28 bytes, yet generic `struct sockaddr` is only 16 bytes
  - So most calls require passing around socket length
  - New `sockaddr_storage` is big enough

Client Skeleton (IPv4)

```
struct sockaddr_in {
        short   sin_family;  /* = AF_INET */
        u_short sin_port;    /* = htons (PORT) */
        struct  in_addr sin_addr;
        char    sin_zero[8];
} sin;

int s = socket (AF_INET, SOCK_STREAM, 0);
bzero (&sin, sizeof (sin));
sin.sin_family = AF_INET;
sin.sin_port = htons (13);  /* daytime port */
sin.sin_addr.s_addr = htonl (IP_ADDRESS);
connect (s, (sockaddr *) &sin, sizeof (sin));
while ((n = read (s, buf, sizeof (buf))) > 0)
  write (1, buf, n);
```

Server Skeleton (IPv4)

```
int s = socket (AF_INET, SOCK_STREAM, 0);
struct sockaddr_in sin;
bzero (&sin, sizeof (sin));
sin.sin_family = AF_INET;
sin.sin_port = htons (9999);
sin.sin_addr.s_addr = htonl (INADDR_ANY);
bind (s, (struct sockaddr *) &sin, sizeof (sin));
listen (s, 5);

for (;;) {
  socklen_t len = sizeof (sin);
  int cfd = accept (s, (struct sockaddr *) &sin, &len);
  /* cfd is new connection; you never read/write s */
  do_something_with (cfd);
  close (cfd);
}
```

Using UDP

- **Call socket with SOCK_DGRAM, bind as before**
- **New calls for sending/receiving individual packets**
  - `sendto(int s, const void *msg, int len, int flags, const struct sockaddr *to, socklen t tolen);`
  - `recvfrom(int s, void *buf, int len, int flags, struct sockaddr *from, socklen t *fromlen);`
  - Must send/get peer address with each packet
- **Example:** `udpecho.c`
- **Can use UDP in connected mode (Why?)**
  - connect assigns remote address
  - `send`/`recv` syscalls, like `sendto`/`recvfrom` w/o last two arguments

Uses of UDP Connected Sockets

- **Kernel demultiplexes packets based on port**
  – Can have different processes getting UDP packets from different peers
- **Feedback based on ICMP messages (future lecture)**
  – Say no process has bound UDP port you sent packet to
  – Server sends port unreachable message, but you will only receive it when using connected socket

Serving Multiple Clients

- **A server may block when talking to a client**
  - Read or write of a socket connected to a slow client can block
  - Server may be busy with CPU
  - Server might be blocked waiting for disk I/O
- **Concurrency through multiple processes**
  - Accept, fork, close in parent; child services request
- **Advantages of one process per client**
  - Don’t block on slow clients
  - May use multiple cores
  - Can keep disk queues full for disk-heavy workloads

Threads

- **One process per client has disadvantages:**
  - High overhead – fork + exit ~100µsec
  - Hard to share state across clients
  - Maximum number of processes limited
- **Can use threads for concurrency**
  - Data races and deadlocks make programming tricky
  - Must allocate one stack per request
  - Many thread implementations block on some I/O or have heavy thread-switch overhead

Rough equivalents to `fork()`, `waitpid()`, `exit()`, `kill()`, plus locking primitives.

Non-blocking I/O

- **fcntl sets O_NONBLOCK flag on descriptor**

```
int n;
if ((n = fcntl(s, F_GETFL)) >= 0)
  fcntl(s, F_SETFL, n|O_NONBLOCK);
```

- **Non-blocking semantics of system calls:**
  - read immediately returns -1 with errno EAGAIN if no data
  - write may not write all data, or may return EAGAIN
  - connect may fail with EINPROGRESS (or may succeed, or may fail with a real error like ECONNREFUSED)
  - accept may fail with EAGAIN or EWOULDBLOCK if no connections present to be accepted

How do you know when to read/write?

```
struct timeval {
  long    tv_sec;          /* seconds */
  long    tv_usec;         /* and microseconds */
};

int select (int nfds, fd_set *readfds, fd_set *writefds,
            fd_set *exceptfds, struct timeval *timeout);
FD_SET(fd, &fdset);
FD_CLR(fd, &fdset);
FD_ISSET(fd, &fdset);
FD_ZERO(&fdset);
```

- **Entire program runs in an *event loop***

Event-driven servers

- **Quite different from processes/threads**
  – Race conditions, deadlocks rare
  – Often more efficient
- **But…**
  – Unusual programming model
  – Sometimes difficult to avoid blocking
  – Scaling to more CPUs is more complex

Coming Up

- **Next class: Physical Layer**
- **Thu 13th: Snowcast milestones: last commit**
  - Let us know if you don't have a slot!
- **Pushed some dates:**
  - Snowcast now due Monday