Principles of Computer Networking

Larry Peterson

Department of Computer Science
University of Arizona

Spring 1996

Copyright Notice

Overview

Expectations

This class **IS** about...

- Principles and Concepts
- General-Purpose Computer Networks
- Internet Perspective
- Network Software
- Designing and Building a System

This class **IS NOT** about...

- Survey of existing protocol standards
- Specialized networks (e.g., CATV, telephone)
- OSI Perspective
- Network Hardware (we do survey)
- Queuing Theory

Perspective

- Network users: services that their applications need e.g., guarantee that each message it sends will be delivered without error within a certain amount of time

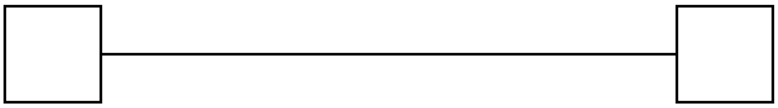
- Network designers: cost-effective design e.g., that network resources are efficiently utilized and fairly allocated to different users

- Network providers: system that is easy to administer and manage e.g., that faults can be easily isolated and it is easy to account for usage

Requirements

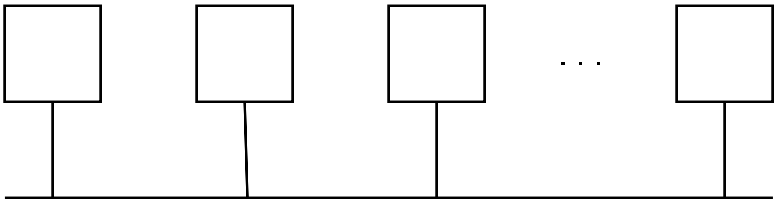
Connectivity

- Building Blocks
  - links: coax cable, optical fiber...
  - nodes: general-purpose workstations...
- Direct Links
  - point-to-point



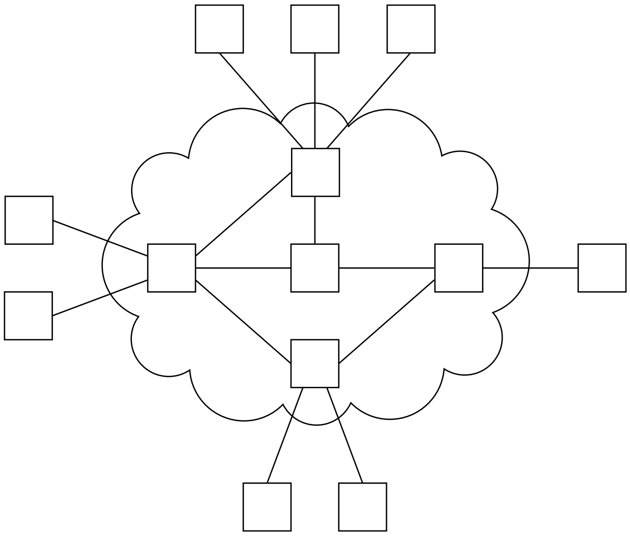
point-to-point network

  - multiple access

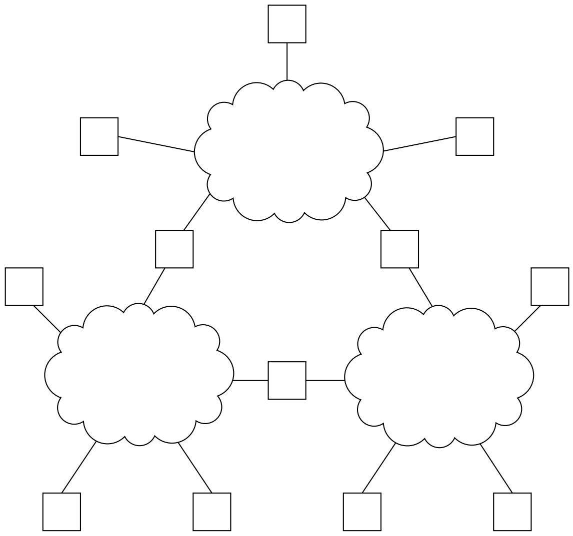


multiple access network

- Indirect Connectivity
  - switched networks



  - internetworks

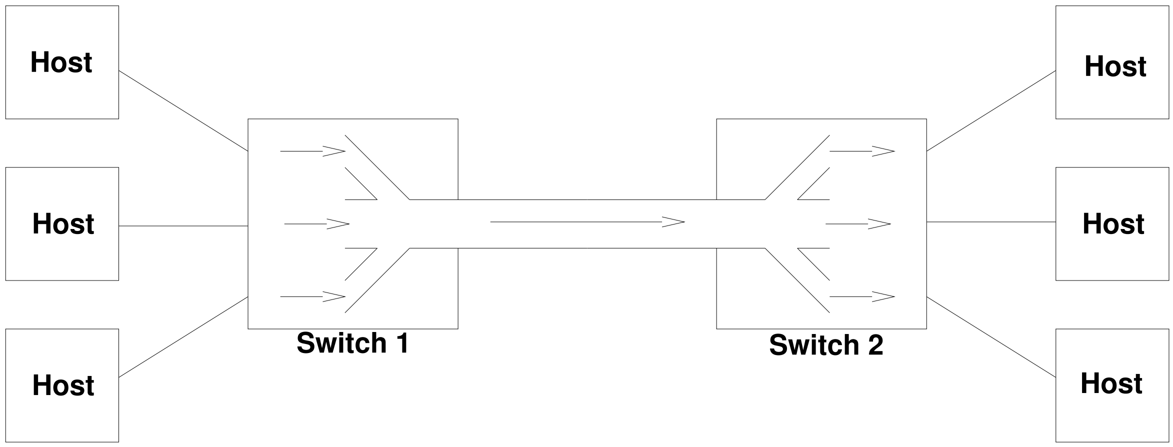


A network can be defined recursively as two or more nodes connected by a physical link, or by two or more networks connected by one or more nodes.

- Switching Strategies
  - circuit switching: dedicated circuit; send/receive a bit stream
  - packet switching: store-and-forward; send/receive messages (packets)
- Addressing and Routing
  - address: byte-string that identifies a node; usually unique
  - routing: process of determining how to forward messages towards the destination node based on its address
  - types of addresses
    - unicast: node-specific
    - broadcast: all nodes on the network
    - multicast: some subset of nodes on the network

Cost-Effective Resource Sharing

Must share (*multiplex*) network resources (nodes and links) among multiple users.

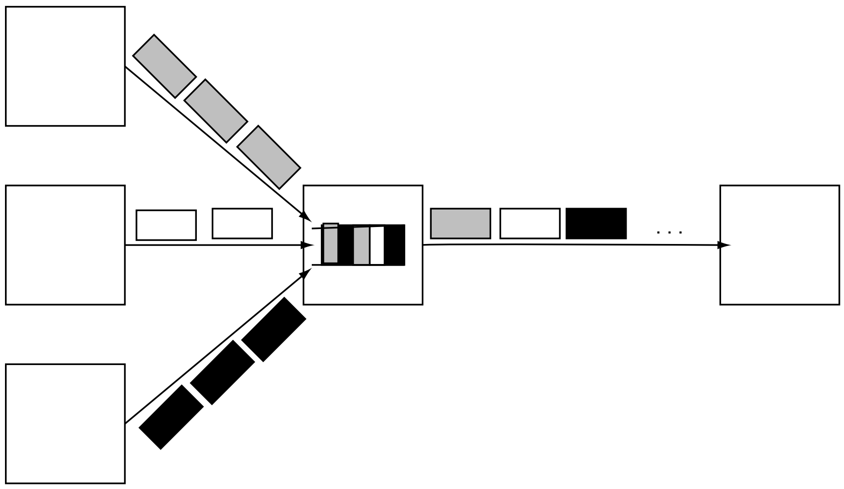


Common Multiplexing Strategies

- Time-Division Multiplexing (TDM)
- Frequency-Division Multiplexing (FDM)

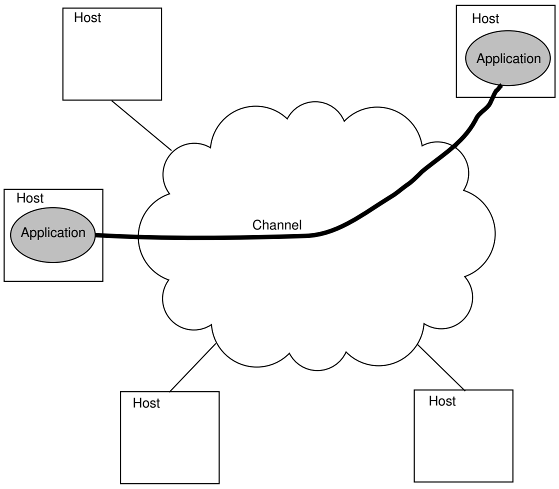
Statistical Multiplexing

- Time-division, but on demand rather than fixed
- Reschedule link on a per-packet basis
- Packets from different sources interleaved on the link
- Buffer packets that are *contending* for the link
- Packet queue may be processed FIFO, but not necessarily
- Buffer overflow is called *congestion*



Functionality

The application programs running on the hosts connected to the network must be able to communicate in a meaningful way.



Network supports common process-to-process channels; e.g.,

- Request/Reply: for file access and digital libraries
- Message Stream: for video applications
  - video: sequence of frames
  - resolution: 1/4 TV-size image = 352x240 pixels;
  - 24-bit color: frame = $(352 \times 240 \times 24)/8 = 247.5$KB;
  - frame rate: 30 fps = 7500KBps = 60Mbps
  - video on-demand versus video-conferencing

What Goes Wrong in the Network?

- Bit-level errors (electrical interference)
- Packet-level errors (congestion)
- Link and node failures

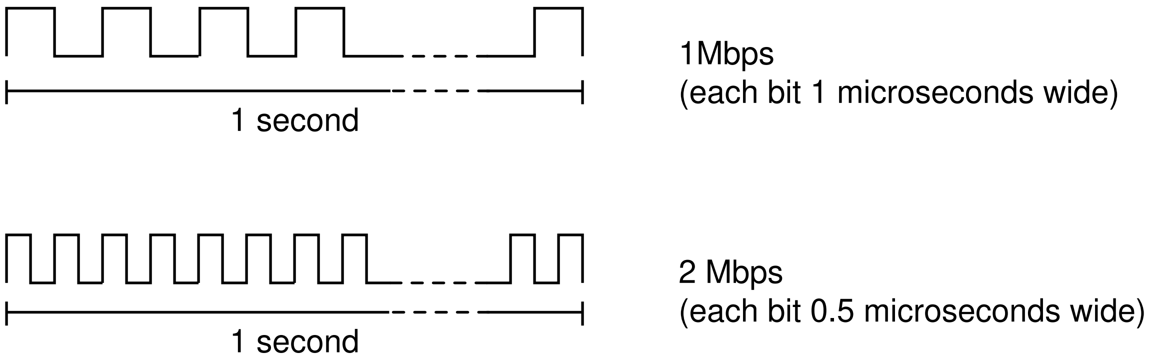
- Messages are delayed
- Messages are deliver out-of-order
- Third parties eavesdrop

The key problem is to fill in the gap between what applications expect and what the underlying technology provides.

Performance

Bandwidth (throughput)

- Amount of data that can be transmitted per time unit
- Example: 10Mbps
- link versus end-to-end
- Notation
  - KB = 2^{10} bytes
  - Mbps = 10^6 bits per second
- Bandwidth related to “bit width”



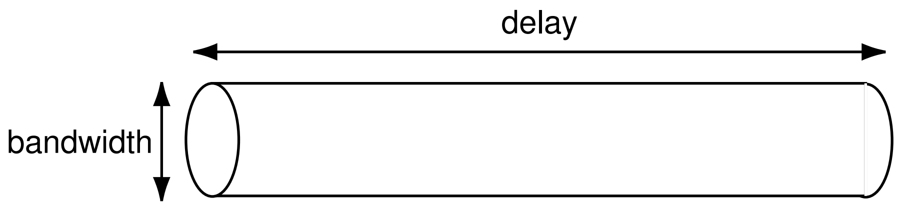
Latency (delay)

- Time it takes to send message from point A to point B
- Example: 24 milliseconds (ms)
- Sometimes interested in in round-trip time (RTT)
- Components of latency

  Latency = Propagation + Transmit + Queue
  Propagation = Distance / SpeedOfLight
  Transmit = Size / Bandwidth

- Speed of light
  - 3.0×10^8 meters/second in a vacuum
  - 2.3×10^8 meters/second in a cable
  - 2.0×10^8 meters/second in a fiber
- Notes
  - no queuing delays in direct link
  - bandwidth not relevant if Size = 1 bit
  - process-to-process latency includes software overhead
  - software overhead can dominate when Distance is small

- Relative importance of bandwidth and latency
  - small message (e.g., 1 byte): 1ms vs 100ms dominates 1Mbps vs 100Mbps
  - large message (e.g., 25 MB): 1Mbps vs 100Mbps dominates 1ms vs 100ms
- Delay $\times$ Bandwidth Product



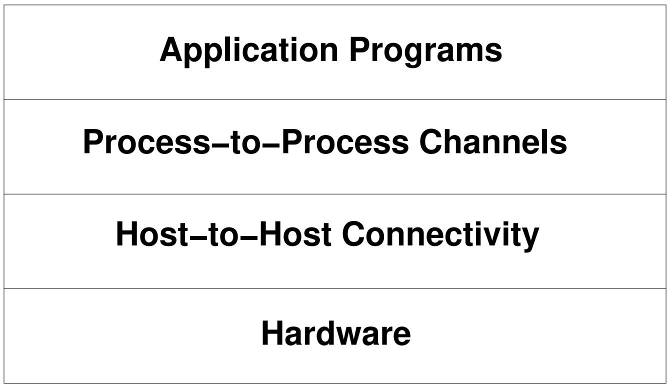
  Example: 100ms RTT and 45Mbps Bandwidth = 560 KB of data

- Application Needs
  - bandwidth requirements: burst versus peak rate
  - jitter: variance in latency (inter-packet gap)

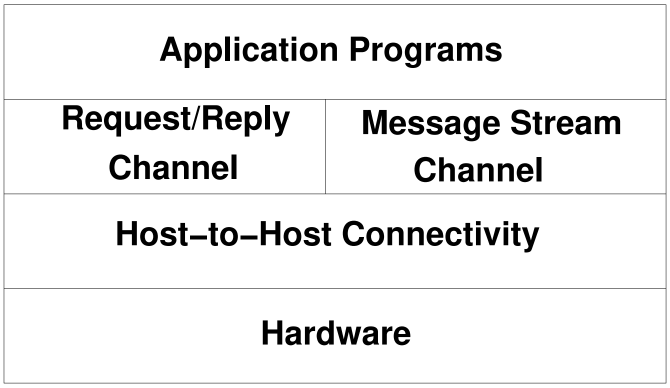
Network Architecture

Layering

- Use abstractions to hide complexity
- Abstraction naturally leads to layering

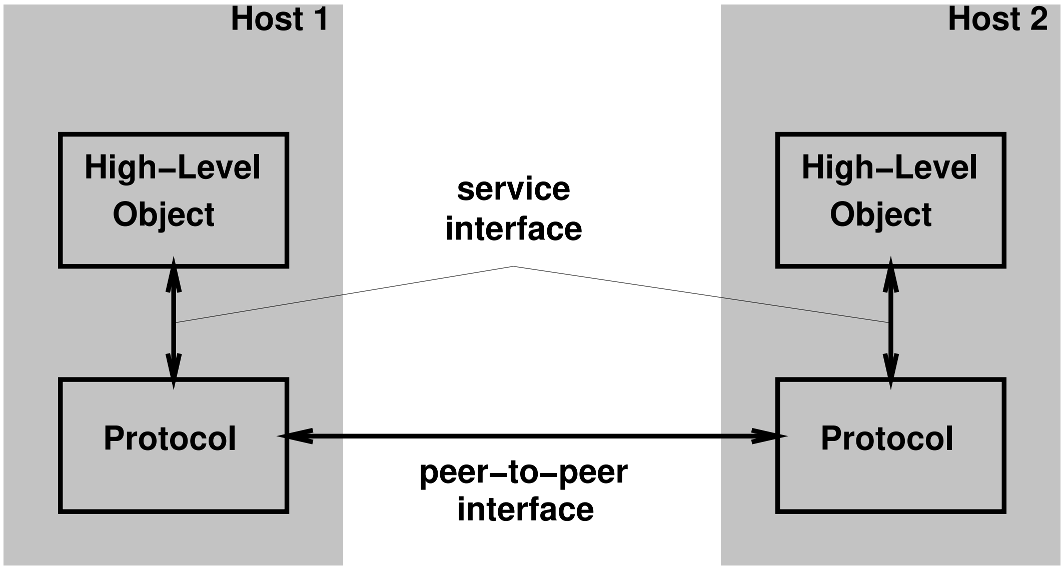


- Can have alternative abstractions at each layer



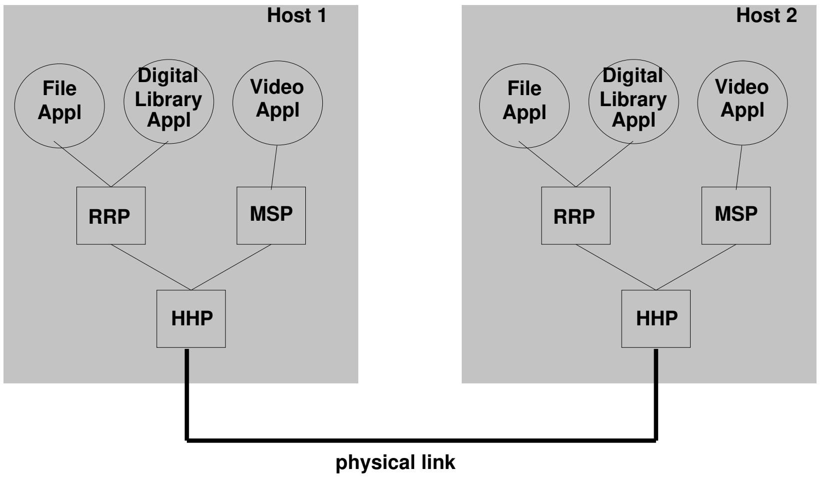
Protocols

- Building blocks of a network architecture
- Each protocol object has two different interfaces
  - *service interface*: defines operations on this protocol
  - *peer-to-peer interface*: defines messages exchanged with peer

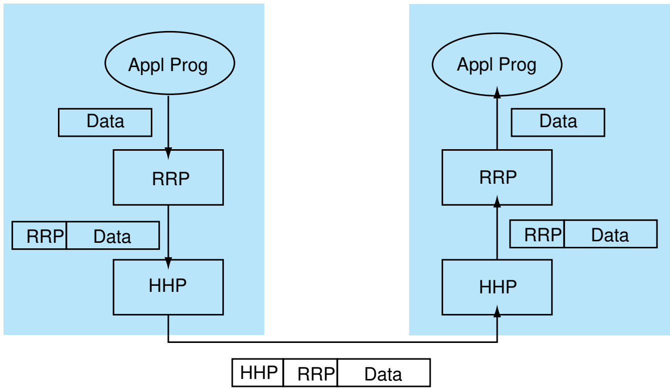


- Term "protocol" is overloaded
  - specification of peer-to-peer interface
  - module that implements this interface

- Protocol Graph
  - collection of protocols and their dependencies
  - most peer-to-peer communication is indirect
  - peer-to-peer is direct only at hardware level



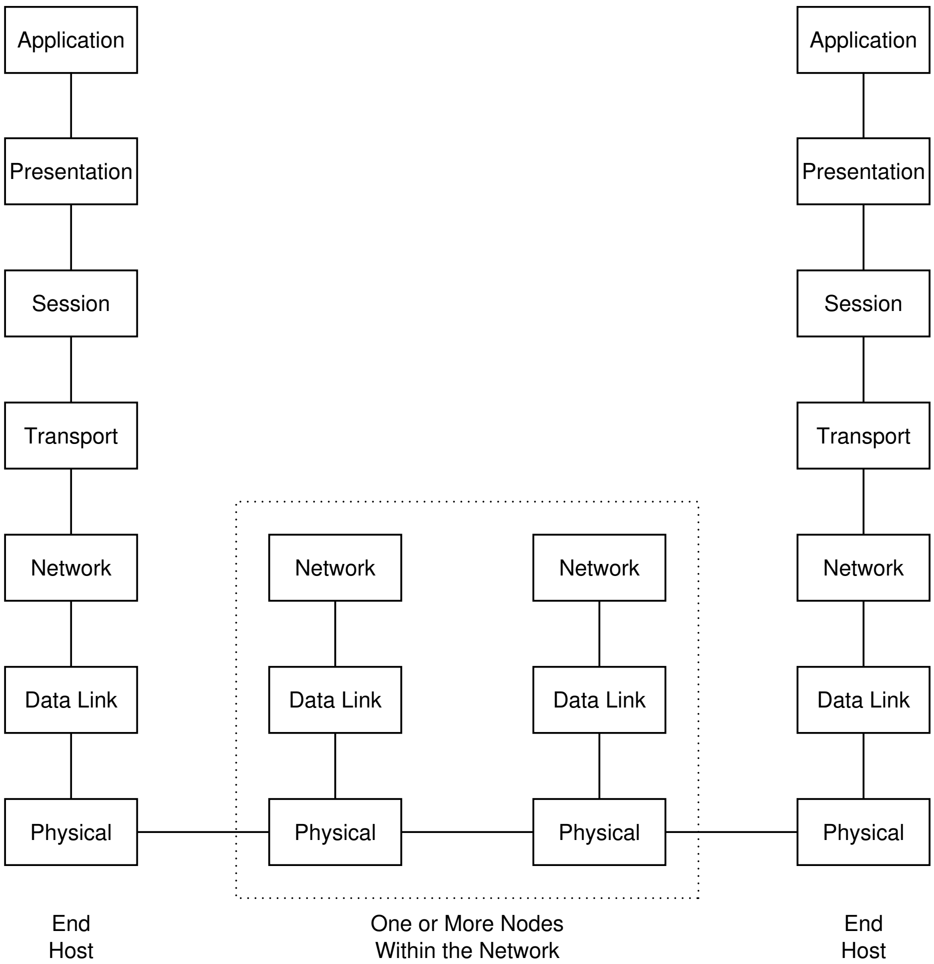
- Multiplexing and Demultiplexing (demux key)
- Encapsulation (header/body)



Standard Architectures

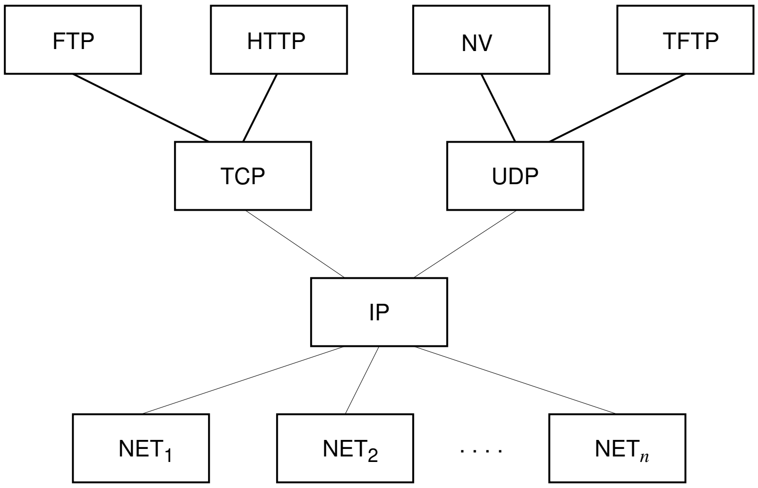
Open Systems Interconnect (OSI) Architecture

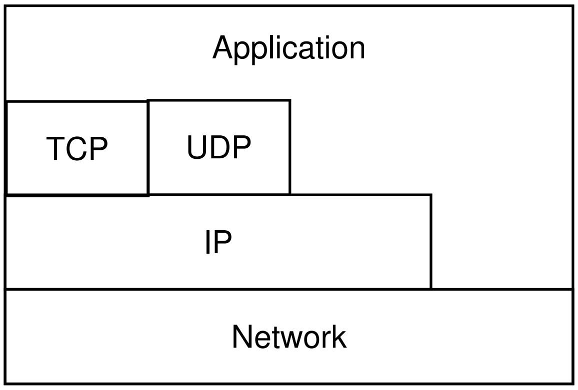
- International Standards Organization (ISO)
- International Telecommunications Union (ITU); formerly CCITT
- "X dot" series: X.25, X.400, X.500
- Reference Model



Internet Architecture

- Internet Engineering Task Force (IETF)





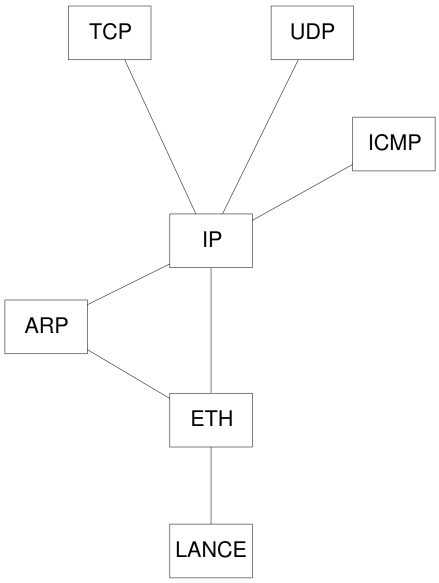
- Application vs Application Protocol (FTP, HTTP)
- Features
  - does not imply strict layering
  - hourglass shape
  - design and implementation go hand-in-hand

x-kernel

Overview

- Framework for Implementing Network Protocols
- Hides OS/Architecture-Specific Details
- Defines Network-Centric Abstractions (Interfaces)
  - uniform protocol interface
  - useful library routines
- Implemented in C (not C++)

Protocol Objects




```
name=lance;
name=eth protocols=lance;
name=arp protocols=eth;
name=ip protocols=eth,arp;
name=icmp protocols=ip;
name=udp protocols=ip;
name=tcp protocols=ip;
```

Operations on Protocol Objects

- Active Open

```
Sessn xOpen(Protl hlp,
            Protl llp,
            Part *participants)
```

- Passive Open

```
XkReturn xOpenEnable(Protl hlp,
                     Protl llp,
                     Part *participant)

XkReturn xOpenDone(Protl hlp,
                   Protl llp,
                   Sessn session,
                   Part *participants)
```

- Demultiplexing

```
XkReturn xDemux(Protl hlp,
                Sessn lls,
                Msg *message)
```

Session Objects

- Local end-point of a channel
- Interpret messages and maintain channel state
- Export operations for sending and receiving messages

Operations

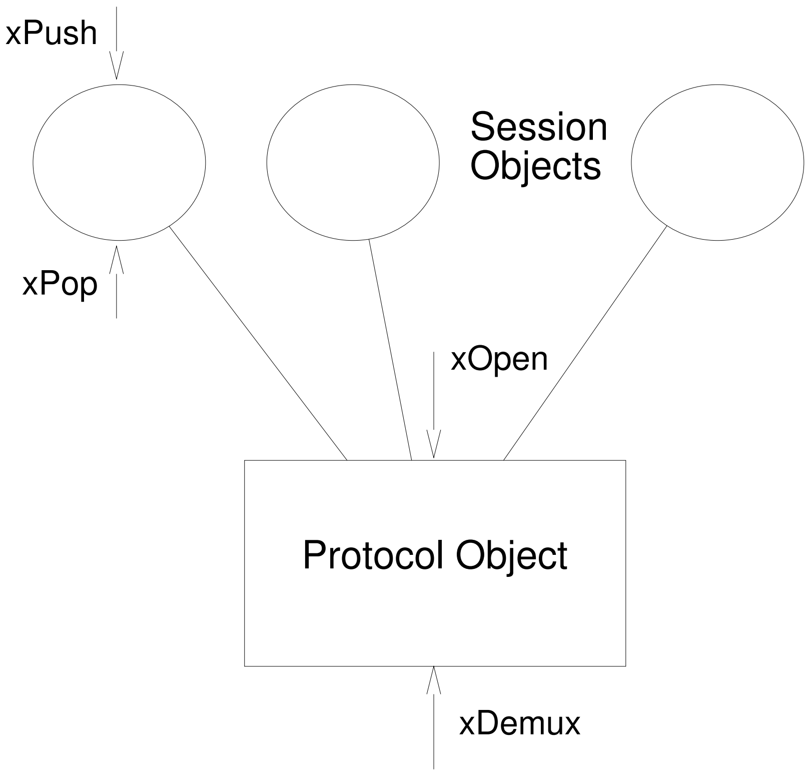
- Send a Message

```
XkReturn xPush(Sessn lls,
               Msg *message)
```

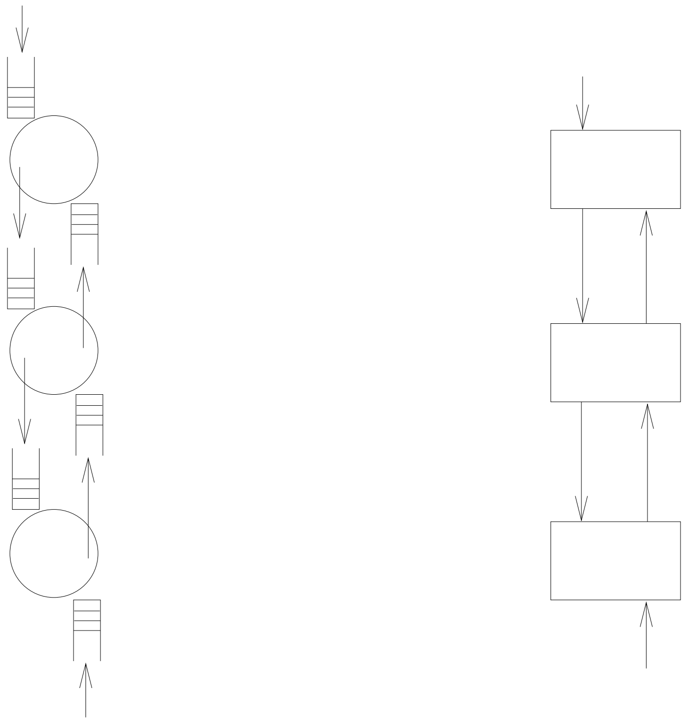
- Deliver a Message

```
XkReturn xPop(Sessn hls,
              Sessn lls,
              Msg *message,
              void *hdr)
```

Relationship Between Protocols and Sessions



Process Model



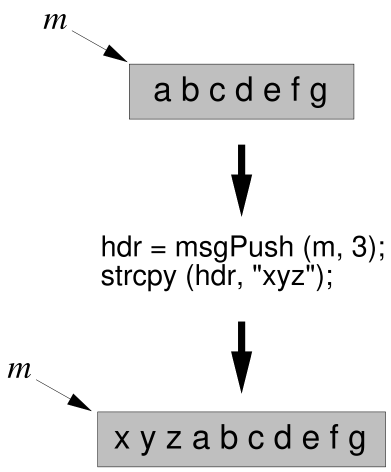
(a) process–per–protocol (b) process–per–message

Synchronization Primitives

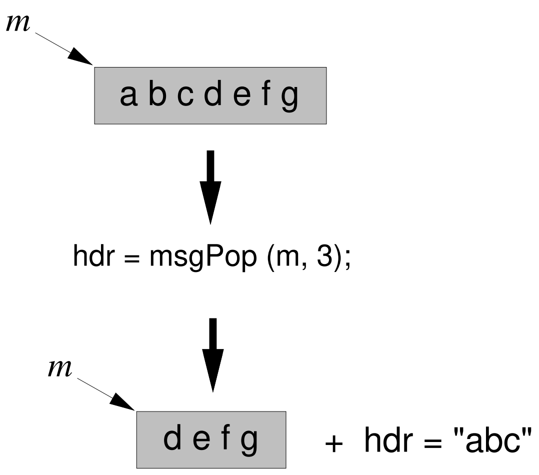
```
void semInit(Semaphore *s, int count)
void semSignal(Semaphore *s)
void semWait(Semaphore *s)
```

Message Library

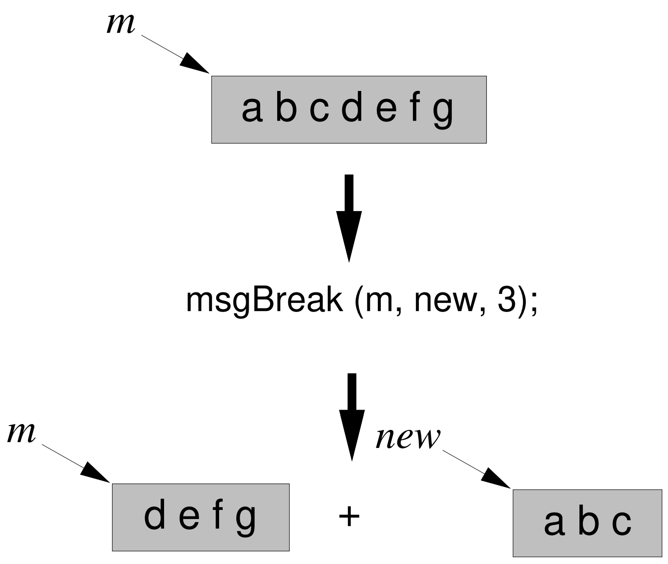
- Add a header to the front of a message



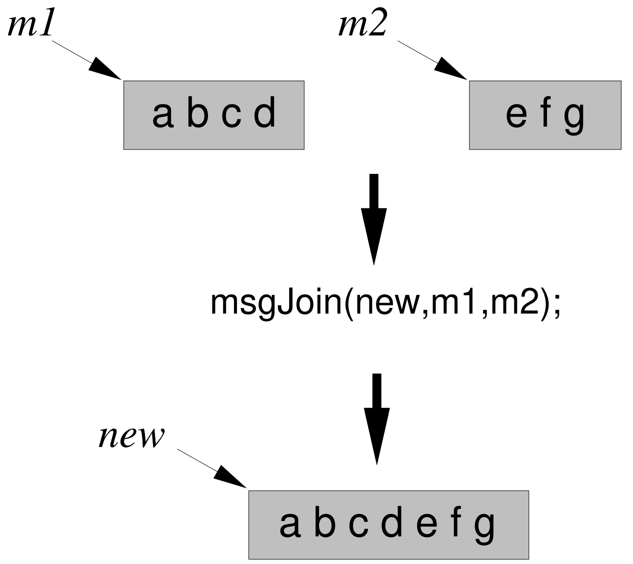
- Strip a header from the front of a message



- Fragment a message into two messages



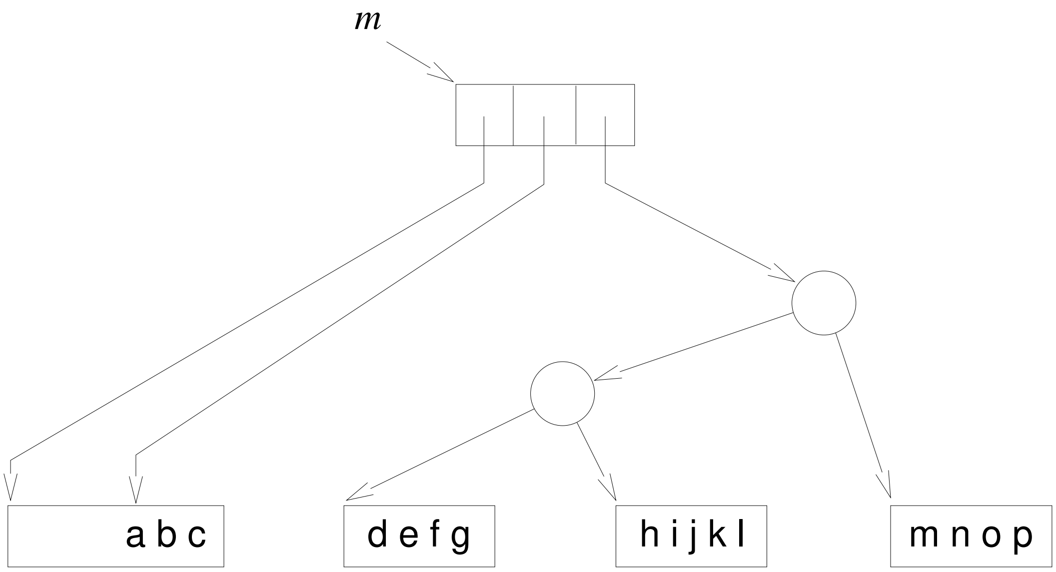
- Reassemble two messages into one message



- Other Operations

```
void msgConstructEmpty(Msg *message)

void msgConstructBuffer(Msg *message,
                        char *buffer,
                        int length)

char *msgConstructAllocate(Msg *message,
                           int length)

void msgAssign(Msg *message_1,
               Msg *message_2)

int msgLength(Msg *message)
```

■ Implementation



Key: Avoid copying packet data from one buffer to another.

■ Traversing Message Data Structure

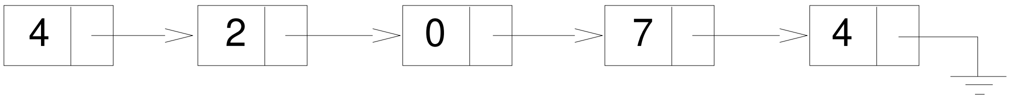
```
void msgWalkInit(MsgWalk *cxt,
                 Msg *message)

char *msgWalkNext(MsgWalk *cxt,
                  int *len)

void msgWalkDone(MsgWalk *cxt)
```

Event Library

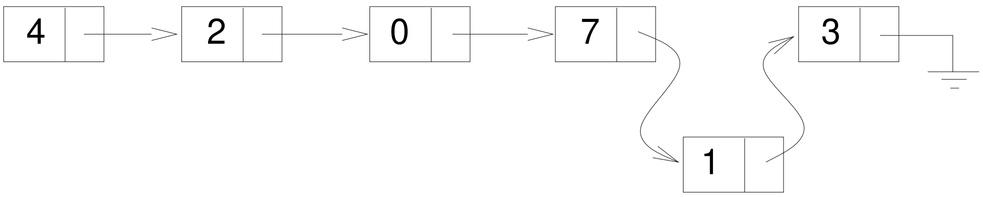
- Scheduling Timeouts

```
Event evSchedule(EvFunc function,
                 void *argument,
                 int time)

EvCancelReturn evCancel(Event event)
```

- Implementation
  - delta list



  - insert event at time 14



Map Library

- Demultiplexing Packets

```
Map mapCreate(int number,
              int size)

Binding mapBind(Map map,
                void *key,
                void *id)

XkReturn mapResolve(Map map,
                    void *key,
                    void **id)
```

- Usage
  - one map to remember active opens (active map)
  - one map to remember passive opens (passive map)
- Implementation
  - hash table
  - specialized code based on key size
  - one-element cache

Example

```
static Sessn
aspOpen(Protl self, Protl hlp, Part *p)
{
    Sessn       asp_s;
    Sessn       lls;
    ActiveId    key;
    ProtlState  *pstate = (ProtlState *)self->state;

    bzero((char *)&key, sizeof(key));

    /* High level protocol must specify both */
    /* local and remote ASP port */
    key.localport = *((AspPort *) partPop(p[0]));
    key.remoteport = *((AspPort *) partPop(p[1]));

    /* Assume failure until proven otherwise */
    asp_s = XK_FAILURE;
```

```
    /* Open session on low-level protocol */
    lls = xOpen(self, xGetDown(self, 0), p);
    if ( lls != XK_FAILURE )
    {
        key.lls = lls;
        /* Check for session in active map */
        if (mapResolve(pstate->activemap, &key,
            (void **)&asp_s) == XK_FAILURE)
        {
         /* Session not in map; initialize it */
            asp_s = asp_init_sessn(self, hlp, &key);
            if ( asp_s != XK_FAILURE )
            {
                /* A successful open */
                return asp_s;
            }
        }

        /* Error has occurred */
        xClose(lls);
    }
    return asp_s;
}
```

```
static XkReturn
aspPush(Sessn s, Msg *msg)
{
    SessnState *sstate;
    AspHdr     hdr;
    void       *buf;

    /* create a header */
    sstate = (SessnState *) s->state;
    hdr = sstate->hdr;
    hdr.ulen = msgLen(msg) + HLEN;

    /* attach header and send */
    buf = msgPush(msg, HLEN);
    aspHdrStore(&hdr, buf, HLEN, msg);
    return xPush(xGetDown(s, 0), msg);
}
```

```
static XkReturn
aspDemux(Protl self, Sessn lls, Msg *msg)
{
    AspHdr       h;
    Sessn        s;
    ActiveId     activeid;
    PassiveId    passiveid;
    ProtlState   *pstate;
    Enable       *e;
    void         *buf;

    pstate = (ProtlState *)self->state;

    /* extract the header from the message */
    buf = msgPop(msg, HLEN);
    aspHdrLoad(&h, buf, HLEN, msg);

    /* construct a demux key from the header */
    bzero((char *)&activeid, sizeof(activeid));
    activeid.localport = h.dport;
    activeid.remoteport = h.sport;
    activeid.lls = lls;
```

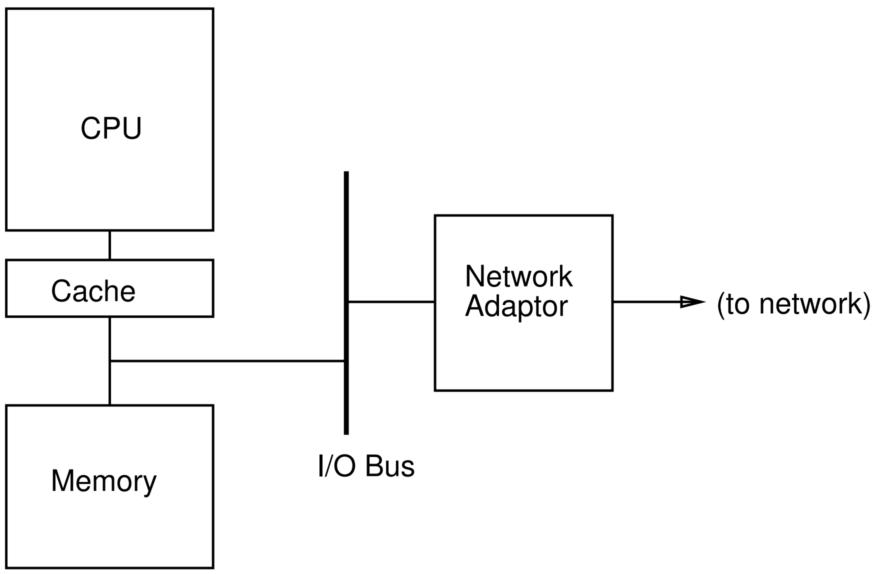
```
    /* see if demux key is in the active map */
    if (mapResolve(pstate->activemap, &activeid,
        (void **)&s) == XK_FAILURE)
    {
        /* didn't find an active session, */
        /* so check passive map */
        passiveid = h.dport;
        if (mapResolve(pstate->passivemap, &passiveid,
            (void **)&e) == XK_FAILURE)
        {
            /* drop the message */
            return XK_SUCCESS;
        }

        /* port was enabled, so create a new */
        /* session and inform hlp */
        s = asp_init_sessn(self, e->hlp, &activeid);
        xOpenDone(e->hlp, s, self);
    }
    /* pop the message to the session */
    return xPop(s, lls, msg, &h);
}
```

```
static XkReturn
aspPop(Sessn s, Sessn ds, Msg *msg, void *inHdr)
{
    AspHdr    *h = (AspHdr *) inHdr;

    /* truncate message to length shown in header */
    if ((h->ulen - HLEN) < msgLen(msg))
        msgTruncate(msg, (int) h->ulen);

    /* pass message up the protocol stack */
    return xDemux(xGetUp(s), s, msg);
}
```

Hardware Building Blocks

Nodes

Assume general-purpose (programmable) computers; e.g., workstations. Sometimes replaced with special-purpose hardware.



- Finite memory (implies limited buffer space)
- Connects to network via a *network adaptor*
- Fast processor, slow memory

Links

Sometimes you install your own

Category 5 twisted pair	10-100Mbps, 100m
50-ohm coax (ThinNet)	10-100Mbps, 200m
75-ohm coax (ThickNet)	10-100Mbps, 500m
Multimode fiber	100Mbps, 2km
Single-mode fiber	100-2400Mbps, 40km

Sometimes leased from the phone company

Service to ask for	Bandwidth you get
ISDN	64 Kbps
T1	1.544 Mbps
T3	44.736 Mbps
STS-1	51.840 Mbps
STS-3	155.250 Mbps
STS-12	622.080 Mbps
STS-24	1.244160 Gbps
STS-48	2.488320 Gbps

Encoding

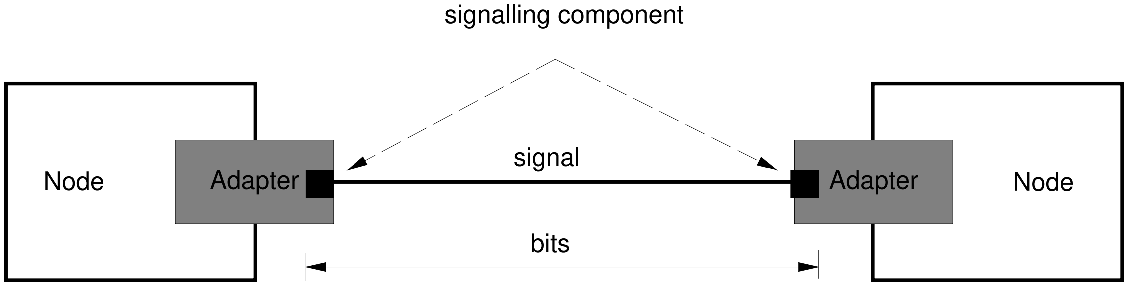
Overview

Signals propagate over a physical medium.

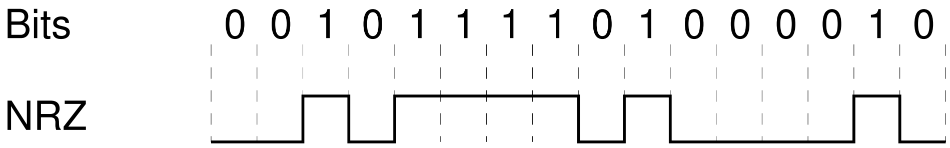
- Digital signals
- Analog signals

Data can be either digital or analog; we're interested in digital data.

Problem: Encode the binary data that the source node wants to send to the destination node into the signal that propogates over



Non-Return to Zero (NRZ)

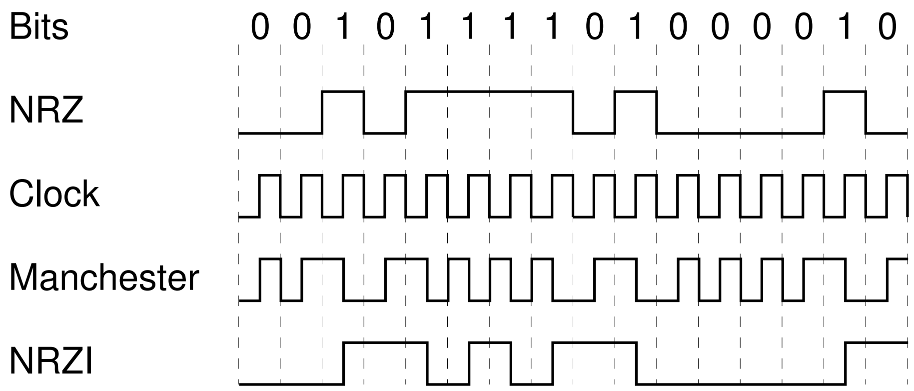


Problem: Consecutive 1s or 0s

- Low signal (0) may be interpretted as no signal
- High signal (1) leads to baseline wander
- Unable to recover clock

NRZI and Manchester

- Non-return to Zero Inverted (NRZI): Make a transition from the current signal to encode a one, and stay at the current signal to encode a zero; solves the problem of consecutive ones.
- Manchester: Transmits the XOR of the NRZ encoded data and the clock; only 50% efficient.



4B/5B

Idea: Every 4 bits of data is encoded in a 5-bit code, with the 5-bit codes selected to have no more than one leading 0 and no more than two trailing 0 (i.e., never get more than three consecutive 0s). Resulting 5-bit codes are then transmitted using the NRZI encoding. Achieves 80% efficiency.

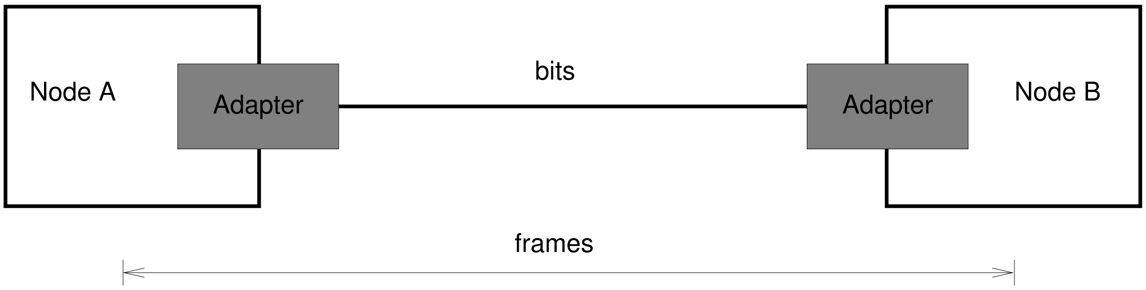
4-bit Data	5-bit Code
0000	11110
0001	01001
0010	10100
0011	10101
0100	01010
0101	01011
0110	01110
0111	01111
1000	10010
1001	10011
1010	10110
1011	10111
1100	11010
1101	11011
1110	11100
1111	11101

Framing

Overview

Problem: Breaking sequence of bits into a frame

- Must determine first and last bit of the frame
- Typically implemented by network adaptor
- Adaptor fetches (deposits) frames out of (into) host memory



Byte-Oriented Protocols

- Sentinel Approach
  - BISYNC

8	8	8		8		8	16
SYN	SYN	SOH	Header	STX	Body	ETX	CRC

  - IMP-IMP

8	8	8	8	128		8	8	16
SYN	SYN	DLE	STX	Header	Body	DLE	ETX	CRC

  - Problem: ETX character might appear in the data portion of the frame.
  - Solution: Escape the ETX character with a DLE character in BISYNC; escape the DLE character with a DLE character in IMP-IMP.
- Byte Counting Approach (DDCMP)

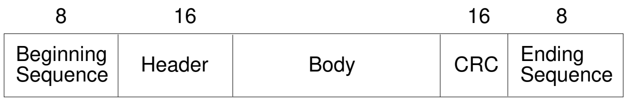
8	8	8	14	42		16
SYN	SYN	Class	Count	Header	Body	CRC

  - Problem: Count field is corrupted (framing error).
  - Solution: Catch when CRC fails.

Bit-Oriented Protocols

HDLC: High-Level Data Link Control (also SDLC and PPP)

Delineate frame with a special bit-sequence: 01111110

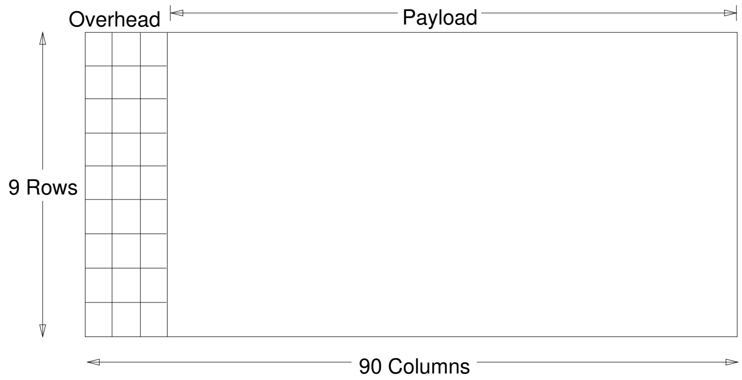


Bit Stuffing

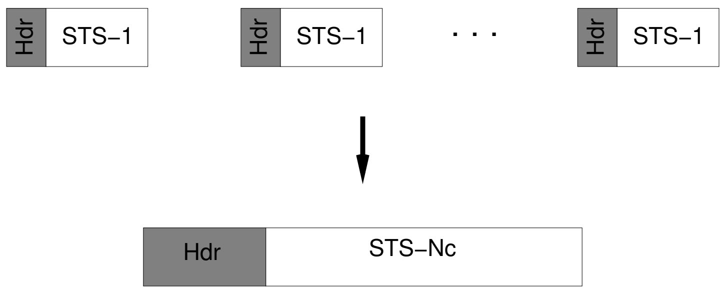
- Sender: any time five consecutive 1s have been transmitted from the body of the message, insert a 0.
- Receiver: should five consecutive 1s arrive, look at next bit(s):
  - if next bit is a 0: remove it
  - if next bits are 10: end-of-frame marker
  - if next bits are 11: error

Clock-Based Framing

- SONET: Synchronous Optical Network
- ITU standard for transmission over fiber
- STS-1 (51.84 Mbps)



- Byte-interleaved multiplexing



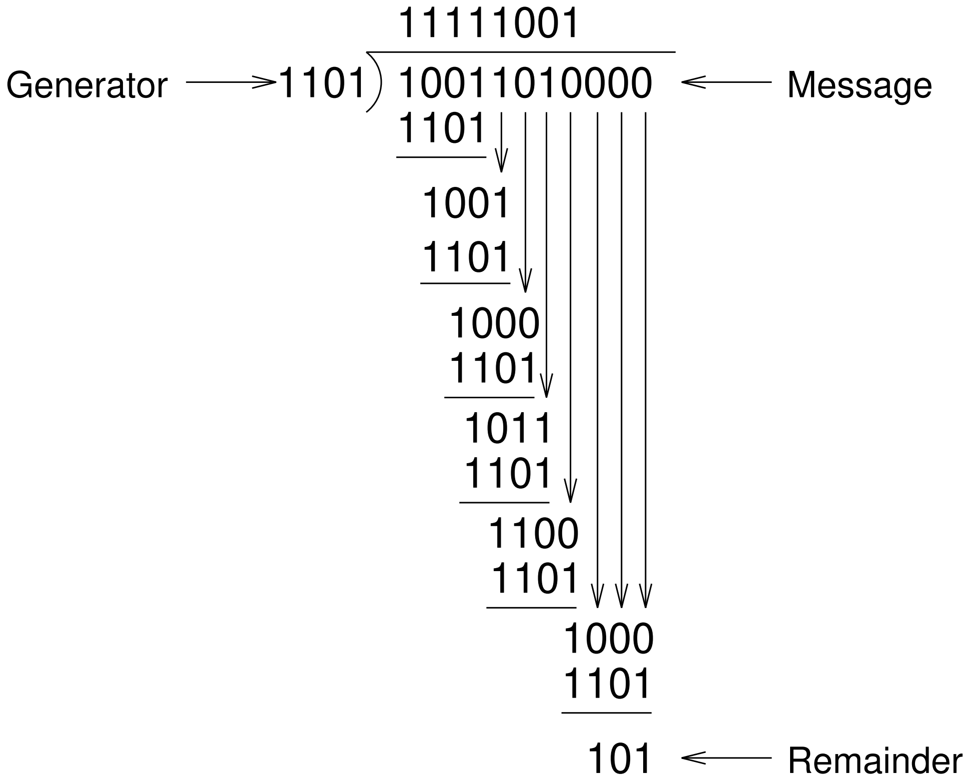
- Each frame is 125μs long.

Error Detection

Cyclic Redundancy Check

- Add k bits of redundant data to an n-bit message.
- Represent n-bit message as an $n-1$ degree polynomial; e.g., MSG=10011010 corresponds to $M(x) = x^7 + x^4 + x^3 + x^1$.
- Let k be the degree of some divisor polynomial $C(x)$; e.g., $C(x) = x^3 + x^2 + 1$.
- Transmit polynomial $P(x)$ that is evenly divisible by $C(x)$, and receive polynomial $P(x) + E(x)$; $E(x)$=0 implies no errors.
- Recipient divides $(P(x) + E(x))$ by $C(x)$; the remainder will be zero in only two cases: $E(x)$ was zero (i.e. there was no error), or $E(x)$ is exactly divisible by $C(x)$. Choose $C(x)$ to make second case extremely rare.

- Sender:
  - multiply $M(x)$ by x^k; for our example, we get $x^{10} + x^7 + x^6 + x^4$ (10011010000);
  - divide result by $C(x)$ (1101);



  - Send 10011010000 - 101 = 10011010101, since this must be exactly divisible by $C(x)$;

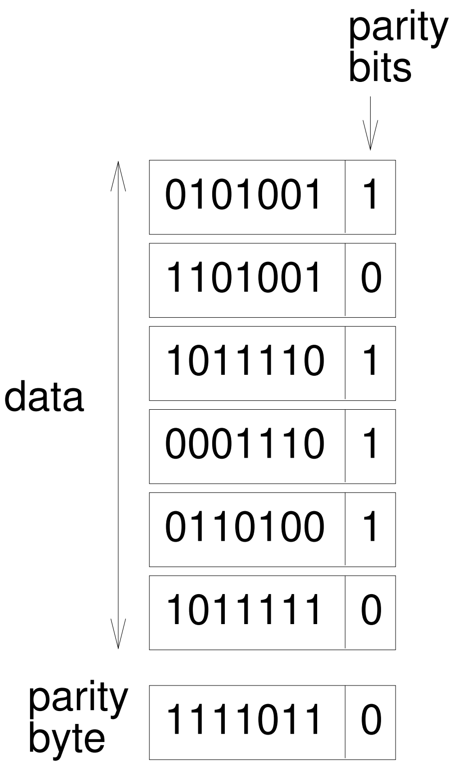
Want to ensure that $C(x)$ does not divide evenly into polynomial $E(x)$.

- All single-bit errors, as long as the x^k and x^0 terms have non-zero coefficients.
- All double-bit errors, as long as $C(x)$ has a factor with at least three terms.
- Any odd number of errors, as long as $C(x)$ contains the factor $(x+1)$.
- Any 'burst' error (i.e sequence of consecutive errored bits) for which the length of the burst is less than k bits.
- Most burst errors of larger than k bits can also be detected.

Common polynomials for $C(x)$:

CRC	$C(x)$
CRC-8	$x^8+x^2+x^1+1$
CRC-10	$x^{10}+x^9+x^5+x^4+x^1+1$
CRC-12	$x^{12}+x^{11}+x^3+x^2+1$
CRC-16	$x^{16}+x^{15}+x^2+1$
CRC-CCITT	$x^{16}+x^{12}+x^5+1$
CRC-32	$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+$
	$x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$

Two-Dimensional Parity



Internet Checksum Algorithm

Idea: view message as a sequence of 16-bit integers. Add these integers together using 16-bit ones complement arithmetic, and then take the ones complement of the result. That 16-bit number is the checksum.

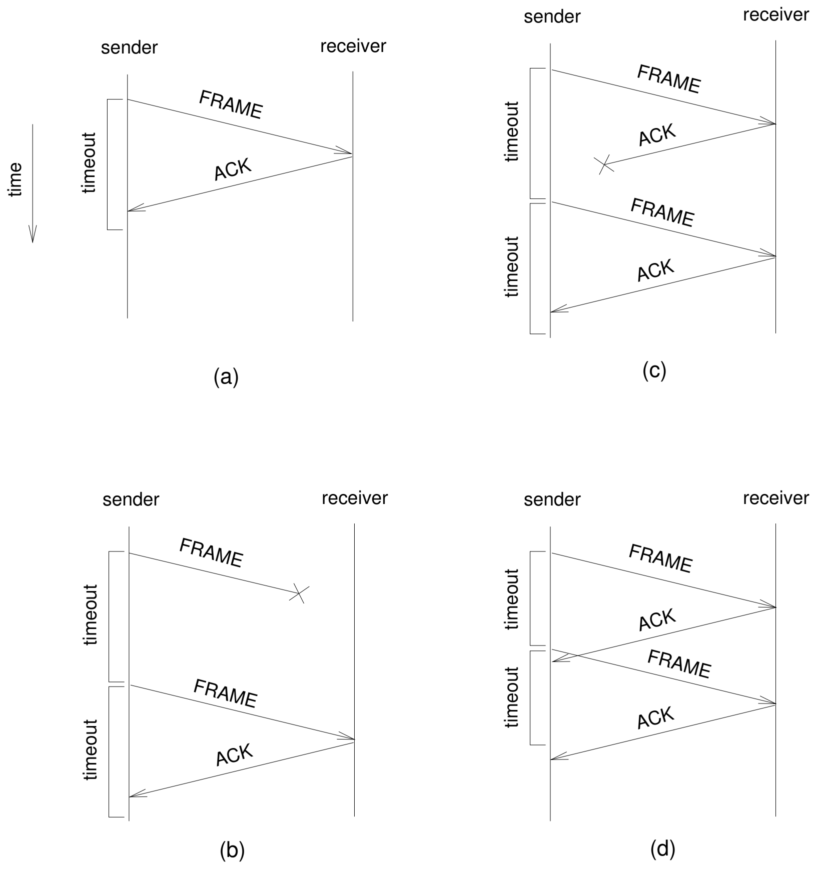
```
u_short
cksum(u_short *buf, int count)
{
    register u_long sum = 0;

    while (count--)
    {
        sum += *buf++;
        if (sum & 0xFFFF0000)
        {
            /* carry occurred, so wrap around */
            sum &= 0xFFFF;
            sum++;
        }
    }
    return ~(sum & 0xFFFF);
}
```

Reliable Transmission

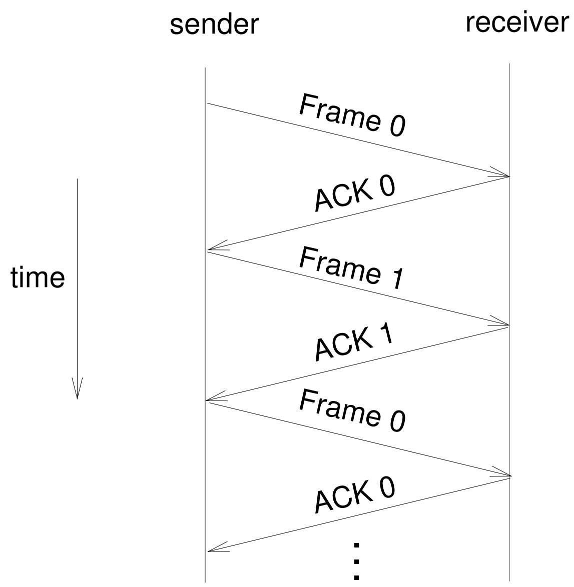
Overview

Recover from Corrupt Frames

- Error Correction Codes (ECC); also called Forward Error Correction (FEC)
- Acknowledgements and Timeouts; also called Automatic Repeat reQuest (ARQ)



Stop-and-Wait

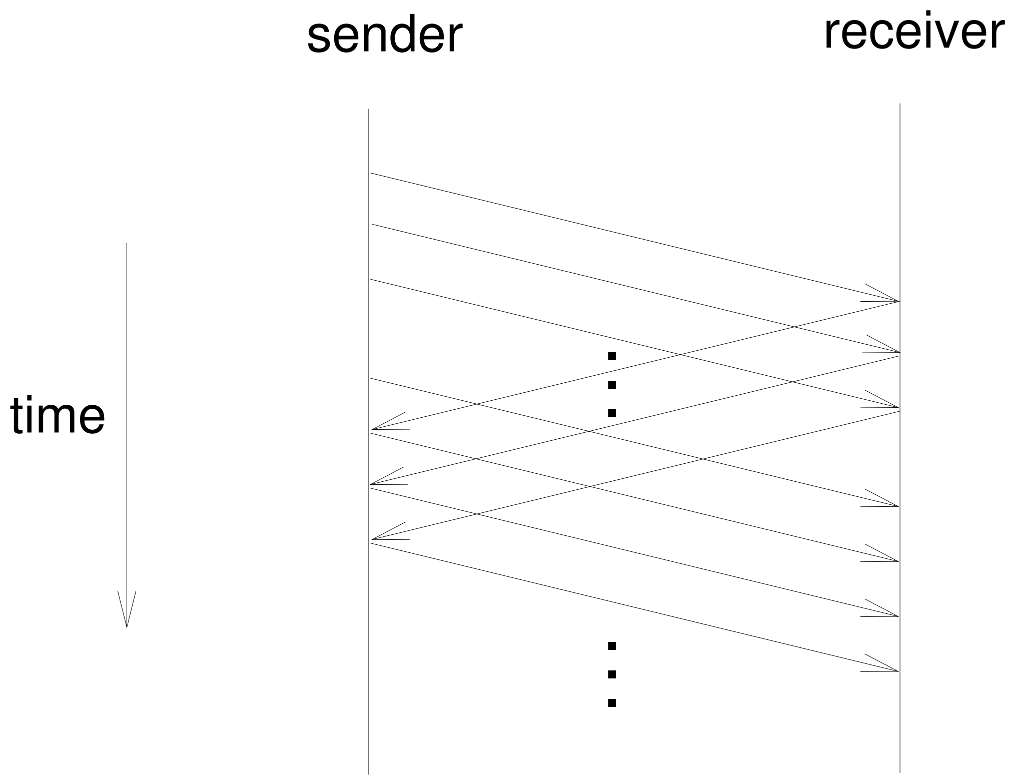


Problem: Keeping the pipe full.

Example: 1.5Mbps link $\times$ 45ms RTT = 67.5Kb (8KB). Assuming frame size of 1KB, stop-and-wait uses about one-eighth of the link's capacity. Want the sender to be able to transmit up to 8 frames before having to wait for an ACK.

Sliding Window

Idea: Allow sender to transmit multiple frames before receiving an ACK, thereby keeping the pipe full. There is an upper limit on the number of outstanding (un-ACKed) frames allowed.



Sender:

- Assign sequence number to each frame (`SeqNum`)
- Maintain three state variables:
  - send window size (`SWS`)
  - last acknowledgment received (`LAR`)
  - last frame sent (`LFS`)
- Maintain invariant: `LFS - LAR + 1` $\leq$ `SWS`

- When ACK arrives, advance `LAR`, thereby opening window
- Buffer up to `SWS` frames

Receiver:

- Maintain three state variables:
  - receive window size (`RWS`)
  - last frame acceptable (`LFA`)
  - next frame expected (`NFE`)
- Maintain invariant: `LFA - NFE + 1` $\leq$ `RWS`

- Frame `SeqNum` arrives:
  - if `NFE` $\leq$ `SeqNum` $\leq$ `LFA` $\longrightarrow$ accept
  - if `SeqNum` $<$ `NFE` or `SeqNum` $>$ `LFA` $\longrightarrow$ discarded
- Send cumulative ACK
- Variations
  - selective acknowledgements
  - negative acknowledgements (NAK)

Sequence Number Space

- `SeqNum` field is finite; sequence numbers wrap around
- Sequence number space must be larger than number of outstanding frames
- `SWS` $\leq$ `MaxSeqNum-1` is not sufficient
  - suppose 3-bit `SeqNum` field (`0..7`)
  - `SWS=RWS=7`
  - sender transmit frames `0..6`
  - arrive successfully, but ACKs lost
  - sender retransmits `0..6`
  - receiver expecting `7,0..5`, but receives second incarnation of `0..5`
- `SWS` $<$ `(MaxSeqNum+1)/2` is correct rule
- Intuitively, `SeqNum` "slides" between two halves of sequence number space

Concurrent Logical Channels

- Multiplex several logical channels over a single point-to-point link; run stop-and-wait on each logical channel.
- Maintain three bits of state for each channel:
  - boolean saying whether the channel is currently busy
  - sequence number for frames sent on this logical channel
  - next sequence number to expect on this logical channel
- ARPANET supported eight logical channels over each ground link (16 over each satellite link).
- Header for each frame included a 3-bit channel number and a 1-bit sequence number, for a total of 4 bits; same number of bits as the sliding window protocol requires to support up to eight outstanding frames on the link.
- Separates reliability from *flow control* and *frame order*.

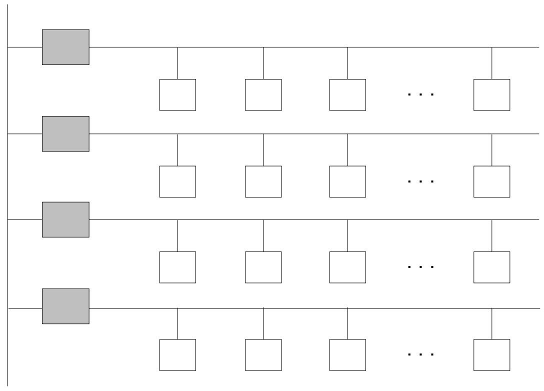
Ethernet

Overview

- History
  - Developed by Xerox PARC in mid-1970s
  - Roots in Aloha packet-radio network
  - Standardized by Xerox, DEC, and Intel in 1978
  - Similar to IEEE 802.3 standard
- CSMA/CD
  - carrier sense
  - multiple access
  - collision detection
- Bandwidth: 10Mbps and 100Mbps
- Problem: Distributed algorithm that provides fair access to a shared medium

Physical Properties

- Classical Ethernet (thick-net)
  - maximum segment of 500m
  - transceiver taps at least 2.5m apart
  - connect multiple segments with repeaters
  - no more than 2 repeaters between any pair of nodes (1500m total)
  - maximum of 1024 hosts
  - also called 10Base5



- Alternative technologies
  - 10Base2 (thin-net): 200m; daisy-chain configuration
  - 10BaseT (twisted-pair): 100m; star configuration

Frame Format

64	48	48	16		32	8
Preamble	Dest Addr	Src Addr	Type	Body	CRC	Postamble

Addresses:

- Unique, 48-bit unicast address assigned to each adaptor
- Example: `8:0:2b:e4:b1:2`
- Broadcast: all `1`s
- Multicast: first bit is `1`

Adaptor receives all frames; it accepts (passes to host):

- Frames addressed to its own unicast address
- Frames addressed to the broadcast address
- Frames addressed to any multicast address it has been programmed to accept
- All frames when in promiscuous mode

Transmitter Algorithm

If line is idle:

- Send immediately
- Upper bound message size of 1500 bytes
- Must wait $51\mu s$ between back-to-back frames

If line is busy:

- Wait until idle and transmit immediately
- Called *1-persistent* (special case of *p-persistent*)

If collision:

- jam for 512 bits, then stop transmitting frame
- minimum frame is 64 bytes (header + 46 bytes of data)
- delay and try again
  - 1st time: uniformly distributed between 0 and $51.2\mu s$
  - 2nd time: uniformly distributed between 0 and $102.4\mu s$
  - 3rd time: uniformly distributed between 0 and $204.8\mu s$
  - give up after several tries (usually 16)
  - exponential backoff

Experiences

Observe in Practice

- 10-200 hosts (not 1024)
- Length shorter than 1500m (RTT closer to 5μ than 51μ)
- Packet length is bimodal
- High-level flow control and host performance limit load

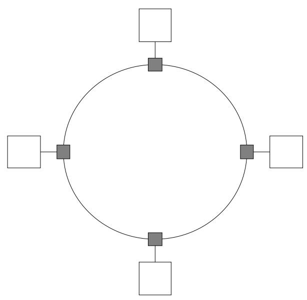
Recommendations

- Do not overload (30% utilization is about max)
- Implement controllers correctly
- Use large packets
- Get the rest of the system right (broadcast, retransmission)

FDDI

Overview

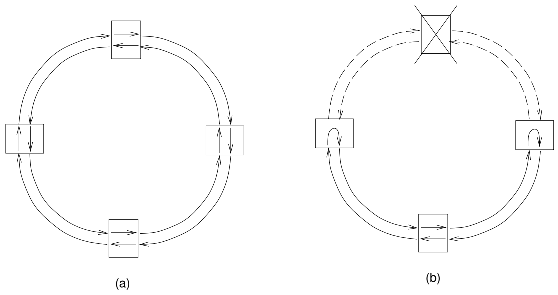
- Token Ring Networks
  - PRONET: 10Mbps and 80 Mbps rings
  - IBM: 4Mbps token ring
  - 16Mbps IEEE 802.5/token ring
  - 100Mbps Fiber Distributed Data Interface (FDDI)



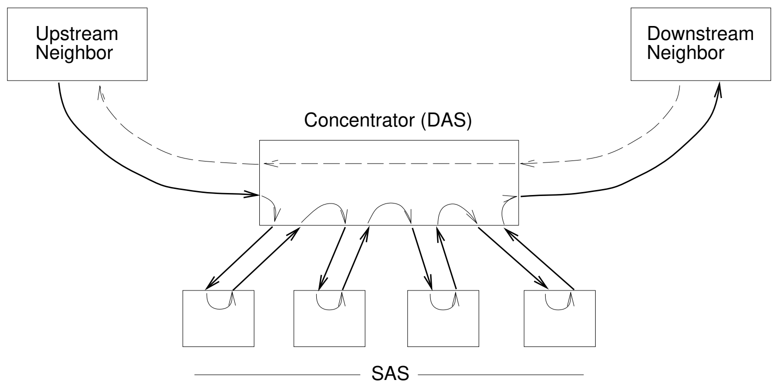
- Basic Idea
  - frames flow in one direction: upstream to downstream
  - special bit pattern (token) rotates around ring
  - must capture token before transmitting
  - release token after done transmitting
    - immediate release
    - delayed release
  - remove your frame when it comes back around
  - stations get round-robin service

Physical Properties of FDDI

Dual Ring Configuration



Single and Dual Attachment Stations



- Each station imposes a delay (e.g., 50ns)
- Maximum of 500 stations
- Upper limit of 100km (200km of fiber)
- Uses 4B/5B encoding
- Can be implemented over copper (CDDI)

Timed Token Algorithm

- Token Holding Time (THT): upper limit on how long a station can hold the token.
- Token Rotation Time (TRT): how long it takes the token to traverse the ring.

  $$\text{TRT} \leq \text{ActiveNodes} \times \text{THT} + \text{RingLatency}$$

- Target Token Rotation Time (TTRT): agreed-upon upper bound on TRT.
- Algorithm
  - each node measures TRT between successive arrivals of the token
  - if measured TRT > TTRT, then token is late so don't send data
  - if measured TRT < TTRT, then token is early so OK to send data
  - define two classes of traffic
    - synchronous data: can always send
    - asynchronous data: can send only if token is early
  - worse case: 2×TTRT between seeing token
  - not possible to have back-to-back rotations that take 2×TTRT time

Token Maintenance

- Lost Token
  - no token when initializing ring
  - bit error corrupts token pattern
  - node holding token crashes
- Generating a Token (and agreeing on TTRT)
  - execute when join ring or suspect a failure
  - each node sends a special *claim frame* that includes the node's *bid* for the TTRT
  - when receive claim frame, update bid and forward
  - if your claim frame makes it all the way around the ring:
    - your bid was the lowest
    - everyone knows TTRT
    - you insert new token
- Monitoring for a Valid Token
  - should see valid transmission (frame or token) periodically
  - maximum gap = ring latency + max frame $\leq$ 2.5ms
  - set timer at 2.5ms and send claim frame if it fires

Frame Format

8	8	48	48		32	8	24
Start of Frame	Control	Dest Addr	Src Addr	Body	CRC	End of Frame	Status

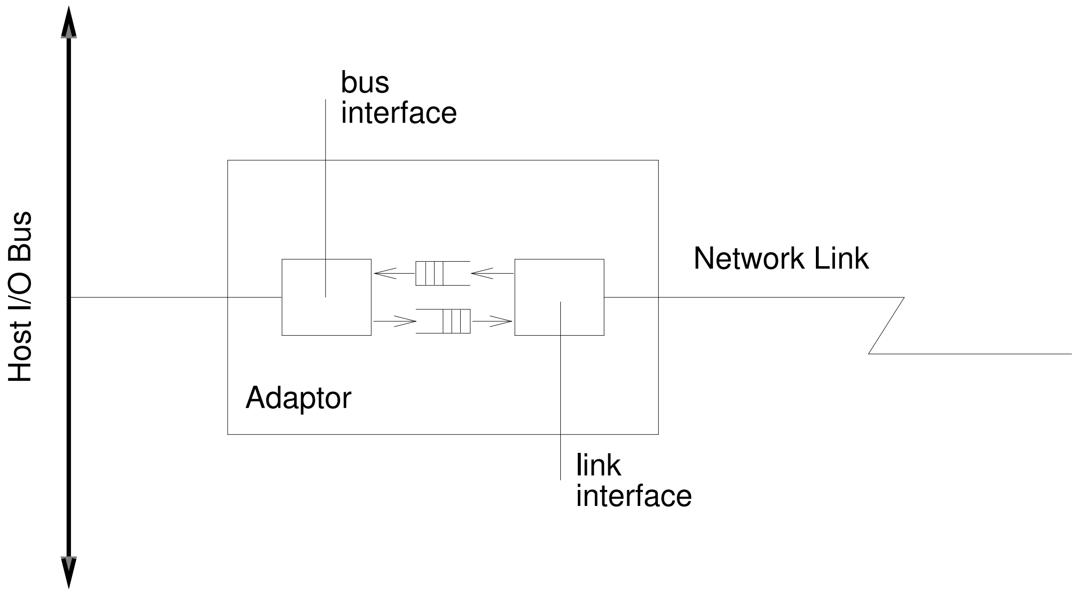
- Control Field
  - 1st bit: asynchronous (0) versus synchronous (1) data
  - 2nd bit: 16-bit (0) versus 48-bit (1) addresses
  - last 6 bits: demux key (includes reserved patterns for token and claim frame)
- Status Field
  - from receiver back to sender
  - error in frame
  - recognized address
  - accepted frame (flow control)

Network Adaptors

Overview

Typically where data link functionality is implemented

- Framing
- Error Detection
- Media Access Control (MAC)



Host Perspective

Control Status Register (CSR)

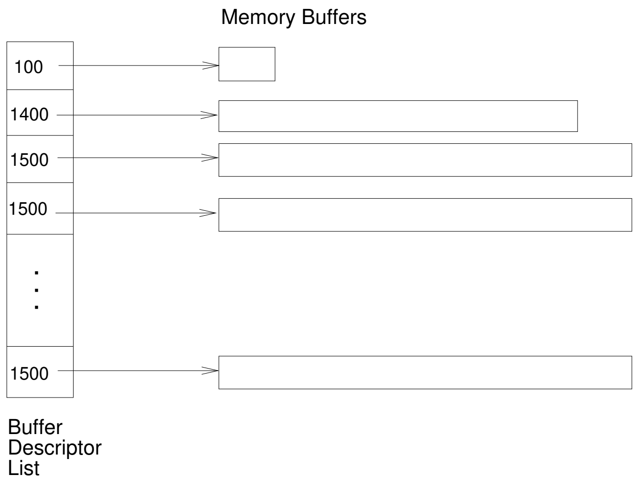
- Available at some memory address
- CPU can read and write
- CPU instructs Adaptor (e.g., transmit)
- Adaptor informs CPU (e.g., receive error)

Example

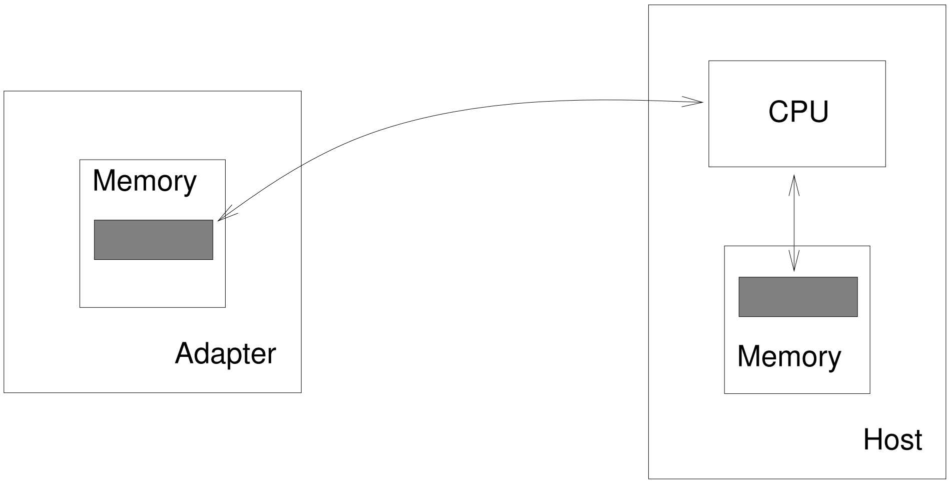
```
LE_RINT  0x0400  Received packet Interrupt (RC)
LE_TINT  0x0200  Transmitted packet Interrupt (RC)
LE_IDON  0x0100  Initialization Done (RC)
LE_IENA  0x0040  Interrupt Enable (RW)
LE_INIT  0x0001  Initialize (RW1)
```

Moving Frames Between Host and Adaptor

Direct Memory Access (DMA)



Programmed I/O (PIO)



Device Driver

Interrupt Handler

```
interrupt_handler()
{
    disable_interrupts();
    /* some error occurred */
    if (csr & LE_ERR)
    {
        print_and_clear_error();
    }
    /* transmit interrupt */
    if (csr & LE_TINT)
    {
        csr = LE_TINT | LE_INEA;
        semSignal(xmit_queue);
    }
    /* receive interrupt */
    if (csr & LE_RINT)
    {
        receive_interrupt();
    }
    enable_interrupts();
    return(0);
}
```

Transmit Routine:

```
transmit(Msg *msg)
{
    char *src, *dst;
    Context c;
    int len;

    semWait(xmit_queue);
    semWait(mutex);
    disable_interrupts();
    dst = next_xmit_buf();
    msgWalkInit(&c, msg);
    while ((src = msgWalk(&c, &len)) != 0)
        copy_data_to_lance(src, dst, len);
    msgWalkDone(&c);
    enable_interrupts();
    semSignal(mutex);
    return;
}
```

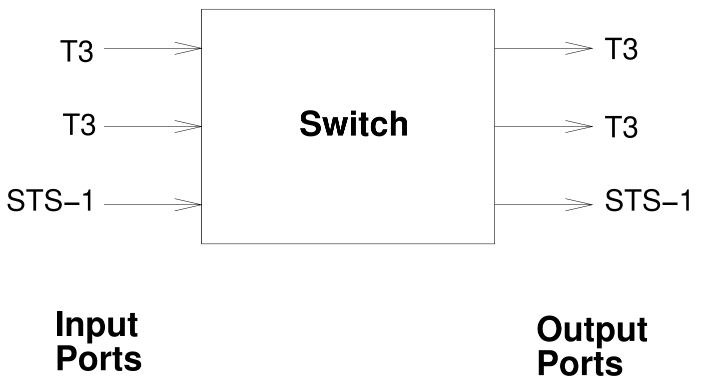
Receive Interrupt Routine

```
receive_interrupt()
{
   Msg  *msg,  *new_msg;
   char *buf;

   while (rdl = next_rcv_desc())
   {
       /* create process to handle this message */
       msg = rdl->msg;
       process_create(ethDemux, msg);

       /* msg eventually freed in ethDemux */
       /* now allocate a replacement */
       buf = msgConstructAllocate(new_msg, MTU);
       rdl->msg = new_msg;
       rdl->buf = buf;
       install_rcv_desc(rdl);
   }
   return;
}
```

Switching and Forwarding

Scalable Networks

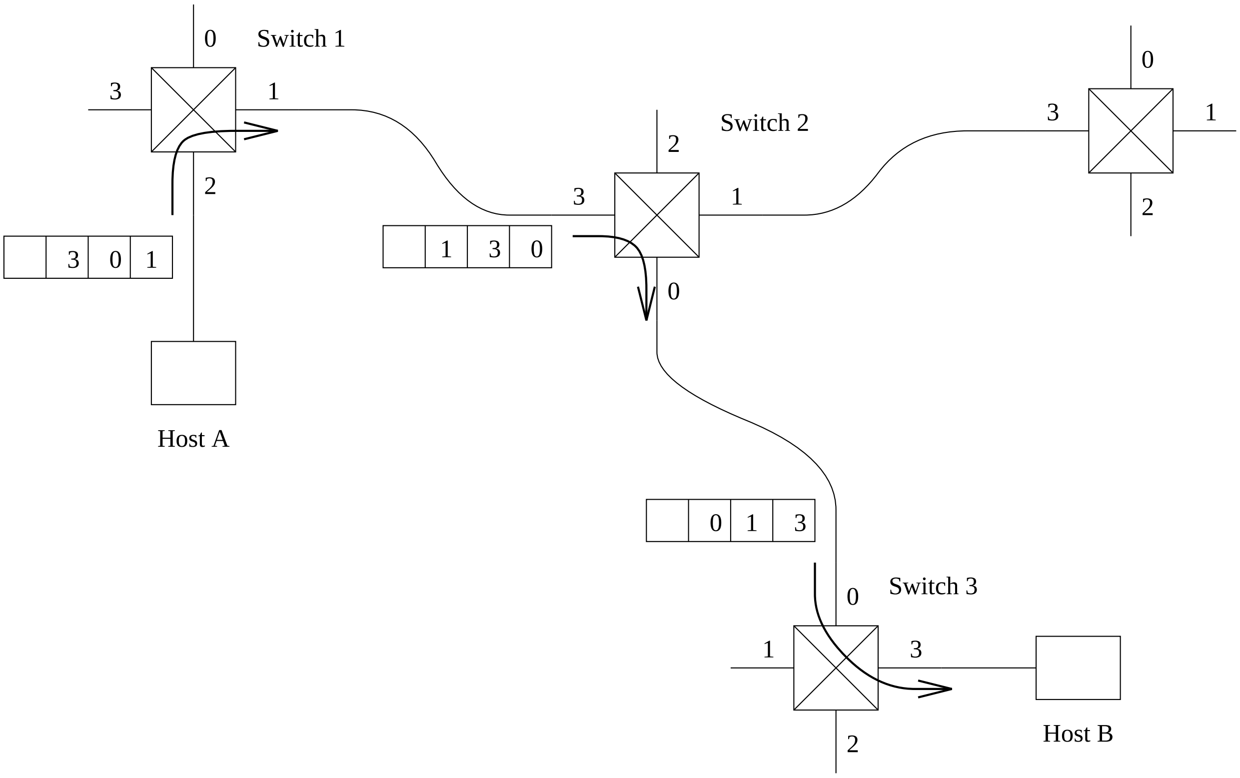
Switch: Forwards packets from input port to output port; port selected based on destination address in packet header.



- Can build networks that cover large geographic area
- Can build networks that support large numbers of hosts
- Can add new hosts without affecting performance of existing hosts

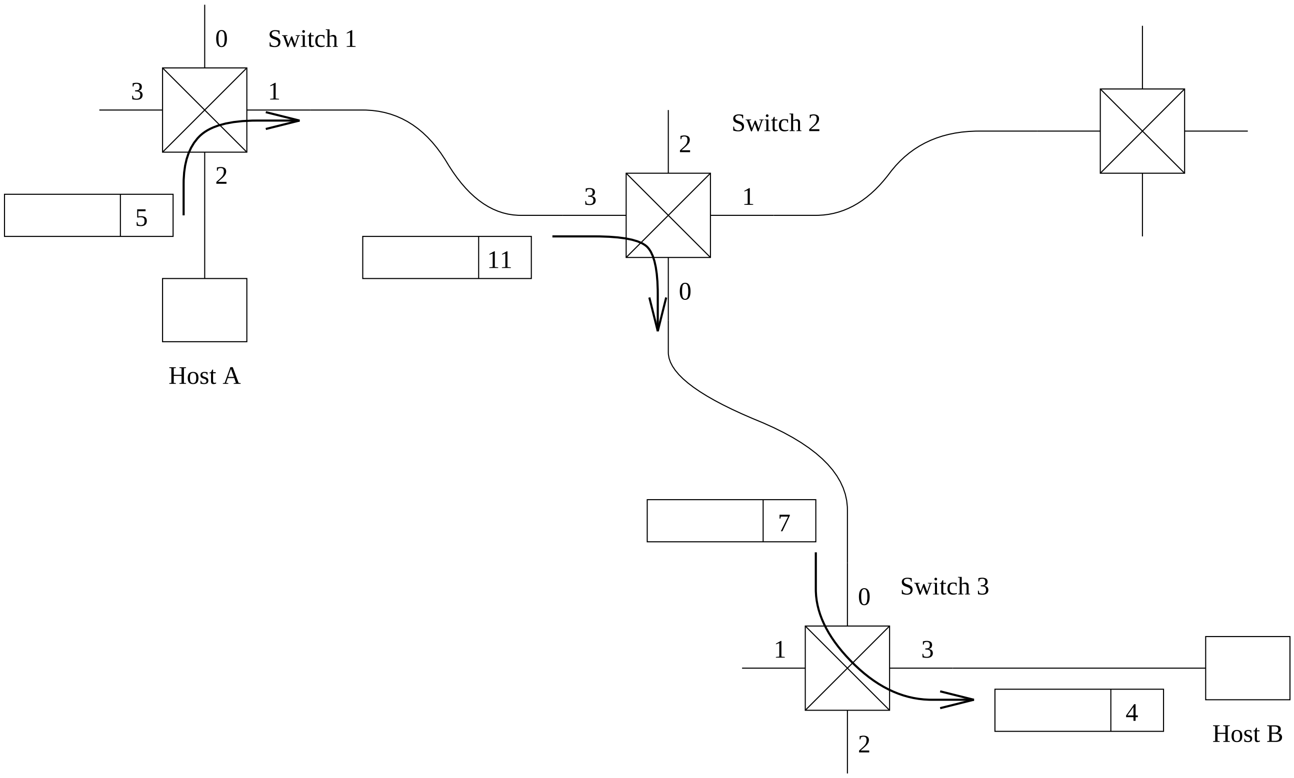
Source Routing

Address contains sequence of ports on path from source to destination.



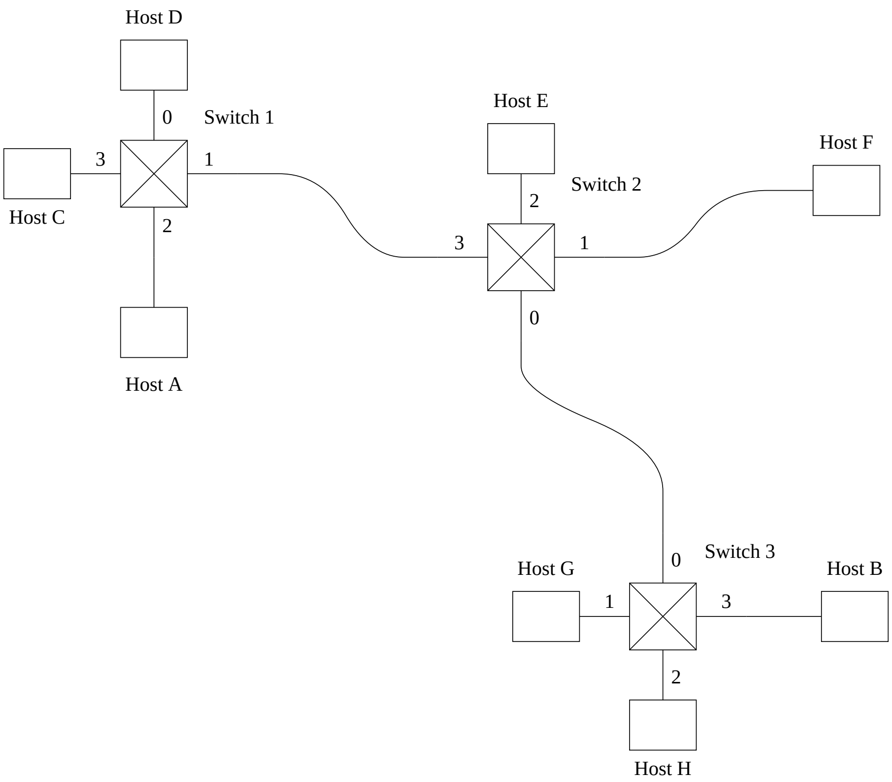
Virtual Circuit Switching

- Explicit connection setup (and tear-down) phase
- Subsequent packets follow same circuit
- Analogy: phone call
- Sometimes called *connection-oriented* model
- Each switch maintains a VC table.



Datagrams

- No connection setup phase
- Each packet forwarded independently
- Analogy: postal system
- Sometimes called *connectionless* model
- Each switch maintains a forwarding (routing) table



Virtual Circuit versus Datagram

Virtual Circuit Model:

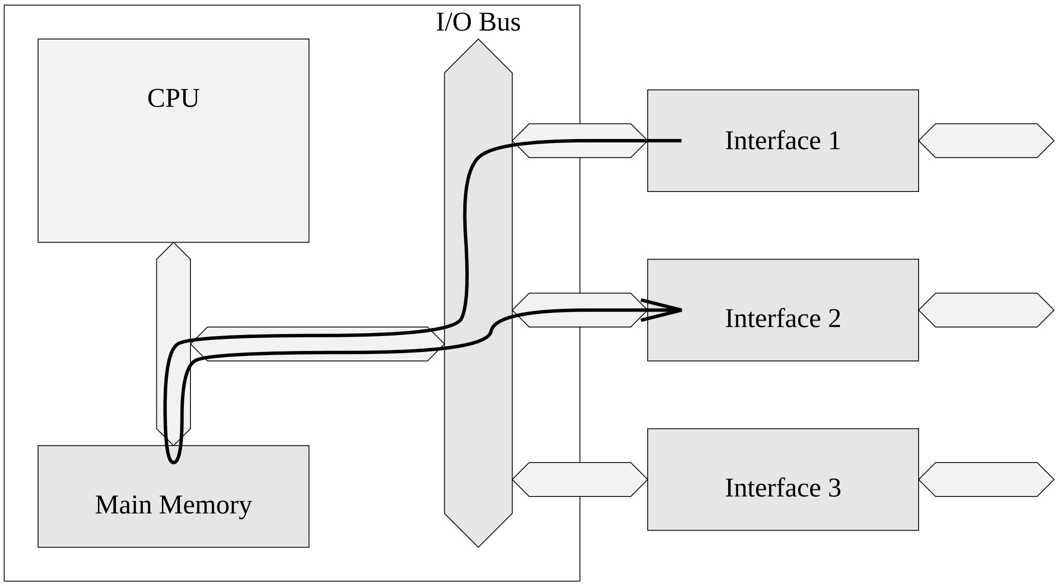
- Typically wait full RTT for connection setup before sending first data packet.
- While the connection request contains the full address for destination, each data packet contains only a small identifier, making the per-packet header overhead small.
- If a switch or a link in a connection fails, the connection is broken and a new one needs to be established.
- Connection setup provides an opportunity to reserve resources.

Datagram Model:

- There is no round trip time delay waiting for connection setup; a host can send data as soon as it is ready.
- Source host has no way of knowing if the network is capable of delivering a packet or if the destination host is even up.
- Since packets are treated independently, it is possible to route around link and node failures.
- Since every packet must carry the full address of the destination, the overhead per packet is higher than for the connection-oriented model.

Performance

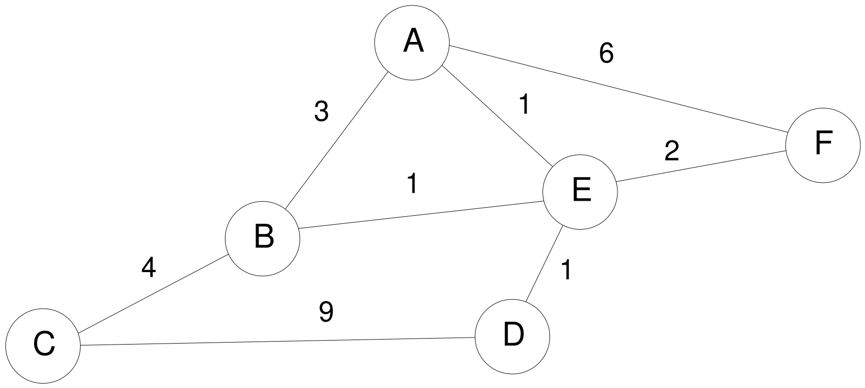
Switches can be built from a general-purpose workstations; will consider special-purpose hardware later.



- Aggregate bandwidth
  - 1/2 of the I/O bus bandwidth
  - capacity is shared among all hosts connected to switch
  - example: 800Mbps bus can support 8 T3 ports
- Packets-per-second
  - must be able to switch small packets
  - 15,000 packets-per-second is an achievable number
  - example: 64-byte packets implies 7.69Mbps

Routing

- Forwarding versus Routing
  - forwarding: to select an output port based on destination address and routing table
  - routing: process by which routing table is built
- Network as a Graph



- Problem: Find the lowest cost path between any two nodes
- Factors:
  - Static: topology
  - Dynamic: load

Distance Vector

- Each node maintains a set of triples:

  `(Destination, Cost, NextHop)`

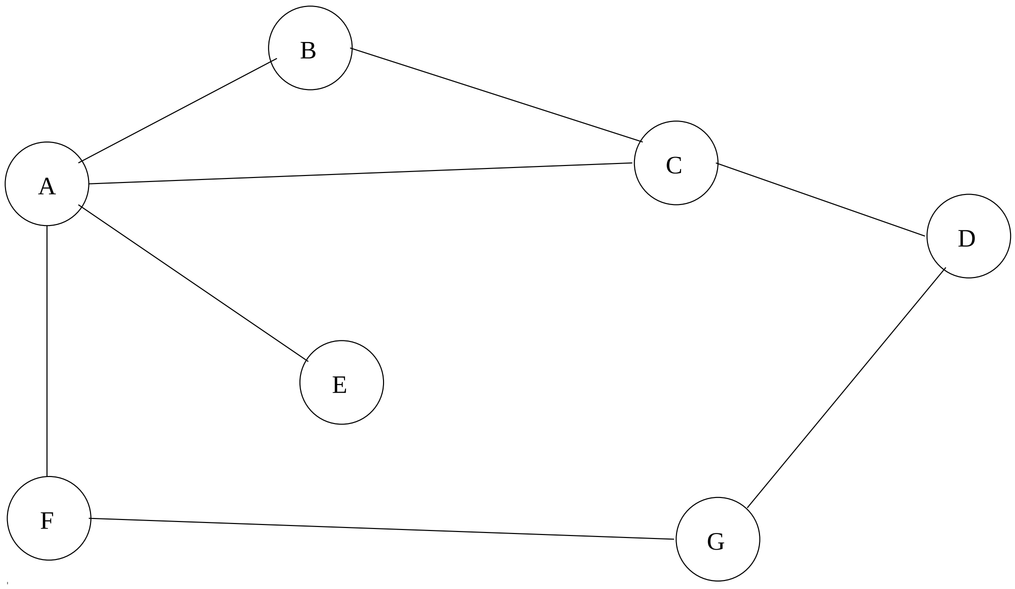
- Each node sends updates to (and receives updates from) its directly connected neighbors
  - periodically (on the order of several seconds)
  - whenever its table changes (called *triggered* update)
- Each update is a list of pairs:

  `(Destination, Cost)`

- Update local table if receive a "better" route
  - smaller cost
  - came from next-hop
- Refresh existing routes; delete if they time out

```
void
mergeRoute (Route *new)
{
   int i;

   for (i = 0; i < numRoutes; ++i)
   {
      if (new->Dest == rt[i].Dest)
      {
         if (new->Cost + 1 < rt[i].Cost)
            break;
         else if (new->NextHop == rt[i].NextHop)
            break;
         else
            return;
      }
   rt[i] = *new;
   rt[i].TTL = MAX_TTL;
   ++rt[i].Cost;
   if (i == numRoutes)
      ++numRoutes;
}
```

Example



Routing table at node B

Destination	Cost	NextHop
A	1	A
C	1	C
D	2	C
E	2	A
F	2	A
G	3	A

Routing Loops

- Example 1
  - F detects that link to G has failed
  - F sets distance to G to infinity and sends update to A
  - A sets distance to G to infinity since it uses F to reach G
  - A receives periodic update from C with 2-hop path to G
  - A sets distance to G to 3 and sends update to F
  - F decides it can reach G in 4 hops via A
- Example 2
  - Link from A to E fails
  - A advertises distance of infinity to E
  - B and C advertise a distance of 2 to E
  - B decides it can reach E in 3 hops; advertises this to A
  - A decides it can reach E in 4 hops; advertises this to C
  - C decides that it can reach E in 5 hops......
- Heuristics to break routing loops
  - set infinity to 16
  - split horizon
  - split horizon with poison reverse

Link State

Strategy: Send to all nodes (not just neighbors) information about directly connected links (not entire routing table).

- Link State Packet (LSP)
  - id of the node that created the LSP
  - cost of link to each directly connected neighbor
  - sequence number (SEQNO)
  - time-to-live (TTL) for this packet
- Reliable Flooding
  - store most recent LSP from each node
  - forward LSP to all nodes but one that sent it
  - generate new LSP periodically; increment SEQNO
  - start SEQNO at 0 when reboot
  - decrement TTL of each stored LSP; discard when TTL=0

Route Calculation (in theory)

- Dijkstra's shortest path algorithm
- N denotes set of nodes in the graph
- $l(i, j)$ denotes non-negative cost (weight) for edge (i, j)
- $s \in N$ denotes this node
- M denotes the set of nodes incorporated so far
- $C(n)$ denotes cost of the path from s to node n

$M = \{s\}$
`for each` n `in` $N - \{s\}$
    $C(n) = l(s, n)$
`while` $(N \neq M)$
    $M = M$ `union` $\{w\}$ `such that` $C(w)$
    `is the minimum for all` w `in` $(N - M)$
    `for each` n `in` $(N - M)$
        $C(n) = \texttt{MIN}(C(n), C(w) + l(w, n))$

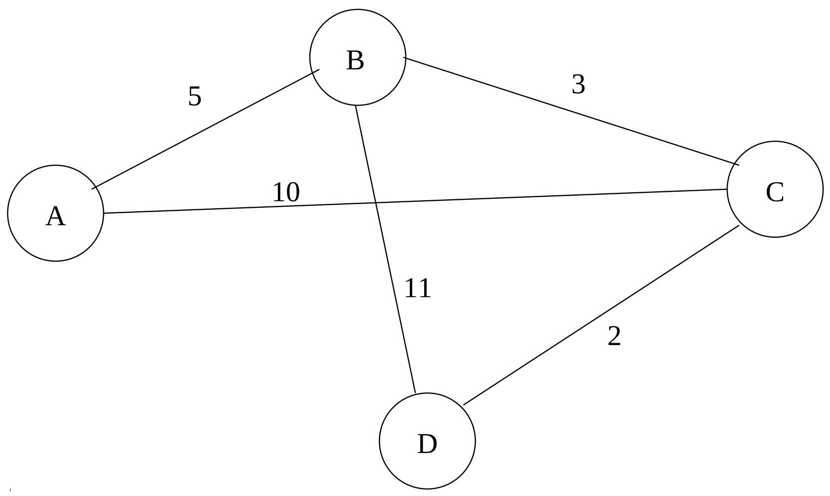
Route Calculation (in practice)

- Forward search algorithm
- Each switch maintains two lists:

  `Tentative` and `Confirmed`

- Each list contains a set of triples:

  `(Destination, Cost, NextHop)`



1. Initialized `Confirmed` with entry for me; cost = 0.

2. For the node just added to `Confirmed` (call it `Next`) select its LSP.

3. For each `Neighbor` of `Next`, calculate the `Cost` to reach this `Neighbor` as the sum of the cost from me to `Next` and from `Next` to `Neighbor`.

    3.1. If `Neighbor` is currently in neither `Confirmed` or `Tentative`, add `(Neighbor, Cost, NextHop)` to `Tentative`, where `NextHop` is the direction to reach `Next`.

    3.2. If `Neighbor` is currently in `Tentative` and `Cost` is less that current cost for `Neighbor`, then replace current entry with `(Neighbor, Cost, NextHop)`, where `NextHop` is the direction to reach `Next`.

4. If `Tentative` is empty, stop. Otherwise, pick entry from `Tentative` with the lowest cost, move it to `Confirmed`, and return to step 2.

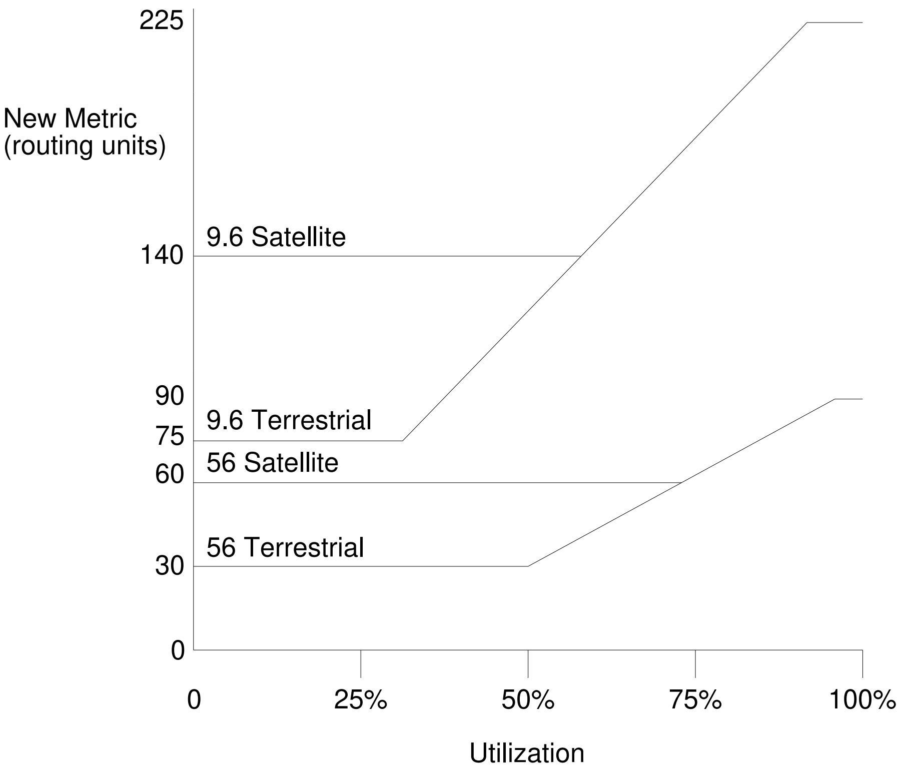
Step	**Confirmed**	**Tentative**
1.	(D,0,-)	
2.	(D,0,-)	(B,11,B) (C,2,C)
3.	(D,0,-) (C,2,C)	(B,11,B)
4.	(D,0,-) (C,2,C)	(B,5,C) (A,12,C)
5.	(D,0,-) (C,2,C) (B,5,C)	(A,12,C)
6.	(D,0,-) (C,2,C) (B,5,C)	(A,10,C)
7.	(D,0,-) (C,2,C) (B,5,C) (A,10,C)	

Metrics

- Original ARPANET metric
  - measured number of packets enqueued on each link
  - took neither latency or bandwidth into consideration
- New ARPANET metric
  - stamp each incoming packet with its arrival time (`AT`)
  - record departure time (`DT`)
  - when link-level ACK arrives, compute

    `Delay = (DT - AT) + Transmit + Latency`
  - if timeout, reset `DT` to departure time for retransmission
  - link cost = average delay over some time period

- Problems with "New" metric
  - under low load, static factors dominated cost; worked OK
  - under high load, congested links had very hight costs; packets oscillated between congested and idle links
  - range of costs too large; prefered path of 126 lightly loaded 56Kbps links to a 1-hop 9.6Kbps path
- Revised ARPANET metric
  - replaced delay measurement with link utilization
  - compressed dynamic range
    * highly loaded link never has a cost more than 3 times its idle cost
    * most expensive link only 7 times the cost of the least expensive
    * high-speed satellite link more attractive than low-speed terrestrial link
    * cost is a function of link utilization only at moderate to high loads.


225
New Metric
(routing units)
9.6 Satellite
140
90
9.6 Terrestrial
75
56 Satellite
60
56 Terrestrial
30
0
0
25%
50%
75%
100%
Utilization

Asynchronous Transfer Mode (ATM)

Overview

- Connection-oriented packet-switched network
- Used in both WAN and LAN settings
- Signalling (connection setup) Protocol: Q.2931
- Specified by ATM Forum
- Packets are called *cells*: 5-byte header + 48-byte payload
- Commonly transmitted over SONET (but not necessarily)

Cells

- Variable versus Fixed-Length
  - no optimal fixed-length
    - if small: high header-to-data overhead
    - if large: low utilization for small messages
  - fixed-length are easier to switch in hardware
    - simpler
    - enables parallelism
- Small size improves queue behavior
  - finer-grained pre-emption point for scheduling link
    - maximum packet = 4KB
    - link speed = 100Mbps
    - transmission time = $4096 \times 8/100 = 327.68\mu s$
    - high priority packet may sit in the queue $327.68\mu s$
    - in contrast, $53 \times 8/100 = 4.24\mu s$ for ATM
  - near cut-through behavior
    - two 4KB packets arrive at same time
    - link idle for $327.68\mu s$ while both arrive
    - at end of $327.68\mu s$, still have 8KB to transmit
    - in contrast, can transmit first cell after $4.24\mu s$
    - at end of $327.68\mu s$, just over 4KB left in queue

- Carrying Voice in Cells
  - voice digitally encoded at 64Kbps (8-bit samples at 8KHz)
  - need full cell's worth of samples before sending cell
  - example: 1000-byte cells implies 125ms per cell (too long)
  - smaller latency implies no need for echo cancellors
- Settled on compromise of 48 bytes: (32+64)/2

Cell Format

- User-Network Interface (UNI)

4	8	16	3	1	8	384 (48 bytes)
GFC	VPI	VCI	Type	CLP	HEC(CRC-8)	Payload

  – host-to-switch format
  – GFC: Generic Flow Control (still being defined)
  – VCI: Virtual Circuit Identifier
  – VPI: Virtual Path Identifier
  – Type: management, congestion control, AAL5 (later)
  – CLP: Cell Loss Priority
  – HEC: Header Error Check (CRC-8)

- Network-Network Interface (NNI)

  – switch-to-switch format
  – GFC becomes part of VPI field

Segmentation and Reassembly

ATM Adaptation Layer (AAL)

- AAL 1 and 2 designed for applications that need guaranteed rate (e.g., voice, video)
- AAL 3/4 designed for packet data
- AAL 5 is an alternative standard for packet data

AAL 3/4

- Convergence Sublayer Protocol Data Unit (CS-PDU)

8	8	16	<64kbytes	8	0-24	8	16
CPI	Btag	BASize	User Data	Pad	0	Etag	Len

  – CPI: common part indicator (version field)
  – Btag/Etag: beginning and ending tag
  – BAsize: hint on amount of buffer space to allocate
  – Length: size of whole PDU

- Cell Format

40	2	4	10	352 (44 bytes)	6	10
ATM header	Type	SEQ	MID	Payload	Length	CRC-10

  – Type
    * BOM: beginning of message
    * COM: continuation of message
    * EOM: end of message
  – SEQ: sequence number
  – MID: message id
  – Length: number of bytes of PDU in this cell

AAL5

- CS-PDU Format

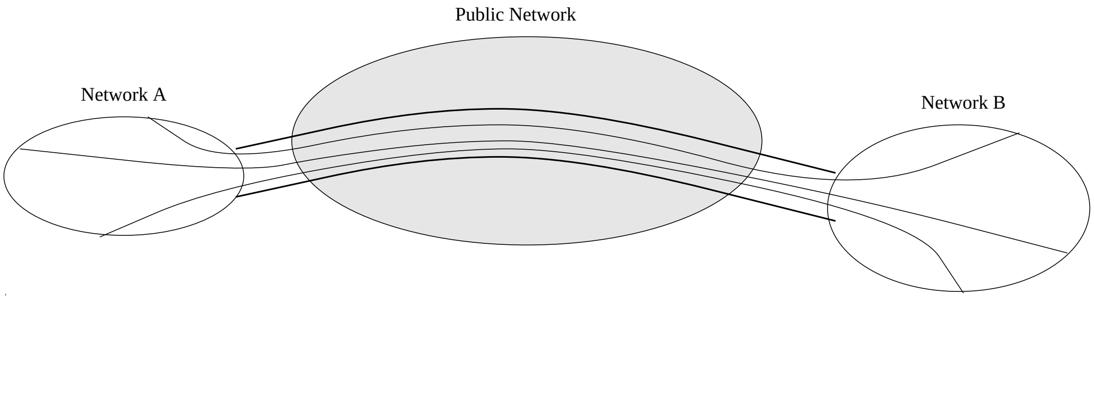
<64kB	0-47 bytes	16	16	32
Data	Pad	Reserved	Len	CRC-32

  – pad so trailer always falls at end of ATM cell
  – Length: size of PDU (data only)
  – CRC-32 (detects missing or misordered cells)

- Cell Format
  – end-of-PDU bit in Type field of ATM header

VPI/VCI

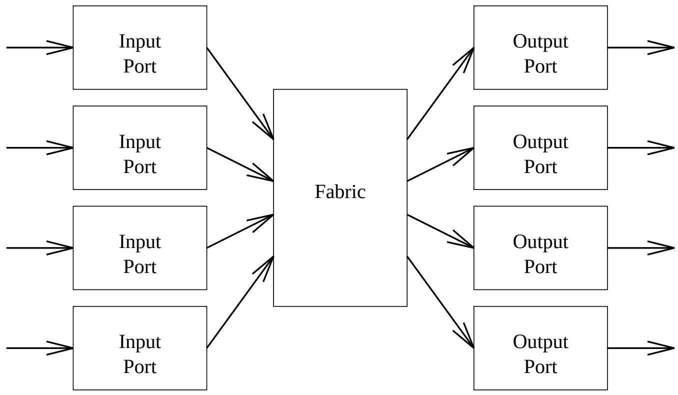
- Host: treat as 24-bit circuit identifier
  - if cheap: one-per application; use for demultiplexing
  - if expensive: multiplex several applications onto one VCI
- Network: aggregate multiple circuits into one path



Switching Hardware

Overview

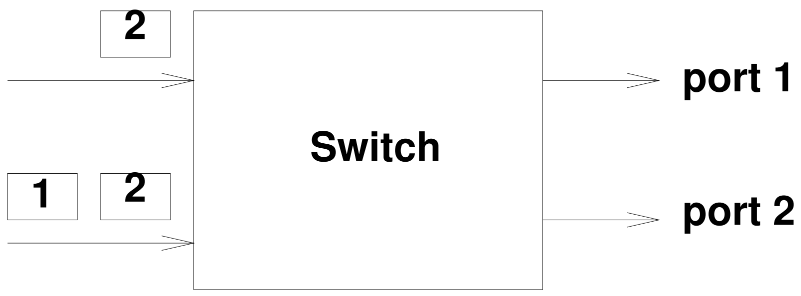
- Terminology: $n \times m$ switch has n inputs and m outputs
- Design Goals
  - throughput (depends on traffic model)
  - scalability (a function of n)
- Ports and Fabrics



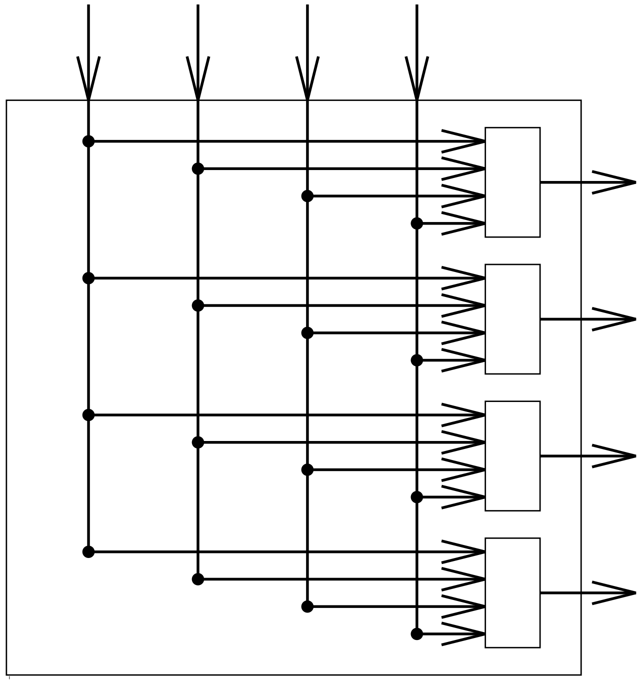
  - ports
    - circuit management (e.g., map VCIs, route datagrams)
    - buffering (input and/or output)
  - fabric
    - as simple as possible
    - sometimes do buffering (internal)

Buffering

- Wherever contention is possible
  - input port (contend for fabric)
  - internal (contend for output port)
  - output port (contend for link)
- Head-of-Line Blocking
  - input buffering

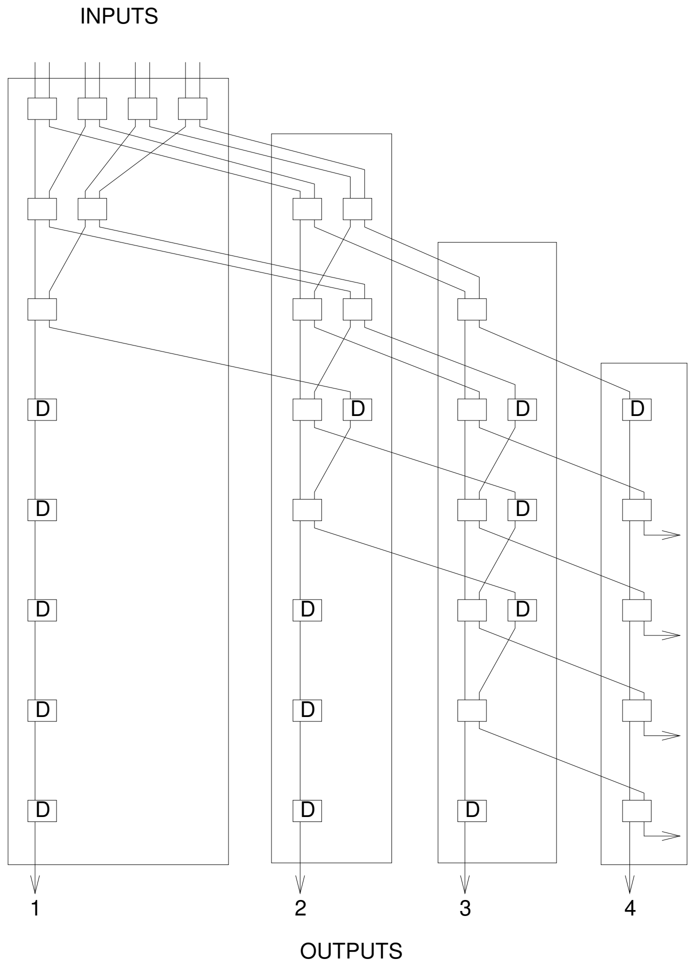


Crossbar Switches



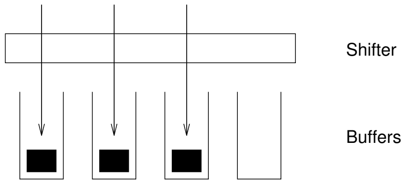
Knockout Switch

- Example of Crossbar
- Concentrator: select l of n packets

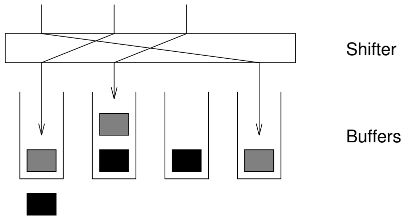


- Complexity: n^2

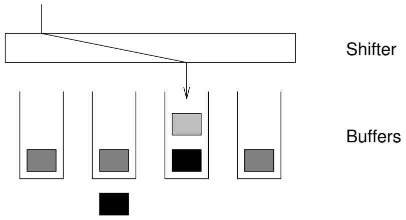
- Output Buffer



(a) Three packets arrive



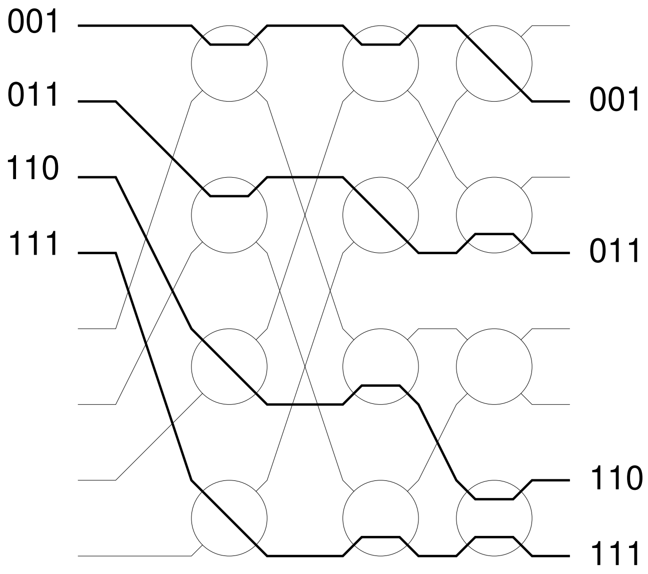
(b) Three packets arrive, one leaves



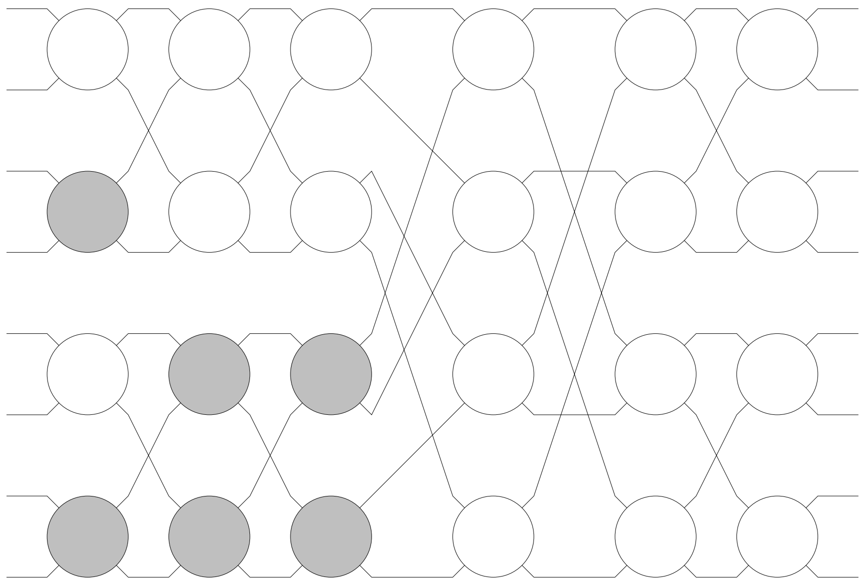
(c) One packets arrives, one leaves

Self-Routing Fabrics

- Banyan Network
  - constructed from simple 2×2 switching elements
  - self-routing header attached to each packet
  - elements arranged to route based on this header
  - no collisions if input packets sorted into ascending order
  - complexity: $n \log_2 n$

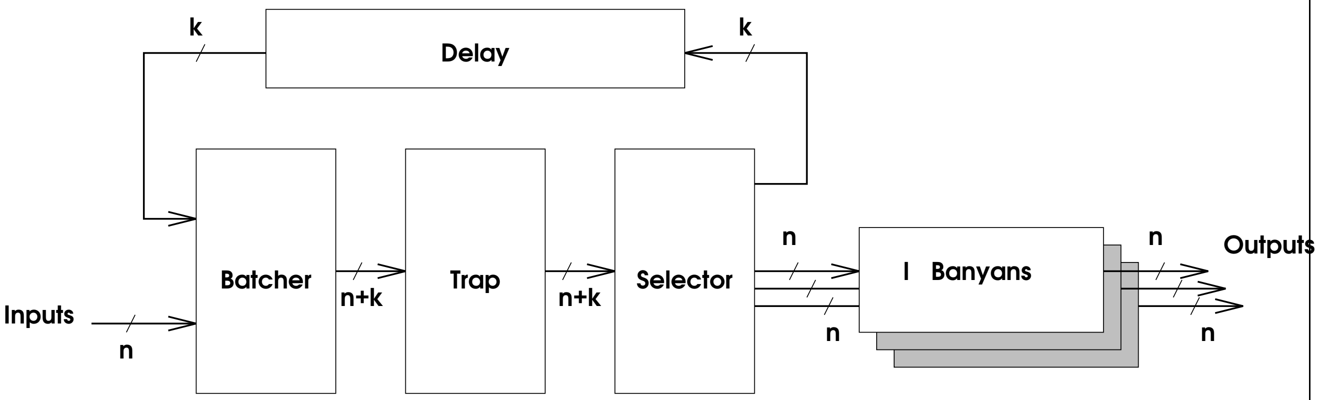


- Batcher Network
  - switching elements sort two numbers
    - ∗ some elements sort into ascending (clear)
    - ∗ some elements sort into descending (shaded)
  - elements arranged to implement merge sort
  - complexity: $n \log_2^2 n$

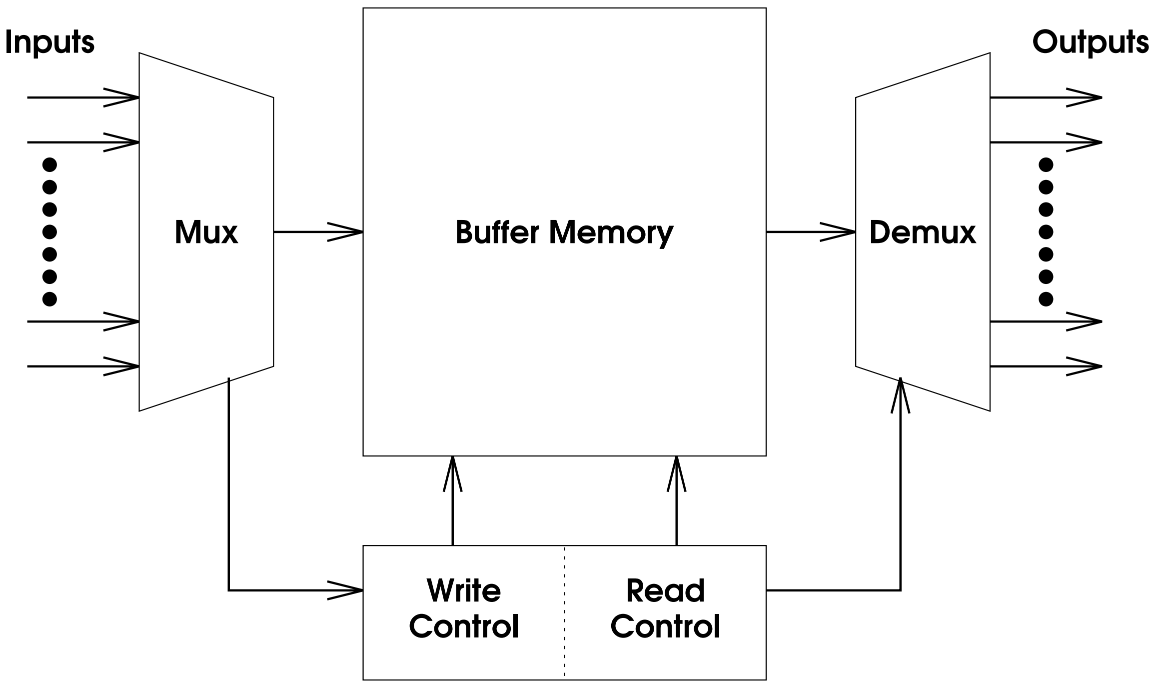


- Common Design: Batcher-Banyan Switching Fabric

- Sunshine Switch



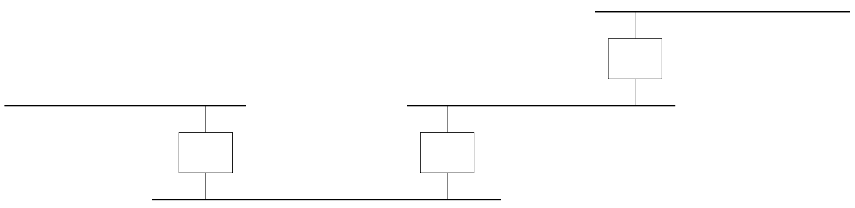
Shared Media Switches



Bridges and Extended LANs

Overview

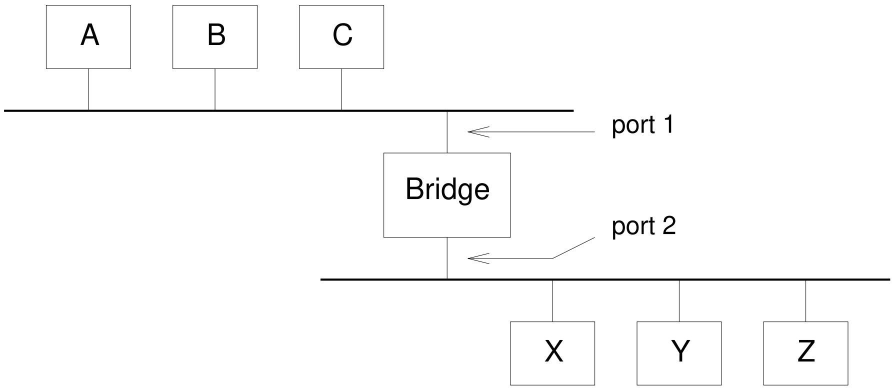
- LANs have physical limitations (e.g., 1500m Ethernet)
- Connect two or more LANs with a *bridge*
  - accept and forward strategy
  - level 2 connection (does not add packet header)



- Collection of LANs connected by bridges called an *extended LAN*

Learning Bridges

- Do not forward when unnecessary



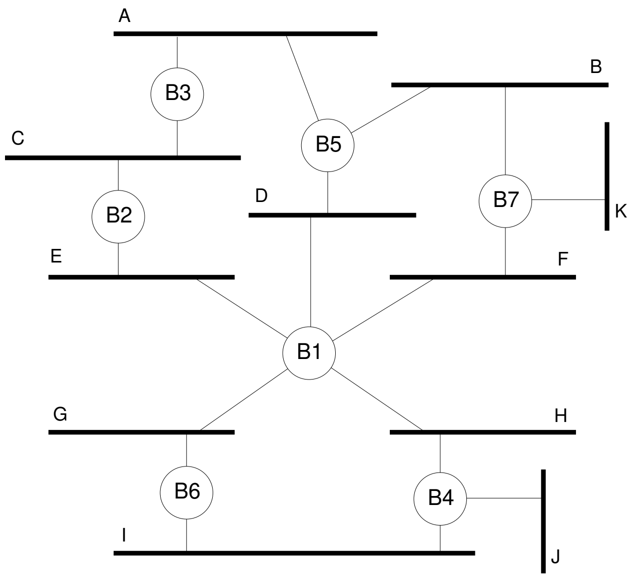
- Maintain forwarding table

Host	Port
A	1
B	1
C	1
X	2
Y	2
Z	2

- Learn table entries based on source address
- Table is an optimization; need not be complete
- Always forward broadcast frames

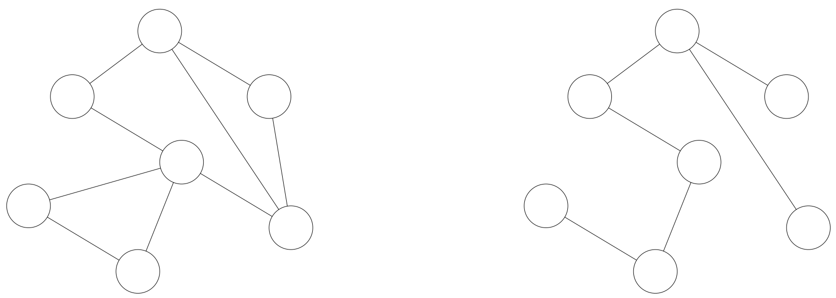
Spanning Tree Algorithm

Extended LANs sometimes have loops



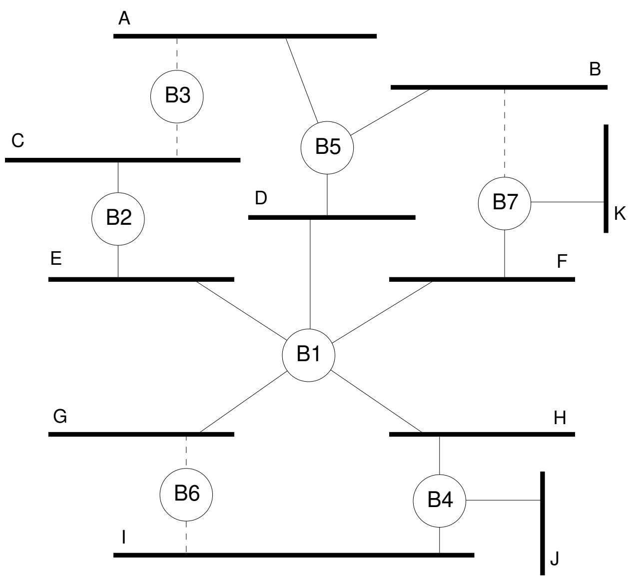
Bridges run a distributed spanning tree algorithm

- Select which bridges actively forward frames
- Developed by Radia Perlman at DEC
- Now IEEE 802.1 specification



Algorithm Overview

- Each bridge has a unique id (e.g., B1, B2, B3)
- Select bridge with smallest id as root
- Select bridge on each LAN that is closest to the root as that LAN's designated bridge (use id to break ties)
- Each bridge forwards frames over each LAN for which it is the designated bridge

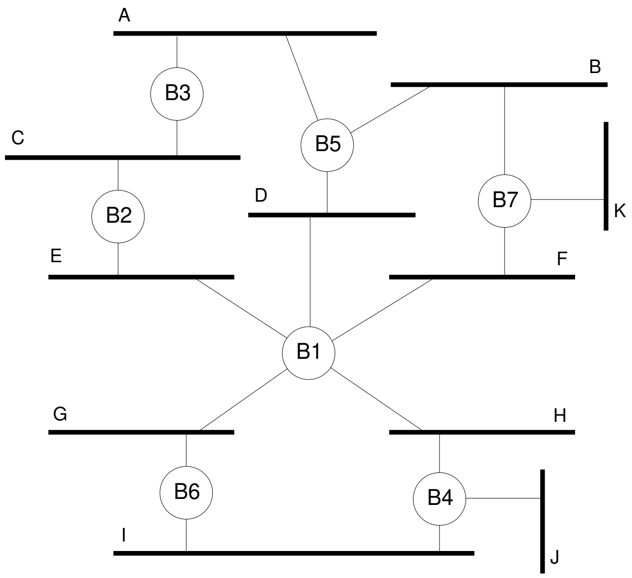


Algorithm Detail

- Bridges exchange configuration messages
  - id for bridge sending the message
  - id for what the sending bridge believes to be root bridge
  - distance (hops) from sending bridge to root bridge
- Each bridge records current best configuration message for each port
- Initially, each bridge believes it is the root
- When learn not root, stop generating configuration message
  - in steady state, only root generates configuration messages
- When learn not designated bridge, stop forwarding configuration messages
  - in steady state, only designated bridges forward configuration messages
- Root bridge continues to send configuration messages periodically
- If any given bridge does not receive configuration message after a period of time, starts generating configuration messages claiming to be to be the root

Broadcast and Multicast

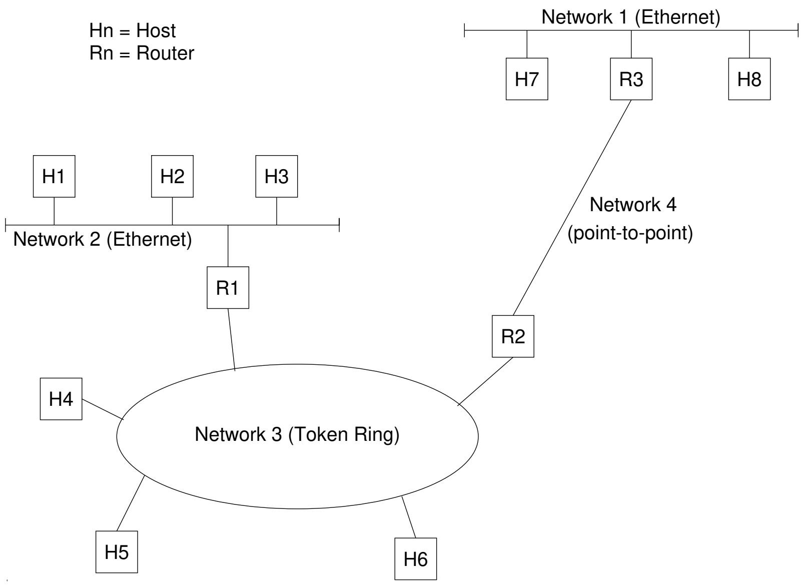
- Forward all broadcast/multicast frames (current practice)
- Learn when no group members downstream
- Accomplished by having each member of group G send a frame to bridge multicast address with G in source field

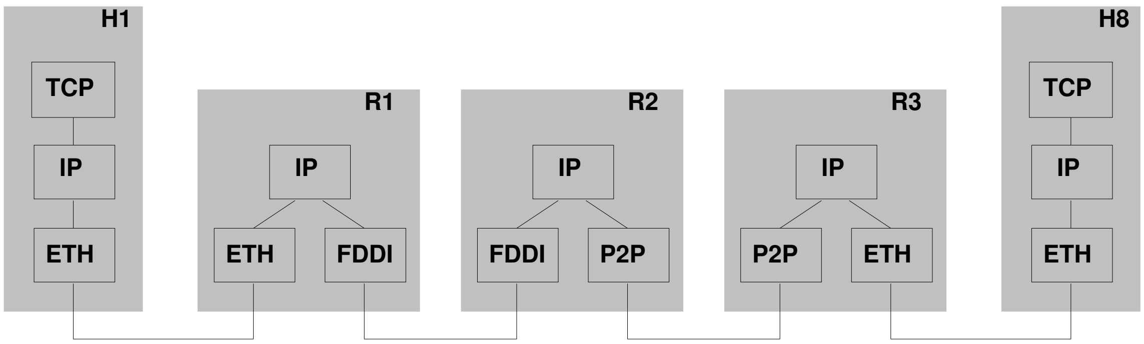


Limitations of Bridges

- Do not scale
  - spanning tree algorithm does not scale
  - broadcast does not scale
- Do not accommodate heterogeneity
- Caution: beware of transparency

Internetworking





Service Model

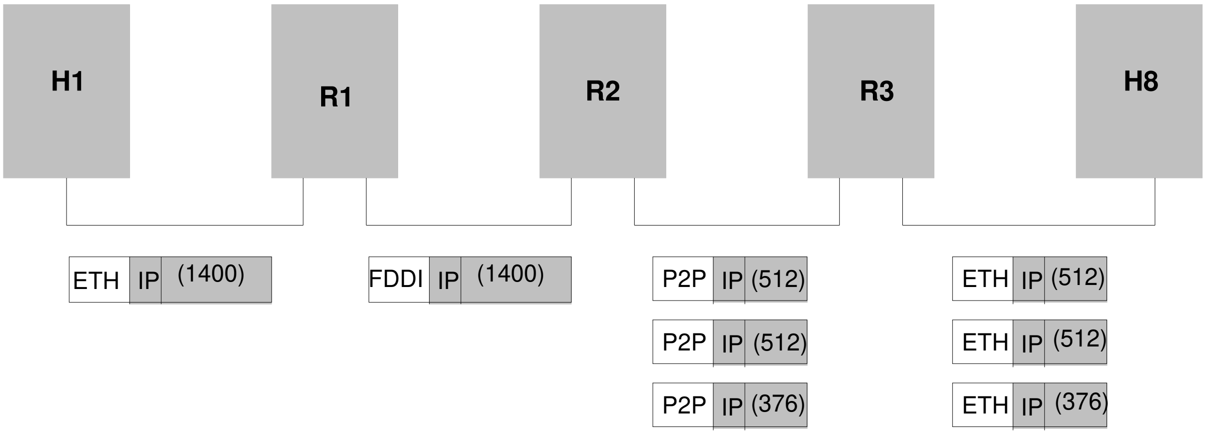
- Packet Delivery Model
- Global Addressing Scheme

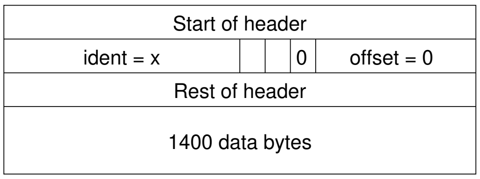
Packet Delivery Model

- Connectionless (datagram-based)
- Best-effort delivery (unreliable service)
  - packets are lost
  - packets are delivered out of order
  - duplicate copies of a packet are delivered
  - packets can be delayed for a long time
- Datagram format
  - Version (4): currently 4
  - Hlen (4): number of 32-bit words in header
  - TOS (8): type of service (not widely used)
  - Length (16): number of bytes in this datagram
  - Ident (16): used by fragmentation
  - Flags/Offset (16): used by fragmentation
  - TTL (8): number of hops this datagram has travelled
  - Protocol (8): demux key (TCP=6, UDP=17)
  - Checksum (16): of the header only
  - DestAddr & SrcAddr (32)

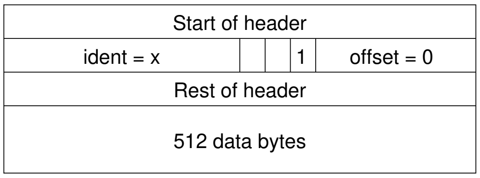
Fragmentation and Reassembly

- Each network has some MTU
- Strategy
  - fragment when necessary (MTU < Datagram)
  - try to avoid fragmentation at source host
  - refragmentation is possible
  - fragments are self-contained datagrams
  - use CS-PDU (not cells) for ATM
  - delay reassembly until destination host
  - do not recover from lost fragments
- Example

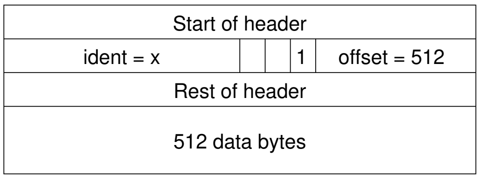




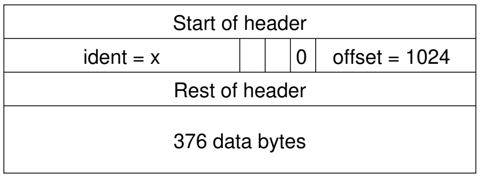
Unfragmented Packet



First Fragment



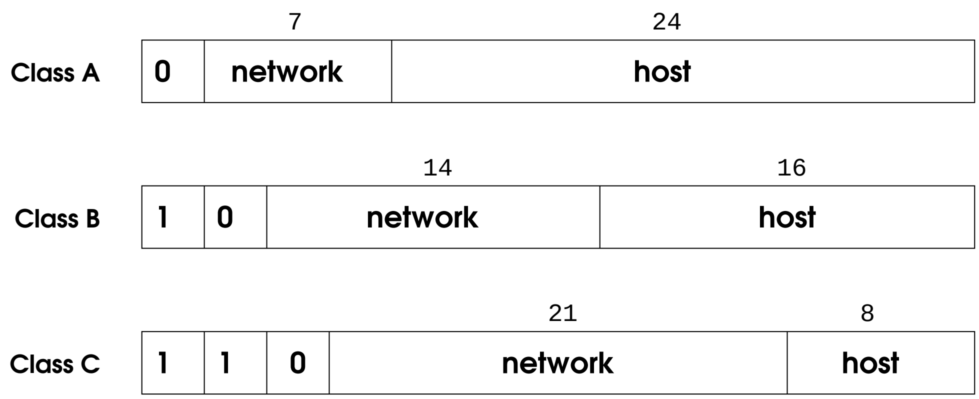
Second Fragment



Last Fragment

Global Addresses

- Properties
  - globally unique
  - hierarchical: network + host
- Format



- Dot notation
  - 10.3.2.4
  - 128.96.33.81
  - 192.12.69.77

Datagram Forwarding

- Strategy
  - every datagram contains destination's address
  - if directly connected to destination network, then forward to host
  - if not directly connected to destination network, then forward to some router
  - forwarding table maps network number into next hop
  - each host has a default router
  - each router maintains a forwarding table
- Example (router R2)

Network Number	Next Hop
1	R3
2	R1
3	interface 1
4	interface 0

Address Translation

- Map IP addresses into physical addresses
  - destination host
  - next hop router
- Techniques
  - encode physical address in host part of IP address
  - table-based
- ARP
  - table of IP to physical address bindings
  - broadcast request if IP address not in table
  - target machine responds with its physical address
  - table entries are discarded if not refreshed

- Request format

<table>
<tr><td colspan="2">HardwareType=1</td><td>ProtocolType=0x0800</td></tr>
<tr><td>HLEN=48</td><td>PLEN=32</td><td>Operation</td></tr>
<tr><td colspan="3">SourceHardwareAddr</td></tr>
<tr><td colspan="2">SourceHardwareAddr</td><td>SourceProtocolAddr</td></tr>
<tr><td colspan="2">SourceProtocolAddr</td><td>TargetHardwareAddr</td></tr>
<tr><td colspan="3">TargetHardwareAddr</td></tr>
<tr><td colspan="3">TargetProtocolAddr</td></tr>
</table>

  – HardwareType: type of physical network (e.g., Ethernet)
  – ProtocolType: type of higher layer protocol (e.g., IP)
  – HLEN & PLEN: length of physical and protocol addresses
  – Operation: request or response
  – Source/Target Physical/Protocol addresses

- Notes

  – table entries timeout in about 10 minutes
  – update table with source when you are the target
  – update table table if already have an entry
  – do not refresh table entries upon reference

Internet Control Message Protocol

- Echo (ping)
- Redirect (from router to source host)
- Destination unreachable (protocol, port, or host)
- TTL exceeded (so datagrams don't cycle forever)
- Checksum failed
- Reassembly failed
- Cannot fragment

Global Internet

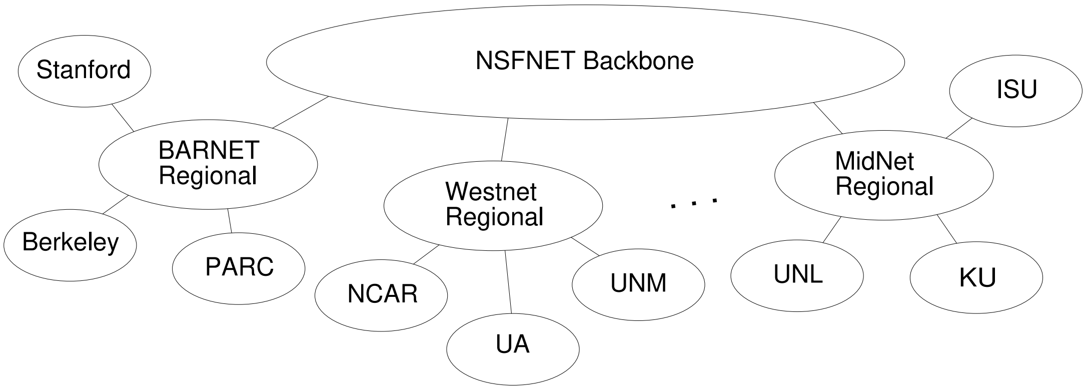
Scalability Issues

IP "hides" hosts in address hierarchy, but...

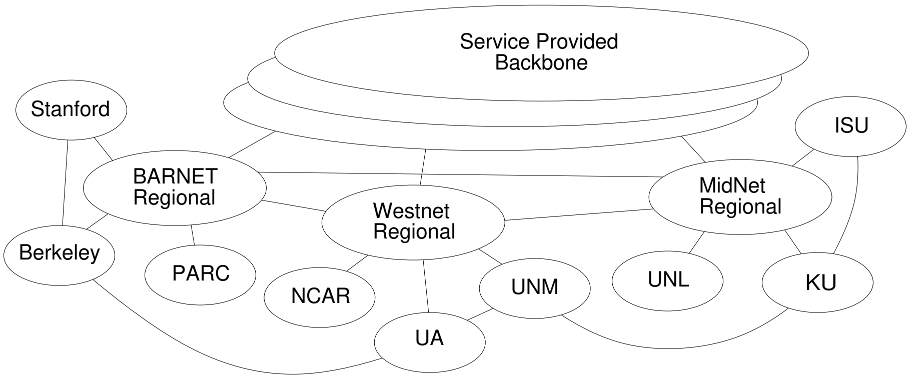
- Inefficient use of address space
  - class C network with 2 hosts (2/255 = 0.78% efficient)
  - class B network with 256 hosts (256/65535 = 0.39% efficient)
- Too many networks
  - today's Internet has tens of thousands of networks
  - routing tables do not scale
  - route propagation protocols do not scale

Internet Structure

Recent Past



Today



Subnetting

- Add another level to address/routing hierarchy: *subnet*
- *Subnet masks* define variable partition of host part of class A and B addresses
- Subnets visible only within site

Network Number	Host Number

Class B address

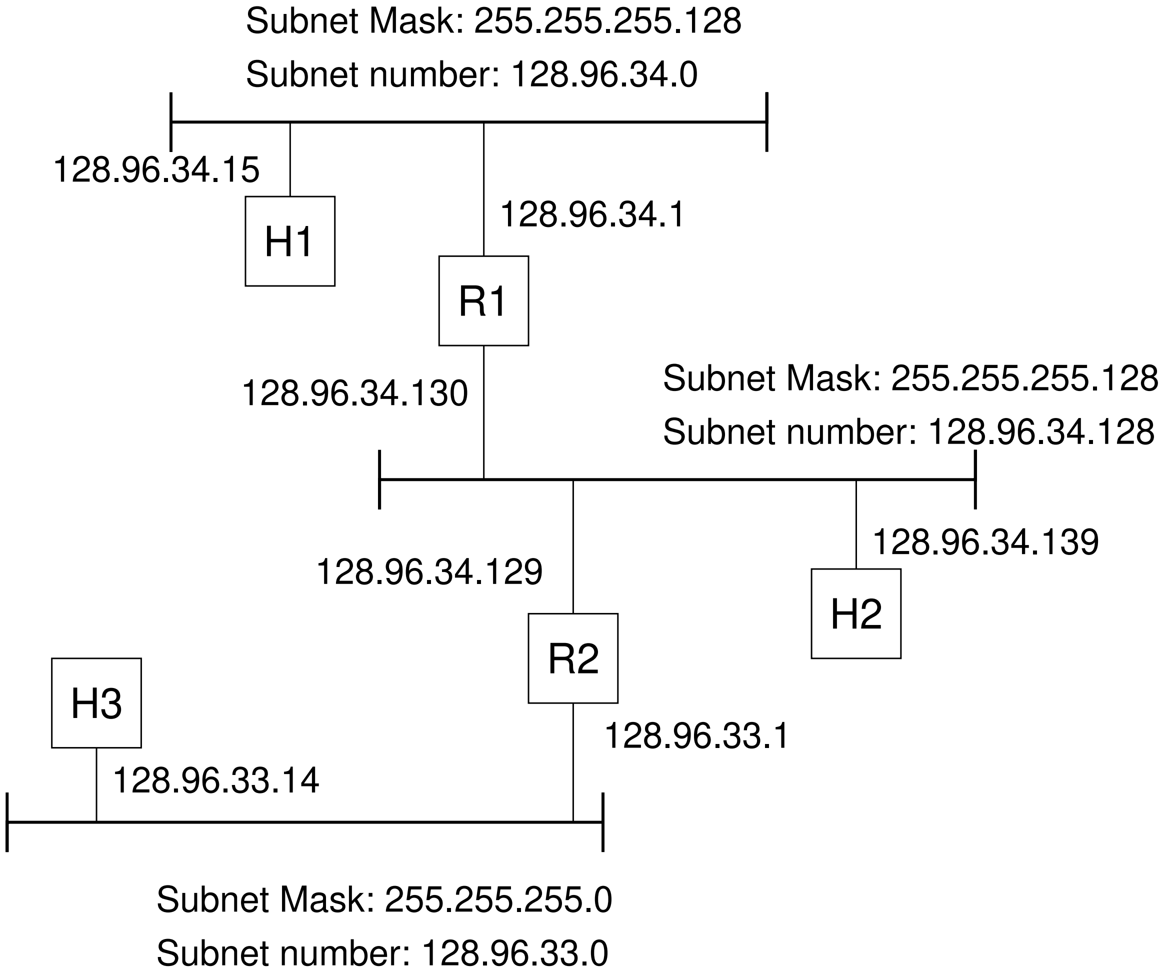
111111111111111111111111	00000000

Subnet Mask (255.255.255.0)

Network Number	Subnet ID	Host ID

Subnetted Address

Subnet Example



Forwarding table at router R1

Subnet Number	Subnet Mask	Next Hop
128.96.34.0	255.255.255.128	interface 0
128.96.34.128	255.255.255.128	interface 1
128.96.33.0	255.255.255.0	R2

Forwarding Algorithm

```
D = destination IP address
for each entry ⟨SubnetNum, SubnetMask, NextHop⟩
   D1 = SubnetMask & D
   if D1 = SubnetNum
      if NextHop is an interface
         deliver datagram directly to destination
      else
         deliver datagram to NextHop (a router)
```

Notes

- Would use a default router if nothing matches
- Not necessary for all ones in subnet mask to be contiguous
- Can put multiple subnets on one physical network
- Subnets not visible from the rest of the Internet

Supernetting

- Assign block of contiguous network numbers to near-by networks
- Called CIDR: Classless Inter-Domain Routing
- Represent blocks with a single pair

  ⟨ `first_network_address, count` ⟩

- Restrict block sizes to powers of 2
- Use a bit mask (CIDR mask) to identify block size
- All routers must understand CIDR addressing

Route Propagation

Idea: Impose a second hierarchy on the network that limits what routers talk to each other. (The first hierarchy is the address hierarchy that governs how packets are forwarded.)

- Autonomous System (AS)
  - corresponds to an administrative domain
  - examples: University, company, backbone network
  - assign each AS a 16-bit number
- Two-level route propagation hierarchy
  - interior gateway protocol (each AS selects its own)
  - exterior gateway protocol (Internet-wide standard)

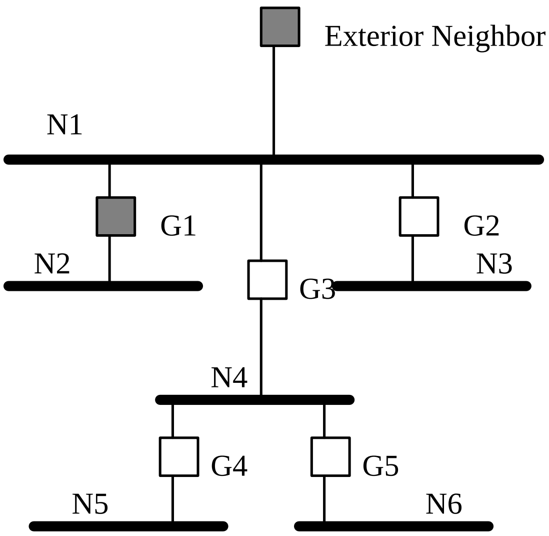
Popular Interior Gateway Protocols

- RIP: Route Information Protocol
  - developed for XNS
  - distributed with Unix
  - distance-vector algorithm
  - based on hop-count
- OSPF: Open Shortest Path First
  - recent Internet standard
  - uses link-state algorithm
  - supports load balancing
  - supports authentication

EGP: Exterior Gateway Protocol

- Overview
  - designed for tree-structured Internet
  - concerned with *reachability*, not optimal routes
- Protocol messages
  - neighbor acquisition: one router requests that another be its peer; peers exchange reachability information
  - neighbor reachability: one router periodically tests to see if the other router is still reachable; exchange HELLO/ACK messages; uses a k-out-of-n rule
  - routing updates: peers periodically exchange their routing tables (distance-vector)

EGP Example



Source Net: N1
G1
1 N2
G2
1 N3
G3
1 N4
2 N5
2 N6

BGP-4: Border Gateway Protocol

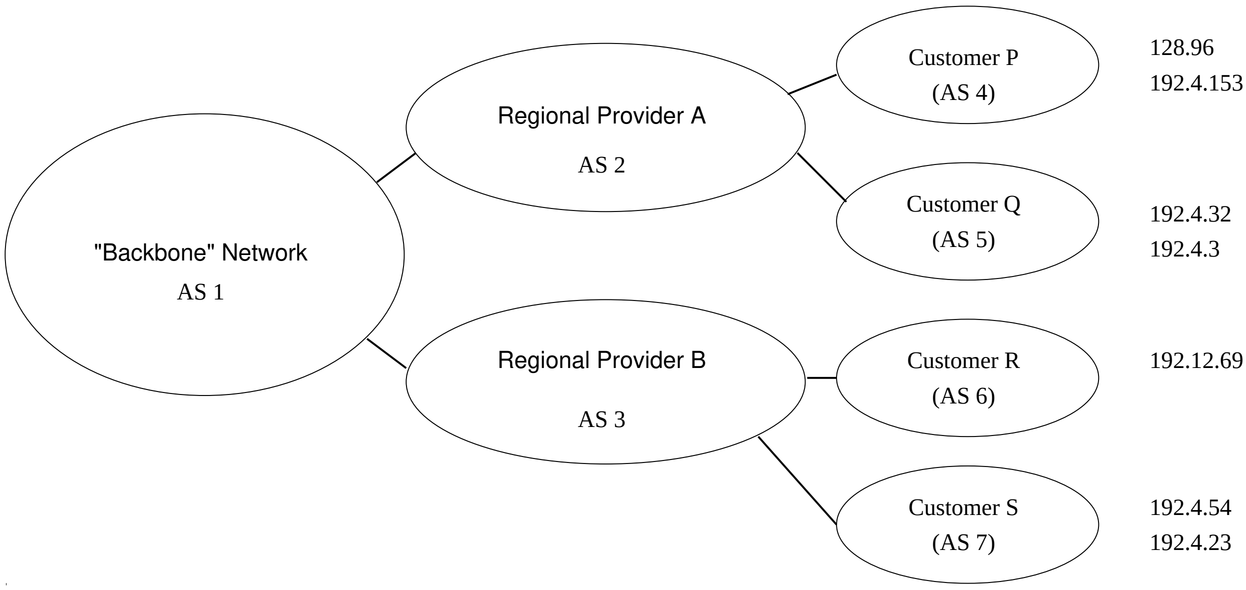
Assumes the Internet is an arbitrarily interconnected set of AS's. Define*local traffic* as traffic that originates at or terminates on nodes within an AS, and *transit traffic* as traffic that passes through an AS, we can classify AS's into three types:

- Stub AS: an AS that has only a single connection to one other AS; such an AS will only carry local traffic.
- Multihomed AS: an AS that has connections to more than one other AS, but refuses to carry transit traffic.
- Transit AS: an AS that has connections to more than one other AS, and is designed to carry both transit and local traffic.

Each AS has:

- One or more border routers
- One BGP *speaker* that advertises:
  - local networks
  - other reachable networks (transit AS only)
  - gives *path* information

BGP Example



- Speaker for AS 2 advertises reachability to P and Q

  Network 128.96, 192.4.153, 192.4.32, and 192.4.3, can be reached directly from AS 2.

- Speaker for backbone network then advertises

  Networks 128.96, 192.4.153, 192.4.32, and 192.4.3 can be reached along the path ⟨AS 1, AS 2⟩.

- Speaker can also cancel previously advertised paths

Next Generation IP (IPv6)

Major Features

- 128-bit addresses
- Multicast
- Real-time service
- Authentication and security
- Autoconfiguration
- End-to-end fragmentation
- Protocol extensions

IPv6 Addresses

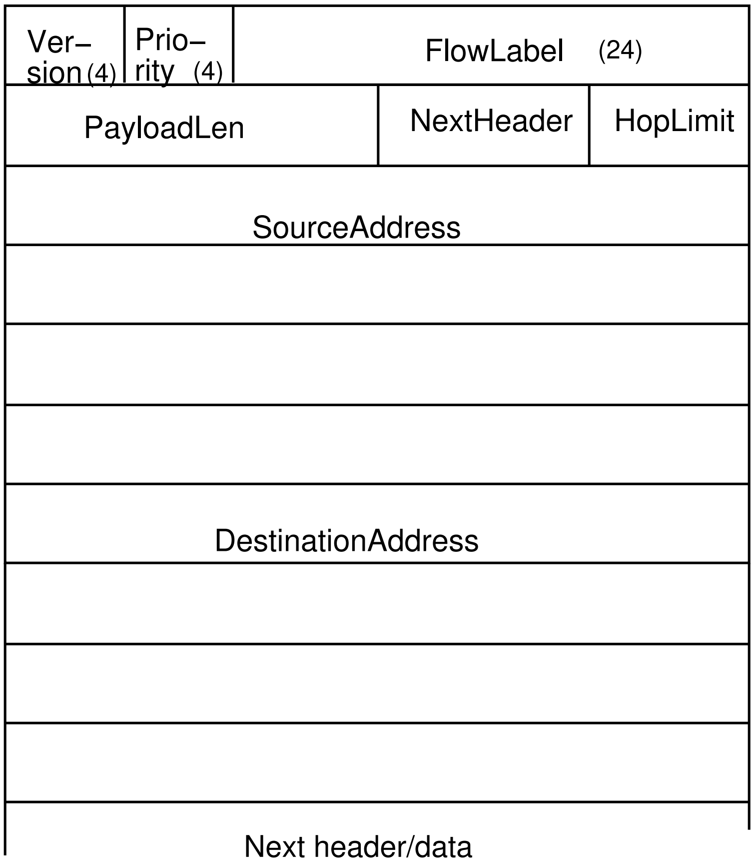
- Classless addressing/routing (similar to CIDR)
- Notation: x:x:x:x:x:x:x:x (x = 16-bit hex number)
  - contiguous 0s are compressed: 47CD::A456:0124
  - IPv6 compatible IPv4 address: ::128.42.1.87
- Address assignment
  - provider-based
  - geographic

3	m	n	o	p	125−m−n−o−p
010	Registry ID	Provider ID	Subscriber ID	Subnet ID	Interface ID

Prefix	Use
0000 0000	Reserved
0000 0001	Unassigned
0000 001	Reserved for NSAP Allocation
0000 010	Reserved for IPX Allocation
0000 011	Unassigned
0000 1	Unassigned
0001	Unassigned
001	Unassigned
010	Provider-Based Unicast Address
011	Unassigned
100	Reserved for Geographic-Based Unicast Addresses
101	Unassigned
110	Unassigned
1110	Unassigned
1111 0	Unassigned
1111 10	Unassigned
1111 110	Unassigned
1111 1110 0	Unassigned
1111 1110 10	Link Local Use Addresses
1111 1110 11	Site Local Use Addresses
1111 1111	Multicast Addresses

IPv6 Header

- 40-byte "base" header
- Extension headers (fixed order, mostly fixed length)
  - fragmentation
  - source routing
  - authentication and security
  - other options



Internet Multicast

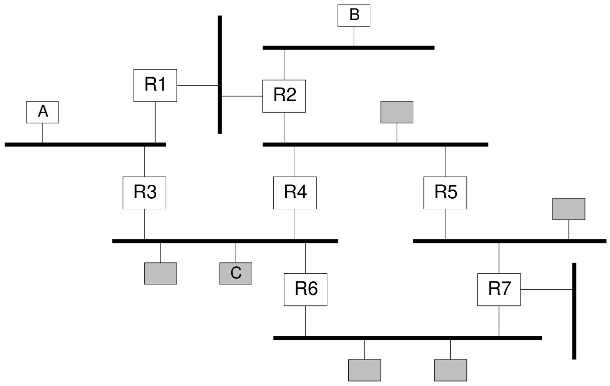
Overview

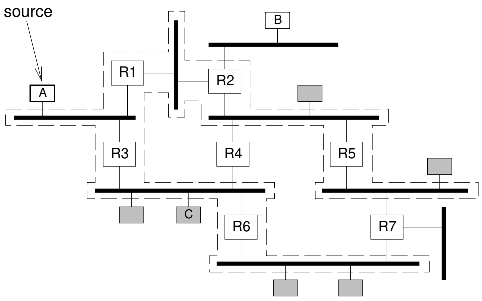
- IPv4
  - class D addresses
  - demonstrated with MBone
  - uses tunneling
- Integral part of IPv6
  - problem is making it scale

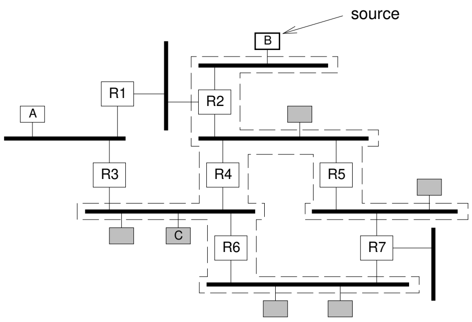
Link-State Multicast

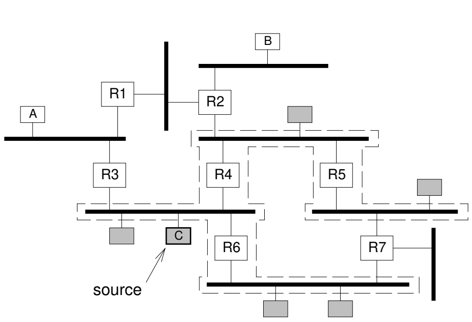
- Each host on a LAN periodically announces the groups it belongs to (IGMP).
- Augment update message (LSP) to include set of groups that have members on a particular LAN.
- Each router uses Dijkstra's algorithm to compute shortest-path spanning tree for each source/group pair.
- Each router caches tree for currently active source/group pairs.

Example


B
R1
R2
A
R3
R4
R5
C
R6
R7


source
B
R1
R2
A
R3
R4
R5
C
R6
R7


source
B
R1
R2
A
R3
R4
R5
C
R6
R7


B
R1
R2
A
R3
R4
R5
C
R6
R7
source

Distance-Vector Multicast

Reverse Path Broadcast (RPB)

- Each router already knows that shortest path to destination S goes through¡ router N.
- When receive multicast packet from S, forward on all outgoing links (except the one on which the packet arrived), iff packet arrived from N.
- Eliminate duplicate broadcast packets by only letting "parent" for LAN (relative to S) forward
  - shortest path to S (learn via distance vector)
  - smallest address to break ties

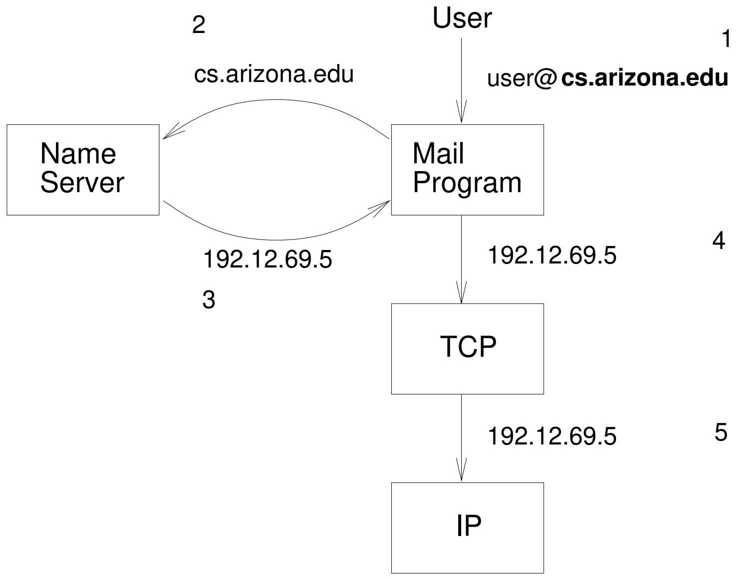
Reverse Path Multicast (RPM)

- Goal: Prune networks that have no hosts in group G
- Step 1: Determine of LAN is a *leaf* with no members in G
  - leaf if parent is only router on the LAN
  - determine if any hosts are members of G using IGMP
- Step 2: Propagate "no members of G here" information
  - augment ⟨`Destination, Cost`⟩ update sent to neighbors with set of groups for which this network is interested in receiving multicast packets.
  - only happens with multicast address becomes active.

Host Names (DNS)

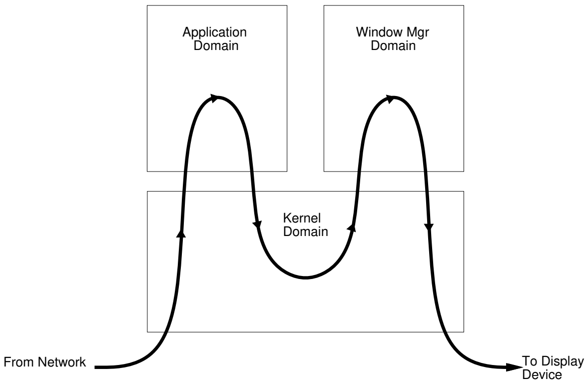
Overview

- Names versus Addresses
  - names are variable length, mnemonic, easy for humans to remember
  - addresses are fixed length, tied to routing, and easy for computers to process
- Name Space
  - defines set of possible names
  - flat versus hierarchical
  - consists of a set of name to value *bindings*



Domain Hierarchy

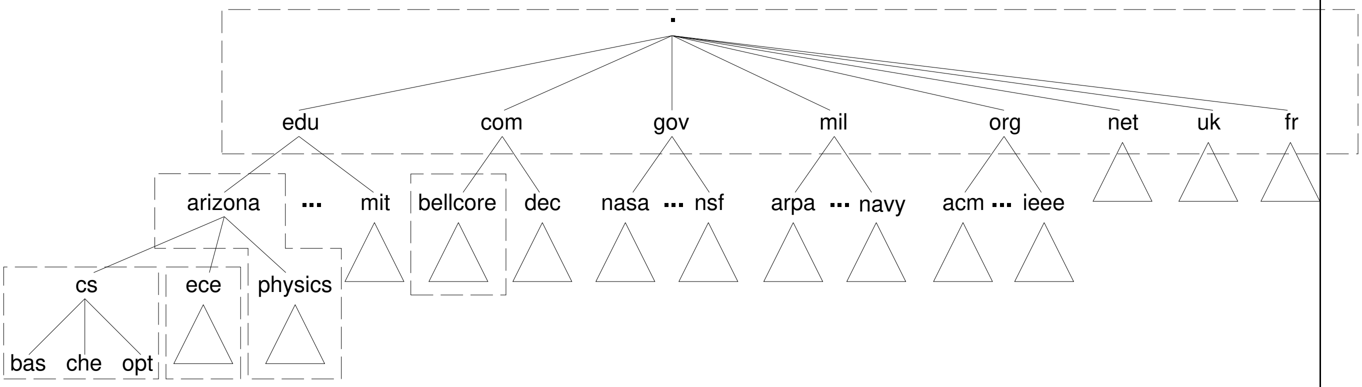
Example hierarchy



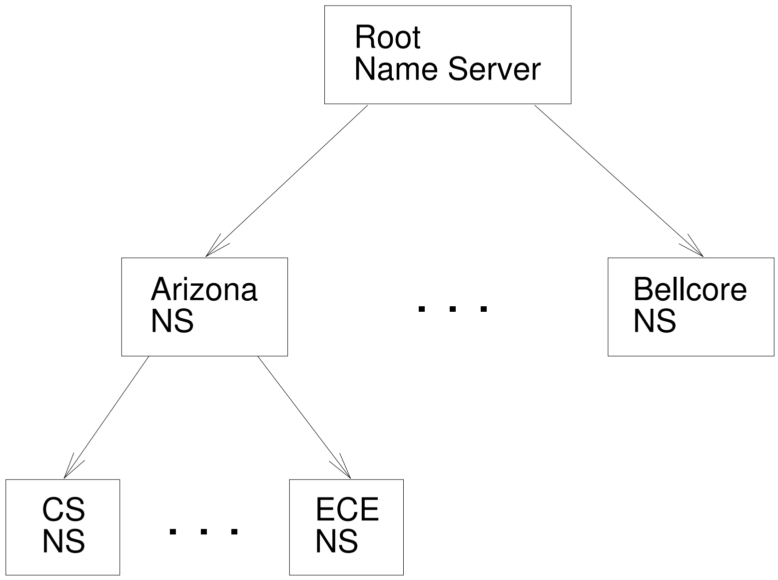
Example name: `cheltenham.cs.arizona.edu`

Name Servers

Partition hierarchy into *zones*



Each zone implemented by two or more *name servers*



Resource Records

- Each name server maintains a collection of *resource records*

  ⟨`Name, Value, Type, Class, TTL`⟩

- `Name/Value`: not necessarily host names to IP addresses
- `Type`
  - `NS`: the **`Value`** field gives the domain name for a host running a name server that knows how to resolve names within the specified domain.
  - `CNAME`: the **`Value`** field gives the canonical name for a particular host; it is used to define aliases.
  - `MX`: the **`Value`** field gives the domain name for a host running a mail server that accepts messages for the specified domain.
- **`Class`**: allow other entities to define types
- `TTL`: how long the resource record is valid

Example

Root server:

⟨`arizona.edu, telcom.arizona.edu, NS, IN`⟩
⟨`telcom.arizona.edu, 128.196.128.233, A, IN`⟩

⟨`bellcore.com, thumper.bellcore.com, NS, IN`⟩
⟨`thumper.bellcore.com, 128.96.32.20, A, IN`⟩

...

Arizona server:

$\langle$`cs.arizona.edu, optima.cs.arizona.edu, NS, IN`$\rangle$
$\langle$`optima.cs.arizona.edu, 192.12.69.5, A, IN`$\rangle$

$\langle$`ece.arizona.edu, helios.ece.arizona.edu, NS, IN`$\rangle$
$\langle$`helios.ece.arizona.edu, 128.196.28.166, A, IN`$\rangle$

$\langle$`jupiter.physics.arizona.edu, 128.196.4.1, A, IN`$\rangle$
$\langle$`saturn.physics.arizona.edu, 128.196.4.2, A, IN`$\rangle$
$\langle$`mars.physics.arizona.edu, 128.196.4.3, A, IN`$\rangle$
$\langle$`venus.physics.arizona.edu, 128.196.4.4, A, IN`$\rangle$

...

CS server:

$\langle$`cs.arizona.edu, optima.cs.arizona.edu, MX, IN`$\rangle$

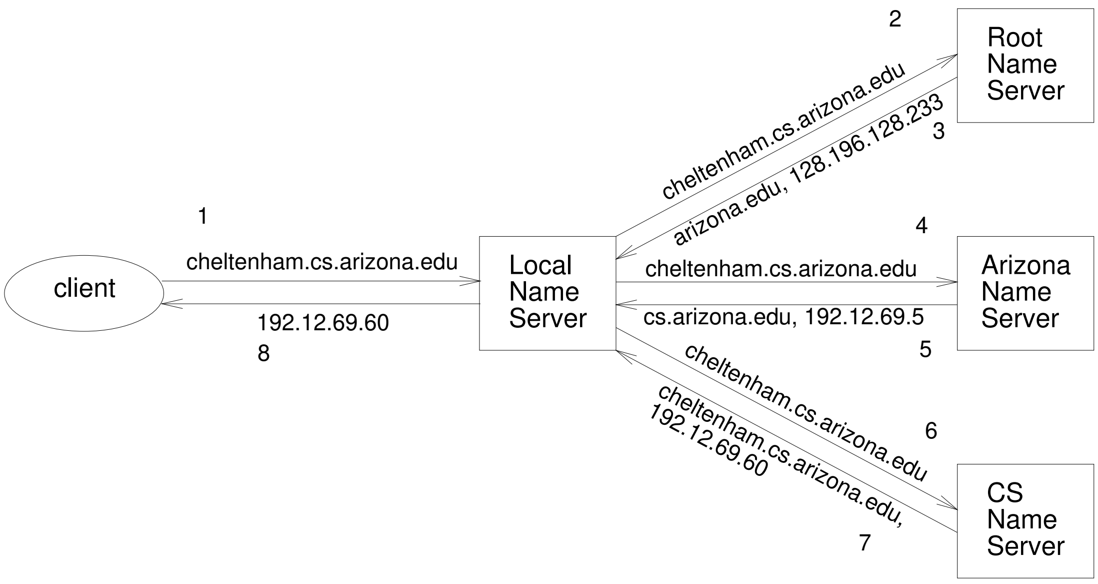
$\langle$`cheltenham.cs.arizona.edu, 192.12.69.60, A, IN`$\rangle$
$\langle$`che.cs.arizona.edu, cheltenham.cs.arizona.edu, CNAME, IN`$\rangle$

$\langle$`optima.cs.arizona.edu, 192.12.69.5, A, IN`$\rangle$
$\langle$`opt.cs.arizona.edu, optima.cs.arizona.edu, CNAME, IN`$\rangle$

$\langle$`baskerville.cs.arizona.edu, 192.12.69.35, A, IN`$\rangle$
$\langle$`bas.cs.arizona.edu, baskerville.cs.arizona.edu, CNAME, IN`$\rangle$

...

Name Resolution



- Strategies
  - forward
  - iterative
  - recursive
- Local server
  - need to know root at only one place (not each host)
  - site-wide cache

End-to-End (Transport) Protocols

- Underlying best-effort network
  - drops messages
  - re-orders messages
  - delivers duplicate copies of a given message
  - limits messages to some finite size
  - delivers messages after an arbitrarily long delay
- Common end-to-end services
  - guarantee message delivery
  - deliver messages in the same order they are sent
  - deliver at most one copy of each message
  - support arbitrarily large messages
  - support synchronization
  - allow the receiver to apply flow control to the sender
  - support multiple application processes on each host

Simple Demultiplexor (UDP)

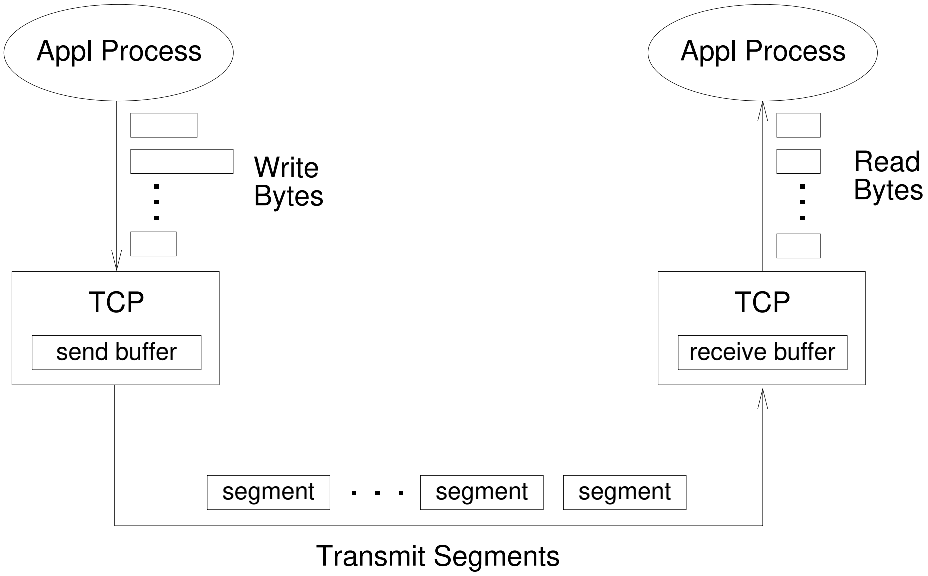
- Unreliable and unordered datagram service
- Adds multiplexing
- No flow control
- Endpoints identified by ports
  - servers have *well-known* ports
  - see `/etc/services` on Unix
- Optional checksum
  - pseudo header + udp header + data
- Header format

Src Port	Dest Port
Check Sum	Length

Reliable Byte-Stream (TCP)

Overview

- Connection-oriented
- Byte-stream
  - sending process writes some number of bytes
  - TCP breaks into *segments* and sends via IP
  - receiving process reads some number of bytes



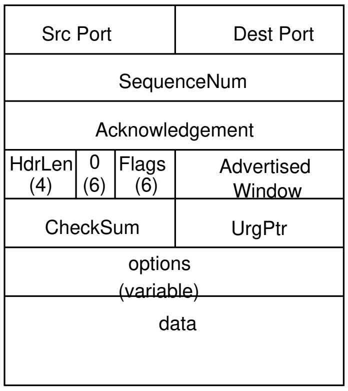
- Full duplex
- Flow control: keep sender from overrunning receiver
- Congestion control: keep sender from overrunning network

End-to-End Issues

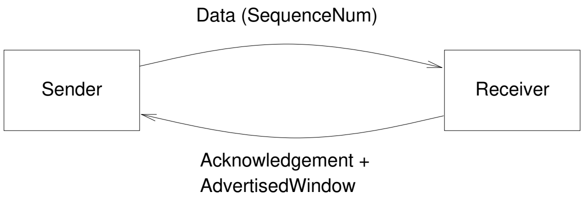
Based on sliding window protocol used at data link level, but the situation is very different.

- Potentially connects many different hosts
  - need explicit connection establishment and termination
- Potentially different RTT
  - need adaptive timeout mechanism
- Potentially long delay in network
  - need to be prepared for arrival of very old packets
- Potentially different capacity at destination
  - need to accommodate different amounts of buffering
- Potentially different network capacity
  - need to be prepared for network congestion

Segment Format



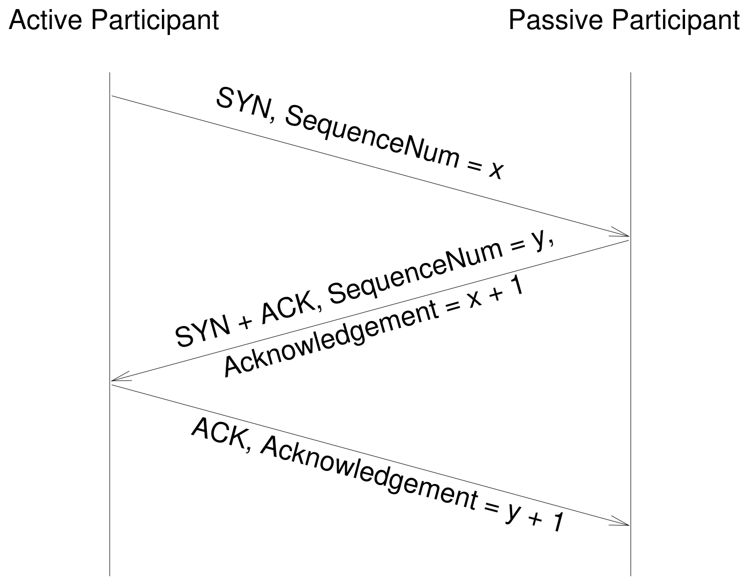
- Each connection identified with 4-tuple:
  - ⟨`SrcPort, SrcIPAddr, DstPort, DstIPAddr`⟩
- Sliding window + flow control
  - `Acknowledgment, SequenceNum, AdvertisedWindow`



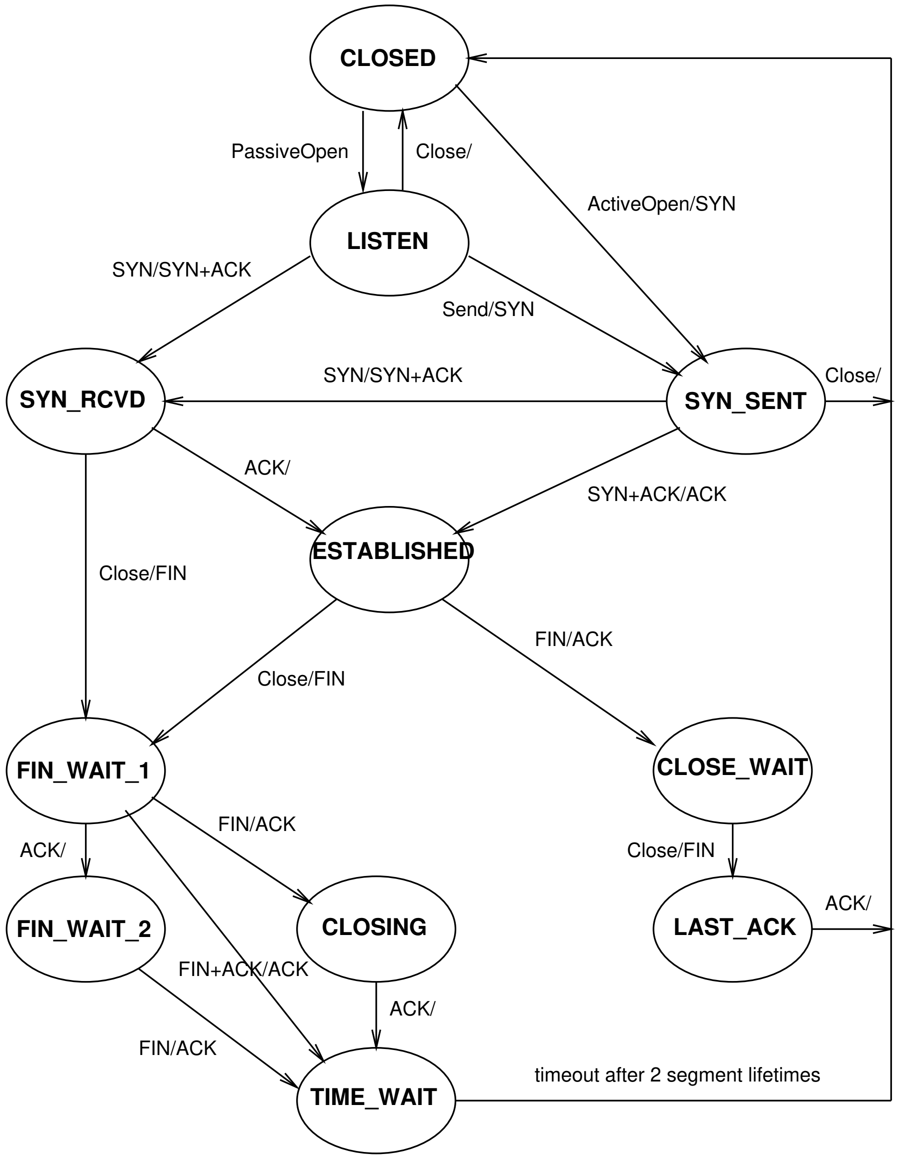
- Flags
  - `SYN, FIN, RESET, PUSH, URG, ACK`
- Checksum
  - pseudo header + tcp header + data

Connection Establishment and Termination

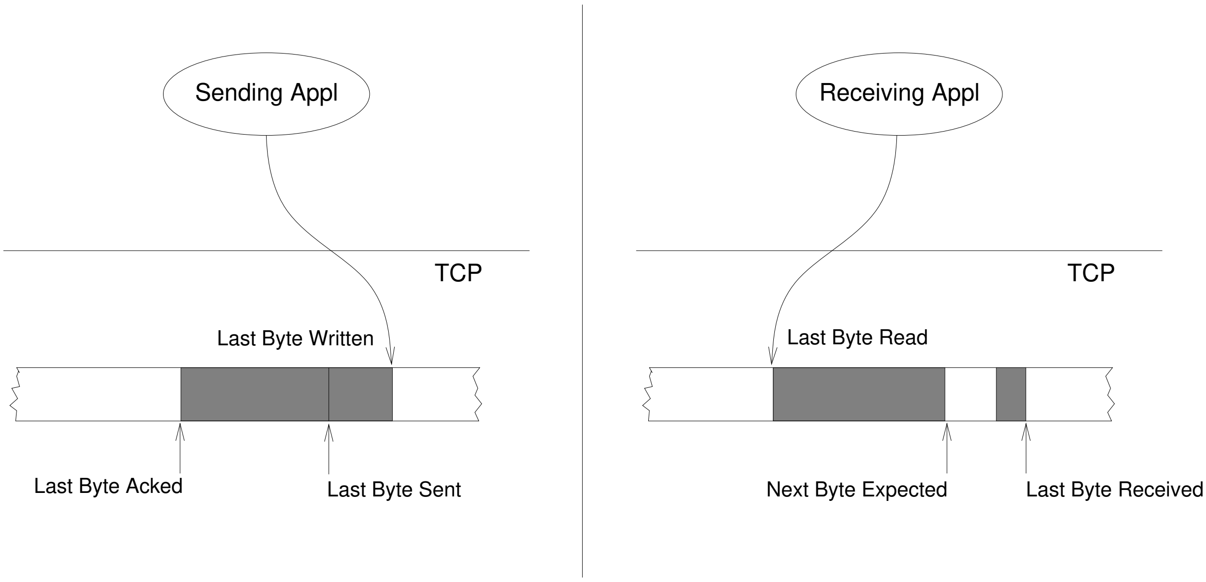
Three-Way Handshake



State Transition Diagram



Sliding Window Revisited



- Each byte has a sequence number
- ACKs are cumulative
- Sending side
  - `LastByteAcked` $\leq$ `LastByteSent`
  - `LastByteSent` $\leq$ `LastByteWritten`
  - bytes between `LastByteAcked` and `LastByteWritten` must be buffered
- Receiving side
  - LastByteRead < NextByteExpected
  - `NextByteExpected` $\leq$ `LastByteRcvd + 1`
  - bytes between `NextByteRead` and `LastByteRcvd` must be buffered

Flow Control

- Sender buffer size: `MaxSendBuffer`
- Receive buffer size: `MaxRcvBuffer`
- Receiving side
  - `LastByteRcvd - NextByteRead` $\leq$ `MaxRcvBuffer`
  - `AdvertisedWindow = MaxRcvBuffer - (LastByteRcvd - NextByteRead)`
- Sending side
  - `LastByteSent - LastByteAcked` $\leq$ `AdvertisedWindow`
  - `EffectiveWindow = AdvertisedWindow - (LastByteSent - LastByteAcked)`
  - `LastByteWritten - LastByteAcked` $\leq$ `MaxSendBuffer`
  - block sender if `(LastByteWritten - LastByteAcked) +` $y >$ `MaxSendBuffer`
- Always send ACK in response to an arriving data segment
- Persist when `AdvertisedWindow=0`

Keeping the Pipe Full

Wrap Around: 32-bit `SequenceNum`

Bandwidth	Time Until Wrap Around
T1 (1.5Mbps)	6.4 hours
Ethernet (10Mbps)	57 minutes
T3 (45Mbps)	13 minutes
FDDI (100Mbps)	6 minutes
STS-3 (155Mbps)	4 minutes
STS-12 (622Mbps)	55 seconds
STS-24 (1.2Gbps)	28 seconds

Bytes in Transit: 16-bit `AdvertisedWindow`

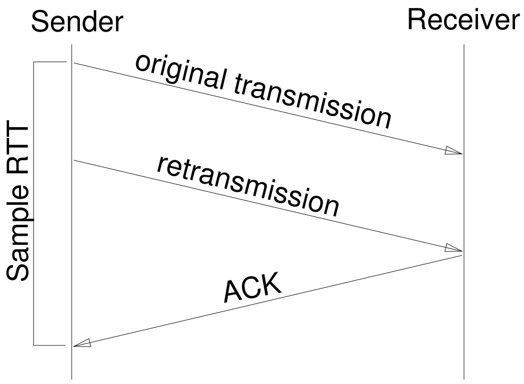
Bandwidth	Delay $\times$ Bandwidth Product
T1 (1.5Mbps)	18KB
Ethernet (10Mbps)	122KB
T3 (45Mbps)	549KB
FDDI (100Mbps)	1.2MB
STS-3 (155Mbps)	1.8MB
STS-12 (622Mbps)	7.4MB
STS-24 (1.2Gbps)	14.8MB

Adaptive Retransmission

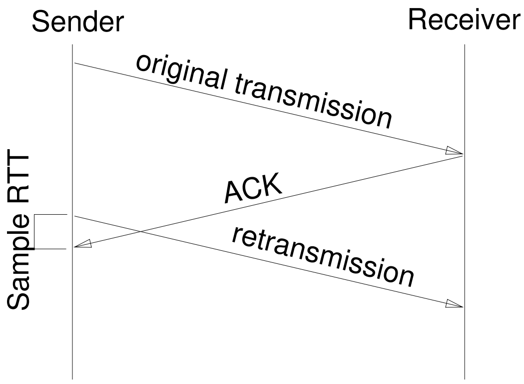
Original Algorithm

- Measure `SampleRTT` for each segment/ACK pair
- Compute weighted average of RTT
  - `EstimatedRTT =` $\alpha \times$ `EstimatedRTT +` $\beta \times$ `SampleRTT`
  - where α `+` β `= 1`
  - α between 0.8 and 0.9
  - β between 0.1 and 0.2
- Set timeout based on `EstimatedRTT`
  - `TimeOut = 2` $\times$ `EstimatedRTT`

Karn/Partridge Algorithm



(a) Sample RTT too long



(b) Sample RTT too short

- Do not sample RTT when retransmitting
- Double timeout after each retransmission

Jacobson/Karels Algorithm

- New calculation for average RTT

  `Difference = SampleRTT - EstimatedRTT`

  `EstimatedRTT = EstimatedRTT + (`$\delta \times$` Difference)`

  `Deviation = Deviation + `δ` ( |Difference |- Deviation)`

  – where δ is a fraction between 0 and 1

- Consider variance when setting timeout value

  `TimeOut = `$\mu \times$` EstimatedRTT + `$\phi \times$` Deviation`

  – where $\mu = 1$ and $\phi = 4$

- Notes

  – algorithm only as good as granularity of clock (500ms on Unix)

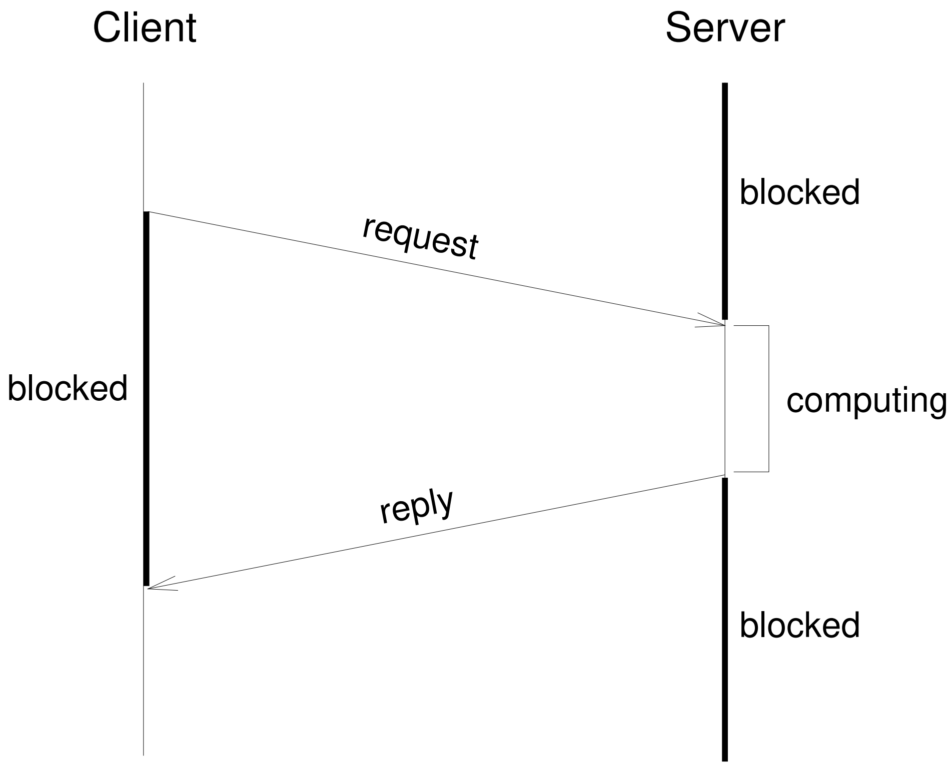
  – accurate timeout mechanism important to congestion control (later)

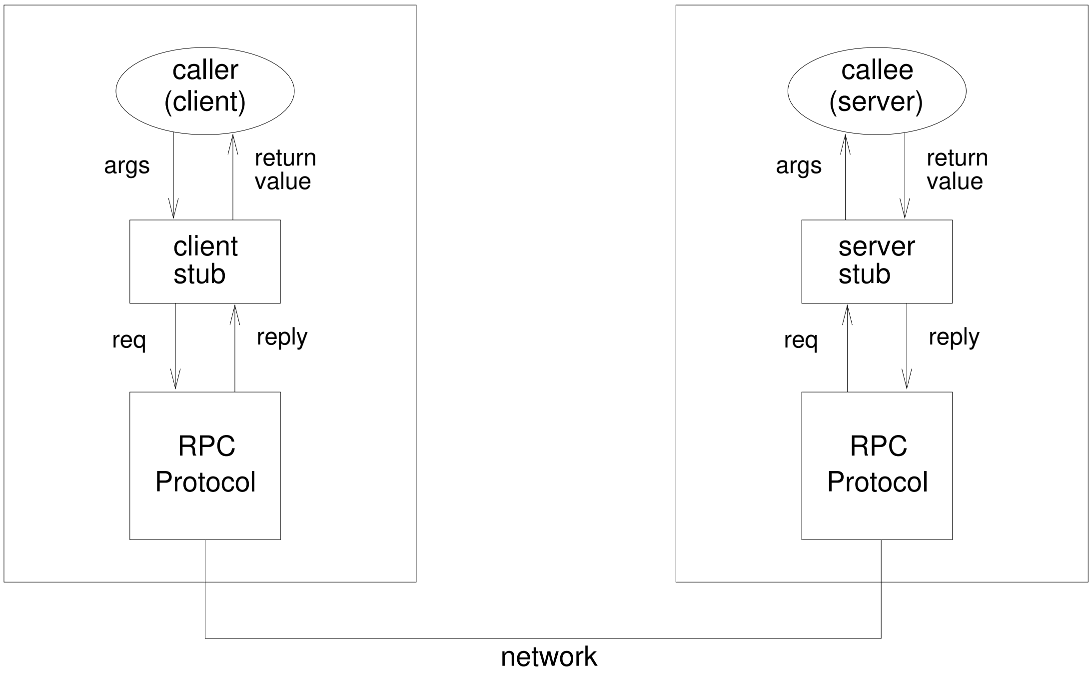
TCP Extensions

- Implemented as header options
- Store timestamp in outgoing segments
- Use 32-bit timestamp to extend sequence space (PAWS)
- Shift (scale) advertised window

Remote Procedure Call

Overview





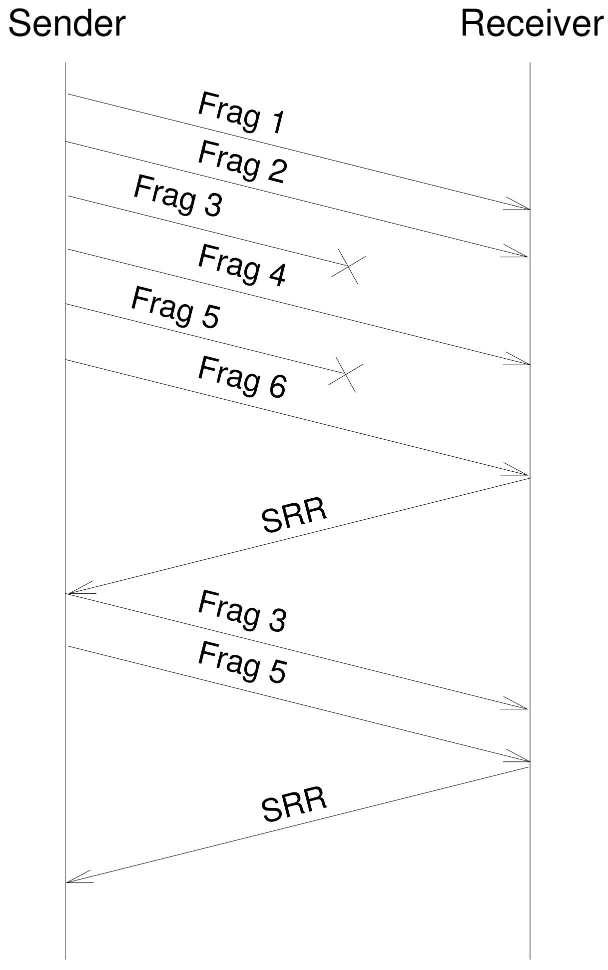
RPC protocol consists of three basic functions

- BLAST: fragments and reassembles large messages
- CHAN: synchronizes request and reply messages
- SELECT: dispatches request messages to the correct process

We consider RPC stubs later.

Bulk Transfer (BLAST)

Unlike AAL and IP in that it tries to recover from lost fragments; persistent, but does not guarantee delivery. Strategy is to use *selective retransmission* (*partial acknowledgements*).



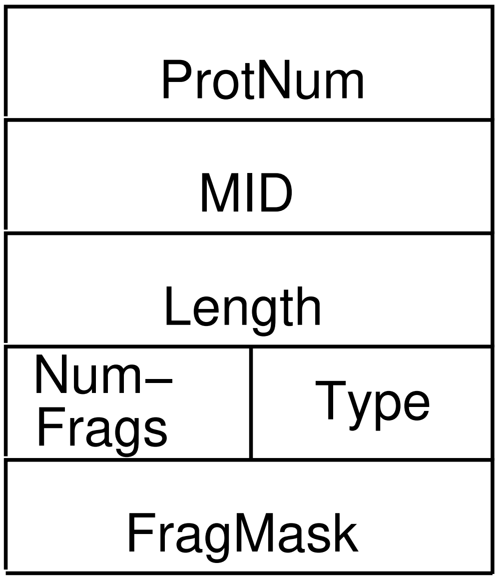
Sender:

- After sending all fragments, set timer DONE
- If receive SRR, send missing fragments and reset DONE
- If timer DONE expires, free fragments

Receiver:

- When first fragment arrives, set timer LAST_FRAG
- When all fragments present, reassemble and pass up
- Four exceptional conditions
  - if last fragment arrives but message not complete
    * send SRR and set timer RETRY
  - if timer LAST_FRAG expires
    * send SRR and set timer RETRY
  - if timer RETRY expires for first or second time
    * send SRR and set timer RETRY
  - if timer RETRY expires for third time
    * give up and free partial message

BLAST Header Format

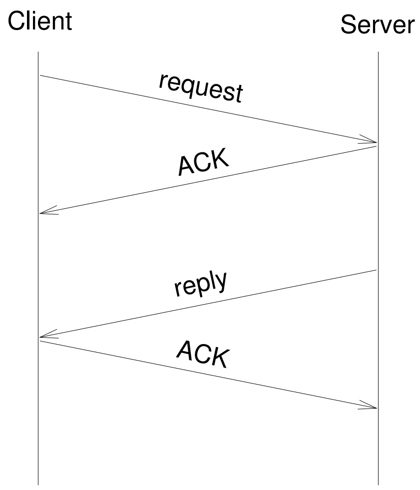


- `MID` must protect against wrap around
- `Type` = `DATA` or `SRR`
- `NumFrags` indicates number of fragments in message
- `FragMask` distinguishes among fragments:
  - if `Type=DATA`, identifies this fragment
  - if `Type=SRR`, identifies missing fragments

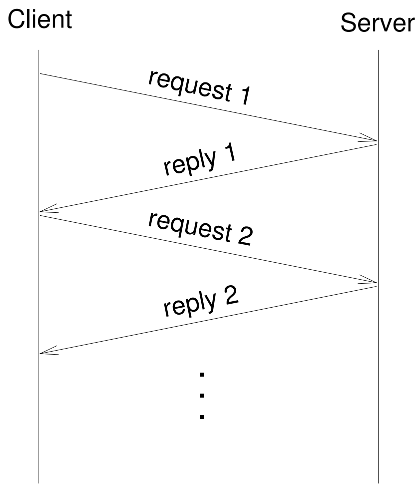
Request/Reply (CHAN)

Guarantees message delivery, and synchronizes client with server; i.e., blocks client until reply received. Supports *at-most-once* semantics.

Simple case:



Implicit Acknowledgements:



Complications:

- Lost message (request, reply, or ACK)
  - set RETRANSMIT timer
  - use message id (MID) field to distinguish
- Slow (long running) server
  - client periodically sends “are you alive” probe, or
  - server periodically sends “I’m alive” notice
- Want to support multiple outstanding calls
  - use channel id (CID) field to distinguish
- Machines crash and reboot
  - use boot id (BID) field to distinguish

CHAN Header Format

```
typedef struct {
    u_short  Type;    /* REQ, REP, ACK, PROBE */
    u_short  CID;     /* unique channel id */
    int      MID;     /* unique message id */
    int      BID;     /* unique boot id */
    int      Length;  /* length of message */
    int      ProtNum; /* high-level protocol */
} ChanHdr;
```

CHAN Session State

```
typedef struct {
    u_char    type;         /* CLIENT or SERVER */
    u_char    status;       /* BUSY or IDLE */
    int       retries;      /* number of retries */
    int       timeout;      /* timeout value */
    XkReturn  ret_val;      /* return value */
    Msg       *request;     /* request message */
    Msg       *reply;       /* reply message */
    Semaphore reply_sem;    /* client semaphore */
    int       mid;          /* message id */
    int       bid;          /* boot id */
} ChanState;
```

Synchronous versus Asynchronous Protocols

Asynchronous Interface

```
xPush(Sessn s, Msg *msg)
xPop(Sessn s, Msg *msg, void *hdr)
xDemux(Protl hlp, Sessn s, Msg *msg)
```

Synchronous Interface

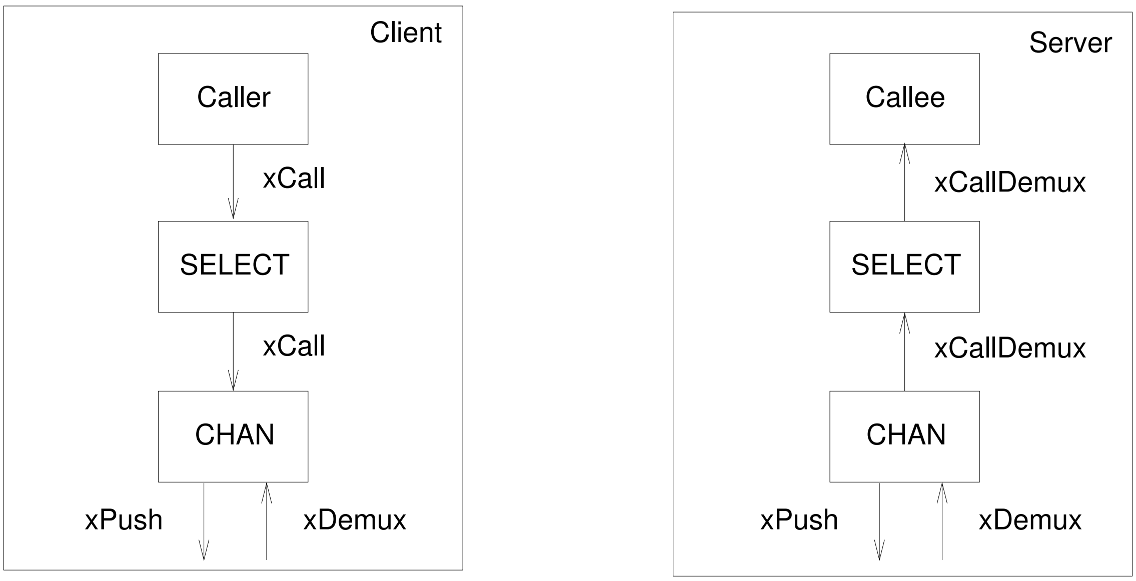
```
xCall(Sessn s, Msg *req, Msg *rep)
xCallPop(Sessn s, Msg *req, Msg *rep,
         void *hdr)
xCallDemux(Protl hlp, Sessn s, Msg *req,
           Msg *rep)
```

CHAN is a Hybrid Protocol

- Synchronous from above: `xCall`
- Asynchronous from below: `xPop/xDemux`

Dispatcher (SELECT)

Dispatches request messages to the appropriate procedure; fully synchronous counterpart to UDP.



Address Space for Procedures

- Flat: unique id for each possible procedure
- Hierarchical: program + procedure within program

Example Code

Client Side

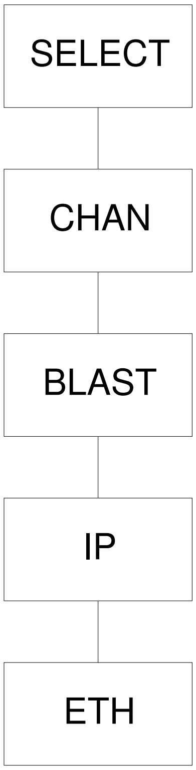
```
static XkReturn
selectCall(Sessn self, Msg *req, Msg *rep)
{
    SelectState *state=(SelectState *)self->state;
    char        *buf;

    buf = msgPush(req, HLEN);
    select_hdr_store(state->hdr, buf, HLEN);
    return xCall(xGetDown(self, 0), req, rep);
}
```

Server side

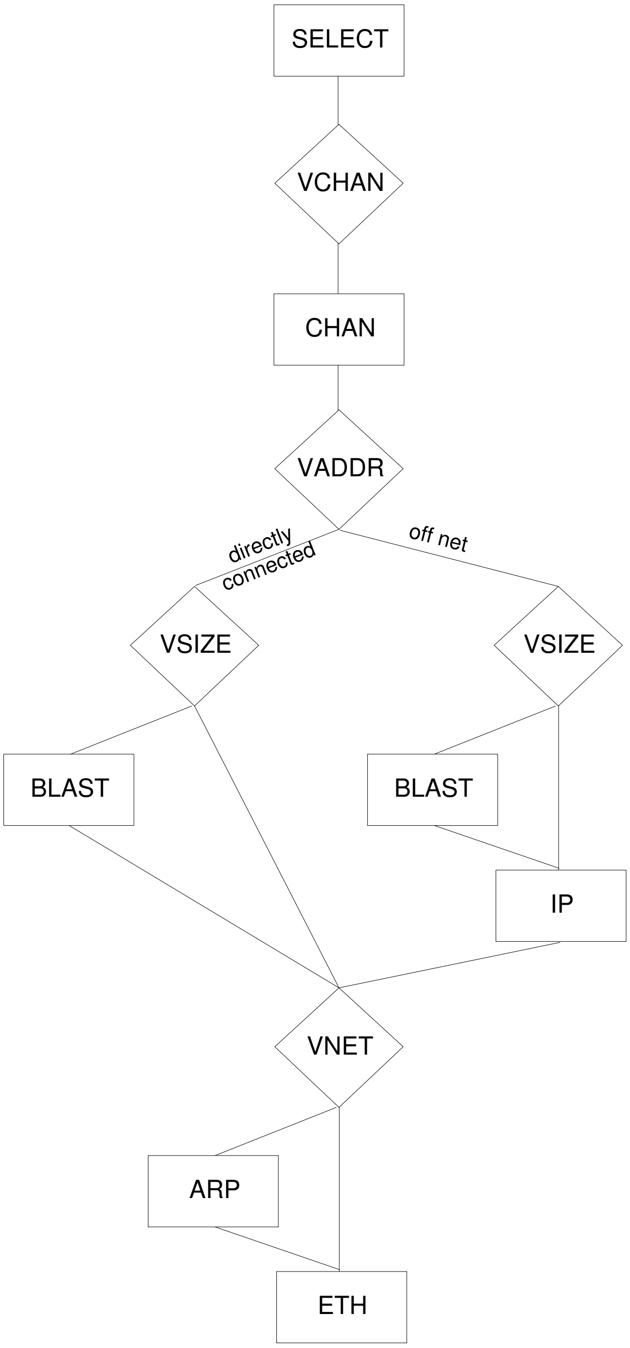
```
static XkReturn
selectCallPop(Sessn s, Sessn lls, Msg *req,
              Msg *rep, void *inHdr)
{
    return xCallDemux(xGetUp(s), s, req, rep);
}
```

Putting it All Together

Simple RPC Stack



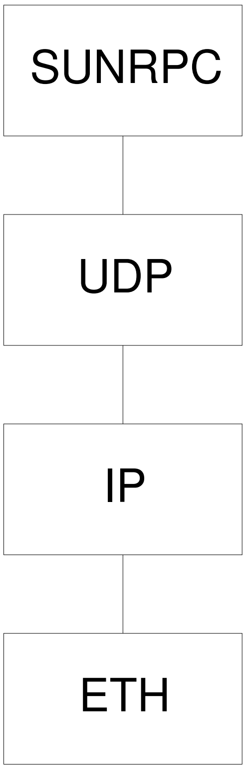
A More Interesting RPC Stack



VCHAN: A Virtual Protocol

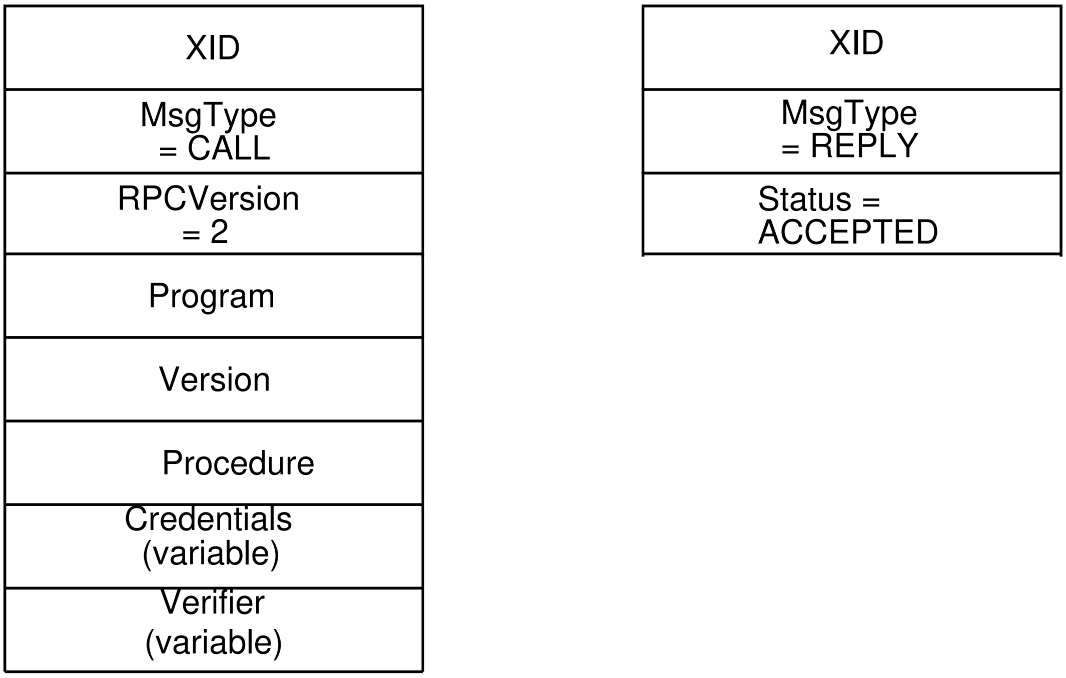
```
static XkReturn
vchanCall(Sessn s, Msg *req, Msg *rep)
{
    Sessn      chan;
    XkReturn   result;
    VchanState *state=(VchanState *)s->state;

    /* wait for an idle channel */
    semWait(&state->available);
    chan = state->stack[--state->tos];

    /* use the channel */
    result = xCall(chan, req, rep);

    /* free the channel */
    state->stack[state->tos++] = chan;
    semSignal(&state->available);

    return result;
}
```

SunRPC



- IP implements BLAST-equivalent
  - except no selective retransmit
- SunRPC implements CHAN-equivalent
  - except not at-most-once
- UDP + SunRPC implement SELECT-equivalent
  - UDP dispatches to program (ports bound to programs)
  - SunRPC dispatches to procedure w/in program

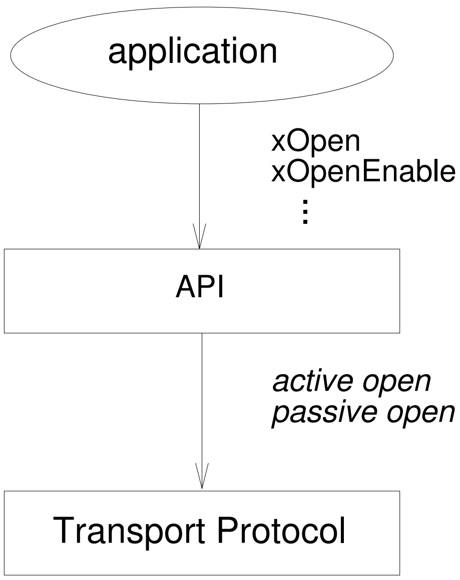
SunRPC Header Format



- `XID` (transaction id) is similar to CHAN's `MID`
- Server does not remember last `XID` it serviced
- Problem if client retransmits request while reply is in

Application Programming Interface

It is important to separate the *implementation* of protocols from the *interface* they export. This is espeically important at the transport layer since this defines the point where application programs typically access the network. This interface is often called the *application programming interface*, or API.



Notes

- The API is usually defined by the OS.
- We now focus on one specific API: *sockets*
- Defined by BSD Unix, but ported to other systems

Socket Operations

- Creating a socket

```
int socket(int domain, int type, int protocol)
```

  - domain=PF_INET, PF_UNIX
  - type=SOCK_STREAM, SOCK_DGRAM

- Passive open on server

```
int bind(int socket, struct sockaddr *address,
         int addr_len)
int listen(int socket, int backlog)
int accept(int socket, struct sockaddr *address,
           int *addr_len)
```

- Active open on client

```
int connect(int socket, struct sockaddr *address,
            int addr_len)
```

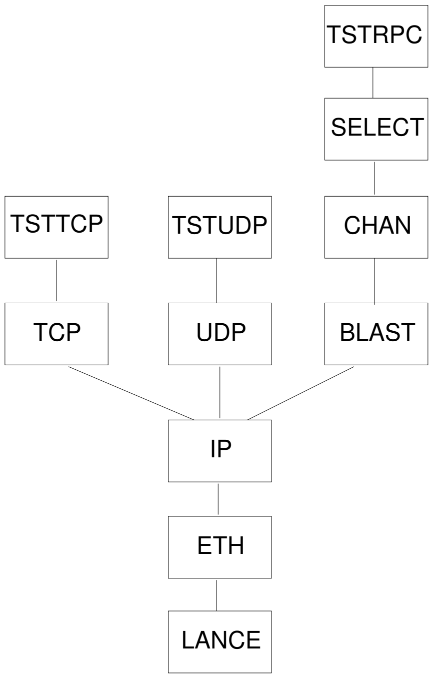
- Sending and receiving messages

```
int write(int socket, char *message, int msg_len,
          int flags)
int read(int socket, char *buffer, int buf_len,
         int flags)
```

Performance

Experimental Method

- DEC 3000/600 workstations (Alpha 21064 at 175MHz)
- 10Mbps Ethernet (Lance controller)
- Ping-pong tests; 10,000 round trips
- Each test repeated five times
- Latency: 1-byte, 100-byte, 200-byte,... 1000-byte messages
- Throughput: 1KB, 2KB, 4KB,... 32KB
- Protocol Graphs



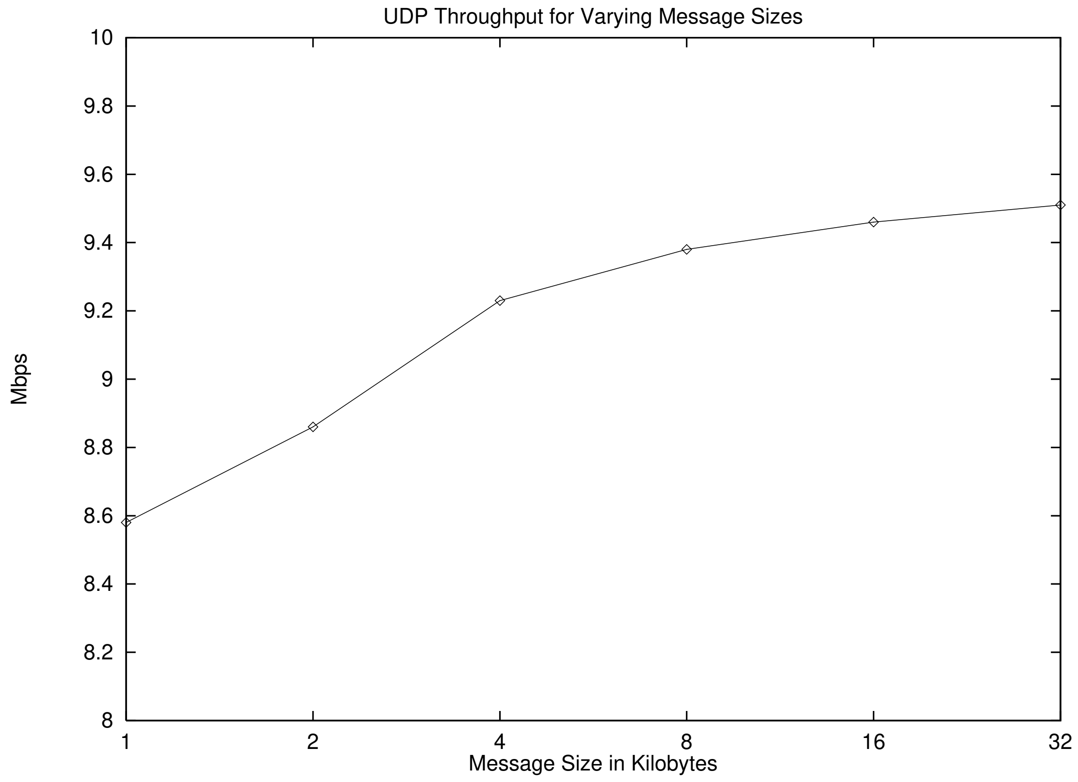
Round-Trip Latency (μs)

Message size (bytes)	UDP	TCP	RPC
1	279	365	428
100	413	519	593
200	572	691	753
300	732	853	913
400	898	1016	1079
500	1067	1185	1247
600	1226	1354	1406
700	1386	1514	1566
800	1551	1676	1733
900	1719	1845	1901
1000	1878	2015	2062

Per-Layer Latency

- ETH + wire: 216μs
- UDP/IP: 58μs

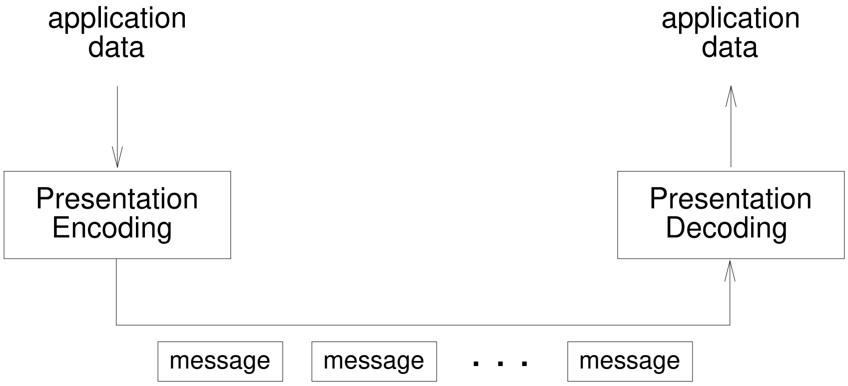
Throughput


UDP Throughput for Varying Message Sizes
10
9.8
9.6
9.4
9.2
9
8.8
8.6
8.4
8.2
8
Mbps
1
2
4
8
16
32
Message Size in Kilobytes

Presentation Formatting

Overview

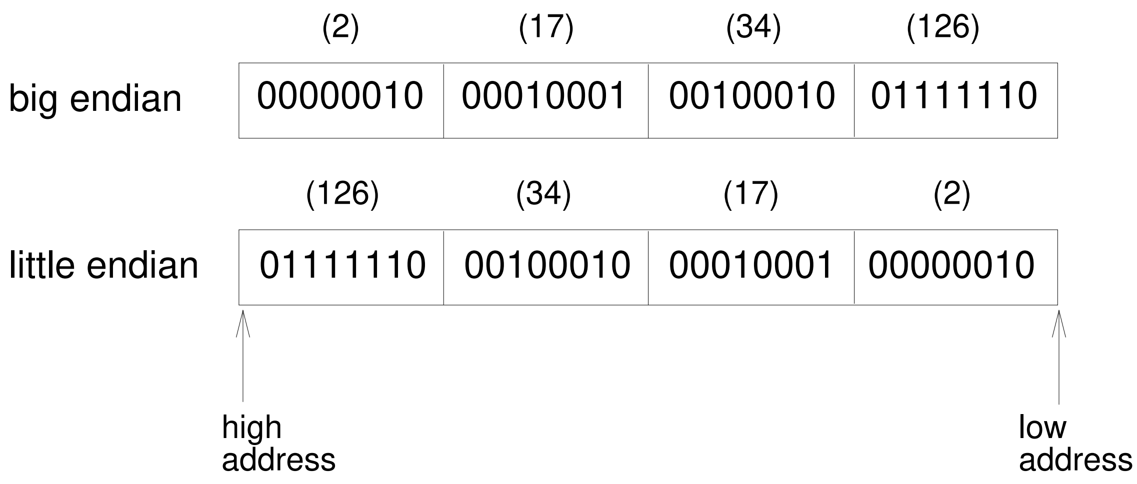
- Marshalling (encoding) application data into messages
- Unmarshalling (decoding) messages into application data



- Data types we consider:
  - integers
  - floating point numbers
  - character strings
  - arrays
  - structures
- Types of data we do not consider (now):
  - images
  - video
  - multimedia documents

Difficulties

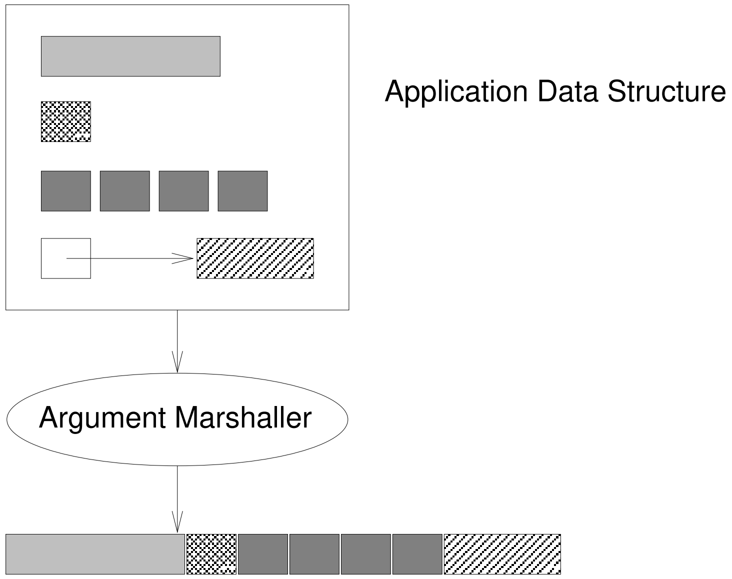
- Representation of base types
  - floating point: IEEE 754 versus non-standard
  - integer: big-endian versus little-endian (e.g., 34,677,374)



- Compiler layout of structures

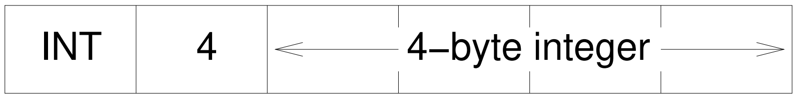
Taxonomy

- Data types
  - base types (e.g., ints, floats); must convert
  - flat types (e.g., structures, arrays); must pack
  - complex types (e.g., pointers); must linearize

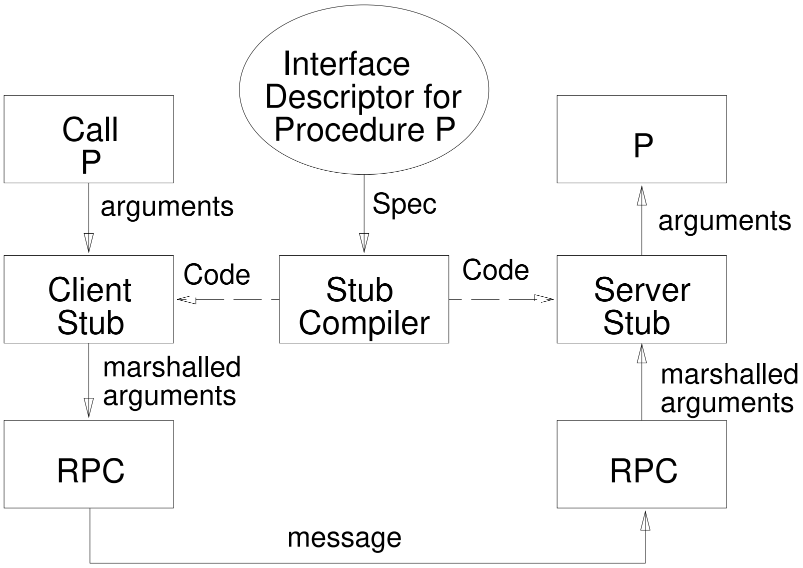


- Conversion Strategy
  - canonical intermediate form
  - receiver-makes-right (an $N \times N$ solution)

- Tagged versus Untagged data



- Stubs
  - compiled
  - interpreted

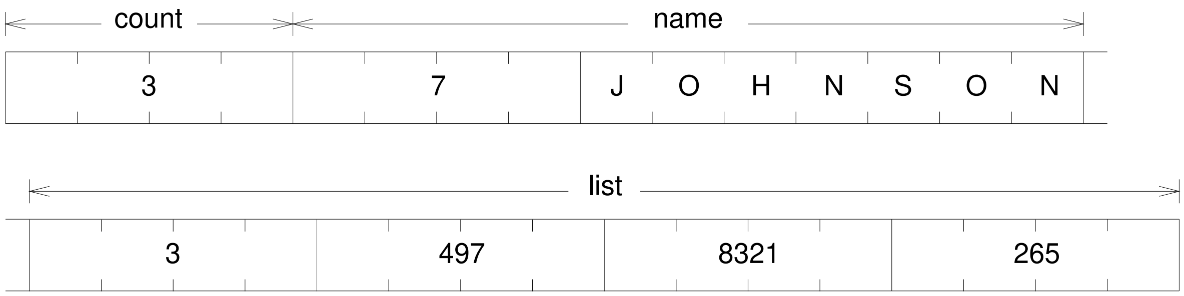


Examples

XDR: eXternal Data Representation

- Defined by Sun for use with SunRPC
- C type system (without function pointers)
- Canonical intermediate form
- Untagged (except array length)
- Compiled stubs

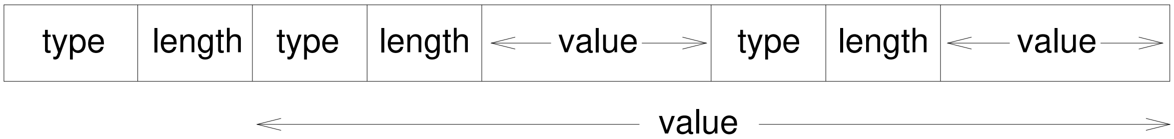
```
#define MAXNAME 256;
#define MAXLIST 100;

struct item {
   int     count;
   char    name[MAXNAME];
   int     list[MAXLIST];
};

bool_t
xdr_item(XDR *xdrs, struct item *ptr)
{
    return(xdr_int(xdrs, &ptr->count) &&
       xdr_string(xdrs, &ptr->name, MAXNAME) &&
       xdr_array(xdrs, &ptr->list, &ptr->count,
                 MAXLIST, sizeof(int), xdr_int));
}
```

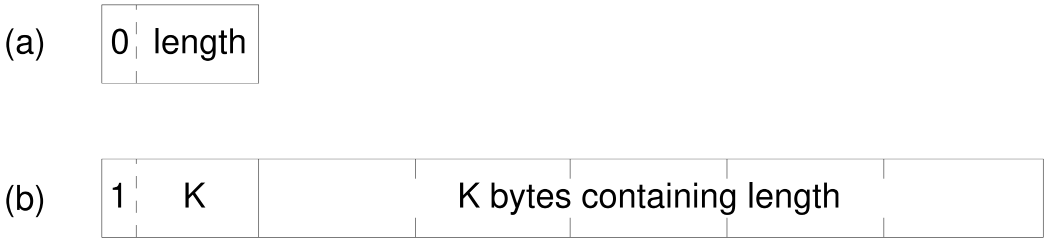


ASN.1: Abstract Syntax Notation One

- An ISO standard
- Essentially the C type system
- Canonical intermediate form
- Tagged
- Compiled or interpretted stubs
- BER: Basic Encoding Rules

  ⟨`tag, length, value`⟩





NDR: Network Data Representation

- Defined by DCE
- Essentially the C type system
- Receiver-makes-right (architecture tag)
- Individual data items untagged
- Compiled stubs from IDL
- 4-byte architecture definition tag

Integer Rep	Char Rep	FloatRep	Extension 1	Extension 2

- `IntegrRep`
  - 0 = big-endian
  - 1 = little-endian
- `CharRep`
  - 0 = ASCII
  - 1 = EBCDIC
- `FloatRep`
  - 0 = IEEE 754
  - 1 = VAX
  - 2 = Cray
  - 3 = IBM

Data Compression

Data must be encoded into a message. Compression is concerned with with removing *redundancy* from that encoding. There are two classes of compression:

- Lossless: ensures that the data recovered from the compression/decompression process is exactly the same as the original data. Commonly used to compress executable code, text files, and numeric data.
- Lossy: does not promise that the data received is exactly the same as the data sent; removes removes information that it cannot later restore. (Hopefully, no one will notice.) Commonly used to compress digital imagery, including video.

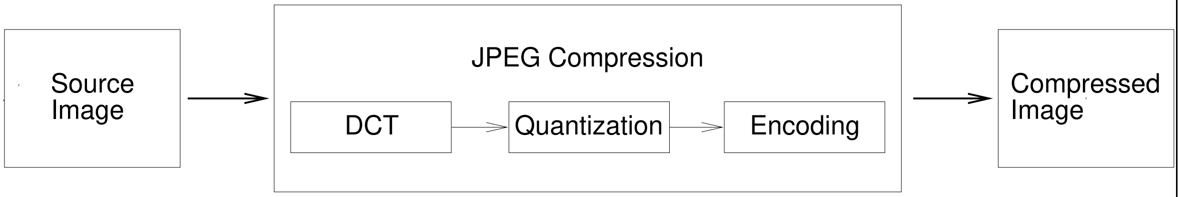
Note: The compression/decompression process takes time. Whether or not you compress data (and how much you compress it) depends on whether you have more cycles (for compression) or bandwidth (for transmission).

Loss-Less Compression Algorithms

- Run Length Encoding (RLE)
  - example: AAABBCDDDD encoded as 3A2B1C4D
  - good for scanned text (8-to-1 compression ratio)
  - can increase size for data with variation (e.g., some images)
- Differential Pulse Code Modulation (DPCM)
  - example: AAABBCDDDD encoded as A0001123333
  - change reference symbol if delta becomes too large
  - works better than RLE for many digital images (1.5-to-1)
- Dictionary-Based Methods
  - build dictionary of common terms (variable length strings)
  - transmit index into dictionary for each term
  - Lempel-Ziv (LZ) compression is the best-known example
  - commonly achieve 2-to-1 ration on text
  - variation of LZ used to compress GIF images
    - first reduce 24-bit color to 8-bit color
    - treat common sequences of pixels as terms in dictionary
    - not uncommon to achieve 10-to-1 compression ($\times 3$)

Image Compression

- JPEG: Joint Photographic Expert Group (ISO/ITU)
- Lossy still image compression
- Three phase process



  – process in 8×8 block chunks (macroblock)
  – greyscale: each pixel is a single value
  – color: each pixel is three values (YUV)
  – DCT: transforms signal from spatial domain into an equivalent signal in the frequency domain (loss-less)
  – apply a quantization to the results (lossy)
  – RLE-like encoding (loss-less)

Quantization Phase

$$\texttt{QuantizedValue}(i,j) = \texttt{IntegerRound}(DCT(i,j)/\texttt{Quantum}(i,j))$$

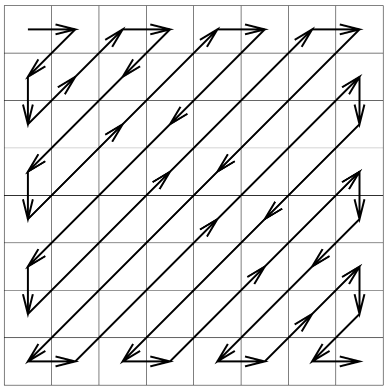
$$\texttt{IntegerRound}(x) = \begin{cases} \lfloor x + 0.5 \rfloor & \text{if } x \geq 0 \\ \lfloor x - 0.5 \rfloor & \text{if } x < 0 \end{cases}$$

Decompression is then simply defined as

$$DCT(i,j) = \texttt{QuantizedValue}(i,j) \times \texttt{Quantum}(i,j)$$

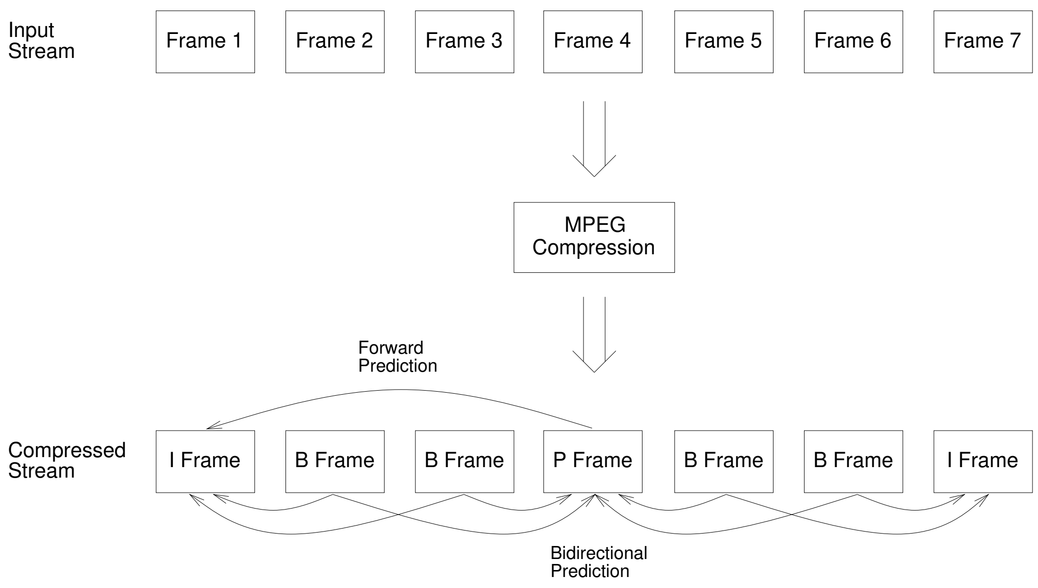
$$\texttt{Quantum} = \begin{bmatrix} 3 & 5 & 7 & 9 & 11 & 13 & 15 & 17 \\ 5 & 7 & 9 & 11 & 13 & 15 & 17 & 19 \\ 7 & 9 & 11 & 13 & 15 & 17 & 19 & 21 \\ 9 & 11 & 13 & 15 & 17 & 19 & 21 & 23 \\ 11 & 13 & 15 & 17 & 19 & 21 & 23 & 25 \\ 13 & 15 & 17 & 19 & 21 & 23 & 25 & 27 \\ 15 & 17 & 19 & 21 & 23 & 25 & 27 & 29 \\ 17 & 19 & 21 & 23 & 25 & 27 & 29 & 31 \end{bmatrix}$$

Encoding Phase



Video Compression

- MPEG: Motion Picture Expert Group
- Lossy compression of video
- First approximation: JPEG on each frame
- Added compression: remove inter-frame redundancy
- Frame types
  - I frames: intrapicture
  - P frames: predicted picture
  - B frames: bidirectional predicted picture

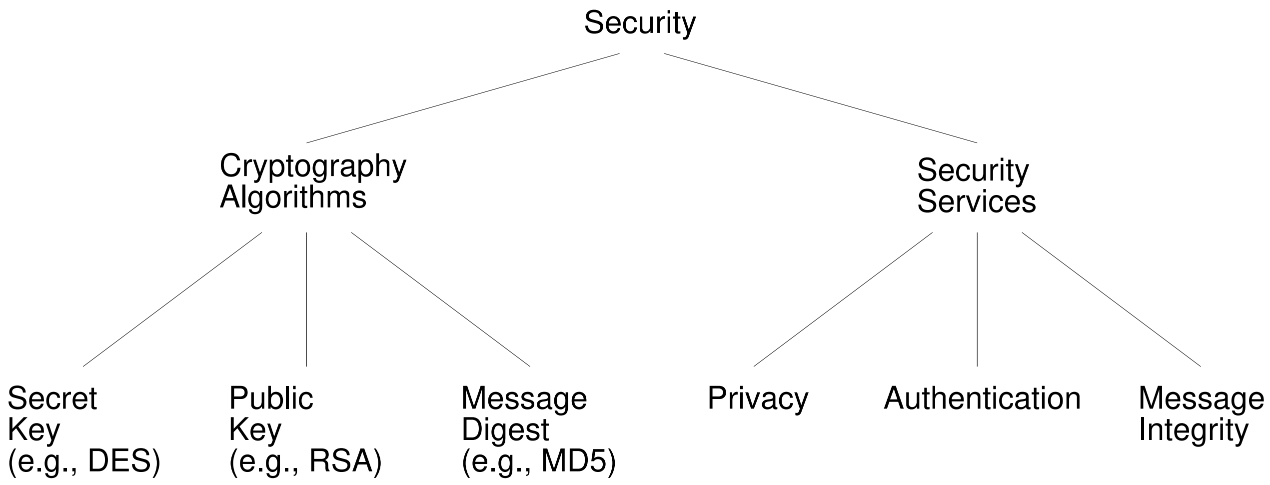


- Example sequence transmitted as I P B B I B B

- B and P frames
  - coordinate for the macroblock in the frame
  - motion vector relative to previous reference frame
  - motion vector relative to subsequent reference frame (B only)
  - delta for each pixel in the macro block
- Effectiveness
  - typically 90-to-1
  - as high as 150-to-1
  - 30-to-1 for I frames
  - P & B frames get another 3 to 5$\times$

Security

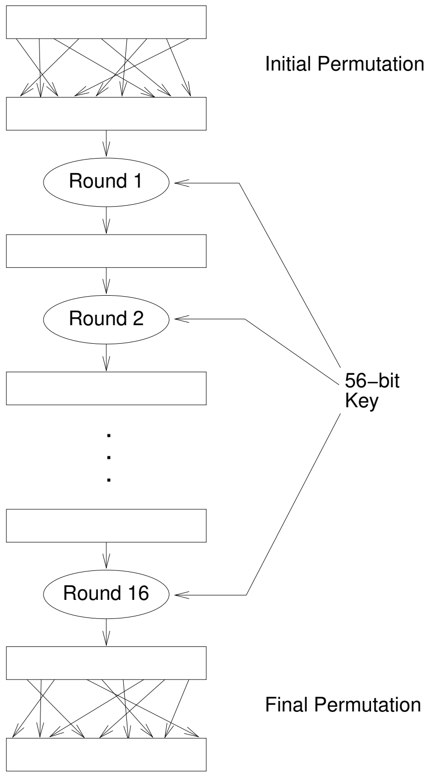
- Cryptography functions
  - Secret key (e.g., DES)
  - Public key (e.g., RSA)
  - Message digest (e.g., MD5)
- Security services
  - Privacy: preventing unauthorized release of information
  - Authentication: verifying identity of the remote participant
  - Integrity: making sure message has not been altered



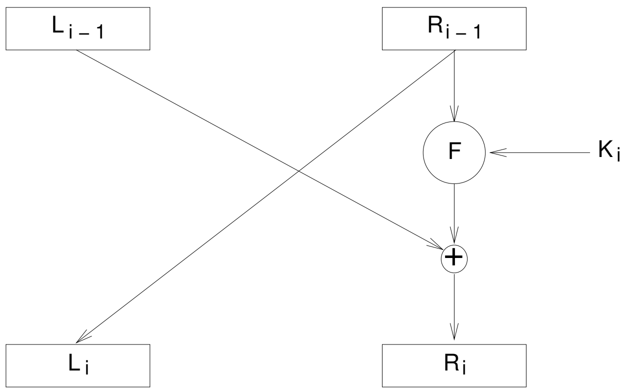
Encryption Algorithms

Private Key (DES)

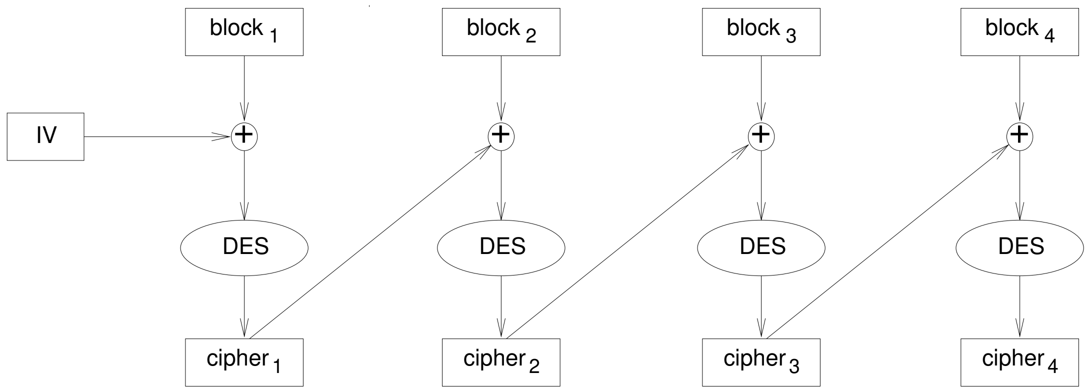
- 64-bit key (56-bits + 8-bit parity)
- 16 rounds



- Each round



- Function F and generation of K_i for each round not shown
- Repeat for larger messages



Public Key (RSA)

- Generate a public and private key
  - choose two large prime numbers p and q (each 256 bits)
  - multiply p and q together to get n
  - chose the encryption key e, such that e and $(p-1) \times (q-1)$ are relatively prime.
  - two numbers are relatively prime if they have no common factor greater than one.
  - compute decryption key d such that

$$d = e^{-1} mod \; ((p-1) \times (q-1))$$

  - construct public key as $\langle e, n \rangle$
  - construct private key as $\langle d, n \rangle$
  - discard (do not disclose) original primes p and q
- Encryption & Decryption

$$c = m^e mod \; n$$

$$m = c^d mod \; n$$

Message Digest

- Cryptographic checksum

  just as a regular checksum protects the receiver from accidental changes to the message, a cryptographic checksum protects the receiver from malicious changes to the message.

- One-way function

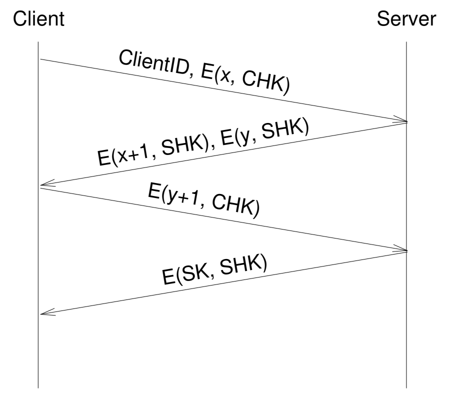
  given a cryptographic checksum for a message, it is virtually impossible to figure out what message produced that checksum; it is not computationally feasible to find two messages that hash to the same cryptographic checksum.

- Relevance

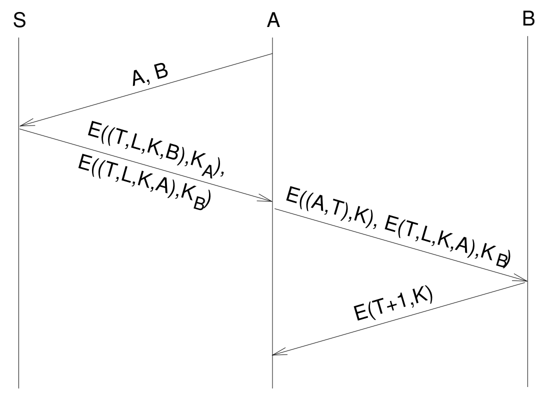
  if you are given a checksum for a message and you are able to compute exactly the same checksum for that message, then it is highly likely this message produced the checksum you were given.

Authentication Protocols

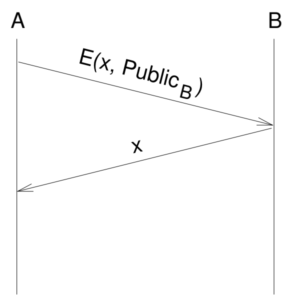
- Three-way handshake



- Trusted third party (Kerberos)



- Public key authentication



Message Integrity Protocols

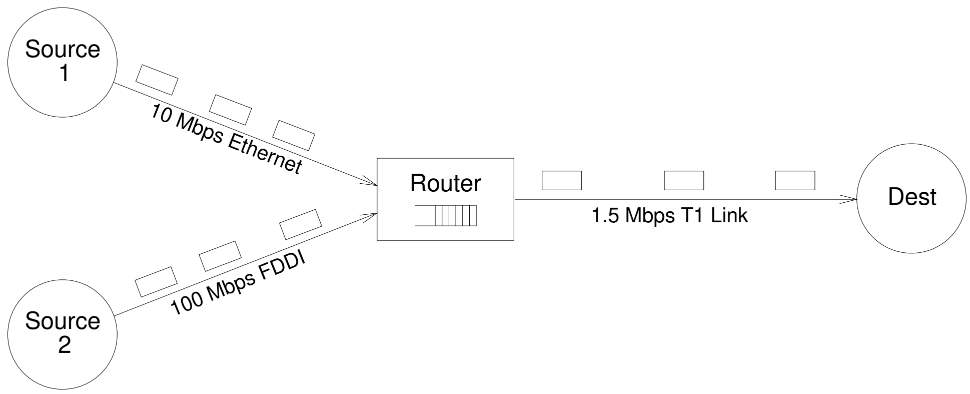
- Digital signature using RSA
  - special case of a message integrity where the code can only have been generated by one participant
  - compute signiture with private key and verify with public key
- Keyed MD5
  - sender

    m + MD5(m + k) + E(k, *private*)
  - receiver
    * recovers random key using the sender's public key
    * applies MD5 to the concatenation of this random key message
    * compares result with checksum sent with message
- MD5 with RSA signature
  - sender

    m + MD5(m) + E(MD5(m), *private*)
  - receiver
    * decrypts signature with sender's pubic key
    * compares result with MD5 checksum sent with message

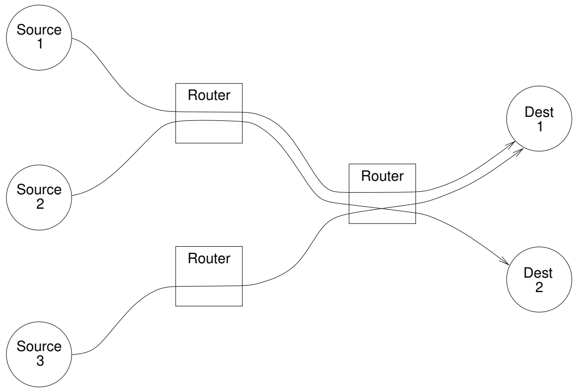
Congestion Control Issues

- Two sides of the same coin
  - pre-allocate resources so at to avoid congestion
  - send data and control congestion if (and when) is occurs

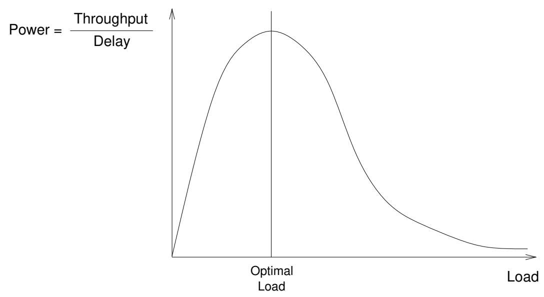


- Two points of implementation
  - hosts at the edges of the network (transport protocol)
  - routers inside the network (queuing discipline)
- Underlying service model
  - best-effort (assume for now)
  - multiple *qualities of service* (later)

- Connectionless flows
  - sequence of packets sent between source/destination pair
  - maintain *soft state* at the routers

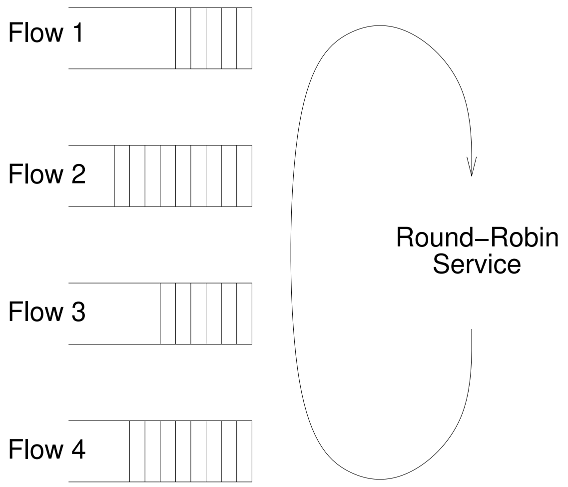


- Taxonomy
  - router-centric versus host-centric
  - reservation-based versus Feedback-based
  - window-based versus rate-based
- Evaluation
  - fairness
  - power (ratio of throughput to delay)



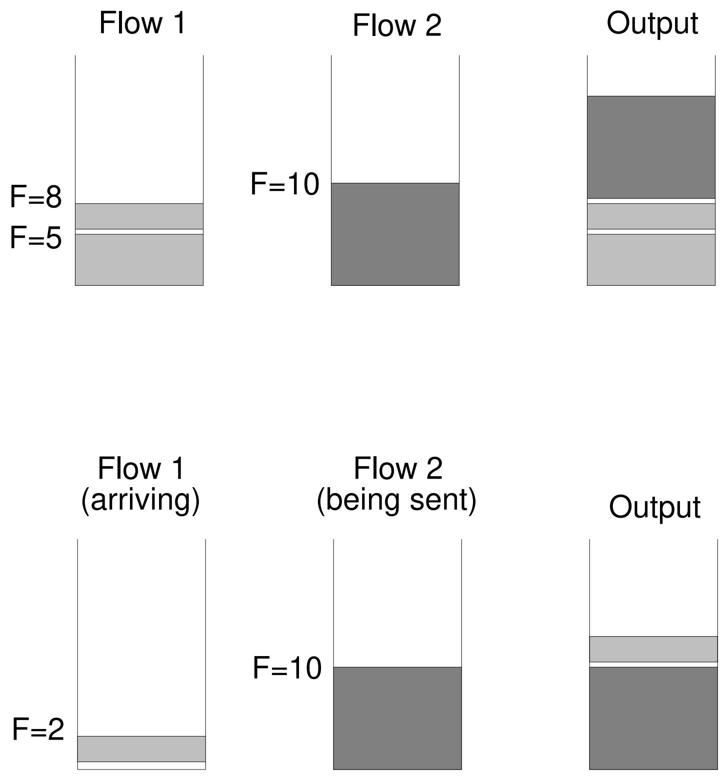
Queuing Disciplines

- First-In-First-Out (FIFO)
  - does not discriminate between traffic sources
- Fair Queuing (FQ)
  - explicitly segregates traffic based on flows
  - ensures no flow captures more than its share of capacity
  - variation: weighted fair queuing (WFQ)



- Problem: packets not all the same length
  - really want bit-by-bit round robin
  - not feasible to interleave bits (schedule on packet basis)
  - simulate by determining when packet would finish
- For a single flow
  - suppose clock ticks each time a bit is transmitted
  - let P_i denote the length of packet i
  - let S_i denote the time when start to transmit packet i
  - let F_i denote the time when finish transmitting packet i
  - $F_i = S_i + P_i$
  - When does router start transmitting packet i?
    * If before router finished packet $i - 1$ from this flow, then immediately after last bit of $i - 1$ (F_{i-1})
    * If no current packets for this flow, then start transmitting when arrives (call this A_i)
  - Thus: $F_i = MAX(F_{i-1}, A_i) + P_i$
- For multiple flows
  - calculate F_i for each packet that arrives on each flow
  - treat all F_i's as timestamps
  - next packet to transmit is one with lowest timestamp

- Not perfect: can't preempt the packet currently being transmitted
- Example



TCP Congestion Control

- Idea
  - assumes best-effort network (FIFO or FQ routers)
  - each source determines network capacity for itself
  - uses implicit feedback
  - ACKs pace transmission (*self-clocking*)
- Challenge
  - determining the available capacity in the first place
  - adjusting to changes in the available capacity

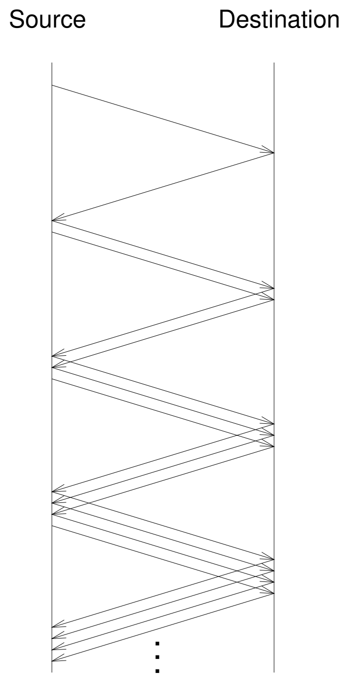
Additive Increase/Multiplicative Decrease

- Objective: adjust to changes in the available capacity
- New state variable per connection: `CongestionWindow`
  - limits how much data source has in transit

```
MaxWin = MIN(CongestionWindow, AdvertisedWindow)
EffWin = MaxWin - (LastByteSent - LastByteAcked)
```

- Idea:
  - increase `CongestionWindow` when congestion goes down
  - decrease `CongestionWindow` when congestion goes up
- Question: how does the source determine whether or not the network is congested?
- Answer: a timeout occurs
  - timeout signals that a packet was lost
  - packets are seldom lost due to transmission error
  - lost packet implies congestion

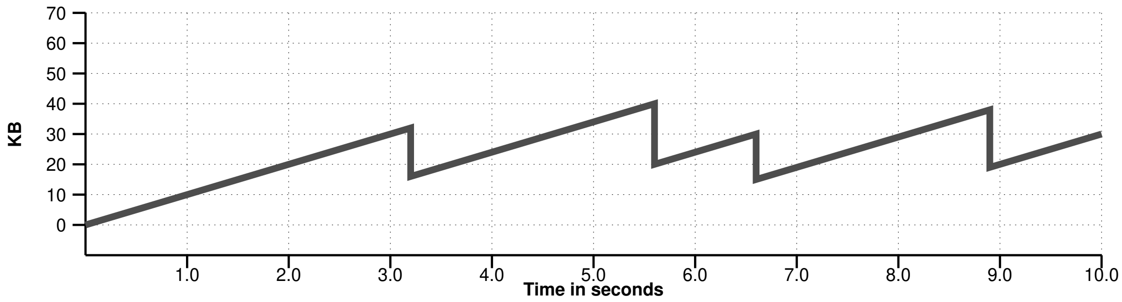
- Algorithm:
  - increment `CongestionWindow` by one packet per RTT (*linear increase*)
  - divide `CongestionWindow` by two whenever a timeout occurs (*multiplicative decrease*)



- In practice: increment a little for each ACK

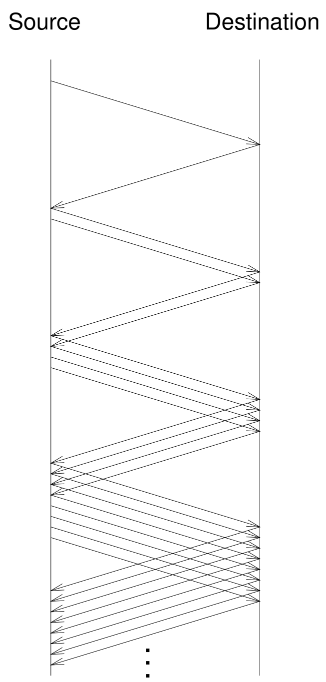
```
Increment = (MSS * MSS)/CongestionWindow
CongestionWindow += Increment
```

- Example trace: sawtooth behavior



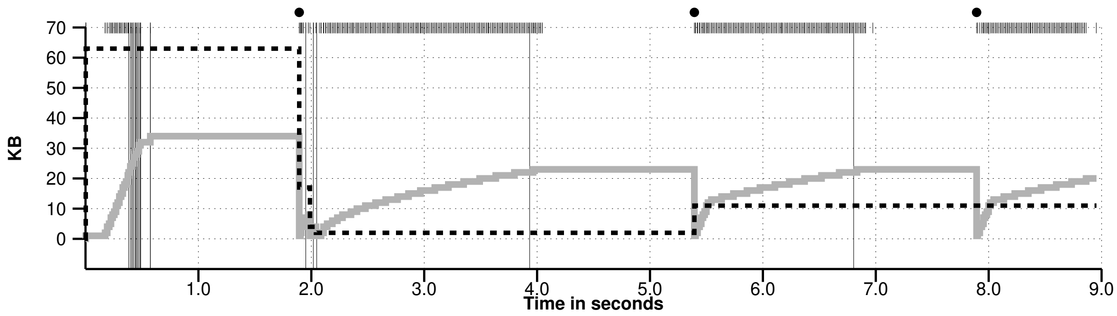
Slow Start

- Objective: determine the available capacity in the first place
- Idea:
  - begin with `CongestionWindow` = 1 packet
  - double `CongestionWindow` each RTT (increment by 1 packet for each ACK)



- Exponential growth, but slower than all in one blast
- Used...
  - when first starting connection
  - when connection goes dead waiting for a timeout

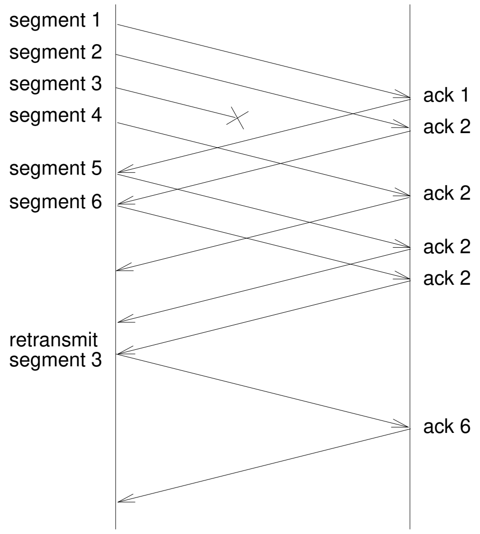
- Example trace



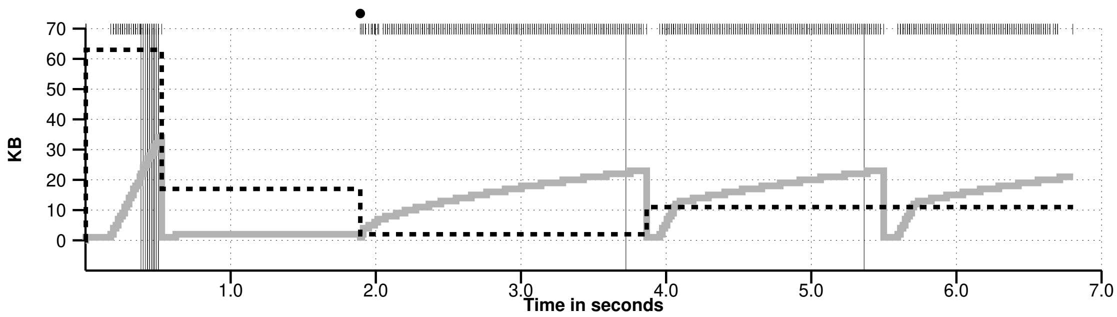
- Problem: lose up to half a `CongestionWindow`'s worth of data

Fast Retransmit and Fast Recovery

- Problem: coarse-grain TCP timeouts lead to idle periods
- Fast retransmit: use duplicate ACKs to trigger retransmission



- Results



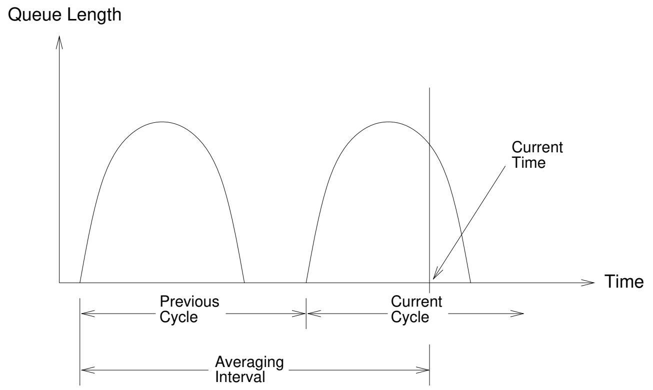
- Fast recovery: remove the slow start phase; go directly to half the last successful `CongestionWindow`

Congestion Avoidance Mechanisms

- TCP's strategy
  - to control congestion once it happens
  - to repeatedly increase load in an effort to find the point at which congestion occurs, and then back off
- Alternative strategy
  - predict when congestion is about to happen, and reduce the rate at which hosts send data just before packets start being discarded
  - we call this congestion *avoidance*, to distinguish it from congestion *control*
- Two possibilities
  - router-centric: DECbit and RED Gateways
  - host-centric: TCP Vegas

DECbit

- Add binary congestion bit to each packet header
- Router
  - monitors average queue length over last busy+idle cycle



  - set congestion bit if average queue length greater than 1 when packet arrives
  - attempts to balance throughput against delay
- End Hosts
  - destination echos bit back to source
  - source records how many packets resulted in set bit
  - if less than 50% of last window's worth had bit set, then increase `CongestionWindow` by 1 packet
  - if 50% or more of last window's worth had bit set, then decrease `CongestionWindow` by 0.875 times

Random Early Detection (RED) Gateways

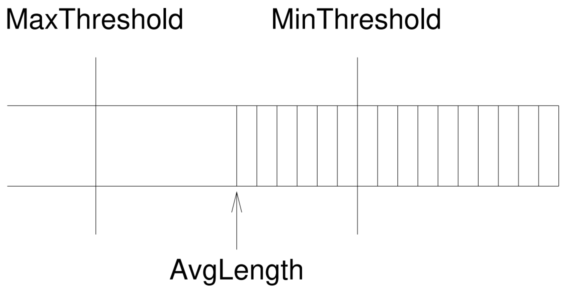
- Notification is implicit
  - just drop the packet (TCP will timeout)
  - could make explicit by marking the packet
- Early random drop
  - rather than wait for queue to become full, drop each arriving packet with some *drop probability* whenever the queue length exceeds some *drop level*
- RED: fills in the details
  - compute average queue length

```
AvgLen = (1 - Weight) * AvgLen
          + Weight * SampleLen
```

    * $0 <$ `Weight` < 1 (usually 0.002)
    * `SampleLen` is queue length each time a packet arrives

– two queue length thresholds

```
if AvgLen <= MinThreshold then
   enqueue the packet
if MinThreshold < AvgLen < MaxThreshold
   calculate probability P
   drop arriving packet with probability P
if MaxThreshold <= AvgLen
   drop arriving packet
```



– probability P

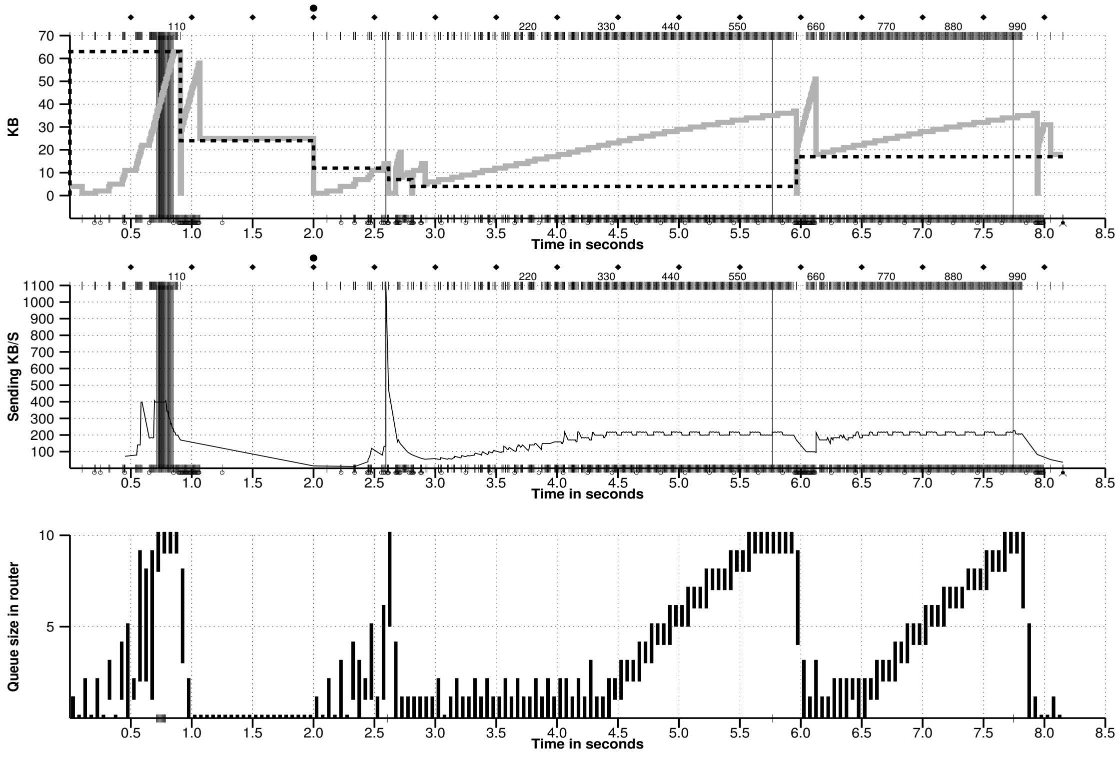
  * not fixed
  * function of `AvgLen` and how long since last drop (`count`) keeps track of new packets that have been queued while `AvgLen` has been between the two thresholds

```
TempP = MaxP * (AvgLen - MinThreshold)
        / (MaxThreshold - MinThreshold)
P = TempP/(1 - count * TempP)
```

- Notes
  - probability of dropping a particular flow's packet(s) is roughly proportional to the share of the bandwidth that flow is currently getting
  - `MaxP` is typically set to 0.02, meaning that when the average queue size is halfway between the two thresholds, the gateway drops roughly one out of 50 packets.
  - if traffic is bursty, then `MinThreshold` should be sufficiently large to allow link utilization to be maintained at an acceptably high level
  - difference between two thresholds should be larger than the typical increase in the calculated average queue length in one RTT; setting `MaxThreshold` to twice `MinThreshold` is reasonable for traffic on today's Internet

TCP Vegas

- Idea: source watches for some sign that some router's queue is building up and congestion will happen soon; e.g.,
  - RTT is growing
  - sending rate flattens
- Intuition



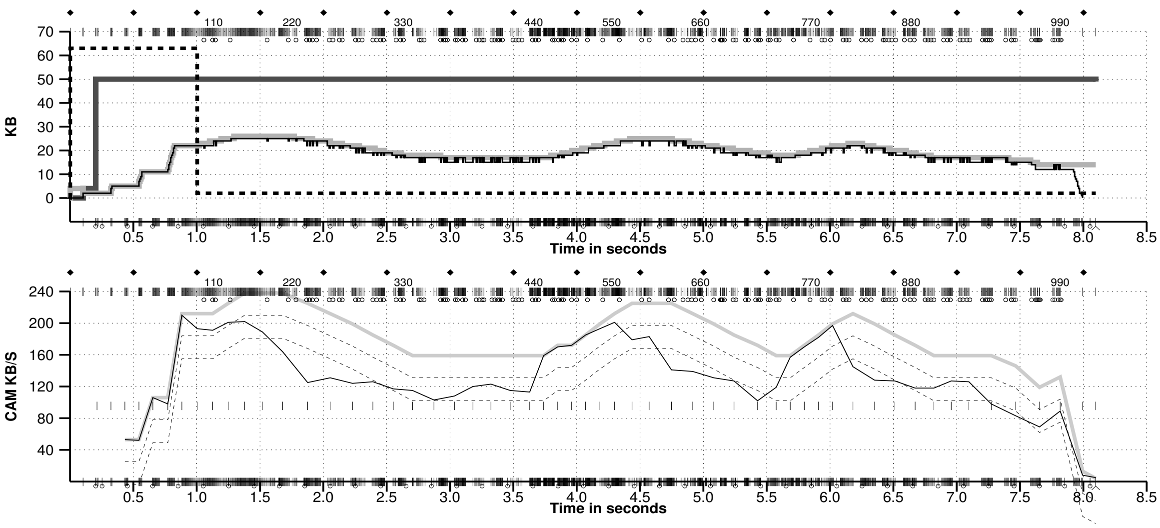
- Algorithm
  - let `BaseRTT` be the minimum of all measured RTTs (commonly the RTT of the first packet)
  - if not overflowing the connection, then

    `ExpectedRate = CongestionWindow / BaseRTT`

  - source calculates current sending rate (`ActualRate`) once per RTT
  - source compares `ActualRate` with `ExpectedRate`

```
Diff = ExpectedRate - ActualRate
if Diff < α
⟶ increase CongestionWindow linearly
else if Diff > β
⟶ decrease CongestionWindow linearly
else
⟶ leave CongestionWindow unchanged
```

- Parameters
  - α: 1 packet
  - β: 3 packets
- Example trace



- Even faster retransmit
  - keep fine-grained timestamps for each packet
  - check for timeout on first duplicate ACK

Virtual Clock

- Different approach
  - uses reservations rather than feedback
  - rate-based rather than window-based
  - router-centric rather than host-centric
- Idea
  - modeled after time-division multiplexing (TDM)
  - statistical multiplexing to accommodate bursty traffic
  - uses logical (rather than real) time

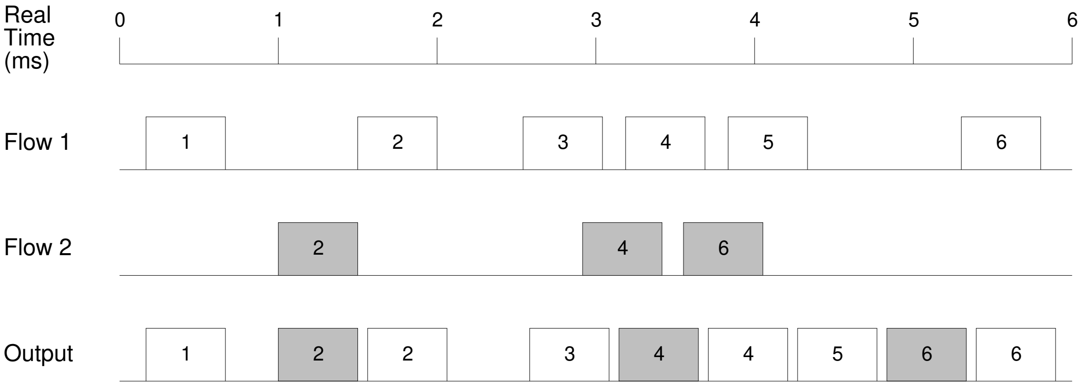
Defining Rate

- Explicit flow setup phase
  - source indicates its needs
  - network grants or denies request
- Resource needs
  - average rate (`AR`)
  - average interval (`AI`)
  - example: `AR` = 10 packets per second and `AI` = 100ms
  - range: `1/AR` $\leq$ `AI` $\leq$ total flow duration
  - datagram traffic gets it's own flow; granted some capacity

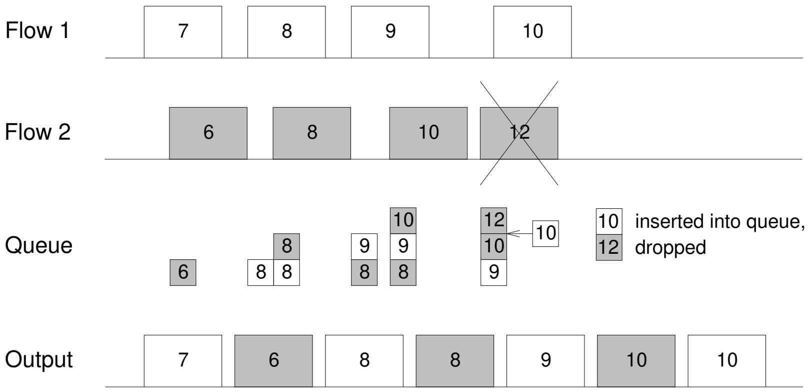
Virtual Clock as a Queuing Discipline

- Similar to weighted fair queuing
- Router variables
  - `VClock`$_i$ for each flow i
  - clock tick (`VTick`$_i$) for flow i
  - `VTick`$_i$ `= 1/AR`$_i$ (assumes all packets on flow i are the same size)
  - example: flow i has `AR` $= 200$ packets per second, then `VTick`$_i = 5$ms
- Router algorithm
  - first packet: `VClock`$_i$ $=$ `RealTime`
  - each packet thereafter: `VClock`$_i$ `+=` `VTick`$_i$
  - timestamp each packet with `VClock`$_i$
  - packets queued and serviced in according to timestamps
  - when buffer space is full, packet with largest timestamp is dropped

- Example of interleaved service proportional to AR
  - flow 1: AR = 1000 packets per second
  - flow 2: AR = 500 packets per second



- Example of discarding a packet



Virtual Clock as a Flow Meter

- Note
  - have not yet usee `AI`
  - possible for flow to "save up" credits and use for burst
- Additional mechanism
  - when setup flow i, compute $\texttt{AIR}_i$ `=` $\texttt{AR}_i \times \texttt{AI}_i$
  - upon receiving $\texttt{AIR}_i$ bytes on flow i...
    - if $\texttt{VClock}_i$ `-` `RealTime` $>$ `Threshold`, then send advisory to source
    - if $\texttt{VClock}_i <$ `RealTime`, then reset $\texttt{VClock}_i$ to `RealTime`
  - source not allowed to accumulate "unused time" during one `AI` period and use it in another

- Last detail
  - source can still accumulate credit *within* an `AI`
  - problem: using one variable (`VClock`) to control both queuing order and flow monitoring
  - solution: introduce auxiliary copy of `VClock`
  - upon receiving each packet...

$\texttt{AuxVClock}_i$ `= MAX(RealTime,` $\texttt{AuxVClock}_i$`)`

$\texttt{VClock}_i$ `=` $\texttt{VClock}_i$ `+` $\texttt{VTick}_i$

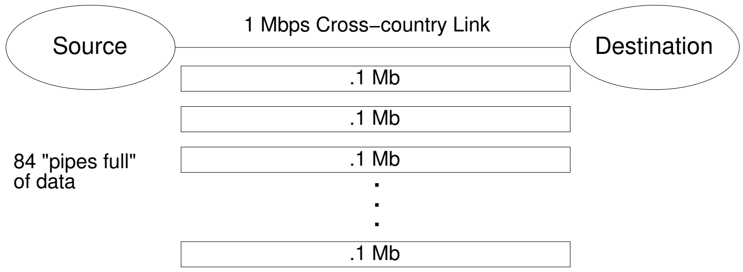
$\texttt{AuxVClock}_i$ `=` $\texttt{AuxVClock}_i$ `+` $\texttt{VTick}_i$

`Stamp packet with current value of` $\texttt{AuxVClock}_i$

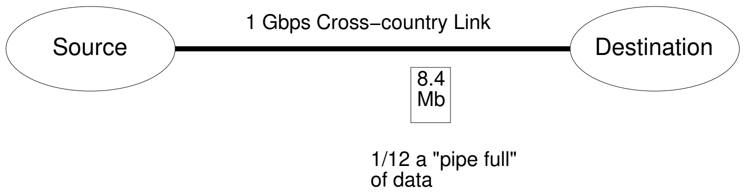
- Thus
  - $\texttt{VClock}_i$ resynchronized with real time once for every $\texttt{AIR}_i$ bytes of data on flow i
  - $\texttt{AuxVClock}_i$ resynchronized with real time once for every packet on flow i

High-Speed Networks

- 1Mbps and 1Gbps cross-country links have the same latency
  - limited by the speed of light
- To transfer a 1MB file takes...
  - 100 RTTs on a 1Mbps link
  - doesn't fill a 1Gbps link (12.5MB delay $\times$ bandwidth)



(a)



(b)

- Said another way:
  - 1MB file is to 1Gbps network what 1KB packet is to 1Mbps network

Latency/Bandwidth Tradeoff

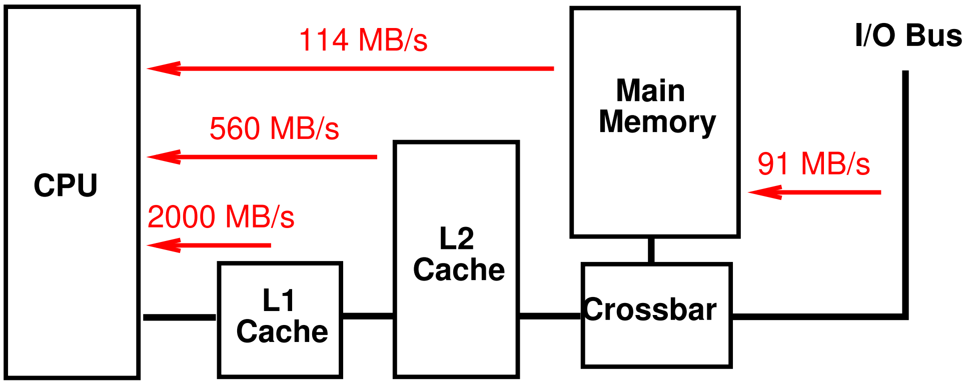
- `Throughput = TransferSize / TransferTime`
  - if it takes 10ms to transfer 1MB, then the effective throughput is 1MB/10ms = 100MBps = 800Mbps
- `TransferTime = Latency + 1/Bandwidth` $\times$ `TransferSize`
  - if network bandwidth is 1Gbps (it takes 1/1Gbps $\times$ 1MB = 0.8ms to transmit 1MB), an end-to-end transfer that requires 1 RTT of 100ms has a total transfer time of 100.8ms
  - effective throughput is 1MB/100.8ms = 79.4Mbps, not 1Gbps
- Notes
  - transferring a large amount of data helps improve the effective throughput; in the limit, an infinitely large transfer size causes the effective throughput to approach the network bandwidth
  - having to endure more than one RTT will hurt the effective throughput for any transfer of finite size, and will be most noticeable for small transfers

Implications

- Congestion control
  - feedback based mechanisms require an RTT to adjust
  - can send 10MB in one 100ms RTT on a 1Gbps network
  - that 10MB might congest a router and lead to massive losses
  - can lose half a delay $\times$bandwidth's of data during slow start
  - reservations work for continuous streams (e.g., video), but require an extra RTT for bulk transfers
- Retransmissions
  - retransmitting a packet costs 1 RTT
  - dropping even one packet (cell) halves effective bandwidth
  - retransmission also implies buffering at the sender
  - possible solution: forward error correction (FEC)
- Trading bandwidth for latency
  - each RTT is precious
  - willing to "waste" bandwidth to save latency
  - example: prefetching

Host Memory Bottleneck

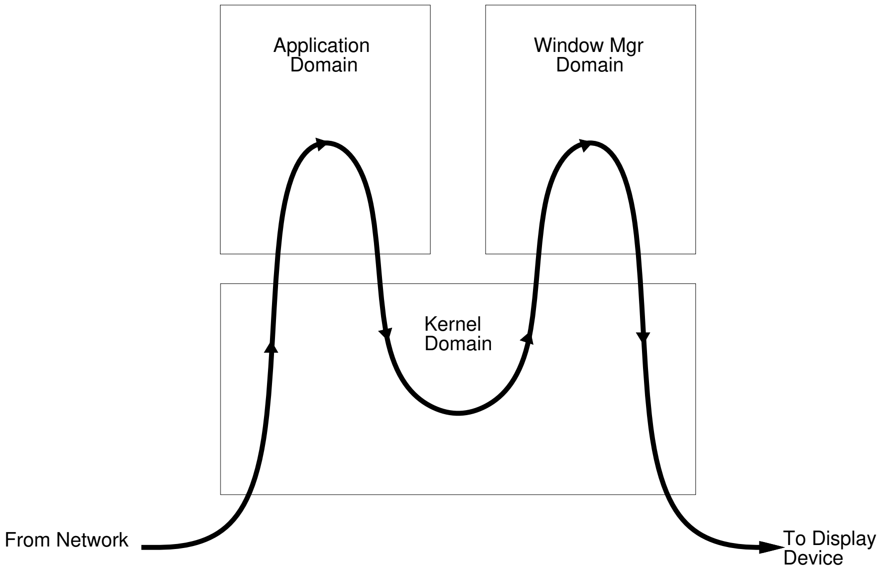
- Issue
  - turning host-to-host bandwidth into application-to-application bandwidth
  - have to deliver data across I/O and memory buses into cache and registers



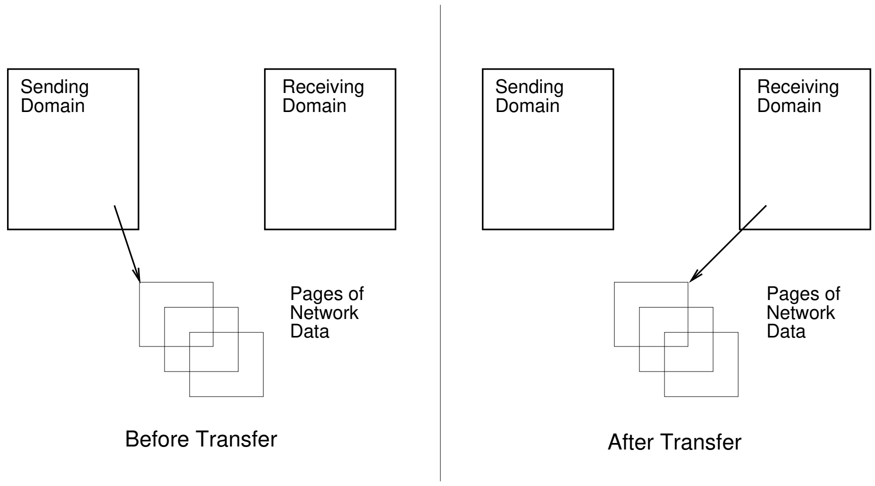
- Memory bandwidth
  - I/O bus must keep up with network speed (currently does for STS-12, assuming peak rate is achievable)
  - 114MBps (measured number) is only slightly faster than I/O bus; can't afford to go across memory bus twice
  - caches are of questionable value (rather small)
  - lots of reason to access buffers
    * user/kernel boundary
    * certain protocols (reassembly, checksumming)
    * network device and its driver
- Same latency/bandwidth problems as high-speed networks

Techniques for Avoiding Data Transfers

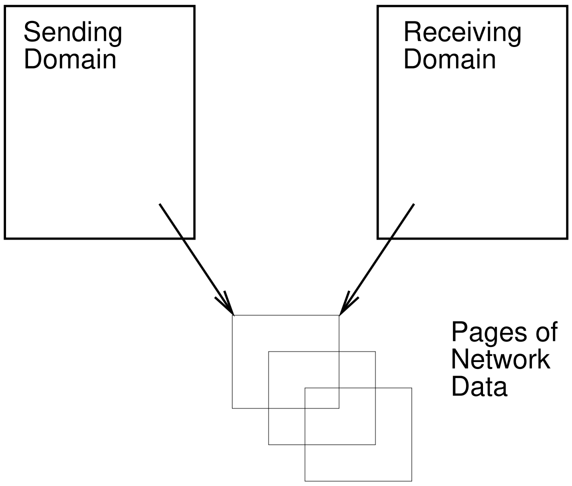
- Device/memory transfers
  - mechanism: DMA vs PIO
  - transfer size: cell vs packet
- Cross-domain transfers



– page remapping



– shared memory



– dynamic page sharing (fbufs)

- Data manipulations
  - compression, encryption, checksum, presentation formatting
  - involves memory loads and stores
  - integrated layer processing (ILP)

```
for( i = 0; i < 10000; i++ )
      msgData[i]++;                    /* LOAD, ADD, STORE */

for( i = 0; i < 10000; i++ )
      msgData[i] = ~msgData[i];        /* LOAD, COMPLEMENT, STORE */
```

(a) Two For−Loops

```
for( i = 0; i < 10000; i++ ){
      temp = msgData[i];               /* LOAD */
      temp++;                          /* ADD */
      temp = ~temp;                    /* COMPLEMENT */
      msgData[i] = temp;               /* STORE */
}
```

(b) Integrated For−Loops

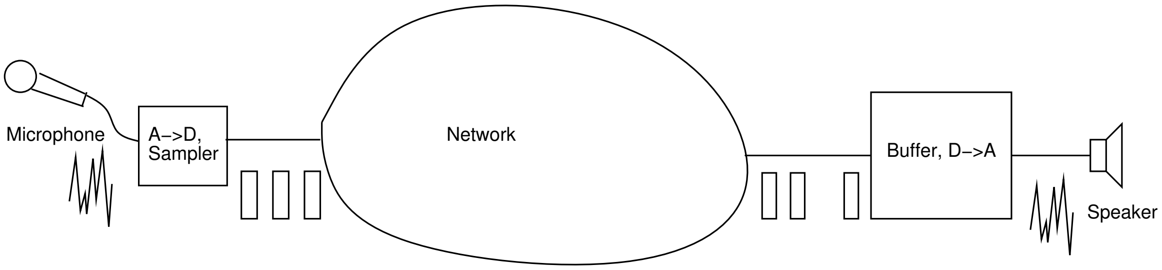
- API design
  - write: application reuses buffer
  - read: application specifies buffer

Integrated Services

- High-speed networks have enabled new applications
  - they also need "deliver on time" assurances from the network
- Applications that are sensitive to the timeliness of data are called *real-time* applications
  - voice
  - video
  - industrial control
- Timeliness guarantees must come from *inside* the network
  - end-hosts cannot correct late packets like they can correct for lost packets
- Need more than best-effort
  - IETF is standardizing extensions to best-effort model

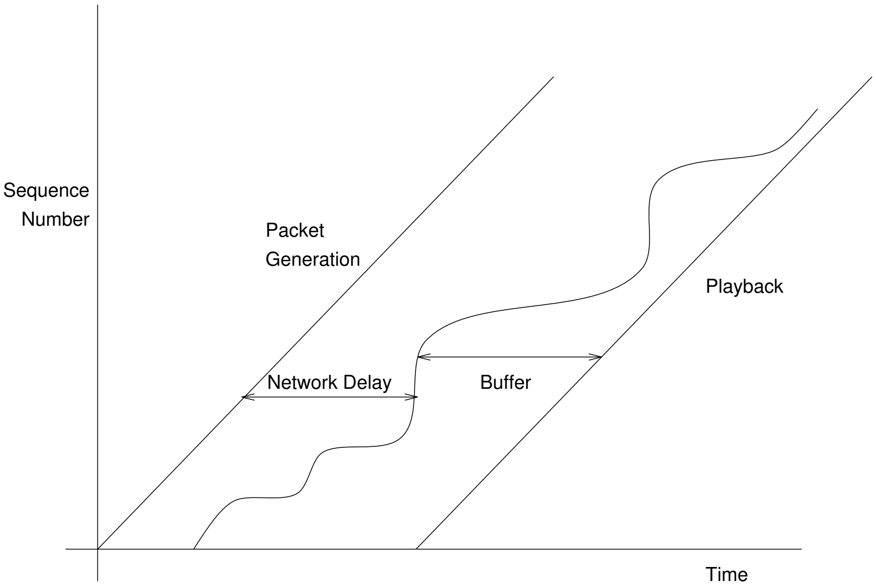
Service Model

- Two types of applications
  - real-time
  - elastic
- Example real-time application (audio)

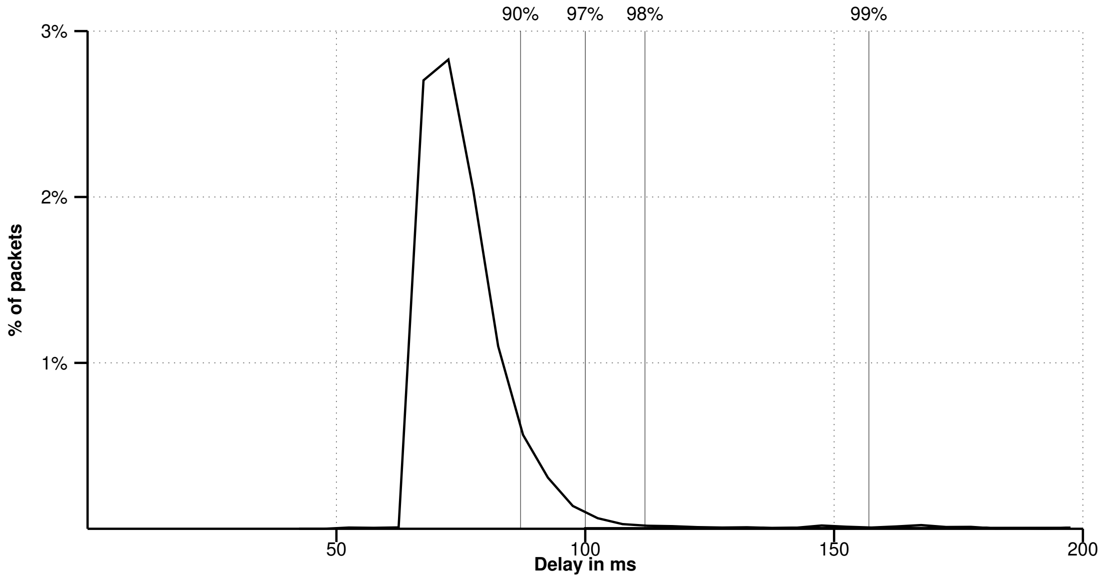


  - sample voice once every 125μs
  - each sample has a *playback time*
  - packets experience variable delay in network
  - add constant factor to playback time: *playback point*

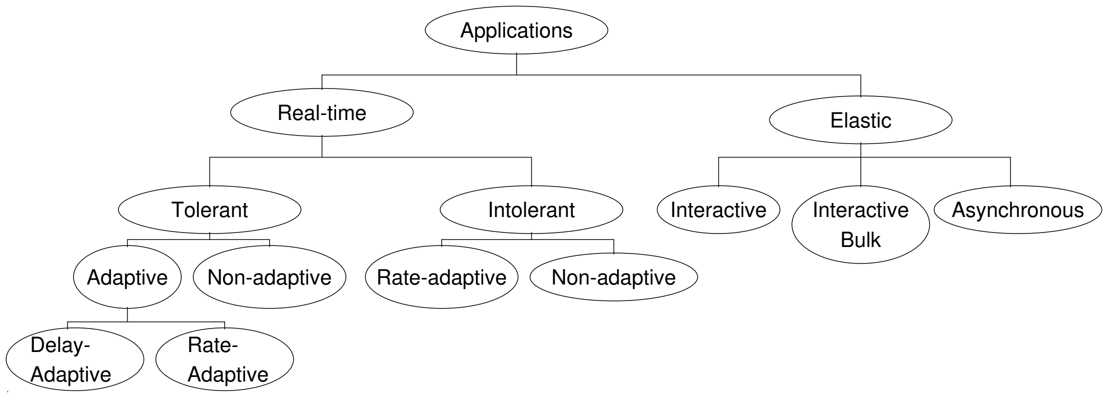
- Playback buffer



- Example distribution of delays



- Taxonomy of real-time applications

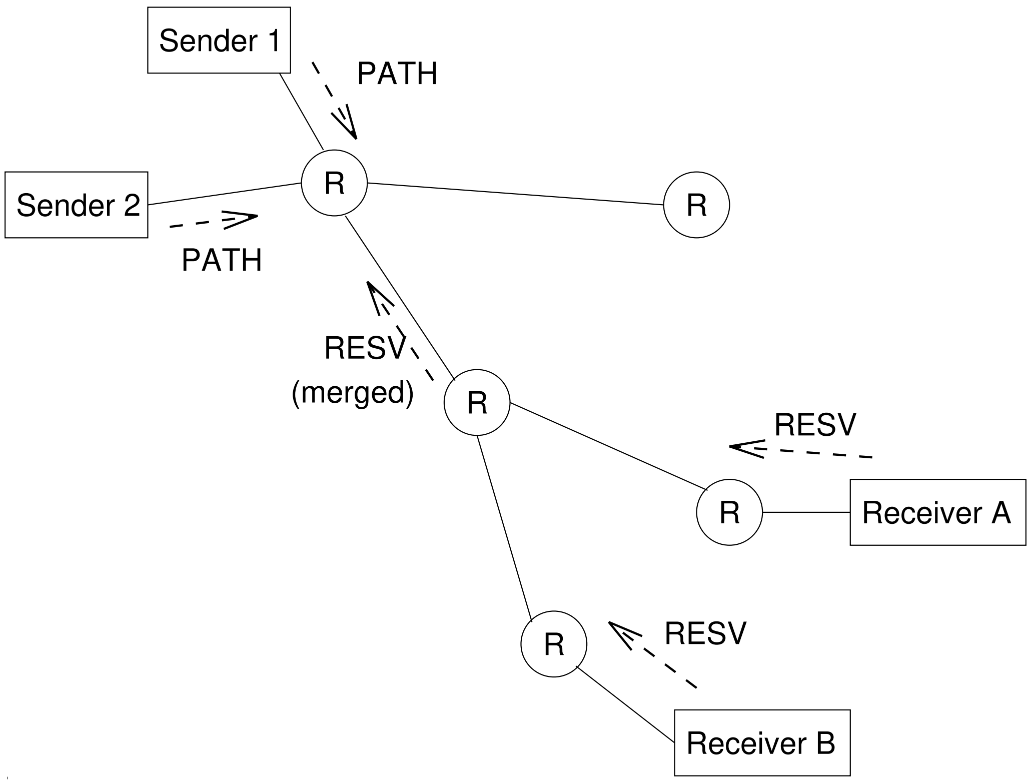


- Service classes
  - guaranteed
  - predictive
  - controlled-delay

Mechanisms

- Flowspecs
  - *Rspec*: describes service requested from network
    - controlled-delay: level of delay required
    - guaranteed/predictive: delay target
  - *Tspec*: describes flow's traffic characteristics
    - average bandwidth + burstiness: *token bucket* filter
    - token rate r
    - bucket depth B
    - must have a token to send a byte
    - must have n tokens to send n bytes
    - start with no tokens
    - accumulate tokens at rate of r per second
    - can accumulate no more than B tokens

- Admission Control
  - decide if a new flow can be supported
  - answer depends on service class
  - not the same as *policing*
- Reservation Protocol
  - called signalling in ATM
  - proposed Internet standard: RSVP
  - consistent with robustness of today's connectionless model
  - uses soft state (refresh periodically)
  - designed to support multicast
  - is receiver oriented
  - two messages: PATH and RESV
  - source transmits PATH messages sent every 30 seconds
  - destination responds with RESV message
  - merge requirements in case of multicast
  - can specify number of speakers


Sender 1
PATH
Sender 2
R
R
PATH
RESV
(merged)
R
RESV
R
Receiver A
R
RESV
Receiver B

- Packet Classifying and Scheduling
  - associate each packet with the appropriate reservation so that it can be handled correctly, a process known as *classifying* packets
  - manage the packets in the queues so that they receive the service that has been requested, a process known as packet *scheduling*
- RSVP versus ATM (Q.2931)
  - RSVP
    - receiver generates reservation
    - soft state (refresh/timeout)
    - separate from route establishment
    - QoS can change dynamically
    - receiver heterogeneity
  - ATM
    - sender generates connection request
    - hard state (explicit delete)
    - concurrent with route establishment
    - QoS is static for life of connection
    - uniform QoS to all receivers