EEC-484/584
Computer Networks

Lecture 15

Wenbing Zhao

wenbing@ieee.org

(Lecture nodes are based on materials supplied by
Dr. Louise Moser at UCSB and Prentice-Hall)

Outline

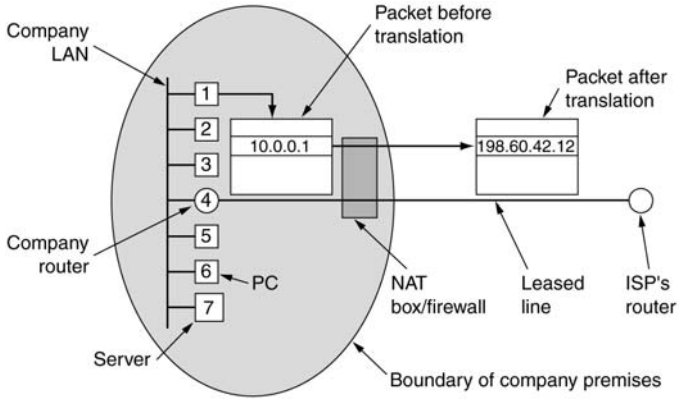
- Review of last lecture
  - The network layer in Internet
- The transport layer
  - Transport service
  - Elements of transport protocols

Review

- Essential topics of last lecture
  - NAT
  - ICMP, ARP, RARP, BOOTP, DHCP
  - OSPF, BGP

NAT – Network Address Translation

Placement and operation of a NAT box

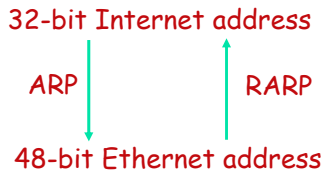


ICMP – Internet Control Message Protocol

- When something unexpected occurs in Internet, the event is reported by routers using ICMP
- It is also used to test Internet
- Principal ICMP message types

Message type	Description
Destination unreachable	Packet could not be delivered
Time exceeded	Time to live field hit 0
Parameter problem	Invalid header field
Source quench	Choke packet
Redirect	Teach a router about geography
Echo request	Ask a machine if it is alive
Echo reply	Yes, I am alive
Timestamp request	Same as Echo request, but with timestamp
Timestamp reply	Same as Echo reply, but with timestamp

ARP and RARP

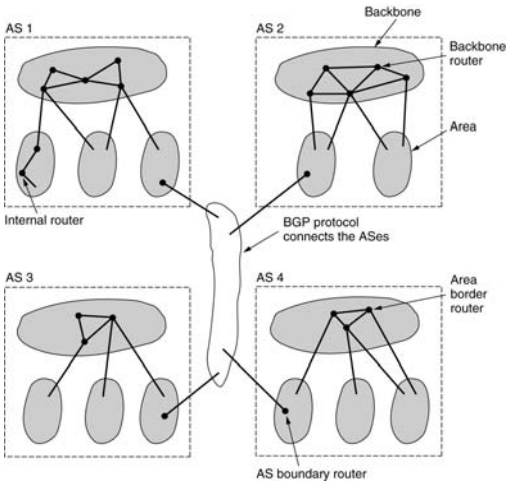


- ARP – find the mapping of IP addresses to data link layer addresses
- RARP - This protocol allows a newly-booted diskless-workstation (e.g., X terminal) to broadcast its Ethernet address and ask for its IP address

DHCP – Dynamic Host Configuration Protocol

- Allows both manual IP address assignment and automatic assignment. DHCP has largely replaced RARP and BOOTP
- A **DHCP relay agent** is needed on each LAN. The only piece of information the relay agent needs is the IP address of the DHCP server.
- To find its IP address, a newly-booted machine broadcasts a DHCP DISCOVER packet. The DHCP relay agent on its LAN receives all DHCP broadcasts
- When it finds a DHCP DISCOVER packet, it sends the packet as a unicast packet to the DHCP server, possibly on a distant network
- IP address assignment is lease-based (to cope with client failure)

Internet Routing Protocols

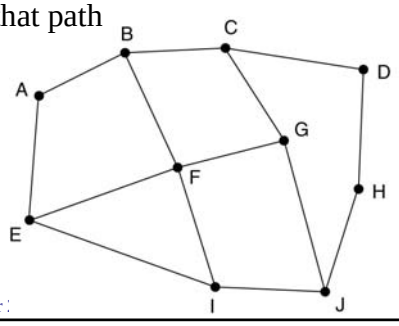


Interior Gateway Routing Protocol

- Uses Open Shortest Path First (OSPF)
  - Open, dynamic, and support multiple distance metrics
  - Routing based on type of service
  - Load balancing
  - Hierarchical
  - Security
  - Tunneling
- Supports 3 types of connections
  - Point-to-point between 2 routers
  - Multiaccess (multiple routers that communicate with each other) – with broadcasting and without broadcasting

Exterior Gateway Routing Protocol

- Border Gateway Protocol (BGP)
  - Used between autonomous systems
  - Main concerns: politics, security, economic
  - Uses distance vector routing except keeps track of exact path instead of cost to destination and periodically tells its neighbors that path



Information F receives from its neighbors about D

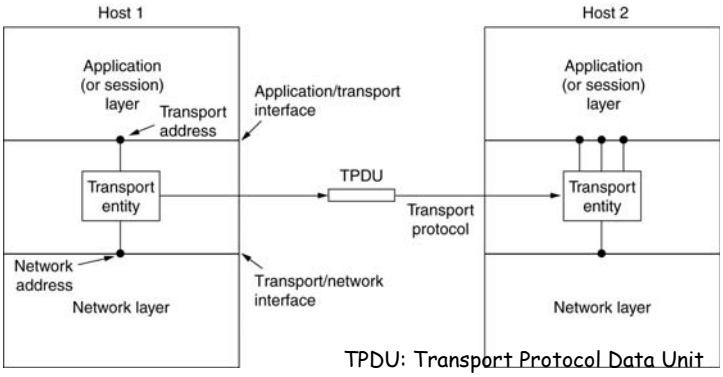
From B: "I use BCD"
From G: "I use GCD"
From I: "I use IFGCD"
From E: "I use EFGCD"

Transport Layer – Design Goals

- To provide efficient, reliable, cost-effective service to the transport layer users
- To allow application programs to be written using a standard set of primitives and to have programs work on a variety of networks
- To enhance the quality of service provided by the network layer below it

Services Provided to the Upper Layers

- **Transport Entity** - The hardware and/or software within the transport layer that does the work. The transport entity can be located in
  - Operating system kernel, a separate user process, a library package bound into network applications, or conceivably on the network interface card



Services Provided to the Upper Layers

- The transport layer fulfills the key function of isolating the upper layers from the technology, design, and imperfections of the subnet
- For this reason, many people have traditionally made a distinction between layers 1 through 4 and layer(s) above 4:
  - **Transport service provider** - The bottom four layers
  - **Transport service user** - the upper layer(s)

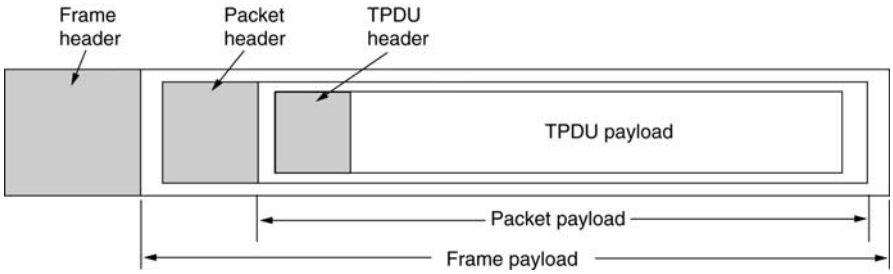
Transport Service Primitives

- Connection oriented
- Provides reliable service on top of unreliable network

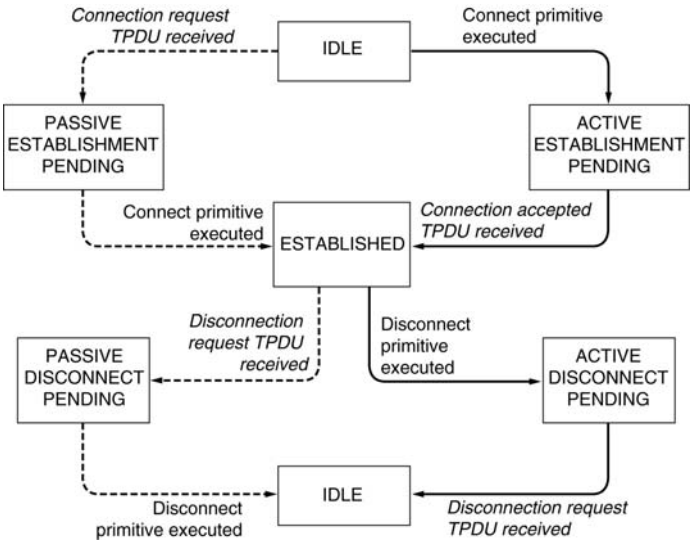
Primitive	Packet sent	Meaning
LISTEN	(none)	Block until some process tries to connect
CONNECT	CONNECTION REQ.	Actively attempt to establish a connection
SEND	DATA	Send information
RECEIVE	(none)	Block until a DATA packet arrives
DISCONNECT	DISCONNECTION REQ.	This side wants to release the connection

Transport Service Primitives

- The nesting of TPDUs, packets, and frames



Transport Service Primitives



A state diagram for a simple connection management scheme.

Transitions labeled in italics are caused by packet arrivals.

The solid lines show the client's state sequence.

The dashed lines show the server's state sequence.

Berkeley Sockets

- Socket – an endpoint to which connections can be attached from bottom (OS) and to which processes can be established from top
- Socket primitives for TCP

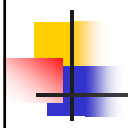
Primitive	Meaning
SOCKET	Create a new communication end point
BIND	Attach a local address to a socket
LISTEN	Announce willingness to accept connections; give queue size
ACCEPT	Block the caller until a connection attempt arrives
CONNECT	Actively attempt to establish a connection
SEND	Send some data over the connection
RECEIVE	Receive some data from the connection
CLOSE	Release the connection

Berkeley Sockets

- Common include:

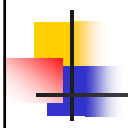
```
#include <sys/types.h>
#include <sys/socket.h>
```

- Socket: int socket(int domain, int type, int protocol);
  - Creates a new end point and allocates table space for it
  - Parameters:
    - Communication domain: PF_UNIX, PF_INET, PF_INET6, etc.
    - Socket type: SOCK_STREAM, SOCK_DGRAM, etc.
    - Protocol: specify a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular type within a given protocol family, in which case 0 is used



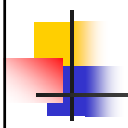
Berkeley Sockets

- Bind:
  int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
  - Each socket has name (local address) by which remote user can send connection request to socket
  - Bind call attaches the given name (local address my_addr) to the socket sockfd. my_addr is addrlen bytes long



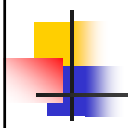
Berkeley Sockets

- Listen: int listen(int s, int backlog);
  - After a socket has been created, listen call allocates space to queue incoming calls in case several clients try to connect simultaneously
  - Listen call is non-blocking



Berkeley Sockets

- Accept:
  int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
  - Accept call is blocking, waits for incoming connection
  - When TPDU requesting a connection arrives, new socket is created and file descriptor is returned for it
  - Server can fork off process or thread to handle connection on new socket and wait for next connection on original socket
  - addr is a pointer to a sockaddr structure. It is filled in which the address of the connecting entity



Berkeley Sockets

- Connect:
  int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);
  - For a connection-based socket such as TCP socket
    - Blocks the caller (default setting) and actively starts the connection process
    - When it completes (appropriate TPDU is received from server), client is unblocked and connection is established
  - For a datagram socket, the specified server address is associated with the socket sockfd

Berkeley Sockets

- Send:

```
ssize_t send(int s, const void *msg, size_t len, int flags);
ssize_t  sendto(int  s,  const  void *msg, size_t len, int flags,
                    const struct sockaddr *to, socklen_t tolen);
ssize_t sendmsg(int s, const struct msghdr *msg, int flags);
```

  - Transmit a message msg to another socket
  - send may be used only when the socket is in a connected state
  - For connectionless communication, sendto and sendmsg should be used instead

Berkeley Sockets

- Receive:

```
ssize_t recv(int s, void *buf, size_t len, int flags);
ssize_t  recvfrom(int s, void *buf, size_t len, int flags, struct
                    sockaddr *from, socklen_t *fromlen);
ssize_t recvmsg(int s, struct msghdr *msg, int flags);
```

  - All three calls are used to receive messages from a socket. They return the length of the message on successful completion
  - If no messages are available at the socket, the receive calls wait for a message to arrive, unless the socket is nonblocking
  - The recv call is normally used only on a connected socket. It is identical to recvfrom with a NULL from parameter
  - The recvfrom and recvmsg calls may be used to receive data on a socket whether or not it is connection-oriented

Berkeley Sockets

- Close

```
#include <unistd.h>
int close(int fd);
```

  - The close call closes a file/socket descriptor, so that it no longer refers to any file/socket
  - When both sides have executed close, connection is released

Socket Programming Example

- Internet File Server
  - Code available at http://authors.phptr.com/tanenbaumcn4/programs/code.zip
  - Compilation flag in Linux is different from what's given in the textbook
  - **setsockopt**() on SO_REUSEADDR: "This socket option tells the kernel that even if this port is busy (in the TIME_WAIT state), go ahead and reuse it anyway. If it is busy, but with another state, you will still get an address already in use error. It is useful if your server has been shut down, and then restarted right away while sockets are still active on its port." quoted from http://www.unixguide.net/network/socketfaq/4.5.shtml

Socket Programming Example: Internet File Server (Client)


```
#define SERVER_PORT 12345          /* arbitrary, but client & server must agree */
#define BUF_SIZE 4096              /* block transfer size */

int main(int argc, char **argv)
{
  int c, s, bytes;
  char buf[BUF_SIZE];              /* buffer for incoming file */
  struct hostent *h;               /* info about server */
  struct sockaddr_in channel;      /* holds IP address */

  if (argc != 3) fatal("Usage: client server-name file-name");
  h = gethostbyname(argv[1]);      /* look up host's IP address */
  if (!h) fatal("gethostbyname failed");

  s = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
  if (s <0) fatal("socket");
  memset(&channel, 0, sizeof(channel));
  channel.sin_family= AF_INET;
  memcpy(&channel.sin_addr.s_addr, h->h_addr, h->h_length);
  channel.sin_port= htons(SERVER_PORT);
```


Socket Programming Example: Internet File Server (Client)


```
  c = connect(s, (struct sockaddr *) &channel, sizeof(channel));
  if (c < 0) fatal("connect failed");

  /* Connection is now established. Send file name including 0 byte at end. */
  write(s, argv[2], strlen(argv[2])+1);

  /* Go get the file and write it to standard output. */
  while (1) {
      bytes = read(s, buf, BUF_SIZE);      /* read from socket */
      if (bytes <= 0) exit(0);             /* check for end of file */
      write(1, buf, bytes);                /* write to standard output */
  }
}

fatal(char *string)
{
  printf("%s\n", string);
  exit(1);
}
```

Socket Programming Example: Internet File Server (Server)

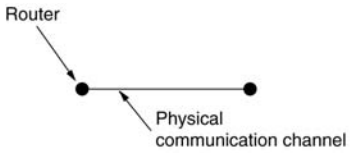

```
#define SERVER_PORT 12345          /* arbitrary, but client & server must agree */
#define BUF_SIZE 4096              /* block transfer size */
#define QUEUE_SIZE 10

int main(int argc, char *argv[])
{
  int s, b, l, fd, sa, bytes, on = 1;
  char buf[BUF_SIZE];              /* buffer for outgoing file */
  struct sockaddr_in channel;      /* hold's IP address */

  /* Build address structure to bind to socket. */
  memset(&channel, 0, sizeof(channel));  /* zero channel */
  channel.sin_family = AF_INET;
  channel.sin_addr.s_addr = htonl(INADDR_ANY);
  channel.sin_port = htons(SERVER_PORT);

  /* Passive open. Wait for connection. */
  s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);   /* create socket */
  if (s < 0) fatal("socket failed");
  setsockopt(s, SOL_SOCKET, SO_REUSEADDR, (char *) &on, sizeof(on));
```


Socket Programming Example: Internet File Server (Server)

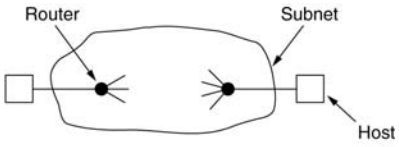

```
  b = bind(s, (struct sockaddr *) &channel, sizeof(channel));
  if (b < 0) fatal("bind failed");

  l = listen(s, QUEUE_SIZE);           /* specify queue size */
  if (l < 0) fatal("listen failed");

  /* Socket is now set up and bound. Wait for connection and process it. */
  while (1) {
      sa = accept(s, 0, 0);            /* block for connection request */
      if (sa < 0) fatal("accept failed");

      read(sa, buf, BUF_SIZE);         /* read file name from socket */

      /* Get and return the file. */
      fd = open(buf, O_RDONLY);        /* open the file to be sent back */
      if (fd < 0) fatal("open failed");

      while (1) {
          bytes = read(fd, buf, BUF_SIZE); /* read from file */
          if (bytes <= 0) break;       /* check for end of file */
          write(sa, buf, bytes);       /* write bytes to socket */
      }
      close(fd);                       /* close file */
      close(sa);                       /* close connection */
  }
}
```

Transport Protocol

- The transport service is implemented by a transport protocol used between two transport entities
- Relationship with data link protocols
  - **Similarities**: deal with error control, sequencing, flow control
  - **Difference**: operating environments



Environment of the data link layer



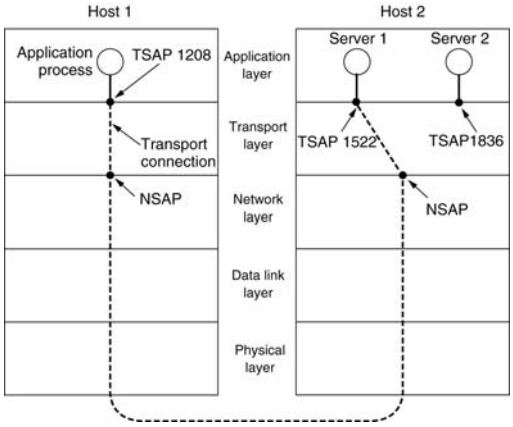
Environment of the transport layer

Connection Management

- Addressing
- Connection establishment
- Connection release

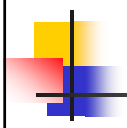
Addressing

- **Transport Service Access Points** (TSAPs) to which processes can attach and wait for connections to arrive



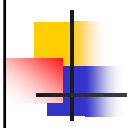
Addressing - A Scenario

- A time of day server process on host 2 attaches itself to TSAP 1522 to wait for an incoming call
- An application process on host 1 wants to find out the time-of-day, so it issues a CONNECT request specifying TSAP 1208 as the source and TSAP 1522 as the destination
- This action results in a transport connection being established between the application process on host 1 and server 1 on host 2.
- The application process then sends over a request for the time.
- The time server process responds with the current time
- The transport connection is then released



Connection Establishment

- Internet Initial Connection Protocol
  - Each machine has process server through which services are requested
  - Idle process server listens on its TSAP
  - User issues connection request specifying TSAP address of process server
  - Once connection established, user sends message to process server asking it to run a particular program

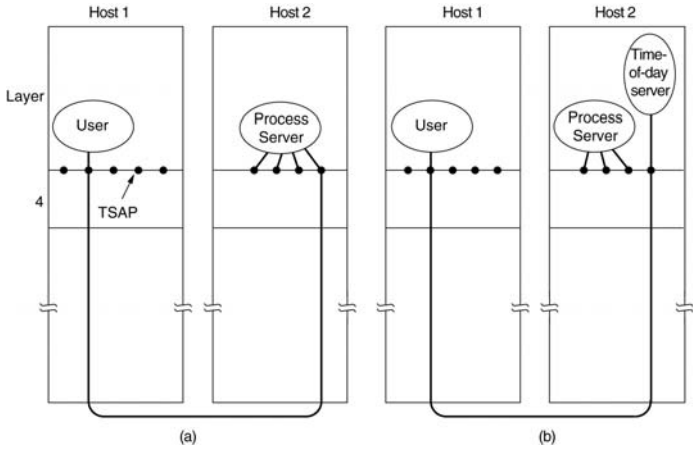


Connection Establishment

- Internet Initial Connection Protocol (cont'd)
  - Process server chooses idle TSAP
    - Spawns new process with that TSAP address
    - Sends that TSAP address to user
    - Terminates connection
    - Returns to listen on its TSAP
  - User releases connection to process server, connects to new process
  - New process executes requested program, terminates when done

Connection Establishment

- Internet Initial Connection Protocol



Connection Establishment

- Name Server (directory server)
  - User sets up connection to Name Server, sends message specifying service name
  - Name Server sends back TSAP address
  - User releases connection with Name Server, establishes new connection with desired service
  - When new service is created, it must register with Name Server giving its service name TSAP address

Connection Establishment

- Problem of delayed duplicates
- Possible Solutions
  - When TSAP address is needed, new unique address is generated based on current time. When connection is released, address is discarded forever
  - Each connection has unique identifier, chosen by initiator of connection put in each TPDU.
    - When connection request arrives, checked against table.
    - When connection released, each transport entity updates its table marking connection obsolete
- Tomlinson's algorithm is **not** required

Coping with Delayed Duplicates

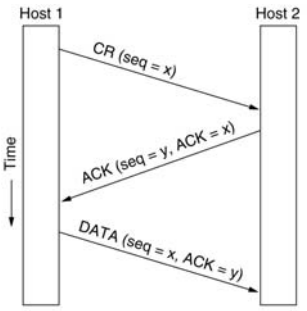
- Sequence number based approach
  - When start, set sequence number = time on processor's local clock
  - If sequence number and time increase indefinitely, no problem of delayed duplicates
- Problem
  - Memory is finite so sequence numbers and time stamps must wrap around, so delayed duplicates is a problem
  - Memory is lost if processor crashes. If processor recovers, it can send message in its new life with same sequence number as in previous life

Coping with Delayed Duplicates

- Essential: ensure each packet has finite life using hop count or time stamp
  - In practice, a packet's life time, T, is multiple of its true life time so that the all its acks are also dead
  - If we wait a time T after a packet is sent, we can be sure that all traces of it are now gone
    - Requiring T dead time after a crash might not be practical

Connection Establishment

- Three-way handshake – to establish a connection, need to agree on initial sequence numbers (instead of using clock). Again, need to prevent delayed duplicates
  - Host 1 chooses sequence number x, sends it to host 2 in a control TPDU
  - Host 2 replies with a control TPDU, acks x, announces its own sequence number y
  - Host 1 sends its first data TPDU, using x, acks y



Connection Establishment

- Three-way handshake (abnormal scenarios)