Chapter 6 - The Transport Layer

General Characteristics -

The lowest layer having endpoint-to-endpoint functionality.
Typically the lowest layer to which applications may directly bind.
Internet family contains no session or presentation layer:
==> Transport is the highest architected layer.

Service Characteristics -

Usual 3 categories possible
Reliable/Connection oriented service is the most interesting/difficult (especially when built on an unreliable network layer.)

OSI Defined Service Classes for network layers

Type	Description
A	Error free service with no N-Resets
B	Error free service except for N-Resets
C	Lost, damaged, reordered or duplicated packets

N-Resets mean that the network layer may occasional suffer packet loss, but always informs the transport layer that a problem has occurred and aborts the connection. SNA path control provides Class B service. ATM and IP provide class C service.

Transport protocols include facilities for

Addressing
Connection management
Flow control
Reliable delivery

Addressing

Transport level addresses are called TSAPs (Transport Service Access Points)

TSAPs are called *Ports* in the TCP/IP domain (NSAPs are *IP addresses*)

Each host may have many active transport connections and thus many TSAPs in concurrent use.

The TSAP address carried in the transport header of an incoming packet provides (possibly in conjuction with the NSAP address) the information needed to identify the process to which the packet is directed.

In TCP delivery is based upon a four tuple:

(Source IP, Source Port, Dest IP, Dest Port)

Connection Management

Objectives are the same as in the datalink layer.

Achieving the objectives are much harder over a Class C network layer than over a physical link

A wire allows *no reordering* and *no packet delays beyond physical latency*
A Class C network permits both

Problems may be caused by the arrival of delayed:
connection requests,
disconnection requests,
data packets (e.g. transfer $1,000,000)
associated with defunct sessions.

For example:

```
connect PORT=2345, HOST=15,NETWORK=10
send  seq=0, move $NNN,NNN from acct xxx to acct yyyy -------   |
:                                                               |
                                                                |
timeout                                                         |
send seq=0, move $NNN,NNN from acct xxx to acct yyyy            |
          <------------------------------------ ack1            |
                                                                |
disconnect                                                      |
connect  PORT=2345, HOST=15,NETWORK=10                          |
                                                                |-->
          <-----------------------------------ack1 ----------------------|
Oh  #^%%#!
```

Possible ways to deal with delayed duplicates:

Ensure **that delayed packets are detected and discarded**

Use unique TSAP addresses (e.g. use the Unix time value which rolls over every 70 years)

Problem: *Now you can't use well known addresses (e.g. mail on port 25)*

Use an incarnation number # ..

At each connection for a TSAP increment the incarnation number.
All TPDU's carry the incarnation number.
Incorrect incarnation number => discard packet.
Problem: *How to remember across crashes which ones have been used.*
Solution: *Write the to disk.. or use the time of day.. or use large random number.*
Essentially expands the sequence # space so that wrap *NEVER* occurs

Reduce **the number of delayed packets and length of the delay**

Possible ways to restrict packet lifetimes:

Restricted subnet design.

Eliminate looping through spanning tree routing.

Unfortunately long-lived loops are not uncommon in TCP/IP

Bound the number of possible hops (TCP/IP)

Each IMP must decrement a hop counter

Bound the actual lifetime of a packet.

Each IMP must decrement the time it holds a packet from a TTL value.

When TTL = 0 packet is discarded.

Bounded lifetime is the most robust but is generally not done.

Tomlinson's Clock Based Sequence #'s.

Objective:

Ensure that two identically numbered packets, *originally created by two distinct connections* are never allowed to simultaneously exist in the network.

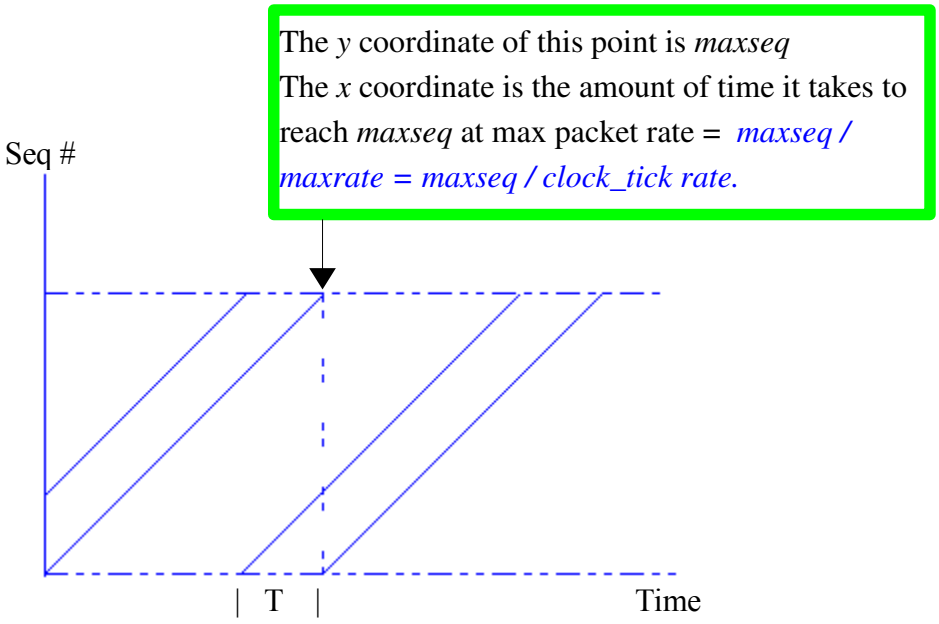
Because of lost, discarded, or damaged packets identically numbered packets belonging to a *single* connection can and do exist all the time.

Prerequisites:

A maximum packet lifetime of T seconds is known to exist.

Each host has a real time clock that continues to run even if host crashes.

Clock ticks at least as fast as the maximum packet generation rate.



Tomlinson's algorithm:

Start sequence # of each session is taken from the clock.

Packets can't be generated faster than the clock ticks.

if (seq# , time) trajectory starts to enter the forbidden zone from above

resynchronization is necessary.

Another example:

Examples

Suppose 16 bit sequence numbers are used; the clock tick rate is 1024 ticks / sec; max packet lifetime is 9 seconds:

a - What is the minimum amount of time between resequencing:

Ans: Sequence number wrap interval - max packet lifetime

2^16 / 2^ 10 = 2 ^6 = 64 secs - 9 seconds = 55 seconds

b - Suppose a packet generation is 1/8 the maximum rate. How often will resequencing be necessary?

Ans: Solve the set of equations

y = x / 8

y = x - 55 => x = (8 * 55) / 7 = 62.85

Connection establishment

Objectives:

Ensure that both parties agree to a connection
Correctly understand the other party's initial sequence number
Avoid halfopen connections
Avoid livelocks and deadlocks in the connection protocol

TCP's "three-way handshake" provides an effective procedure

Normally initiated by one TCP and responded to by another TCP.
Also works if two TCP simultaneously initiate the procedure.
Initialization packets carry:

Seq #, Ack #
Flag bits:
SYN ==> The seq # specified is my initial seq #.
ACK ==> The ack # is valid
RST ==> Error!
In an initializing state return to LISTEN.
In a connected state abort the connection.

Connection model has 4 states:

Closed:	Not involved in session activation or in an established session:
SYN-Sent:	Actively trying to open a session.
SYN-Recvd:	Received a SYN and responded with an ACK:
Established:	SYN Acked.

Normal connection establishment:

```
1.  CLOSED                                                    LISTEN
2.  SYN-SENT    --> <SEQ=100><CTL=SYN>               --> SYN-RECEIVED
3.  ESTABLISHED <-- <SEQ=300><ACK=101><CTL=SYN,ACK>  <-- SYN-RECEIVED
4.  ESTABLISHED --> <SEQ=101><ACK=301><CTL=ACK>       --> ESTABLISHED
5.  ESTABLISHED --> <SEQ=101><ACK=301><CTL=ACK><DATA> --> ESTABLISHED
```

Simultaneous connection establishment:

```
   TCP A                                                TCP B
1.  CLOSED                                               CLOSED
2.  SYN-SENT     --> <SEQ=100><CTL=SYN>              ...
3.  SYN-RECEIVED <-- <SEQ=300><CTL=SYN>              <-- SYN-SENT
4.               ... <SEQ=100><CTL=SYN>              --> SYN-RECEIVED
5.  SYN-RECEIVED --> <SEQ=100><ACK=301><CTL=SYN,ACK> ...
6.  ESTABLISHED  <-- <SEQ=300><ACK=101><CTL=SYN,ACK> <-- SYN-RECEIVED
7.               ... <SEQ=101><ACK=301><CTL=ACK>     --> ESTABLISHED
```

Recovery from old duplicate SYN.

```
    TCP A                                                TCP B

1.  CLOSED                                               LISTEN

2.  SYN-SENT    --> <SEQ=100><CTL=SYN>               ...

3.  (duplicate) ... <SEQ=90><CTL=SYN>               --> SYN-RECEIVED

4.  SYN-SENT    <-- <SEQ=300><ACK=91><CTL=SYN,ACK>  <-- SYN-RECEIVED

5.  SYN-SENT    --> <SEQ=91><CTL=RST>               --> LISTEN

6.              ... <SEQ=100><CTL=SYN>               --> SYN-RECEIVED

7.  SYN-SENT    <-- <SEQ=400><ACK=101><CTL=SYN,ACK>  <-- SYN-RECEIVED

8.  ESTABLISHED --> <SEQ=101><ACK=401><CTL=ACK>      --> ESTABLISHED
```

Note that TCP A remains in the SYN-SENT state after sending the RST.

Dealing with half open connections:

```
      TCP A                                                TCP B

1.  (CRASH)                                  (send 300,receive 100)

2.  CLOSED                                                ESTABLISHED

3.  SYN-SENT --> <SEQ=400><CTL=SYN>              --> (??)

4.  (!!)     <-- <SEQ=300><ACK=100><CTL=ACK>     <-- ESTABLISHED

5.  SYN-SENT --> <SEQ=100><CTL=RST>              --> (Abort!!)

6.  SYN-SENT                                              CLOSED

7.  SYN-SENT --> <SEQ=400><CTL=SYN>              -->
```

Note that TCP B responds to the unexpected SYN-1 with an *ack* not a *rst*. This is done because the SYN-1 might be an old delayed duplicate and it would not be productive to abort a perfectly good connection in that case. TCP A responds with *rst* because it *knows* that an established session does not exist.

Dealing with delayed SYNs:

```
    TCP A                                                TCP B

1.  LISTEN                                                 LISTEN

2.        ... <SEQ=Z><CTL=SYN>                  -->  SYN-RECEIVED

3.  (??) <-- <SEQ=X><ACK=Z+1><CTL=SYN,ACK>    <--  SYN-RECEIVED

4.        --> <SEQ=Z+1><CTL=RST>                -->  (return to LISTEN!)

5.  LISTEN                                                 LISTEN
```

Crash Recovery in the Transport Layer:

Consider a stop and wait file transfer protocol in which a receiver crashes, restarts, and notifies sender that crash occurred.

Sender can be in one of two states:

S0 ==> No outstanding unacked packets.

S1 ==> One outstanding unacked packets.

Receiver can use one of two possible strategies:

R0 ==> ACK then write segment to disk.

R1 ==> Write segment to disk then ACK.

Sender can attempt to recover in 4 ways:

P1 ==> Always retransmit last packet sent before crash.

P2 ==> Never retransmit last packet sent before crash.

P3 ==> Retransmit only if in state S0 at time of crash.

P4 ==> Retransmit only if in state S1 at time of crash.

Most intuitively plausible strategy is P4.

Example scenarios: () means the event did not happen because it logically followed the crash.

Receiver Policy	*Event*	*Sender Policy*	*Result*
R0	C(AW)	P4	OK
R1	WC(A)	P4	Duplicate

All possible Sender / Receiver strategies:

	Receiver Strategy					
	ACK then Write			Write then ACK		
Sender Strategy	*AC(W)*	*AWC*	*C(AW)*	*C(WA)*	*WAC*	*WC(A)*
Always Retransmit	OK	DUP	OK	OK	DUP	DUP
Never Retransmit	LOST	OK	LOST	LOST	OK	OK
Retransmit in S0	OK	DUP	LOST	LOST	DUP	OK
Retransmit in S1	LOST	OK	OK	OK	OK	DUP

A successful Sender/Receiver strategy pair requires (OK, OK, OK) in some row in either the left or right half of the table.

P4 is the best policy but it does not work correctly.

===> A layer can't be expected to recover from its own crashes.

Window size considerations in a transport protocol.

The term window size is used to refer to the number of unacked packets or bytes that are outstanding in the network. There are at least three different objectives of window size management:

1. Allow the sender to send consistently without ever having to stop to wait for an ACK. WS = 1 packet is a stop and wait protocol that is very inefficient in the presence of high propagation or store and forward delays.

2. Don't congest the network. Objective 2 applies to the transport protocol, not the application. Objectives 1 and 2 are inherently conflicting.

3. Don't allow a fast sender to generate packets faster than a slower receiver can consume them

Dealing with objectives 1 and 2

The problem of dealing with objectives 1 and 2 is similar to (but more difficult than) the datalink layer counterpart.

In the datalink layer our objective was to make the window sufficiently large that the sender could *send continuously* in the absence of errors.

The window size that is sufficient to do this is given by
bandwidth-delay product.

For example suppose the link speed is 100 packets / second, and round trip latency is ½ second. Then a sender window size of 50 is sufficient.

At the link layer both bandwidth and delay are *fixed parameters of the link.* However, both the *bandwidth* and the *delay* provided to the transport layer by the network layer are *variable.*

Therefore, using the bandwidth delay product to dynamically adjust window size is problematic:

Suppose
c = network *end to end* throughput (bandwidth) (e.g. 10 packets / second)
r = roundtrip propagation delay in seconds (e.g. 2 seconds)
$win = cr$ = 20 packets

The problem: Suppose congestion causes $r = 4$. Increasing r increases *win to 40.* Increasing *win* further increases congestion.. which increases r *to 8* which increases *win* to 80.. which further increases congestion....

The TCP Vegas window size adjustment mechanism fixes r = *min(all observed r's)* and adjust *win* based upon changes to throughput c. Thus, in TCP Vegas, when throughput decreases due to congestion so does *win.*

Standard TCP gradually increases the *congestion window* and cuts it in half, in response to a packet loss.

Objective 3 - Flow control

Objective:

Synchronize a fast sender and a slow receiver so that the sender produces packets at the rate at which the receiver is consuming them

Example:

Sender of a file is a high performance workstation with a fast disk.
Receiver is a low performance PC with a disk slower than the LAN.

or

Producer is synthesizing random numbers
Consumer is using them in a time-consuming simulation at a slower rate than they are being produced.

Problem:

Unless the sender is forced to produce at the same rate the receiver consumes, all buffer space at receiver will fill up and newly arriving packets will have to be discarded. Discarded packets will have to be retransmitted.

Possible flow control mechanisms

A bad solution: *Withhold acks*

Use a sliding window protocol like a DLC.
Sender window full of unacked packets ==> sender must stop.
But if the receiver is *really* slow the sender will timeout and retransmit.
==> *witholding acks is not a good way to do flow control.*

A better solution:

Use a RR / RNR mechanism similar to the one used in SDLC. But in a deeply pipelined environment, by the time RNR reaches the sender the overflow packets may already be in the pipe.

The ultimate solution:

Have the receiver explicitly advertise available buffer space to the sender.
When the sender has transmitted sufficient un-acked packets to fill all buffer space that the receiver advertises, the sender must stop.
This is a refinement of the binary RR (buffer space is available) RNR (buffer space is not available) and is more suitable for use in a *pipelined* environment.

Example:

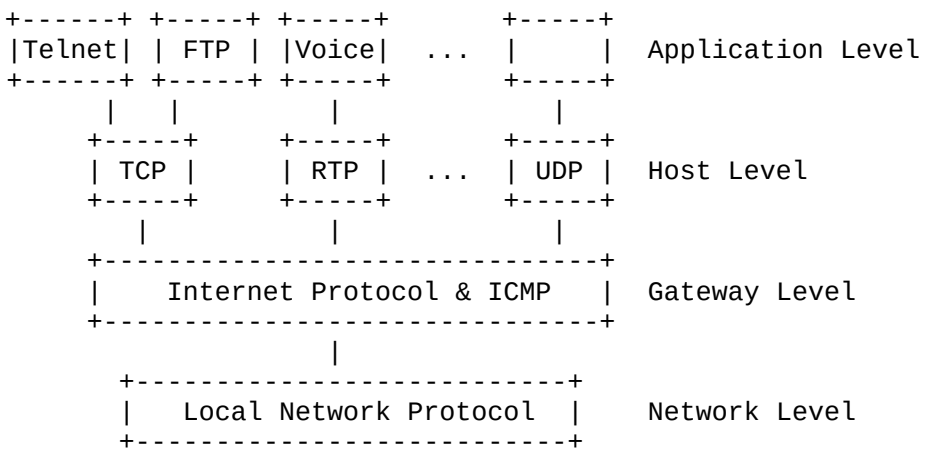
Receiver Ack=25,Wnd=4 pkts

This says that the receiver has correctly received packets through 24 and is willing to receive 25,26,27,28

Thus after the sender sends packet 28, it must stop and wait for another ack.

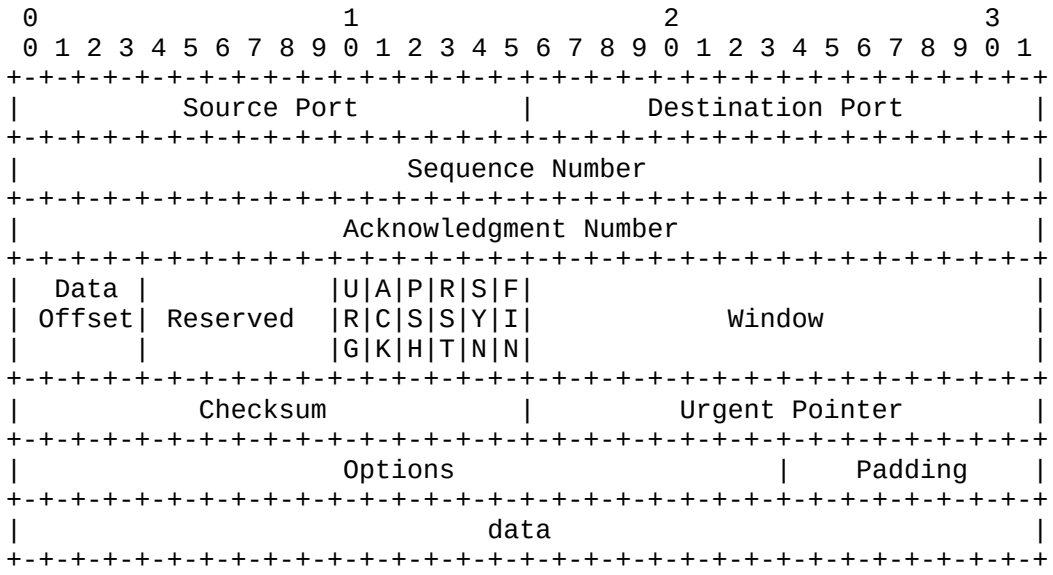
Examples of the transport layer

Transport protocols in TCP/IP



Protocol Relationships

The TCP header



TCP Header Format

TSAP Addresses and packet routing in TCP

A connection is uniquely identified by a connection identifier.

Local Port number
Local IP address
Remote Port number
Remote IP address

Local and Remote IP addresses can, of course, be the same. This connection identifier is used to deliver packets to the proper input queue

TCP State Variables and Sequence Number Space Management.

Sequence #'s

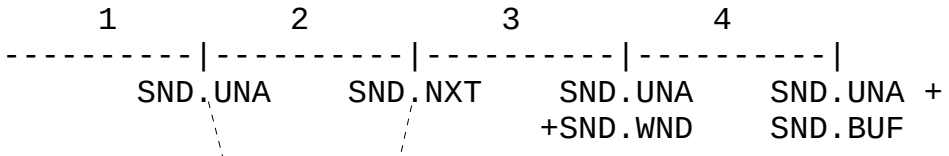
32 bits long.

Identify a specific data byte rather than packet.

Seq = First byte sent in an outgoing packet

Ack = Last byte received correctly and in sequence (no gaps) + 1

Send Sequence Space



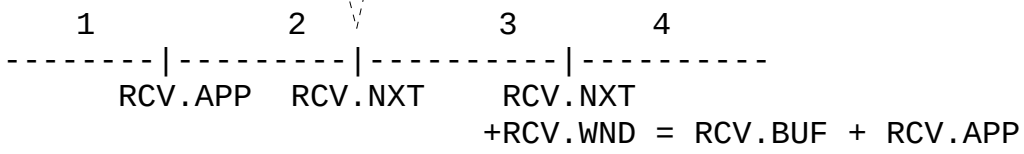
1 - old sequence numbers which have been acknowledged

2 - sequence numbers of unacknowledged data

3 - sequence numbers allowed for new data transmission

4 - future sequence numbers which are not yet allowed

Receive Sequence Space



1 - sequence numbers which have been acknowledged acked and consumed by app

2 - sequence numbers which have been acked but not yet consumed by app

3 - sequence numbers allowed for new reception

4 - future sequence numbers which are not yet allowed

The amount of buffer space available to any connection is limited by the OS.

So

rcv.wnd = rcv.buf – (rcv.nxt – rcv.app) (modulo seq # wrap)

Hence *rcv.wnd* is updated when and only when the application consumes some data.

Examples

Suppose a TCP connection has SND.UNA = 3800 and SND.NXT = 4400 and
a packet arrives with ACK = 4000 and WINDOW = 2000.

a. How many more bytes may be transmitted before the sending TCP must stop sending because it runs out of usable space in the sender window.

After the ack is processed SND.UNA = 4000 and Window = 2000. Therefore the right edge of the usable window is at 6000 and 6000-4400 = 1600 more bytes may be sent.

b. Suppose the sending application has an allocation of 8000 bytes of transmit buffer space. The application will be put to sleep when it tries to send a packet which contains what sequence number??

SND.UNA + SND.BUF = 12,000

Suppose a receiver application has an allocation of 12,000 bytes of TCP receive buffer space. The last byte that the application has consumed is 4000 and the present offered window is 7000. What is RCV.NXT?

RCV.APP + RCV.BUF = 16,000
RCV.NXT = 16,000 - 7000 = 9,000

Now suppose the application consumes 2000 additional bytes of data and the receiver sends a standalone ACK. This packet will carry

ACK = RCV.NXT = 9,000 and WINDOW = 7,000 + 2000 = 9,000

State management summary

Sender side state management

- Sending a segment increases SND.NXT by the amount sent
- Receiving an ACK updates SND.UNA and SND.WND with values carried in the ACK packet

Receiver side state management

- Receiving a segment that include RCV.NXT updates RCV.NXT and reduces RCV.WND by the number of bytes following RCV.NXT.
- Receiving a segment that covers buffer space beyond RCV.NXT changes neither RCV.WND nor RCV.ACK.
- When an application consumes data, RCV.WND is increased by the amount consumed.

TCP connection state machine

The states of a TCP connections are:

LISTEN - represents waiting for a connection request from any remote TCP and port.

SYN-SENT - represents waiting for a matching connection request after having sent a connection request.

SYN-RECEIVED - represents waiting for a confirming connection request acknowledgment after having both received and sent a connection request.

ESTABLISHED - represents an open connection, data received can be delivered to the user. The normal state for the data transfer phase of the connection.

FIN-WAIT-1 - represents waiting for a connection termination request from the remote TCP, or an acknowledgment of the connection termination request previously sent.

FIN-WAIT-2 - represents waiting for a connection termination request from the remote TCP.

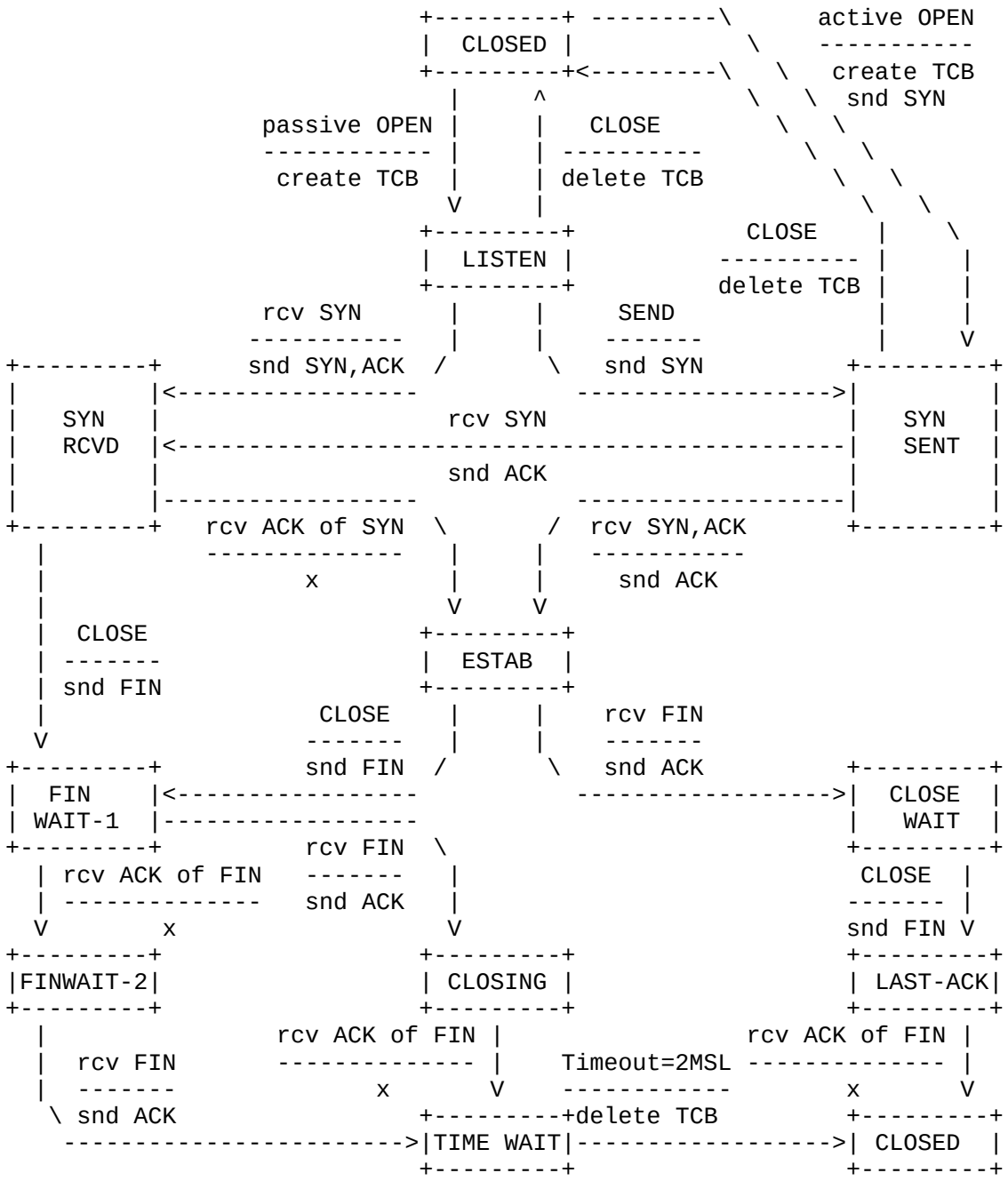
CLOSE-WAIT - represents waiting for a connection termination request from the local user.

CLOSING - represents waiting for a connection termination request acknowledgment from the remote TCP.

LAST-ACK - represents waiting for an acknowledgment of the connection termination request previously sent to the remote TCP (which includes an acknowledgment of its connection termination request).

TIME-WAIT - represents waiting for enough time to pass to be sure the remote TCP received the acknowledgment of its connection termination request.

CLOSED - represents no connection state at all.



TCP Connection State Diagram

TCP/IP Programming model

Addressing in TCP/IP.

Addresses are of the form (Port address, IP address)

IP addresses are assigned by the Internet Administration.
Port addresses are 16 bits
Addresses < 1024 require privileged access.
Well known servers uses address in this range.
Other addresses may be requested explicitly or a free port may be requested.

Sockets are the TSAP's through which applications interact with TCP/IP.

Sockets may be bound to addresses.
Sockets may be in either a connected or unconnected state.
Multiple different connected sockets may be bound to the same local address, but only a single non-connected socket.
When a data message comes in on a connection it is routed to the socket that is bound to its source.
When a connection request message comes in it is directed to the unconnected socket.

An example server:

```
/* rcmdserv.c */

/* This program is an example of how a remote command server */
/* like rsh might work.                                      */
```

Include files for both low level (read, write) I/O and for network I/O are needed .

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

char buffer[200];
struct hostent *hp;
struct hostent *gethostbyname();
```

There are a variety of sockaddr type structures...
The _in indicates that this one is compatible with internet form addresses.

```
struct sockaddr_in name;
struct sockaddr_in client;

extern int errno;

main(argc, argv)
int argc;
char *argv[];
{
   int i;
   int session = 0;
   unsigned char c;
   int sock;
   int msgsock;
   int status;
   int ramt;
   int wamt;
   int namesize;
   int hostaddr;
```

```
/* Create a stream (TCP) socket */

   sock = socket(AF_INET, SOCK_STREAM, 0);
   if (sock < 0)
   {
      printf("Socket create failed \n");
      exit(1);
   }

/* Get nameserver to convert host name to address */

   hp = gethostbyname(argv[1]);
   if (hp == 0)
   {
      printf("Host %s not found", argv[1]);
      exit(2);
   }
```

You should always use the functions provided form manipulating network and port names and addresses. There are some nasty big/little endian portability problems with network addresses.

```
/* Fill in host and port address in name structure */

   bcopy((char *)hp->h_addr,
            (char *)&name.sin_addr, hp->h_length);
   name.sin_family = AF_INET;
   name.sin_port = htons(atoi(argv[2]));

/* Bind socket to local network,port address */

   status = bind(sock, (struct sockaddr *)&name, sizeof(name));
   printf("Bind status = %d \n", status);
   if (status < 0)
     exit(3);
```

```
/* Signify availabilty for network connections */
/* This call is NON-BLOCKING                   */

   status = listen(sock, 4);
   if (status < 0)
   {
      printf("Listen failed with code %d. \n", errno);
      exit(4);
   }

   namesize = sizeof(name);
   while(1)
   {

   /* Accept a connection from the network                  */
   /* Address of the connecting party will be in name after */
   /* the connection is established.                        */
   /* This call BLOCKS until a connection req arrives       */

      msgsock= accept(sock, (struct sockaddr *)&name, &namesize);
      if (msgsock < 0)
      {
         printf("Accept  failed with code %d. \n", errno);
         exit(5);
      }


   /* Invoke the remote command processor */

      session += 1;
      if (fork() == 0)
      {
         rcmd(msgsock, session);
         close(msgsock);
         exit(6);
      }
   }
}
```

```
/* This is the remote command server.. It reads commands */
/* from a connected client, redirects the standard out   */
/* back across the net and executes the commands.        */

rcmd(comfile, id)
int comfile;
int id;

{
   int infile;
   int amtread;
   char msgbuf[10];

/* Redirect standard output back to network connection */

   sprintf(msgbuf, "SVR%d>", id);
   close(1);
   dup(comfile);

   while(1)
   {
      write(comfile, msgbuf, 6);

   /* @ informs the other end its ok to send.. This is a SNA  */
   /* "Dataflow" protocol.                                    */

      write(comfile, "@", 1);
      amtread = read(comfile, buffer, sizeof(buffer));
      *(buffer + amtread - 1) = 0;
      fprintf(stderr, "Cmd was: %s \n", buffer);
```

```
   /* Session level protocol ... x ==> terminate this session */
   /*                            z ==> terminate server.      */

      if (buffer[0] == 'x')
      {
         close(comfile);
         return(1);
      }
      if (buffer[0] == 'z')
      {
         close(comfile);
         return(-1);
      }

    /* Execute the command that was transferred. */

      system(buffer);
   }
}
```

The Client application:

/* rcmdproc.c */

```
/* This is the local host interface to the remote command server
*/

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
char buffer[200];

struct hostent *hp;
struct hostent *gethostbyname();
struct sockaddr_in name;
extern int errno;

main(argc, argv)
int argc;
char *argv[];
{
   int i;
   unsigned char c;
   int sock;
   int status;
   int ramt;
   int wamt;

/* Obtain a socket */

   sock = socket(AF_INET, SOCK_STREAM, 0);
   if (sock < 0)
   {
      printf("Socket create failed \n");
      exit(1);
   }
```

```
/* Map remote host name to network address */

   hp = gethostbyname(argv[1]);
   if (hp == 0)
   {
      printf("Host %s not found", argv[1]);
      exit(1);
   }
```

The client will explicitly bind this socket to any specific port. TCP will automagically select an available port on the source machine. The network address parameter block used in the client will specify the IP address and port of the ***server system.*** *That is, you* ***bind*** *to a* ***local address*** *but* ***connect*** *to a* ***remote address.***

```
/* Complete network address parameter block */

   bcopy((char *)hp->h_addr,
      (char *)&name.sin_addr, hp->h_length);
   name.sin_family = AF_INET;
   name.sin_port = htons(atoi(argv[2]));

/* Try to connect to specified server [name,port] */

   status = connect(sock,
         (struct sockaddr *)&name, sizeof(name));

   if (status < 0)
   {
      printf("Connect failed with code %d. \n", errno);
      exit(4);
   }

   printf("Connect status = %d \n", status);

   wcmd(sock);
}
```

```
/* This function implements the "session level" protocol */
/* (half duplex flip flop) between the command server    */
/* and its clients.                                      */

wcmd(comfile)
int comfile;

{
   int infile;
   int amtread;
   int i;

   while(1)
   {
/* Write output from previous command and server prompt   */
/* Server sends an @ sign indicating end of transmission. */

      while ((amtread =
            read(comfile, buffer, sizeof(buffer))) > 0)
      {
         for (i = 0; i < amtread; i++)
           if (buffer[i] == '@')
              break;
         if (buffer[i] == '@')
         {
            buffer[i] = ' ';
            write(1, buffer, amtread);
            break;
         }
         write(1, buffer, amtread);
      }
   /* Read command from terminal and write to server */

      amtread = read(0, buffer, sizeof(buffer));

   /* Commands starting with x or z cause termination */

      if (buffer[0] == 'x')
         break;
      if (buffer[0] == 'z')
         break;
      write(comfile, buffer, amtread);

   }
   write(comfile, buffer, 1);
}
```

An alternate approach to client / server applications:

Connections are setup with the assistance of one or more "superservers"

Superserver listens on a well known address for connection requests.

When a request arrives superserver:

Acquires a new socket and binds any available free port to it.
Sends the identity of the free port back to the requester.
Forks and creates an instance of the requested server.

Requested server listens on the new port.

Client disconnects from the superserver and reconnects to requested server.

Advantages of this approach:

A standard mnemonic way of requesting servers can be established.
Clients only have to know one port number.

Disadvantages:

Race conditions that exist between disconnect/reconnect.
Possible negative implications of race on security.

UDP does not support the *listen/accept* mechanism and so UDP servers commonly work in this way.

AALs (ATM Adaptation Layers)

Similar to transport layer protocols because:

Are implemented only in hosts and not switches
Perform end-to-end functions

Original plan was to have AAL1, 2, 3, and 4
(corresponding to service classes A, B, C, and D)

The telephone industry developed AAL 1, 2, 3, and 4
They never finished AAL 2 and it is now being redefined
Then they merged 3 and 4 yielding 3/4

When the computer industry became aware of their good works the computer industry invented AAL-5 (originally called SEAL in a none too subtle dig at the telcos).

	Class A	**Class B**	**Class C**	**Class D**
Timing	Real	Real	None	None
Bit Rate	Const	Var	Var	Var
Connection	Oriented	Oriented	Oriented	Less
AAL	1	2	3/4, 5	3/4

An AAL Layer consists of two sublayers:

CS - Convergence sublayer is further subdivided

AAL Service Specific part (aka SSCS)
Common part (aka CPCS)

SAR - Segmentation / reassembly sublayer of logical packets to / from ATM cells

Cell formats differ among the AAL types

AAL -1 (Constant bit rate)

Convergence sublayer functions :

Detect lost cells (via seq #)

Smooths incoming traffic to provide CBR service

AAL-1 cell format: CH P SN SNP Par Payload (47 bytes)

CH Standard 5 byte cell header

P 1 bit

P = 0 => don't preserve message boundaries

P = 1 => Preserve messages a boundaries and contains a 1 byte Message separator pointer and 46 byte payload)

SN - 3 bits sequence #

SNP - 3 bits sequence # protection ($X^3 + X + 1$)

Par 1 bit parity bit

AAL -2 (Originally designed for RT VBR)

AAL - 2 original cell format: CH SN IT Payload(45 bytes) LI CRC

CH Standard 5 byte cell header

SN - sequence #

IT - info type (start - middle - end) of a message SN + IT = 1 bytes

LI - length indicator

CRC - cyclic check LI + CRC = 2 bytes

(Field sizes -- *not* -- part of the standard. Oops).

AAL -2 (New edition: designed for low bit rate, real time, small packets)

e.g. cell phones.

Multiplexing of multiple logical channels into a single ATM connection so as to avoid partially filled cells is the objective.

CS Header (3 Bytes)

CID - 8 bits - logical connection identifier

LI - 6 bits - length of logical packet (64 bytes is the max!)

PPT - 2 bits - payload type (3 => OAM = signaling)

UUI - 3 bits - User to user

HEC - 5 bits - Header Error control

AAL2 (new) cell format CH OSF SN P

CH Standard 5 byte cell header

OSF - 6 bits offset to start of next CS header

SN - 1 bit sequence #

P - 1 bit parity

AAL - 3/4

CS Framing

CPI	1 byte	Message type and BA size units
Btag	1 byte	Begin tag, incremented by 1 for each message
BA size	2 bytes	Amount of buffer space needed.
Payload	1 to 64K bytes	
Padding	0 - 3 bytes	
Etag	1 byte	Same as Btag
Length	2 bytes	

AAL - 3/4 cell format CH ST SN MID Payload (44 bytes) LI CRC

CH	Standard cell header	
ST -	2 bits	Middle, end, beginning, first and only cell of message
SN -	4 bits	Sequence #
MID -	10 bits	Multiplexing ID - Multiple sessions *per virtual channel*
LI -	6 bits	length indicator
CRC -	10 bits	cyclic check

AAL5 Details

higher - level - PDU
|
CS sublayer
|
higher - level - PDU - [pad] - CPCS trailer (Whole mess is mult of 48 bytes)
|
SAR sublayer
|
ATM cell - ATM cell - ATM cell .... ATM - cell

The CPCS trailer

CPCS UU	- 1 byte	(User to User)
CPI	- 1 byte	(Common part indicator)
Length	- 2 byte	(Of CPCS payload)
CRC	- 4 bytes	

Recall that the AAL5 cell is ATM header followed by 48 byte payload

Implications of this on the SAR include:

No sequence #'s => ATM cells must arrive *in order* at SAR for reassembly

No ST=> each successive cell must be:
Part of the current CPCS PDU or
The first cell of the next CPCS PDU
The third bit of the PTI field (aka the AAU bit) carries this info:
0 => not the last cell of the CPCS PDU
1 => the last cell of the CPCS PDU
(This hack is a major violation of layered architecture principles.)

No LI => last cell must be padded so that CPCS trailer is "right justified"

Lost cells can be detected using the:
CRC
Length field in CPCS trailer