Today

- TCP congestion control
- non-blocking i/o
- performance

TCP stream mgt

- checksum'd (dropped by recvr, if fails)
- sequenced and ACK'd segments
- timeouts on send (RTT estimates and retransmissions)
- end-to-end flow control: sliding window
- congestion control with Nagle, delayed ACK, slow start (try to avoid congesting a busy net)

congestion control

- avoid tiny grams
- exponential backoff
- slow start, fast retransmit, fast recovery

delayed ACK, RFC 813

```
wait (up to 200 ms) to send ACK if no data to send
send ACK if 2nd full segment arrives and read by app.
```

Nagle algorithm (disable with TCP_NODELAY sockopt)

```
don't send less than MSS-sized segment if un-acked segment
sender will accumulate data to send (interactive sessions)
```


assignment 3

sending 2K packet to echo server, we get Nagle'd and slow ACK'd

```
   menkar echoing 2k packets to whisper
39.975513 MENKAR.2319 > whisper.8989: . 1:1461(1460) ack 1 win 8192
39.975562 whisper.8989 > MENKAR.2319: . ack 1461 win 30660 (DF)
39.976000 MENKAR.2319 > whisper.8989: P 1461:2001(540) ack 1 win 8192
39.978696 whisper.8989 > MENKAR.2319: . 1:1461(1460) ack 2001 win 32120 (D

40.178545 MENKAR.2319 > whisper.8989: . ack 1461 win 8192
40.178591 whisper.8989 > MENKAR.2319: P 1461:2001(540) ack 2001 win 32120
40.183020 MENKAR.2319 > whisper.8989: . 2001:3461(1460) ack 2001 win 8192
40.183134 whisper.8989 > MENKAR.2319: P 2001:3461(1460) ack 3461 win 32120
40.183507 MENKAR.2319 > whisper.8989: P 3461:4001(540) ack 2001 win 8192
40.194201 whisper.8989 > MENKAR.2319: . ack 4001 win 32120 (DF)

40.378562 MENKAR.2319 > whisper.8989: . ack 3461 win 8192
40.378596 whisper.8989 > MENKAR.2319: P 3461:4001(540) ack 4001 win 32120
40.383012 MENKAR.2319 > whisper.8989: . 4001:5461(1460) ack 4001 win 8192
40.383118 whisper.8989 > MENKAR.2319: P 4001:5461(1460) ack 5461 win 32120
40.383501 MENKAR.2319 > whisper.8989: P 5461:6001(540) ack 4001 win 8192
40.394205 whisper.8989 > MENKAR.2319: . ack 6001 win 32120 (DF)
```

2KBytes every .2 seconds

TCP_NODELAY on server would fix it

SO_DEBUG

- setsockopt() with SO_DEBUG
- kernel trace of messages in small circular kernel buffer
- use **trpt** to display buffer for TCP trace (cryptic)
- includes TCP state info
- you can add SO_DEBUG as an option to your own applications
- many clients support debug (telnet, ftp, ttcp)
- only SunOS ?

ttcp/trpt

```
  FDDI (MTU 4352) ttcp between two Sun's -- slow???
    sender's trace.   sender window 2K,  receiver window 8k
0.36 SYN_SENT:output [677d2a00..677d2a04)@0(win=1000)<SYN> -> SYN_SENT
0.36 SYN_SENT:input 2e93d400@677d2a01(win=1000)<SYN,ACK> -> E
0.36 E:output 677d2a01@2e93d401(win=1000)<ACK> -> ESTABLISHED
0.37 E:output [677d2a01..677d2e01)@2e93d401(win=1000)<ACK,PUSH> -> ESTABLISHED
0.37 E:output [677d2e01..677d3601)@2e93d401(win=1000)<ACK,PUSH> -> ESTABLISHED
0.37 E:input 2e93d401@677d2e01(win=8000)<ACK> -> ESTABLISHED

0.54 E:input 2e93d401@677d3601(win=8000)<ACK> -> ESTABLISHED
0.54 E:output [677d3601..677d3e01)@2e93d401(win=1000)<ACK,PUSH> -> ESTABLISHED

0.74 E:input 2e93d401@677d3e01(win=8000)<ACK> -> ESTABLISHED
0.74 E:output [677d3e01..677d4601)@2e93d401(win=1000)<ACK,PUSH> -> ESTABLISHED

0.94 E:input 2e93d401@677d4601(win=8000)<ACK> -> ESTABLISHED
0.94 E:output [677d4601..677d4e01)@2e93d401(win=1000)<ACK,PUSH> -> ESTABLISHED

If the sender's window size is less than 2*MSS, the receiver will
delay after every data packet (waiting for a 2nd full-sized segment
which is not going to arrive).
```


slow start

RFC 2001

- used on non-local connections
- TCP sender could send up to receivers recvbuf amount of data
- slow start sends one segment, waits for ACK
- then 2 segments waits for ACKs, then 4 ...
- if timeout waiting for ACK, slow start re-invoked
- **fast retransmit**
  - if receive out of order segment, immediately re-issue ACK for last in-order segment
  - assuming sender has sent several segements and it receives 3 duplicate ACKs, immediately (don't wait for timeout) send missing segment
- **fast recovery** does modified slow-start after fast retransmit

slow start tcpdump

```
CETUS1C.47323 > thistle.2000: S 295015721:295015721(0) win 8760 <mss 1460> (DF)
thistle.2000 > CETUS1C.47323: S 2667590736:2667590736(0) ack 295015722 win 32736 <mss 1460>
CETUS1C.47323 > thistle.2000: . ack 1 win 8760 (DF)
CETUS1C.47323 > thistle.2000: P 1:1025(1024) ack 1 win 8760 (DF)
thistle.2000 > CETUS1C.47323: . ack 1025 win 32736 (DF)
CETUS1C.47323 > thistle.2000: . 1025:2049(1024) ack 1 win 8760 (DF)
CETUS1C.47323 > thistle.2000: . 2049:3073(1024) ack 1 win 8760 (DF)
thistle.2000 > CETUS1C.47323: . ack 3073 win 32736 (DF)
CETUS1C.47323 > thistle.2000: . 3073:4097(1024) ack 1 win 8760 (DF)
CETUS1C.47323 > thistle.2000: . 4097:5121(1024) ack 1 win 8760 (DF)
CETUS1C.47323 > thistle.2000: . 5121:6145(1024) ack 1 win 8760 (DF)
thistle.2000 > CETUS1C.47323: . ack 6145 win 32736 (DF)
CETUS1C.47323 > thistle.2000: . 6145:7169(1024) ack 1 win 8760 (DF)
CETUS1C.47323 > thistle.2000: . 7169:8193(1024) ack 1 win 8760 (DF)
CETUS1C.47323 > thistle.2000: P 8193:9217(1024) ack 1 win 8760 (DF)
CETUS1C.47323 > thistle.2000: . 9217:10241(1024) ack 1 win 8760 (DF)
CETUS1C.47323 > thistle.2000: P 10241:11265(1024) ack 1 win 8760 (DF)
thistle.2000 > CETUS1C.47323: . ack 10241 win 32736 (DF)
...
```

limits & timers

/usr/include/netinet/tcp_timers.h

- snd/rcv bufs: 8K
- max UDP pkt: 9000
- max settable: 52,400 (SunOS), 64K
- connect timeout: 75s
- delayed-ACK timeout: 200ms
- keepalive: 2hr+
- retransmit: 5-9+ minutes
- close wait: 30s (2MSL)
- 0-window persist: forever @ 60s
- IP frag assembly: 30s

TCP uses a 200ms and 500ms timer

netstat -s

```
TCP     tcpRtoAlgorithm     =     4     tcpRtoMin           =   200
        tcpRtoMax           = 60000     tcpMaxConn          =    -1
        tcpActiveOpens      = 86813     tcpPassiveOpens     =589652
        tcpAttemptFails     =  2201     tcpEstabResets      = 31590
        tcpCurrEstab        =    80     tcpOutSegs          =96278649
        tcpOutDataSegs      =86696691   tcpOutDataBytes     =284330482
        tcpRetransSegs      =806887     tcpRetransBytes     =592177765
        tcpOutAck           =9476125    tcpOutAckDelayed    =4754620
        tcpOutUrg           =   379     tcpOutWinUpdate     =   365
        tcpOutWinProbe      = 14270     tcpOutControl       =1490125
        tcpOutRsts          = 79302     tcpOutFastRetrans   = 67374
        tcpInSegs           =84971880
        tcpInAckSegs        =67297742   tcpInAckBytes       =239210115
        tcpInDupAck         =1188481    tcpInAckUnsent      =     0
        tcpInInorderSegs    =56656001   tcpInInorderBytes   =1107905274
        tcpInUnorderSegs    = 27191     tcpInUnorderBytes   =9233710
        tcpInDupSegs        = 16578     tcpInDupBytes       =6707303
        tcpInPartDupSegs    =   913     tcpInPartDupBytes   =372814
        tcpInPastWinSegs    =   163     tcpInPastWinBytes   =134543765
        tcpInWinProbe       =   165     tcpInWinUpdate      = 12141
        tcpInClosed         = 43275     tcpRttNoUpdate      =390895
        tcpRttUpdate        =7448238    tcpTimRetrans       =1271701
        tcpTimRetransDrop   =  1447     tcpTimKeepalive     =  1988
        tcpTimKeepaliveProbe=   906     tcpTimKeepaliveDrop =    34
        tcpListenDrop       =    88     tcpListenDropQ0     =     0
        tcpHalfOpenDrop     =     0
```


socket options

- **SO_RCVBUF SO_SNDBUF** socket buffers (performance enhancers)
- **SO_LINGER** change close behavior
- **SO_REUSEADDR** avoid "port in use", TIME_WAIT
- **TCP_NODELAY** disable Nagle
- **SO_KEEPALIVE** 2hr idle check
- **SO_DEBUG** enable kernel trace

non-blocking I/O

fork/thread/select/non-blocking/signal/aio

- need fcntl() to set nonblocking
- if socket call would block, you get error EWOULDBLOCK (connect() gives you EINPROGRESS)
- need to poll or use select()

```
#include <fcntl.h>

val = Fcntl(sockfd, F_GETFL, 0);
Fcntl(sockfd, F_SETFL, val | O_NONBLOCK);
```

signal driven I/O

- signal-driven I/O (SIGIO)
- works for tty I/O and sockets (only)
- if multiple fd's enabled for asynch. I/O, on interrupt, you don't know which fd. Need **select()**
- steps:
  1. establish handler for SIGIO
  2. set socket owner
  3. enable asynch I/O on a fd
- for UDP, signal generated when datagram arrives or asynch error occurs (need connect())
- NTP uses timestamping UDP arrivals
- useless for TCP, too many signals (could use for listen socket)

oldcat.c

```
/*
 * Copy standard input to standard output.
 */

#define BUFFSIZE        4096

main()
{
        int             n;
        char            buff[BUFFSIZE];

        while ( (n = read(0, buff, BUFFSIZE)) > 0)
                if (write(1, buff, n) != n)
                        err_sys("write error");

        if (n < 0)
                err_sys("read error");

        exit(0);
}
```


aiocat.c

```
#include <signal.h>
#include <fcntl.h>

#define  BUFFSIZE 4096

int   sigflag;
sigio_func()
{
   sigflag = 1;
}

main()
{
   int      n;
   char     buff[BUFFSIZE];
   int      sigio_func();

   signal(SIGIO, sigio_func);

   if (fcntl(0, F_SETOWN, getpid()) < 0)
      err_sys("F_SETOWN error");

   if (fcntl(0, F_SETFL, FASYNC) < 0)
      err_sys("F_SETFL FASYNC error");
```



```
   for ( ; ; ) {
      sigblock(sigmask(SIGIO));  /* block sigs */
      while (sigflag == 0)
         sigpause(0);      /* unblock and wait for a signal,then block */

      /*
       * We're here if (sigflag != 0).  Also, we know that the
       * SIGIO signal is currently blocked.
       */

      if ( (n = read(0, buff, BUFFSIZE)) > 0) {
         if (write(1, buff, n) != n)
            err_sys("write error");
      } else if (n < 0)
         err_sys("read error");
      else if (n == 0)
         exit(0);    /* EOF */

      sigflag = 0;         /* turn off our flag */

      sigsetmask(0);       /* and reenable signals */
   }
}
```

socket signals

- **SIGIO** socket ready for asynch I/O
- **SIGURG** arrival of urgent data, need to have SETOWN
- **SIGPIPE** can no longer write to a socket
- **EINTR** for interrupted blocking socket calls (**accept(), recvfrom(), read() ...**

advanced socket I/O

dont' try this at home

```
  stevens 13.3-.5
#include <sys/socket.h>

int recv(int s, void *buf, int len, unsigned int flags);
int  send(int  s,  const  void *msg, int len, unsigned int
       flags);

 flags
 MSG_DONTWAIT  nonblocking I/O for single operation
 MSG_OOB       send or read out-of-band data
 MSG_PEEK      look at the data but leave in system buff.
 MSG_WAITALL   does what our readn() does
```


advanced socket I/O

readv/writev

- read or write from more than one buffer
- handy for avoiding lots of write's
- for UDP, writev generates a single datagram

```
    scatter/gather

#include <sys/uio.h>

int  readv(int  fd,  const  struct  iovec * vector, size_t
       count);
int writev(int fd, const struct  iovec  *  vector,  size_t
       count);

 2nd argument points to an array of iovec structures

       struct iovec {
         void *iov_base; /* Starting address.  */
         size_t iov_len; /* Length in bytes.  */
       };
```


sendmsg/recvmsg

- most general of all I/O functions
- could implement read/readv/recv/recvfrom with recvmsg
- recvmsg for UDP can tell if multicast/broadcast packet was received, or if datagram was truncated
- same flags as send/recv

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

int  recvmsg(int  s,  struct  msghdr  *msg,  unsigned  int
       flags);
int  sendmsg(int s, const struct msghdr *msg, unsigned int
       flags);
struct msghdr {
   caddr_t msg_name;              /* optional address */
   int     msg_namelen;           /* size of address */
   struct  iovec *msg_iov;        /* scatter/gather array */
   int     msg_iovlen;            /* # elements in msg_iov */
   caddr_t msg_accrights;         /* access rights sent/received */
   int     msg_accrightslen;
};

  newer version includes msg_flags (Stevens 13.5)
```

Mogul's rules

- You can buy better bandwidth, but not latency.
- Bigger packets are better.
- Congestion avoidance is better than congestion recovery.
- CPU speed is more important than LAN speed.
- Avoid timeouts and retransmissions.

```
timeouts are bad for performance, means net has been idle
  for at least 1 RTT.
  100 pkts/sec connection -- assume timeout is 1 sec.
   1% loss: after 99 pkts, 1 second pause ==> throughput halved
   10% loss  ==>  90% drop in throughput !
```


Stevens rules

- Can't run faster than slowest link
- Can't go faster than memory bandwidth of slowest machine (reduce data copies, see Jacobson paper)
- You can't go faster than the window-size offered by the receiver divided by RTT

```
Stevens client performance pg 409
  batch str_cli  (echo'ing 200 lines, RTT of 175 ms)

  stop-and-wait         354   seconds
  select & blocking I/O  12.3 seconds
  fork                    8.7 seconds
  threads                 8.5 seconds
  non-blocking I/O        6.9 seconds
```

TCP speed is determined by size of TCP window and speed of light

server performance

- iterative server has limited performance
- concurrent servers
  - select
  - fork per client
  - thread per client
- pre-fork (pool of servers)
- pre-threading (textbook 27.1)
- DNS round-robin
- router load-balancing

long fat pipes

bandwidth-delay product

- bandwidth and delay of path dictates buff size

```
T1 (1.5 Mbs) and RTT 60 ms:  need 11.5 KBytes buffer or pipe
  will not stay full
gigabit/sec and RTT 60 ms: need 7.5 MBytes
```

- faster links or slower RTT (satellite) requires bigger buffs
- may exceed max window size (65535)

new TCP options

RFC 1323 (both sides must comply)

- needed for long fat pipes
- window scale option (1 GByte)
- that much data in flight, can't afford to resend: need selective ACK (SACK)
- RTT calculated only once every roundtrip, with time-stamp option can do on every send-ack
- sequence numbers may wrap on high speed link

```
packets should be discarded after 2MSL
on Ethernet, takes 60 minutes to send 4GB and wrap sequence number
on FDDI, takes 5 minutes
giganet, 34 seconds

PAWS -- protection against wrapped sequence numbers
```

- use time-stamp option to discard wrapped sequence numbers

UDP vs TCP

- must use UDP for multicast/broadcast (would need to send N copies with TCP)
- UDP for simple request-reply apps
  - TCP connection set up expensive (packets, RTT)
  - may not need flow control
  - might still need timeout/retransmission/sequencing
- active flows to lots (100s) of hosts from a server may exceed file descriptors for OS (TCP), then use UDP
- bulk transfer, use TCP
- can emulate datagram with readn() in TCP
- multimedia app's use UDP

next time

review