Textbook Server – C

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define SERVER_PORT  5432
#define MAX_PENDING  5
#define MAX_LINE     256

int
main()
{
  struct sockaddr_in sin;
  char buf[MAX_LINE];
  int len;
  int s, new_s;

  /* build address data structure */
  bzero((char *)&sin, sizeof(sin));
  sin.sin_family = AF_INET;
  sin.sin_addr.s_addr = INADDR_ANY;
  sin.sin_port = htons(SERVER_PORT);

  /* setup passive open */
  if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    perror("simplex-talk: socket");
    exit(1);
  }
  if ((bind(s, (struct sockaddr *)&sin, sizeof(sin))) < 0) {
    perror("simplex-talk: bind");
    exit(1);
  }
  listen(s, MAX_PENDING);

  /* wait for connection, then receive and print text */
  while(1) {
    if ((new_s = accept(s, (struct sockaddr *)&sin, &len)) < 0) {
      perror("simplex-talk: accept");
      exit(1);
    }
    while ((len = recv(new_s, buf, sizeof(buf), 0))) {
      buf[len] = '\0' ;
      fputs(buf, stdout);
    }
    close(new_s);
  }
}
```

Better server – C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <errno.h>
#include <sys/time.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <pthread.h>

#include "parseInetAddr.h"
#include "logServ.h"
#include "lineIOServ.h"

#define SERVERPORT 3630
#define SERVERLOG  "h1.log"

#define MAX_INLINE_SIZE (100 < MAX_MESSAGE_SIZE ? 10 : MAX_MESSAGE_SIZE)

#define INLINE_TIME_LIMIT 10

struct pthreadArgs {
  FILE *logFile ;
  int connectSock ;
  struct sockaddr_in clientSockAddr ;
} ;
```

Pthread routine to log and process a connection

```
void *process_request(void *myArgs) {
  FILE *logFile;
  int sio;
  struct sockaddr_in clientAddr ;
  char clnt_name[MAX_CLIENT_NAME_SIZE+1] ;
  char user_inline[MAX_INLINE_SIZE+2] ;
  int numRead ;
  struct timeval timeLimit ;

  /* Copy my arguments and free the memory */
  logFile = ((struct pthreadArgs *)myArgs) -> logFile ;
  sio = ((struct pthreadArgs *)myArgs) -> connectSock ;
  (void) memcpy((void *)&clientAddr,
              (void *)&(((struct pthreadArgs *)myArgs) -> clientSockAddr),
              sizeof(struct sockaddr_in)) ;
  (void) free(myArgs) ;

  parse_inet_address(&clientAddr, clnt_name, MAX_CLIENT_NAME_SIZE+1) ;
  logServerAction(logFile, clnt_name, "connected") ;
  timeLimit.tv_sec  = INLINE_TIME_LIMIT ;
  timeLimit.tv_usec = 0 ;
  if ((numRead=readLine(sio, user_inline, MAX_INLINE_SIZE, &timeLimit)) <= 0)
    logServerAction(logFile, clnt_name, "no input") ;
  else
    logClientInput(logFile, clnt_name, user_inline, numRead) ;
  close(sio) ;
  logServerAction(logFile, clnt_name, "finished") ;
  return ((void *)NULL) ;
}
```

main routine – creation and binding of socket

```
main(int argc, char *argv[])  {
  int srendezvous, sio, clntsize ;
  struct sockaddr_in servsock, clntsock;
  char *logFileName ;
  FILE *logFile ;
  struct timeval ptime ;
  pthread_t newThread ;
  pthread_attr_t attribThread ;
  struct pthreadArgs *newThreadArgs ;

  (void) pthread_attr_init(&attribThread) ;
  (void) pthread_attr_setdetachstate(&attribThread, PTHREAD_CREATE_DETACHED) ;

  /* Initialize random number generate */
  (void) gettimeofday(&ptime, (struct timezone *)NULL) ;
  srand(ptime.tv_usec) ;

  /* Open log file */
  if (argc > 1)
    logFileName = argv[1] ;
  else
    logFileName = SERVERLOG ;
  logFile = fopen(logFileName, "a") ;
  if (logFile == (FILE *)NULL) {
    fprintf(stderr, "Unable to open log file: %s\n", logFileName) ;
    exit(1) ;
  }

  if ((srendezvous = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    fputs("Unable to create socket\n", stderr) ;
    exit(1) ;       /* Now this is unlikely to happen */
  }

  servsock.sin_family = AF_INET ;
  servsock.sin_port = htons(SERVERPORT) ;
  servsock.sin_addr.s_addr = INADDR_ANY ;

  if (bind(srendezvous, (struct sockaddr *) &servsock,
          sizeof(struct sockaddr_in))) {
    fprintf(stderr, "Unable to bind to port %d\n", SERVERPORT) ;
    if (errno == EADDRINUSE)
      fputs("Check if another server is still running\n\n"
           "If a server recently terminated, wait before restarting.\n",
           stderr) ;
    exit(1) ;
  }

  (void) listen(srendezvous, SOMAXCONN) ;      /* No way to fail */
```

main routine – accept connection and start Pthread

```
  while (1) {
    clntsize = sizeof(struct sockaddr_in) ;
    sio = accept(srendezvous, (struct sockaddr *)&clntsock, &clntsize) ;
    if (sio<0) {
      write_log(logFile, "accept failure", strerror(errno)) ;
      exit(1) ;
    } else {
      void *newData ;
      if ((newData = malloc(sizeof(struct pthreadArgs))) == NULL) {
      write_log(logFile, "out of memory", "") ;
      exit(1) ;
      } else {
      newThreadArgs = (struct pthreadArgs *)newData ;
      newThreadArgs->logFile = logFile ;
      newThreadArgs->connectSock = sio ;
      memcpy((void *)&newThreadArgs->clientSockAddr,
             (void *)&clntsock,
             sizeof(struct sockaddr_in)) ;
      pthread_create(&newThread, &attribThread,
                  process_request, (void *)newThreadArgs) ;
      }
    }
  }
}
```

Nonblocking socket input routine

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include "lineIOServ.h"

int readLine(int sio, char *lineBuff, int lineSize,
             struct timeval *timeLimit ) {
  int lineRead, rc ;
  struct timeval presTime, alarmTime, selLimit ;
  fd_set sioOn ;

  /* Set the alarm clock */
  (void) gettimeofday(&presTime, (struct timezone *)NULL) ;
  /* timeradd(&presTime, timeLimit, &alarmTime) ; */
  alarmTime.tv_usec = presTime.tv_usec + timeLimit->tv_usec ;
  alarmTime.tv_sec  = presTime.tv_sec  + timeLimit->tv_sec  ;
  if (alarmTime.tv_usec >= 1000000) {
    alarmTime.tv_usec -= 1000000 ;
    alarmTime.tv_sec++ ;
  }

  /* Set the initial select limit */
  selLimit = *timeLimit ;

  lineRead = 0 ;
  while ((lineRead < lineSize) &&
       ((lineRead<2) || (lineBuff[lineRead-2] != '\r') ||
          (lineBuff[lineRead-1] != '\n'))) {
    FD_ZERO(&sioOn) ;
    FD_SET(sio, &sioOn) ;
    if (select(sio+1, &sioOn, (fd_set *)NULL, (fd_set *)NULL, &selLimit)) {
      rc = read(sio, &lineBuff[lineRead], lineSize-lineRead) ;
      if (rc <= 0)
      return -1 ;
      lineRead += rc ;
      /* Recompute the time limit */
      (void) gettimeofday(&presTime, (struct timezone *)NULL) ;
      /* timersub(&alarmTime, &presTime, &selLimit) ; */
      selLimit.tv_usec = alarmTime.tv_usec - presTime.tv_usec ;
      selLimit.tv_sec  = alarmTime.tv_sec  - presTime.tv_sec  ;
      if (selLimit.tv_usec < 0) {
      selLimit.tv_usec += 1000000 ;
      selLimit.tv_sec-- ;
      }
      if (selLimit.tv_sec < 0)      /* Seems extremely unlikely */
      selLimit.tv_sec = selLimit.tv_usec = 0 ;
    } else
      return -2 ;
  }
  return lineRead ;
}
```

Java server

“main” routine

```
import java.net.*;
import java.io.* ;

class E1srv {

  private static final int serverport = 3630 ;

  public static void main (String args[]) {

    ServerSocket Rendezvous ;
    Socket theConnection ;

    String logFileName = "h1.log" ;
    if (args.length >= 1) {
      logFileName = args[1] ;
    }

    try {
      PrintWriter logWriter =
       new PrintWriter(new FileOutputStream(logFileName), true) ;
      E1logServ logFile = new E1logServ(logWriter) ;
      try {
        ServerSocket srendezvous = new ServerSocket(serverport) ;
        while (true) {
          try {
            Socket sio = srendezvous.accept() ;
            (new Thread(new E1child(sio, logFile))).start() ;
          } catch (IOException ioe) {
            System.err.println ("Failed to accept new connection") ;
            break ;
          }
        }
      } catch (IOException ioe) {
        System.err.println ("Unable to open port: " + serverport) ;
      }
    } catch (IOException ioe) {
      System.err.println ("Unable to open log file: " + logFileName) ;
    }
  }
}
```

Java thread routine – serves a single connection

```
import java.net.*;
import java.io.* ;

class E1child implements Runnable {
  E1logServ logFile ;
  Socket sio ;

  public E1child( Socket s, E1logServ l) {
    logFile = l ;
    sio = s ;
  }

  public void run() {

    InetSocketAddress clientAddr =
                  (InetSocketAddress) sio.getRemoteSocketAddress() ;
    String clnt_name = clientAddr.getHostName() ;
    logFile.logServerAction(clnt_name, "connected") ;

    InputStream myIS ;
    try {
      myIS = sio.getInputStream() ;
    } catch (IOException ie) {
      logFile.logServerAction(clnt_name, "no input stream") ;
      return ;
    }

    byte[] myBuff = new byte[1024] ;
    E1buffer myEBuff = new E1buffer(myBuff) ;

    Thread gomer = new Thread(new E1lineread(myIS, myEBuff)) ;
    gomer.start() ;

    try {
      gomer.join(10 * 1000) ;
    } catch (InterruptedException ie) { }
    try {
      sio.close() ;     // If gomer is waiting too long, this will stop him
    } catch (IOException ie) { }
    try {
      myIS.close() ;    // is this necessary?
    } catch (IOException ie) { }

    String status = "exited" ;
    if (gomer.isAlive()) {
      try {
        gomer.join() ;    // Closing the socket will cause gomer to stop
      } catch (InterruptedException ie) { }
      status = "killed" ;
    }
    logFile.logClientInput(clnt_name, myEBuff.Data, myEBuff.Size) ;
    logFile.logServerAction(clnt_name, status) ;
  }
}
```

Java thread needed for non-blocking I/O

```
import java.net.*;
import java.io.* ;

class E1lineread implements Runnable {
  private E1buffer myBuffer ;
  private InputStream mySocket ;

  public E1lineread( InputStream s, E1buffer b) {
    mySocket = s ;
    myBuffer = b ;
  }

  public void run() {
    byte[] lineBuff = myBuffer.Data ;
    int lineSize = lineBuff.length ;     // Size of buffer
    int lineRead = 0 ;                   // Number read into buffer
    int lineProc = 0 ;                   // Number of processed characters
    boolean Done = false ;               //
    while (!Done) {
      if (lineProc == lineSize)
        Done = true ;
      else if (lineProc == lineRead) {
        try {
          int rc = mySocket.read(lineBuff, lineRead, lineSize-lineRead) ;
          if (rc == -1)
            Done = true ;
          else
            lineRead += rc ;
        } catch (IOException ioe) {
          // if "parent" closes the socket, IOException is thrown
          Done = true ;
        }
      } else if (lineBuff[lineProc] == '\n') {
        ++lineProc ;
        Done = true ;
      } else
        ++lineProc ;
    }
    myBuffer.Size = lineProc ;
    myBuffer.Done = true ;
    return ;
  }
}
```

Java logger class

```
import java.io.* ;

class E1logServ {

  PrintWriter myLog ;

  public static final int max_log_rec_size = 100 ;
  public static final int max_client_name_size = 50 ;
  public static final int max_message_size =
      max_log_rec_size - max_client_name_size - 4;

  public E1logServ(PrintWriter tLog) {
    myLog = tLog ;
  }
```

Only one thread at a time, please

```
  private synchronized void write_log(String clnt_name, String log_msg) {
    // Weren't C format statements nice
    if (clnt_name.length() > max_client_name_size)
      myLog.print(clnt_name.substring(0, max_client_name_size)) ;
    else {
      myLog.print(clnt_name) ;
      for (int i=clnt_name.length(); i<max_client_name_size; ++i)
        myLog.print(' ') ;
      }
    myLog.print(" -- ") ;
    if (log_msg.length() > max_message_size)
      myLog.print(log_msg.substring(0, max_message_size)) ;
    else
      myLog.print(log_msg) ;
    myLog.println() ;
  }

  private char hexIt(int i) {
    final String hexChars = "0123456789ABCDEF" ;
    return hexChars.charAt(i & 0xF) ;
  }

  void logServerAction(String clnt_name, String log_msg) {
    write_log(clnt_name, "*** " + log_msg + " ***") ;
  }


  void logClientInput(String clnt_name,
                      byte[] user_rsp, int user_rsp_size) {
```

lots of boring stuff omitted

```
    write_log(clnt_name, new String(outBuff, 0, respP )) ;
  }
}
```

Perl server

```
#!/usr/bin/perl -w
use strict;
use IO::Socket::INET ;
use Fcntl ':flock';
use Fcntl ':seek' ;
my($max_log_rec_size) = 100 ;
my($max_client_name_size) = 50 ;
my($max_message_size) = $max_log_rec_size - $max_client_name_size - 4 ;
```

Perl logger – notice the file locking

```
sub write_log {
    my ($outFile, $clnt_name, $log_msg) = @_ ;

    if (length($log_msg) > $max_log_rec_size) {
        $log_msg = substr($log_msg, $max_log_rec_size) ;
    }
    my $wrz = sprintf("%-*s -- %s\n", $max_client_name_size,
                      $clnt_name, $log_msg) ;
    flock ($outFile, LOCK_EX) ;
    seek ($outFile, 0, SEEK_END) ;
    print $outFile $wrz ;
    flock ($outFile, LOCK_UN) ;
}

sub logServerAction {
    my ($outFile, $clnt_name, $log_msg) = @_ ;
    write_log ($outFile, $clnt_name, '*** ' . $log_msg . ' ***') ;
}

sub hexIt {
    substr ('0123456789ABCDEF', $_[0] & 0xF, 1) ;
}
sub logClientInput {
    my ($logFile, $clnt_name, $user_rsp) = @_ ;
    my $outBuff = '"' ;
    for (my $userP=0; $userP < length($user_rsp) &&
                      length($outBuff)<($max_message_size-4); ++$userP) {
        my $nextC = substr($user_rsp, $userP, 1) ;
        if ($nextC eq "\r") {
            $outBuff .= "\\r" ;
        } elsif ($nextC eq "\n") {
            $outBuff .= "\\n" ;
        } elsif ($nextC eq "\0") {
            $outBuff .= "\\0" ;
        } elsif ($nextC eq "\\") {
            $outBuff .= "\\\\" ;
        } elsif ($nextC ge ' ' && $nextC le '~') {
            $outBuff .= $nextC ;
        } else {
            $outBuff .= ( "\\x" . &hexIt(ord($nextC) >> 4) . &hexIt(ord($nextC)) ) ;
        }
    }
    if (length($outBuff) < ($max_message_size-4)) {
        $outBuff .= '"' ;
    }
    write_log($logFile, $clnt_name, $outBuff) ;
}
```

Perl non-blocking I/O

```
sub readLine {
    my ($sio, $pLineBuff, $lineSize, $timeLimit) = @_ ;
    my $alarmtime = time() + $timeLimit + 1;
    my $lineRead = 0 ;
    $$pLineBuff = '' ;
    while ($lineRead < $lineSize &&
           (($lineRead < 1) ||
             substr($$pLineBuff, $lineRead-1, 1) ne "\n")) {
        my $sio0n = '' ;
        vec($sio0n, fileno($sio), 1) = 1 ;
        my $selLimit = $alarmtime - time() ;
        my $numReady = select($sio0n, undef, undef, $selLimit) ;
        if ($numReady == 0) {
            return -1 ;
        }
        my $rc = sysread($sio, $$pLineBuff,
                         $lineSize-$lineRead, $lineRead) ;
        if ($rc < 0) {
            return -1 ;
        } elsif ($rc == 0) {
            return $lineRead ;
        }
        $lineRead += $rc ;
    }
    return $lineRead ;
}
```

Perl connection handler – runs as a separate Unix process

```
sub process_request {
    my ($logFile, $sio) = @_ ;
    my $clnt_name = gethostbyaddr($sio->peeraddr(), AF_INET) ;
    &logServerAction($logFile, $clnt_name, 'connected') ;
    my $outBuff = '' ;
    if (&readLine($sio, \$outBuff, 1024, 10) >= 0) {
        &logClientInput($logFile, $clnt_name, $outBuff) ;
        &logServerAction($logFile, $clnt_name, 'exited') ;
    } else {
        &logServerAction($logFile, $clnt_name, 'killed') ;
    }
    close($sio) ;
}
```

Perl main routine

```
my $logFileName = 'h1.log' ;
if ( $#ARGV >= 0) {
    $logFileName = $ARGV[0] ;
}

my $logFile ;
if (!open($logFile, ">${logFileName}")) {
    print STDERR "Unable to open ${logFileName}\n" ;
    exit 1 ;
}

my $serverport = 3630 ;
my $rendezvous = IO::Socket::INET->new(
        Listen    => 5,
        LocalPort => $serverport,
        Proto     => 'tcp') ;

die "Unable to bind tcp port ${serverport}: $!" unless $rendezvous ;

$SIG{CHLD} = 'IGNORE' ;
while (1) {
    my $sio = $rendezvous->accept() ;
    die "Unable to accept: $!" unless $sio ;
    if (fork() > 0) {
        Parent process goes here
        close($sio) ;
    } else {
        Child process goes here
        close($rendezvous) ;
        &process_request($logFile, $sio) ;
        exit 0 ;
    }
}
```

Needed for a server

- Lots of error checking
- Multi-processing to handle concurrent requests
- Non-blocking I/O to handle client disconnects
- Synchronization to handle resource access