TCP and Concurrency

The third assignment at DAB728

2006-12-06
Jonas Lundberg/Ola Flygt
Matematiska och systemtekniska institutionen, MSI
Växjö universitet

(email: Ola.Flygt@msi.vxu.se)

This assignment should be finished and handed in before January 10, 2007.

Introduction
The aim of this third assignment is to introduce the fundamentals of network programming using the TCP transport protocol. We also introduce two techniques to implement concurrency into servers.
The Reliable Transport Service TCP

This is only a brief introduction. We refer to the textbook for a more detailed presentation of the various features of TCP.

In the previous assignment we developed simple applications that used the transport protocol UDP. In this assignment we will develop applications that use TCP (Transport Control Protocol). The main difference between UDP and TCP is the reliability. UDP provides unreliable packet delivery. Packets can be lost or destroyed when transmission errors interfere with data, when network hardware fails, or when networks become too heavily loaded. Furthermore, networks that route packets dynamically can deliver them out of order, deliver them after a substantial delay, or deliver duplicates. Since application programs often need to send large volumes of data across the Internet, using an unreliable transport protocol like UDP forces programmers to build error detection and recovery into each application. The method Positive Acknowledgement with Retransmission that we used in the last assignment is one way to introduce reliability into applications using UDP.

The TCP transport protocol was developed to provide a general purpose solution to the problem of unreliable transport, making it possible for experts to build a single instance of transport software that all applications can use. Having a single transport protocol helps isolate application programs from details concerning the reliability of the underlying network. The TCP protocol can be characterized by the following features that provide reliability:

- The application data is broken, and later gathered, into what TCP considers the best sized packets to send. This is totally different from UDP where each write by the application generates a UDP datagram of that size unless the maximum unit of transmission is exceeded, and where no gathering at the destination endpoint takes place. The unit of data passed by TCP is called a segment.
- When TCP sends a segment it maintains a timer, waiting for the other end to acknowledge reception of the segment. If an acknowledgement isn’t received in time, the segment is retransmitted.
- When TCP receives data from the other end, it sends an acknowledgement. This acknowledgement is not sent immediately, but normally delayed a fraction of a second to allow so called piggybacking.
- TCP maintains a checksum in its header in order to detect any modification of the data it transports. If a segment arrives with an invalid checksum, TCP returns no acknowledgment and thus forces the sender to retransmit.
- Since IP datagrams may arrive out of order, so do TCP segments. A receiving TCP resequences the arrived segments (if necessary) and passes them correct order to the application. The receiving TCP also discards duplicate segments.
- TCP provides flow control. A receiving TCP only allows the other end to send as much data as the receiver has buffers for. This prevents a fast host from taking all the buffer space from a slow host.

In order to implement all this reliability, TCP uses a rather complicated transmission scheme based on sequenc numbers and a variable sized sliding windows. For more details about TCP we refer to the course litterature. The important thing to remember when developing applications using TCP is that reliability is not a problem, TCP takes care of that.

An Echo Client/Server Pair Using TCP

In the last assignment we presented and used a simple echo client/server pair based on UDP. In this assignment we will use TCP to do the same thing. Since TCP provides us with a reliable transport service we don’t have to worry about error detection and retransmission of lost packets.

In Figure 1 we show the various system calls involved in a single echo session. Notice the similarities between this set of calls and the one we used to build the client/server pair in the previous assignment. It is only the two server calls listen() and accept() that are new. However,
The amount of messages sent between the server and the client has increased substantially from 2 to 11. This is the price we have to pay for the high reliability that TCP provides. However, the use of piggybacking makes the number of segments sent less than the number of messages. It is in the `socket()` call we determine what kind of transport protocol to be used. The argument `SOCK_STREAM` implies TCP whereas the argument `SOCK_DGRAM` implies UDP.

If we start to look at the echo session from the client point of view we first notice that the `connect()` call works differently when using TCP instead of UDP. In the UDP case all `connect()` did was to add remote endpoint information to the socket. In the TCP case it does the same and initiates contact with the remote part using a three-way handshake procedure. Notice also that in the TCP case both `write()` and `read()` have acknowledgement handling built in automatically. It is somewhat confusing at the beginning that a call has different meanings depending on whether we use TCP or UDP. (i.e. Weather we choose `SOCK_STREAM` or `SOCK_DGRAM` as arguments in the `socket()` call). The positive side is that we have fewer calls to bother about and that applications developed using UDP easily can be transformed to TCP. That is particularly true in the case of a simple echo client.

In the server we have two calls, `listen()` and `accept()` that we have’t seen before. The `listen()` call is straight forward and easy to use whereas the `accept()` call is somewhat more difficult. However, it is essential that we understand the functionality of `accept()` in order to use it correctly. Notice also that we don’t use `recvfrom()` and `sendto()` to receive and send data as we did in the UDP echo server case. One of the features of `accept()` is that it accepts incoming connections and creates a new socket containing both parties’ endpoint information. The TCP echo server is therefore a connection-oriented application.
listen() - listen for connections on a socket

SYNOPSIS

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

int listen(int sockfd, int backlog);

DESCRIPTION

Servers use listen() to make the socket sockfd ready to accept incoming requests. The listen() call applies only to sockets used with TCP. The backlog parameter defines the maximum number of incoming connection requests that the application should enqueue for a given socket while the server handles another request.

RETURN VALUES

Upon successful completion, listen() returns 0. Otherwise, -1 is returned and errno is set to indicate the error.

accept() - accept a connection on a socket

SYNOPSIS

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

int accept(int sockfd, struct sockaddr *addr, int *addrlen);

DESCRIPTION

The argument sockfd is a socket that has been created with the call socket() and bound to a local endpoint with bind(), and that is listening for connections after a call to listen(). The accept() function extracts the first connection on the queue of pending connections, creates a new socket newfd with the same properties (PF_INET, SOCK_STREAM) as sockfd but with both parties' endpoint information stored. If no pending connections are present, accept() blocks the caller until a connection is present. This is the call on which the incoming connection will be accepted. The accepted socket, newfd, is used to read and write data to and from the socket that connected. The original socket sockfd remains open for accepting further connections.

The argument addr is a result parameter that is filled in with the address of the remote part. It works the same way as in recvfrom. That is, in order to use a struct sockaddr_in as an argument, we have to cast it while we make the call.

The argument addrlen is a value-result parameter. Initially, it contains the amount of space pointed to by addr; on return it contains the length in bytes of the address returned.

The accept() function applies only to stream sockets created using SOCK_STREAM.

RETURN VALUES

The accept() function returns -1 on error. If it succeeds, it returns a non-negative integer newfd that is a descriptor for the accepted socket.

EXAMPLE OF USAGE

addrsize = sizeof(struct sockaddr_in);
if ((newfd = accept(sockfd, (struct sockaddr *) theiraddr, &addrsize)) == -1) {
    perror("accept");
    exit(-1);
}

Notice that the use of accept() requires two file descriptors. One (sockfd) that is used to wait for incoming connections, and a second (newfd) that is used in the following communication with the client.
Like the connectionless server described in the previous assignment, a server using TCP must run forever. After creating a socket `sockfd` that listens at the well-known port specified in the `bind` call, the server enters an infinite loop in which it accepts and handles connections using `newfd`.

**Concurrent Servers**

The term *concurrent servers* is used to describe servers that are capable of handling many connections at the same time. The primary reason for introducing concurrency into a server arises from a need to provide faster response times to multiple clients. Concurrency implies faster response time if:

1. forming a response requires significant I/O,
2. the processing time required varies among requests,
3. the server executes on a computer with multiple processors.

In the first case, concurrency means that a server can overlap the use of the processor and external units needed for the I/O. While the processor computes one response, the I/O device can handle other responses. In the second case, timeslicing permits a single processor to handle short requests without waiting for requests that take longer. In the third case, concurrent execution allows one processor to handle one request while another processor computes a response to another request.

In what follows we will present two different techniques to implement concurrency into servers.

**Multi-Process Concurrency Using `fork()`**

This technique is based on the system call `fork()` that creates a new process. The parent process opens a socket at the well-known port, waits for the next request, and creates a child process to handle any incoming connections. The parent never communicates directly with a client - it passes that responsibility to a child. After a child forms a response and sends it to a client, it exits. The code fragments that follow show how we can use `fork()` and `signal()` to implement a concurrent server that immediately after a child is created to handle the request, returns to the `accept` call, ready to accept a new request. We only give a short description since we expect that you are familiar with the `fork` call.

```c
/* Code fragment that uses fork() and signal() to implement concurrency */
/* various include and define statements */

void signal_handler(int sig) {
  int status;
  switch (sig) {
    case SIGINT:
      printf("Ctrl-C caught\n");
      close(sockfd);
      exit(-1);
      case SIGCHLD:
      wait(&signal);
      signal(SIGCHLD,&signal_handler); /* restarts signal handler */
      break;
    default:;
  }
}

main(int argc, char *argv[]) {
  /* Various declarations */
  /* The calls Socket(), GetMyAddres(),Bind(), and Listen() */
```
signal(SIGCHLD,&signal_handler);
signal(SIGINT,&signal_handler);

while(1) { /* infinite accept() loop */
    newfd = accept(sockfd,(struct sockaddr *)&theiraddr,&sinsize);
    if (newfd < 0) { /* error in accept() */
        if (errno = EINTR)
            continue;
        else {
            perror("accept");
            exit(-1);
        }
    }

    switch (fork()) {
    case -1: /* fork() error */
        perror("fork");
        exit(-1);
    case 0: /* child handles request */
        close(sockfd);
        /* read msg and form a response */
        close(newfd);
        exit(-1);
    default: /* parent returns to wait for another request */
        close(newfd);
    }
}

The first thing to notice is the call signal() that informs the system what to do when the signal SIGCHLD is raised (i.e. when a child exits). In this case we want the system to execute the function signal_handler that calls wait() to prevent the child from becoming a zombie, and after that, restart the signal handler. The advantage of using signal to catch the signal SIGCHLD is that the parent can continue to work. (A wait() without a signal handler blocks the parent.)

The server you will write will run until you interrupt its execution by pressing Ctrl-C. The function signal_handler() can handle this case to perform the needed operations before exiting the program. The port the server listens to should be closed in this case to make it easier to restart the server again. If the port is not closed the port will not be available to open as a listen port for some time. This is a security measurement taken by the operating system to make it harder for programs to kidnap ports. If there were no delay a program could interrupt a service, steal its port and in this way compromise the security of the system. This is why we catch the SIGINT interrupt and close the port with sockfd as its descriptor.

The next thing to notice is the way we handle errors in the accept call. If an error of type EINTR (i.e. accept was interrupted by a signal) occurs, we don’t call exit(), instead we call continue() that restarts the loop and thereby redirects the execution point to the accept() call once again. If we didn’t take care of this particular situation, then every time a child exits, and the signal SIGCHLD is raised, while the parent is waiting at accept for a new connection, an error would be raised and the program would exit. The construction where we treat EINTR separately is a necessary measure if we want the program to quit when errors different from EINTR occurs. And that’s what we want!

The concurrency achieved on a single processor system when we use fork() is based on time-slicing. That is, the operating systems capability to divide its CPU resources among different processes. Thus, the server is not strictly concurrent, it just appears to be so. A different approach is taken in the next section where we use select() to implement concurrency. This is a single process technique where “concurrency” means that a server is capable of handling many connections at the same time, although the various requests are handled sequentially. Remember that a connection may involve many requests.
Single-Process Concurrency Using select()

This technique is based on the system call select (presented in assignment 2) that works with sets of file descriptors. The select() function indicates which of the specified file descriptors is ready for (in this case) reading. If the specified condition is false for all of the specified file descriptors, select() blocks until the specified condition is true for at least one of the specified file descriptors.

In the example server that follows each incoming connection results in a new socket that is placed in file descriptor set fd_set. We then use select() to wait for requests to arrive on any of them. Once select() indicates an activity, we use FD_ISSET to find out which one of the filedescriptors that is ready for reading.

/* Code fragment that uses select() to implement concurrency */
/* various include and define statements */
main(int argc, char *argv[]) {
    /* various declarations */
    int result;
    fd_set readfds, testfds;
    /* Socket(), GetMyAddress(), Bind(), Listen() */
    FD_ZERO(&readfds);
    FD_SET(sockfd,&readfds);
    while(1) {
        int fd;
        testfds = readfds;
        result = select(FD_SETSIZE, &testfds, NULL, NULL, NULL);
        /* error check for select */
        for (fd = 0; fd < FD_SETSIZE; fd++) {
            if (FD_ISSET(fd,&testfds)) { /* find activated socket fd */
                if (fd == sockfd) { /* ==> New connection */
                    newfd = accept(sockfd,(struct sockaddr *)&theiraddr,&sinsize);
                    FD_SET(newfd,&readfds); /* update file descriptor set with newfd */
                } else { /* ==> request from ‘old’ connection */
                    close(fd);
                    FD_CLR(fd,&readfds); /* remove fd when connection finnished */
                }
            }
        }
    return 1;
    }
}

Notice here that the socket sockfd is only used to “catch” new connections. If select() indicates an activity on sockfd, we therefore conclude that it is a new connection, accept it with accept(), and stores the new file descriptor in readfds with FD_SET.

If select() indicates an activity on a file descriptor in readfds that is not sockfd, we conclude that this is a request from an already connected client, and calls read() to handle that request.

The advantage of using select() is that the server now can switch between different users instead of handling them one at the time sequentially. We no longer allow a single client to block
the server. For example, if a client that is already connected is slow in transmitting a request, the server can handle another client between connection and request of the former. Notice however that a client’s request never gets interrupted once the server has started to work on it, no matter how long time the server needs to handle it.

Exercises

1. Simple echo using TCP: Implement an echo client that uses TCP according to the description given in Figure 1 and call them tcp_echo_clnt.c and tcp_echo_srv.c. Pick a string of your own choice to be the echo message. We recommend you to reuse the functions already declared in udp.h (assignment 2) and augment it with a set of TCP specific functions. If you modify any of the “old” functions, make sure that they still work in the UDP case. Furthermore, all system calls should handle errors using perror.

2. Echo questions: Monitor an echo session with snoka using the default option. Describe briefly the purpose of each one of the messages that is sent. How many packets are involved in one echo session? Modify tcp_echo_clnt.c so that you can easily change the size (measured in bytes) of the data transmitted. Then try to find the upper limit for a single segment. What happens when a packet is transmitted that is larger than the upper limit? What determines the upper limit? (No code is needed in the report. Only thorough answers based on facts from various executions and snoka sessions.)

3. Echo server using both UDP and TCP: Implement an echo server (echo_server.c) that can handle both UDP and TCP. Hint: Use two sockets (one for each protocol) that both listen to the same port and use select() to decide which socket to be used for each incoming echo request. This is the server program to be handed in together with both a UDP and TCP echo client.

4. Simple fetch: In assignment 2 you made a client-server pair (fetch.c and fetchserver.c) that used UDP to fetch files from a predefined directory and saved them in a local directory. Implement the same functionality using TCP, i.e. implements the same fetch protocol. This exercise is just a preparation for the following two exercises. However, we recommend it as a starting point.

5. Fetch with fork: Implement a concurrent fetch server (and call it fetchserver_fork.c) that uses fork() according to the description earlier in this text.

6. Fetch with select: Implement a concurrent fetch server (and call it fetchserver_select.c) that uses select() according to the description earlier in this text.

7. Fetch questions: Discuss the differences between using fork() and select() when you implement a concurrent server. Advantages and drawbacks? Under which circumstances should they be used? Try also to figure out methods that verify that the two servers (fetchserver_fork.c and fetchserver_select.c) really work concurrently. Hand in results from executions that prove concurrency and describe how the execution was done.

What to be handed in

1. Exercise 3: udp_echo_clnt.c, tcp_echo_clnt.c, and echo_server.c together with any header and definition files used.

2. Exercise 4, 5 and 6: fetch.c, fetchserver_fork.c, and fetchserver_select.c together with any header and definition files used.

3. A single text document containing the answers from exercises 2 and 7.

Don not forget to follow the instructions on how to hand in your results, which can be found on the last page of the PM for assignments 1 and 2.

The tutor will be happy if you, together with the programs, also hand in a suitable makefile. If you are not familiar with makefiles and how to create one, you can search Google using the keywords makefile tutorial. One of the results from this search is the URL http://www.eng.hawaii.edu/Tutor/Make/.