TCP and Concurrency

The third assignment at DA2402

2009-03-05
Jonas Lundberg/Ola Flygt
adapted to Java by Marcus Edvinsson
maintained by Marcus Edvinsson
Matematiska och systemtekniska institutionen, MSI
Växjö universitet

(email: Ola.Flygt@vxu.se)

This assignment should be finished and handed in before April 3, 2009.

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 sequences numbers and a variable sized sliding windows. For more details about TCP we refer to the course literature. 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 method calls involved in a single echo session. Notice the similarities and differences between this set of calls and the one we used to build the client/server pair in the previous assignment. It is still only one short string that is sent between the client
and the server. 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.

If we start to look at the echo session from the client point of view we first notice that there is a `connect` call. In the UDP case we used `bind` to add remote endpoint information to the socket. In the TCP case `connect` initiates contact with the remote part using a *three-way handshake* procedure. Since TCP uses a stream-based approach to transfer data, we send and receive data using the method calls `read` and `write` instead of `receive` and `send`. Notice also that in the TCP case both `write()` and `read()` have acknowledgement handling built in automatically.

In the server we have one new call, `accept()` that we have’t seen before. 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. The socket created when a client connects is used to communicate with the client from a new thread that lives as long as the communication lasts between the server and that client. The server goes back to listen for incoming connections as soon as the new thread is created and started.

Like the connectionless server described in the previous assignment, a server using TCP must run forever. After creating the socket `ServerSocket` that listens at the well-known port specified when instantiating the `ServerSocket` class, the server enters an infinite loop in which it accepts
and handles connections using \texttt{clientSocket}.

\section*{Concurrent Servers}

The term \textit{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.

\subsection*{Multi-Process Concurrency Using Threads}

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

```java
/* Various class import statements */
public static void main(String[] args) throws IOException {
    /* Various declarations needed */
    while (true) { // The server runs forever
        final Socket clientSocket = serverSocket.accept();
        new Thread() {
            public void run() {
                InputStream is = clientSocket.getInputStream();
                /* Reading the client request and echoing the response */
            }
        }.start();
    }
}
```

The server you will write will run until you interrupt its execution (by pressing Ctrl-C or otherwise kills the running server). For each iteration of the eternal loop, the server calls \texttt{accept} that blocks until a client connects on the port the server listens to. When a client connects a \texttt{Socket} object is returned. This object is used to communicate with the client.

A new thread object is created using an anonymous class. The code that should run is written in the \texttt{run} method. When the \texttt{start} method is called the code in the \texttt{run} method is executed in a new thread, i.e., \textit{concurrently} with other client requests and the server code. The \texttt{clientSocket} variable is declared \texttt{final} since it is referenced from within the thread object. The \texttt{clientSocket} should not be able to change after it has been initialized.
Exercises

1. **Simple echo using TCP:** Implement an echo client that uses TCP according to the description given in Figure 1 and call them TCPEchoClient.java and TCPEchoServer.java. Pick a string of your own choice to be the echo message.

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 TCPEchoClient.java 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. **Simple fetch:** In assignment 2 you made a client-server pair (Fetch.java and FetchServer.java) 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.

4. **Fetch with threads:** Implement a concurrent fetch server (and call it FetchServer.java) that uses java.lang.Thread according to the description earlier in this text.

5. **Fetch questions:** Try to figure out methods that verify that the server (FetchServer.java 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 2: TCPEchoClient.java and TCPEchoClient.java together with any other files used.

2. Exercise 4: Fetch.java and FetchServer.java together with any other files used.

3. A single text document containing the answers from exercises 2 and 5.

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.