

MAWLANA BHASHANI SCIENCE AND TECHNOLOGY UNIVERSITY

Santosh, Tangail – 1902



Course Title: Computer Networks Lab

Lab Report Name: Socket Programming(Daytime protocol)

Lab Report No: 05

Submitted by,

Name: Tazneen Akter &

Jannatul Ferdush Dhina

ID: IT-18056 & 18012

Session: 2017-18

Dept. of ICT, MBSTU

Submitted to,

NAZRUL ISLAM

Assistant Professor

Dept. of ICT, MBSTU

Socket Programming:

Sockets provide the communication mechanism between two computers using TCP. A client program creates a socket on its end of the communication and attempts to connect that socket to a server.

When the connection is made, the server creates a socket object on its end of the communication. The client and the server can now communicate by writing to and reading from the socket

. The `java.net.Socket` class represents a socket, and the `java.net.ServerSocket` class provides a mechanism for the server program to listen for clients and establish connections with them.

The following steps occur when establishing a TCP connection between two computers using sockets –

- The server instantiates a `ServerSocket` object, denoting which port number communication is to occur on.
- The server invokes the `accept()` method of the `ServerSocket` class. This method waits until a client connects to the server on the given port.
- After the server is waiting, a client instantiates a `Socket` object, specifying the server name and the port number to connect to.
- The constructor of the `Socket` class attempts to connect the client to the specified server and the port number. If communication is established, the client now has a `Socket` object capable of communicating with the server.
- On the server side, the `accept()` method returns a reference to a new socket on the server that is connected to the client's socket.

1) Briefly explain the term IPC in terms of TCP/IP communication.

Answer:

InterProcess Communication is a term we use for interactions between two processes on the same host. William Westlake mentions TCP/IP, which is used for interactions with another host. It can be used locally as well, but it is relatively inefficient. Unix domain sockets are used in the same way, but are only for local use and a bit more efficient.

The answer will be different each Operating System. Unix offers System V IPC, which gives you message queues, shared memory, and semaphores.

Message queues are easy to use: processes and threads can send variable-sized messages by appending them to some queue and others can receive messages from them so that each message is received at most once. Those operations can be blocking or non-blocking. The difference with UDP is that messages are received in the same order as they are sent. Pipes are simpler, but pass a stream of data instead of distinct messages. Line feeds can be used as delimiters.

Shared memory allows different processes to share fixed regions of memory in the same way that threads have access to the same memory. Unix allocates a certain amount of memory after which it can be accessed like private memory. Since two threads updating the same data can lead to inconsistencies, semaphores can be used to achieve mutual exclusion. A similar mechanism is the memory-mapped file: the difference is that the memory segment is initialised from a disc file and changes can be permanent. The size of a file can change, which complicates shared files.

2) What is the maximum size of a UDP datagram? What are the implications of using a packet-based Protocol as opposed to a stream protocol for transfer of large files?

Answer:

It depends on the underlying protocol i.e., whether you are using IPv4 or IPv6.

- In IPv4, the maximum length of packet size is 65,536. So, for UDP datagram you have maximum data length as:

65,535 bytes - 20 bytes(Size of IP header) = 65, 515 bytes (including 8 bytes UDP header)

- In IPv6, the maximum length of packet size allowed is 64 kB, so, you can have UDP datagram of size greater than that.

NOTE: This size is the theoretical maximum size of UDP Datagram, in practice though, this limit is further constrained by the MTU of data-link layer(which varies for each data-link layer technology, but cannot be less than 576 bytes), considering that, maximum size of UDP datagram can be further calculated as (for IPv4):

- 576 bytes - 20 bytes(IP header) = 556 (including 8 bytes UDP header)

3) TCP is a reliable transport protocol, briefly explain what techniques are used to provide this reliability.

Answer:

A number of mechanisms help provide the reliability TCP guarantees. Each of these is described briefly below.

Checksums. All TCP segments carry a checksum, which is used by the receiver to detect errors with either the TCP header or data

. Duplicate data detection. It is possible for packets to be duplicated in packet switched network; therefore TCP keeps track of bytes received in order to discard duplicate copies of data that has already been received.

Retransmissions. In order to guarantee delivery of data, TCP must implement retransmission schemes for data that may be lost or damaged. The use of positive acknowledgements by the receiver to the sender confirms successful reception of data. The lack of positive acknowledgements, coupled with a timeout period calls for a retransmission.

Sequencing. In packet switched networks, it is possible for packets to be delivered out of order. It is TCP's job to properly sequence segments it receives so it can deliver the byte stream data to an application in order.

Timers. TCP maintains various static and dynamic timers on data sent. The sending TCP waits for the receiver to reply with an acknowledgement within a bounded length of time. If the timer expires before receiving an acknowledgement, the sender can retransmit the segment.

4) Why are the htons(), htonl(), ntohs(), ntohl() functions used?

Answer:

These are used for:

htons() host to network short

htonl() host to network long

ntohs() network to host short

ntohl() network to host long

5) What is the difference between a datagram socket and a stream socket?

Answer:

The difference is given below:

- Stream sockets enable processes to communicate using TCP. A stream socket provides a bidirectional, reliable, sequenced, and unduplicated flow of data with no record boundaries. After the connection has been established, data can be read from and written to these sockets as a byte stream. The socket type is SOCK_STREAM.
- Datagram sockets enable processes to use UDP to communicate. A datagram socket supports a bidirectional flow of messages. A process on a datagram socket might receive messages in a different order from the sending sequence. A process on a datagram socket might receive duplicate messages. Messages that are sent

over a datagram socket might be dropped. Record boundaries in the data are preserved. The socket type is SOCK_DGRAM.

Daytime Protocol implementation: 2.2.1

Server Program:

```
package je3.nio;

import java.nio;
import java.nio.channels;
import java.nio.charset;
import java.net;
import java.util.logging;
import java.util;

/**
 * A more robust daytime service that handles TCP and UDP connections and
 * provides exception handling and error logging.
 */
public class DaytimeServer {

    public static void main(String args[ ]) {

        try {

            // Handle startup exceptions at the end of this block

            // Get an encoder for converting strings to bytes

            CharsetEncoder encoder = Charset.forName("US-ASCII").newEncoder( );
```

```
// Allow an alternative port for testing with non-root accounts

int port = 13; // RFC867 specifies this port.

if (args.length > 0) port = Integer.parseInt(args[0]);


// The port we'll listen on

SocketAddress localport = new InetSocketAddress(port);


// Create and bind a TCP channel to listen for connections on.

ServerSocketChannel tcpserver = ServerSocketChannel.open( );

tcpserver.socket( ).bind(localport);


// Also create and bind a DatagramChannel to listen on.

DatagramChannel udpserver = DatagramChannel.open( );

udpserver.socket( ).bind(localport);


// Specify non-blocking mode for both channels, since our
// Selector object will be doing the blocking for us.

tcpserver.configureBlocking(false);

udpserver.configureBlocking(false);


// The Selector object is what allows us to block while waiting
// for activity on either of the two channels.

Selector selector = Selector.open( );
```

```
// Register the channels with the selector, and specify what
// conditions (a connection ready to accept, a datagram ready
// to read) we'd like the Selector to wake up for.
// These methods return SelectionKey objects, which we don't
// need to retain in this example.
tcpserver.register(selector, SelectionKey.OP_ACCEPT);
udpserver.register(selector, SelectionKey.OP_READ);

// This is an empty byte buffer to receive empty datagrams with.
// If a datagram overflows the receive buffer size, the extra bytes
// are automatically discarded, so we don't have to worry about
// buffer overflow attacks here.
ByteBuffer receiveBuffer = ByteBuffer.allocate(0);

// Now loop forever, processing client connections
for() {
    try {
        // Handle per-connection problems below
        // Wait for a client to connect
        selector.select( );

        // If we get here, a client has probably connected, so
        // put our response into a ByteBuffer.
```



```
String date = new java.util.Date( ).toString( ) + "\r\n";

ByteBuffer response=encoder.encode(CharBuffer.wrap(date));


// Get the SelectionKey objects for the channels that have
// activity on them. These are the keys returned by the
// register( ) methods above. They are returned in a
// java.util.Set.

Set keys = selector.selectedKeys( );


// Iterate through the Set of keys.
for(Iterator i = keys.iterator( ); i.hasNext( ); ) {

    // Get a key from the set, and remove it from the set
    SelectionKey key = (SelectionKey)i.next( );
    i.remove( );


    // Get the channel associated with the key
    Channel c = (Channel) key.channel( );


    // Now test the key and the channel to find out
    // whether something happened on the TCP or UDP channel
    if (key.isAcceptable( ) && c == tcpserver) {

        // A client has attempted to connect via TCP.
        // Accept the connection now.
        SocketChannel client = tcpserver.accept( );
```

```

        // If we accepted the connection successfully,
        // then send our response back to the client.
        if (client != null) {
            client.write(response); // send response
            client.close( );      // close connection
        }
    }

    else if (key.isReadable( ) && c == udpserver) {
        // A UDP datagram is waiting. Receive it now,
        // noting the address it was sent from.
        SocketAddress clientAddress =
            udpserver.receive(receiveBuffer);

        // If we got the datagram successfully, send
        // the date and time in a response packet.
        if (clientAddress != null)
            udpserver.send(response, clientAddress);
    }
}

catch(java.io.IOException e) {
    // This is a (hopefully transient) problem with a single
    // connection: we log the error, but continue running.
    // We use our classname for the logger so that a sysadmin
    // can configure logging for this server independently

```

```

        // of other programs.

        Logger l = Logger.getLogger(DaytimeServer.class.getName( ));

        l.log(Level.WARNING, "IOException in DaytimeServer", e);
    }

    catch(Throwable t) {

        // If anything else goes wrong (out of memory, for example),
        // then log the problem and exit.

        Logger l = Logger.getLogger(DaytimeServer.class.getName( ));

        l.log(Level.SEVERE, "FATAL error in DaytimeServer", t);

        System.exit(1);

    }

}

}

catch(Exception e) {

    // This is a startup error: there is no need to log it;

    // just print a message and exit

    System.err.println(e);

    System.exit(1);

}

}

}

```

Client Program:

```

package je3.nio;

import java.net;

```

```

/**
 * Connect to a daytime server using the UDP protocol.
 * We use java.net instead of java.nio because DatagramChannel doesn't honor
 * the setSoTimeout( ) method on the underlying DatagramSocket
 */
public class DaytimeClient {

    public static void main(String args[ ]) throws java.io.IOException {

        // Figure out the host and port we're going to talk to

        String host = args[0];

        int port = 13;

        if (args.length > 1) port = Integer.parseInt(args[1]);

        // Create a socket to use

        DatagramSocket socket = new DatagramSocket( );

        // Specify a 1-second timeout so that receive( ) does not block forever.

        socket.setSoTimeout(1000);

        // This buffer will hold the response. On overflow, extra bytes are
        // discarded: there is no possibility of a buffer overflow attack here.

        byte[ ] buffer = new byte[512];

        DatagramPacket packet = new DatagramPacket(buffer, buffer.length,
                                                    new InetSocketAddress(host,port));
    }
}

```

```

// Try three times before giving up
for(int i = 0; i < 3; i++) {
    try {
        // Send an empty datagram to the specified host (and port)
        packet.setLength(0); // make the packet empty
        socket.send(packet); // send it out

        // Wait for a response (or time out after 1 second)
        packet.setLength(buffer.length); // make room for the response
        socket.receive(packet); // wait for the response

        // Decode and print the response
        System.out.print(new String(buffer, 0, packet.getLength( ),
                                   "US-ASCII"));

        // We were successful, so break out of the retry loop
        break;
    }
    catch(SocketTimeoutException e) {
        // If the receive call timed out, print error and retry
        System.out.println("No response");
    }
}

```

```
// We're done with the channel now  
socket.close( );  
}  
}
```