Homework 5 - Data Link Protocol Implementation

Document last modified: 09/15/2004 11:18:42

Protocols Overview Several data link layer protocol examples are given in the Tanenbaum text. The *network* and *physical layer* operations used by the data link layer in Tanenbaum have been implemented in Java using the *User Datagram Protocol* of the Internet protocol set. The Java implementation is very similar to that given in the text on page 202 for C. Essentially, the network and physical layer protocols have been implemented. Based upon those functions, it is possible to implement data link protocols 1 through 4 on a single or multiple workstations running Java.

Protocol Discussion

The Java implementation follows that of the text as closely as reasonably possible, remembering that Java is object-oriented and the text example is not. The Java definitions that correspond to that of page 202 follow:

```
public class Protocol {
        public final static int frame_arrival = 0;
        public final static int chksum_err = 1;
        public final static int timeout = 2;

        public static void to_network_layer(String s);
        public static String from_network_layer();
}

public class Frame {
        public String info;
        public int seq;
        public int ack;
```

```
    Frame(int socket);
    public String from_physical_layer();
    public void to_physical_layer(String remoteAddress, int remotePort);
    public int wait_for_event();
    public void start_timer(int k);
    public void stop_timer(int k);
}
```

Differences between Java and Text Implementation

Note that only the functionality needed for Protocols 1-4 has been implemented in the two following Java classes.

1. **Protocol**
   - packet - A packet is the data load of a frame. Instead of *packet* uses the Java *String* class.
     1. public static void to_network_layer(String s); - Use String rather than *packet*.
     2. public static String from_network_layer( ); - Returns a String as a function result rather than a *packet* as a parameter.
2. **Frame**
   1. Frame(int port); - Creates a new Frame given a port number.
   2. public String from_physical_layer( ); - Returns the message data load as a String from the physical layer, and the fields of the frame object (*seq, ack,* and *info)* are initialized.
   3. public void to_physical_layer(String remoteAddress, int remotePort); - Sends a Frame through the physical layer to the host at *remoteAddress* and port number *remotePort.*
   4. public int wait_for_event( ); - Wait for an event and return the *event* number, one of either *frame_arrival, chksum_err,* or *timeout.*
   5. public void start_timer(int k); - Start timer number *k.*
   6. public void stop_timer(int k); - Stop timer number *k.*

Assignment

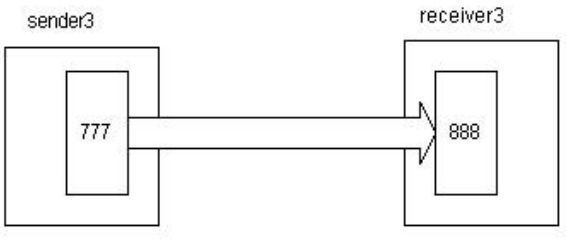
1. Verify protocol 3 operation (given below) and implement protocols 1, 2, and 4 from the text to further your understanding of each. Note that the *Protocol from_network_layer* and *to_network_layer* functions do not issue prompts in case you're using this protocol interactively.
2. When running the protocols on *one* host, the *to_physical_layer* parameter for *remoteAddress* will be "*localhost".* The *sender* and *receiver* must send to the other's ports.

   If the receiver listens for frames to arrive on port 888, the sender must

transmit to port 888. In *sender3* of protocol 3, the following instruction defines the *local* port that will be used to send and receive frames:

```
Frame s = new Frame(777);
```

The program instruction:




```
s.to_physical_layer("localhost",
888);
```

actually sends the *frame* s to, in this case on the same host *localhost* and the *receiver3* port 888.

Note that protocol 4 is *symmetric*, having only one program that both sends and receives. Since two programs cannot share the same *port* on the same machine, two versions of protocol 4 are needed when run on one host. Name one *protocol4A* and the other *protocol4B* and use two *different ports* in the *Frame(port)* constructor in each version. Note that this is not necessary when running on two *different host* machines, both machines can use the same ports.

3. Run protocols 1, 2, 4 on two different hosts, sending on one and receiving on the other. You will need to determine the IP of each machine by entering *arp -a* at a DOS window (or by entering *ipconfig /all* for W2K) on each which should display the machine's IP in a format similar to *Interface: 149.160.29.93,* then edit the IP into the *to_physical_layer* parameter for *remoteAddress* of each program. In Protocol 3 below, the change would be:

```
s.to_physical_layer("149.160.29.93", 888);
```

Alternatively make two separate directories with complete copies of files and change one protocol copy to use different send and receive ports as was done in protocol 3.

4. Modify the *timeout* parameter in the Constants.java file for protocol 4 until timeouts occur.

```
public static final int buffersize = 13;
public static final int lostframes = 0;
public static final int timeout = 1000;
```

You may need to insert an:

if (event != frame_arrival) System.out.println("timeout");

in protocol 4 to notify you that a timeout actually occurred.

5. Modify the *Constants.java* file to vary the size of the data buffers. The *buffersize* determines the number of bytes sent to the physical layer and received from the physical layer. Run for protocols 1, 2, 4 for a large and small data buffer size. Note that both hosts should use the same size buffers or truncation will occur.
6. Modify the *Constants.java* file to vary the number of frames lost. Run for protocols 1, 2, 4 for a large and small number of lost frames.

public static final int buffersize = 13;
public static final int lostframes = 0;
public static final int timeout = 1000;

Turn In

1. Cover sheet with your name, date, Homework 5. Please staple all pages together.
2. Listing of each sender/receiver file for protocol 1, 2, and 4 (protocol 4 needs only one).
3. Observations of running protocols on *single* or *multiple* machines. Discuss differences and give supporting outputs or other evidence.
4. Observations on modifying *timeout* parameter in file *Constants.java* on appropriate protocols. Use at least three different timeouts from 10 ms. to 5000 ms. Discuss differences, if any, witnessed for each protocol tested and whether the behavior was as expected for each protocol. Give supporting outputs or other evidence.
5. Observations on modifying *buffersize* parameters in file *Constants.java* and sending a large file for protocols 1, 2, 4. Use at least three different sizes from 1 to 2000. Discuss differences witnessed for each and give supporting outputs or other evidence. Note that the *testdata* file can be sent by:
   - *java -cp . sender3 < testdata*
6. Observations on modifying l*ostframes* parameter in file *Constants.java* and sending a file for protocols 1, 2, 4. Use at least three different numbers from 0 to 20. Discuss differences witnessed for each protocol and give supporting outputs or other evidence. Note that the *testdata* file can be sent by:
   - *java -cp . sender3 < testdata*
7. The *testdata* is a large file filled with sequential digits 0..9 to more easily recognize when a protocol failure occurs.
8. The protocols are designed for two communicating hosts. Consider whether any (or all) of the protocols could be spoofed by a third sender. Try one and discuss the results and give supporting outputs or other evidence. Remember that the

ports of all senders and receivers must be unique on the same machine by changing the following:

```
Frame s = new Frame(777);
```

Hints

- Protocol 4 is symmetric, the same program runs on both hosts (with the exceptions for different IP numbers noted earlier in the assignment). It requires that *both* hosts have something to send, such as a file.
- Remember that the timeout event should occur on the *receive* frame. In Protocol 3 below: *r.start_timer(s.seq);*
- Protocol 4 of the text uses two frames, *r* for receiving, and *s* for sending. Because of the way frames are defined in the Java implementation (the text didn't have to worry with details such as whether the protocol would actually work) you will need to associate a *port* for each frame. A reasonable definition of *s* and *r* is:

```
Frame s = new Frame(777);
Frame r = new Frame(888);
```

  which defines *s* on port 777 and *r* on port 888.

- When running Protocol 4 on one machine, remember that ports must be unique for the machine. If one Protocol 4 has the port definitions above, the other Protocol 4 program must use different port numbers in the *Frame* definition of *r* and *s.*

Example - Protocol 3

Protocol 3 from Tanenbaum is implemented below as an example using the Java network and data link definitions. Note that one weakness of all the protocols is that the network layer is expected to always have data available. Consult the text and notes for details of Protocol 3.

Use

- Protocol 3 consists of 2 separate programs, *sender3.java* and *receiver3.java,* which are listed below, along with three other files that implement the network and part of the data link layer. Create a directory and copy all of the following files to that directory just created. Click on each to copy, the right button in IE will allow the file to be saved directly.

1. sender3.java
2. receiver3.java
3. Protocol.java
4. Frame.java
5. DataGram.java
6. Constants.java
7. testdata.txt

- Compile each of the files at IUS (see Java Development Kit for home use) by opening a DOS window and entering:
  - *v:\common\user\b438\forJava*
  - javac *.java.
- Open a second DOS windows and change to the directory where the above files were compiled.
- Enter in one DOS window (need to start the *receiver* before the *sender* in order not to miss anything):
  - *v:\common\user\b438\forJava*
  - java -cp . receiver3
- Enter in the other window (note the use of < for redirecting a file as input to the *sender3* program*)*:
  - *v:\common\user\b438\forJava*
  - java -cp . sender3 < testdata.txt

  The *sender3* program should transmit to the *receiver3* program the *testdata* file for display to the screen. Note that both programs continue to execute after the file is finished because of our assumption that there will *always* be data available from the network layer. Control C will terminate a Java program.

- Modify the *Constants.java* file to use a larger or smaller buffersize from the current setting of 13 bytes per frame. When the buffersize is small the overhead of transmitting a frame is greater causing the protocol to run more slowly. Delete all class files by:

  del *.class

- Modify the *Constants.java* file to lose frames by setting *lostframes* to a nonzero value. A value of 5 will cause 1 frame in 5 to be lost, this is not random, in this example the fifth frame of five is simply discarded. To clearly see the effects use a buffersize of 3 and and lostframes of 2.

```
public class Constants {
        public static final int buffersize = 3;
        public static final int lostframes = 2;
        public static final int timeout = 1000;
}
```

- Modify the *timeout* of the the *Constants.java* file for sender3.java

acknowledgement receive frame to reduce the delay when resending lost frames.

- Using different hosts for *sender3* and *receiver3* requires a minor program modification to each. Both *sender3* and *receiver3* use *localhost* when running on the same machine. To run on separate machines each must send frames through the physical layer to the IP of the other. Necessary changes are:

receiver3.java on host with IP 149.160.27.222 s.to_physical_layer("**149.160.27.111**", 777);	**sender3**.java on host with IP 149.160.27.111 s.to_physical_layer("**149.160.27.222**", 888);

Sender3 and Receiver3 - Changeable parameters:

- IP is *localhost* or the machine on which the program is running.

```
public class sender3 extends Protocol {
   public static void main(String args[])
throws Exception
   {  int next_frame_to_send;
      Frame r = new Frame(777);
      Frame s = new Frame(666);
      String buffer;
      int event;

      next_frame_to_send = 0;
      buffer = from_network_layer();
      while (true)
      {      s.info = buffer;
             s.seq =
next_frame_to_send;

s.to_physical_layer("localhost",
888);
             r.start_timer(s.seq);
             event = r.wait_for_event();
             if (event == frame_arrival)
            {   r.from_physical_layer();
                if (r.ack ==
next_frame_to_send)
                {   buffer =
from_network_layer();
                    next_frame_to_send
=

(next_frame_to_send + 1) % 2;
                }
            }
```

```
public class receiver3 extends Protocol {
   public static void main(String args[])
throws Exception
   {   int frame_expected;
       Frame r = new Frame(888);
       Frame s = new Frame(999);
       int event;

       frame_expected = 0;
       while (true)
       {   event = r.wait_for_event();
           if (event == frame_arrival)
          {   r.from_physical_layer();
              if (r.seq == frame_expected)
             {   to_network_layer(r.info);
                 Frame_expected =
(frame_expected + 1) % 2;
             }
             s.ack = 1 - frame_expected;

s.to_physical_layer("localhost", 777);
          }
       }
   }
}
```

```
    }
  }
}
```

Java Listing of Data Link Layer Implementation

The following four class definitions, Constants, Protocol, Frame, and DataGram, implement the functionality described by the text. Only Constants will need to be modified.

Constants.java - Defines three constants.

- *buffersize* fixes the maximum number of bytes sent and received by a protocol. When the buffersize of the sender is greater than that of the receiver a larger number of bytes are sent than the receiver can receive at one time. The receiver discards any excess bytes.
- *lostframes* defines the number of frames that will be lost by the physical layer. A value of 0 will lose zero frames, a value of 5 will lose 1 out of 5 frames.
- *timeout* defines time interval of a timer. After being started, a timer will timeout after *timeout* milliseconds.

```
public class Constants {
	public static final int buffersize = 13;
	public static final int lostframes = 0;
	public static final int timeout = 1000;
}
```

Protocol.java - Serves to define *events* and *network* functions. When implementing a protocol, one must inherit Protocol to have access to these definitions, for example: *public class **sender3** extends Protocol.*

```
public class Protocol {
	public final static int frame_arrival = 0;
	public final static int chksum_err = 1;
	public final static int timeout = 2;

	public static void to_network_layer(String s)
	{	System.out.print(s);
	}

	public static String from_network_layer()
	{	byte buffer[] = new byte[Constants.buffersize];
		try { System.in.read(buffer); }
		catch (Exception e) {};
		String s = new String(buffer,0);
		return s;
```

```
    }
}
```

Frame.java - Frame is actually the majority of the Data Link layer, providing access to the *physical layer.* One key difference between the text and Java implementation is the requirement that Frame's be created for a *port* number on the host for sending and receiving and a *time limit* for waiting a Frame arrival. Note that the *port* can only be in use by one Frame. Another point of difference is the need to address the Frame to a specific destination *host* and *port* on the destination.

```
public class Frame {
    public String info;
    public int seq=0;
    public int ack=0;

    private boolean timer = false;
    private int
timelimit=Constants.timeout;
    private DataGram datagram;
    private int lostframes=0;

    Frame(int port)
    {    datagram = new
DataGram(port);
    }

    public String from_physical_layer()
    {    String s = datagram.receive();

         int index;
         index = s.indexOf(" ");
         seq = (new
Integer(s.substring(0,index))).intValue();
         s = s.substring(index+1);
         index = s.indexOf(" ");
         ack = (new
Integer(s.substring(0,index))).intValue();
         info = s.substring(index+1);
         return info;
    }
```

```
  public void to_physical_layer(
     String remoteAddress, int
remotePort)
  {
datagram.send(remoteAddress,
remotePort,
          "" + seq + " " + ack + " "
+ info);
   }

   public int wait_for_event()
   {  int timelimit = 0;
      if(timer) timelimit =
this.timelimit;
      try {
datagram.start_receive(timelimit); }
      catch(Exception e) { return
Protocol.timeout; }
      if(Constants.lostframes > 0 &&
            ++lostframes %
Constants.lostframes == 0)
          return Protocol.timeout;
       return Protocol.frame_arrival;
  }

  public void start_timer(int k)
  {  timer = true;
   }

   public void stop_timer(int k)
   {   timer = false;
    }
}
```

DataGram.java - Used by Frame to access the *physical layer* for sending and receiving. Implemented using User Datagram Protocol. Whenever a datagram containing a Frame is to be received, a thread is started that waits for its arrival. If a timer runs out while waiting for arrival, the thread is stopped, terminating the wait for arrival.

```
// Low-level User Datagram class to send/receive datagram.

import java.net.*;

public class DataGram {
  private byte buffer[]=new byte[Constants.buffersize+4];
  private int remotePort;              // Allow for 1 digit Ack and Seq
  private String remoteAddress;
  private DatagramSocket ds = null;
  private String info;

  DataGram(int localPort)
  {      try { ds = new DatagramSocket(localPort);}
         catch (Exception e) {System.out.println("Exception " + e
                                                 + " port " + localPort ); }
   }

   public void send(String remoteAddress, int remotePort, String s)
   {   byte buffer[]=new byte[s.length()];
       s.getBytes(0, s.length(), buffer, 0);
       try { ds.send(new DatagramPacket(buffer, s.length(),

InetAddress.getByName(remoteAddress), remotePort));
       }
       catch (Exception e) { System.out.println("Exception " + e); };
  }
```

```
 public void start_receive(int timeout)
                  throws Exception
 { ds.setSoTimeout(timeout);
   info = "";
   DatagramPacket p =
        new DatagramPacket(buffer, buffer.length);
   ds.receive(p);
   info = new String(p.getData(), 0, 0, p.getLength());
 }

 public String receive()
 {      return info;
  }
}
```

Java Development Kit

Java requires Win95, NT, W2K, Unix, or other multitasking operating system.

- **Installing**
  - **Home:** It is available free as JDK (Java Development Kit) of SDK via the Sun Java site. For the Sun Java In AUTOEXEC.BAT add**:**

    path = %path%;c:\sdk1.4.0\bin

    assuming that the sdk1.4.0 was installed.

- **Compiling**
  - Create a directory with the Java program, for example d*:\B438\java* on a Zip drive.
  - Change to the directory with the Java program.
  - At DOS enter:

    javac sender3.java

    to compile sender3.java source. To compile all Java files enter:

    *javac *.java*

- **Executing**

  Note that since there is a sender and receiver program, both must be executed, each in a separate DOS window, executing the receiver before the sender (so the receiver is waiting on the sender). At DOS enter:

  java -cp . sender3 < sender3.java

  to execute *sender3* and have it transmit *sender3.java* .

- **Documentation**

  The Sun Java documentation is available using a browser when the JDK is installed. Look at file *index.html* in the Java *docs* directory. Note that you must have installed the documentation.

Document last modified: 09/15/2004 11:18:42