Appendix A
JAVA-Based TFRC Implementation

AckWindowList.java
Control Stream.java
DataStream.java
TFRCCongestion.java
TFRCInputStream.java
TFRCOutputStream.java
TFRCPacket.java
TFRCPacketHeader.java
TFRCServerSocket.java
TFRCSession.java
TFRCSocket.java

AckWindowList.java

```
/*-----------------------------------------------------------------

AckWindowList keeps a list of 8 TFRCPacketHeaders.
They can be updated by calling insert.
There is a toObject and toByteArray functions to switch
   between a byte array and an AckWindowList Object.

 -----------------------------------------------------------------*/

import java.util.LinkedList;
import java.util.Date;

public class AckWindowList
{
   public static final int SIZE = 8;
   public LinkedList LastAcks;

   public AckWindowList(LinkedList Acks)
   {
          LastAcks = Acks;
   }

   public AckWindowList(int start)
   {
          long time = new Date().getTime();
          LastAcks = new LinkedList();
          for (int i = 0; i < SIZE; ++i) {
             LastAcks.add(new TFRCPacketHeader(start, time));
          }
   }

   public synchronized void insert(TFRCPacketHeader TPH)
   {
          LastAcks.removeLast();
          LastAcks.addFirst(TPH);
   }

   //converts AckWindowList into a byte array

   public byte[] toByteArray()
   {
```

```
        byte[] Data = new byte[TFRCPacketHeader.SIZE * SIZE];

        for (int i = 0; i < SIZE; ++i) {
           System.arraycopy(((TFRCPacketHeader)LastAcks.get(i)).toByteArray(),
                            0, Data, i * TFRCPacketHeader.SIZE,
                            TFRCPacketHeader.SIZE);
        }

        return (Data);
    }

    //converts a byte array to an AckWindowList Object

    public static AckWindowList toObject(byte[] Data)
    {
        LinkedList LL = new LinkedList();
        byte[] data = new byte[TFRCPacketHeader.SIZE];

        for (int i = 0; i < SIZE; ++i) {
           System.arraycopy(Data, i * TFRCPacketHeader.SIZE,
                            data, 0, TFRCPacketHeader.SIZE);
           LL.add(TFRCPacketHeader.toObject(data));
        }

        return (new AckWindowList(LL));
    }
}
```

ControlStream.java

```
/*-----------------------------------------------------------------

Control stream is used for transferring control information.
It sends a AckWindowList.
It has a running thread awaiting packets from a receiving application.
There are functions to get the port addresses.
The object uses UDP for transfering packets.

  -----------------------------------------------------------------*/

import java.net.*;
import java.io.*;
import java.util.Date;

public class ControlStream extends Thread{

   // private variables for the class object

   private DatagramSocket dsock = null;  // Socket for receiving
   private TFRCSocket tsock     = null;  // Pointer to Socket
   private InetAddress address  = null;  // Address to send control
   private int receivePort;              // Port to send control


   //-------------------------------------------------------------------
   // Constructor requring a DatagramSocket for receiving control,
   // InetAddress and port number for sending, and a handle to
   // the TFRCSocket creating the ControlStream.
   //-------------------------------------------------------------------

   public ControlStream(DatagramSocket d, InetAddress iadd, int port,
                                TFRCSocket Ts) throws IOException
   {
           tsock       = Ts;
           dsock       = d;
           receivePort = port;
           address     = iadd;

           start();
   }

   //-------------------------------------------------------------------
   // Thread to run and listen for AckWindowList
   //-------------------------------------------------------------------

   public void run(){

           // Variables for listening

           AckWindowList AWL = null;
           DatagramPacket dpack = new DatagramPacket(new byte[100], 100);
```

```
        while(true) {

           // Wait for a DatagramPacket

           try { dsock.receive(dpack); }
           catch (IOException e) { e.printStackTrace(); }

           // Reconstruct data into the AckWindowList Object

           AWL = AckWindowList.toObject(dpack.getData());

           // Send AckWindowList to TFRCSocket recvControl method

           tsock.recvControl(AWL);
        }
}


public void send(AckWindowList AWL)
{
        // private variables for sending

        byte[] data                = null;
        DatagramPacket dpack       = null;

        // Create a ByteArray to be placed in a created DatagramPacket

        data  = AWL.toByteArray();
        dpack = new DatagramPacket(data, data.length,
                                   address, receivePort);

        // Send the DatagramPacket

        try { dsock.send(dpack); }
        catch (IOException e) { e.printStackTrace(); }
}

//---------------------------------------------------------------------
// Close the receiving Socket
//---------------------------------------------------------------------

public void close()
{
        dsock.close();
}

//---------------------------------------------------------------------
// Get the local control port
//---------------------------------------------------------------------

public int getLocalPort()
{
        return (dsock.getLocalPort());
}
```

```
    //---------------------------------------------------------------------
    // Get the remote control port
    //---------------------------------------------------------------------

    public int getRemotePort()
    {
            return (receivePort);
    }
}
```

DataStream.java

```
/* ------------------------------------------------------------

Data stream is used for sending data in the form of a TFRC Packet.
There is a thread running awaiting for sent data by a sending datastream.
The ports and addresses are available by calls.

  ------------------------------------------------------------*/

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

public class DataStream extends Thread
{
   private DatagramSocket dsock;
   private InetAddress clientAddr;
   private int clientPort;
   private TFRCSocket tsock;

   public DataStream(DatagramSocket ds, InetAddress addr,
                           int port, TFRCSocket ts)
   {
           dsock = ds;
           clientAddr = addr;
           clientPort = port;
           tsock = ts;

           start();
   }

   public void write(TFRCPacket packet)
   {
           byte[] data;
           DatagramPacket dpack;

           /* Convert TFRCPacket to byte array */

           data = packet.toByteArray();

           //converts the byte array into a DatagramPacket

           dpack = new DatagramPacket(data, data.length, clientAddr, clientPort);

           //sends the DatagramPacket

           try { dsock.send(dpack); }
           catch (IOException e) {
              System.err.println("DataStream: Error sending Datagram");
              e.printStackTrace();
              return;
           }
   }
```

```
public void run()
{
        DatagramPacket dpack         = null;
        TFRCPacket tpack             = null;
        byte[] data                  = null;
        int len;

        while(true)
           {
                dpack = new DatagramPacket(new byte[1024], 1024);

                //receiving a datagram packet

                try { dsock.receive(dpack); }
                catch (IOException e) {
                   System.err.println("DataStream: Error " +
                                      "receiving Datagram");
                   e.printStackTrace();
                   continue;
                }

                /* Convert byte array to TFRCPacket */

                len  = dpack.getLength();
                data = new byte[len];
                System.arraycopy(dpack.getData(), 0, data, 0, len);
                tpack = TFRCPacket.toObject(data);

                /* Send TFRCPAcket up to TFRCSocket */
                tsock.recvData(tpack);
           }
}

//---------------------------------------------------------------------
// closes data socket
//---------------------------------------------------------------------

public void close()
{
        dsock.close();
}

//---------------------------------------------------------------------
 // Return Local Port
//---------------------------------------------------------------------

public int getLocalPort()
{
        return dsock.getLocalPort();
}

//---------------------------------------------------------------------
// Return Remote Port
//---------------------------------------------------------------------
```

```
    public int getRemotePort()
    {
            return  clientPort;
    }

    //---------------------------------------------------------------------
    // Return Local Address
    //---------------------------------------------------------------------

    public InetAddress getLocalAddress()
    {
            return (dsock.getLocalAddress());
    }

    //---------------------------------------------------------------------
    // Return Remote Address
    //---------------------------------------------------------------------

    public InetAddress getRemoteAddress()
    {
            return clientAddr;
    }

}
```

TFRCCongestion.java

```
/* ---------------------------------------------------------------------

TFRCCongestion tracks bandwidth and sending rate.
It is calculated using the TFRC Formula.
A call to update is made to update the round trip time and
loss event rate necesary to calculate bandwidth and sending rate.
Update requires an ACKWindowList.
Calls can be made to getRTT (round trip time), getBandwidth, and
 getRate (sendingrate).

  ---------------------------------------------------------------------*/


import java.io.*;
import java.lang.*;
import java.util.*;

public class TFRCCongestion {

   /* The necessary values needed to get a bandwidth of 44K
      if the first packet sent is lost with a packet size of 95 and
      round trip time = 80ms */

   private int packet_size = 950;
   private float round_trip_time = (float)0.08;
    private int[] loss_intervals = new int[] {31, 31, 31, 31, 31, 31, 31, 31};
   private float loss_event_rate = (float) 0.048192771;

   //initalizing variables

    private int curr_index = 7;
   private int last_packet_lost = 0;
    private float last_packet_lost_time = 0;

   //constant variables used to calculate the loss_event_rate

   private static final float[] WEIGHTS =
           new float[] {(float)1.0, (float)1.0, (float)1.0, (float)1.0,
                        (float)0.8, (float)0.6, (float)0.4, (float)0.2};
   private static final float SQRT_2thirds = (float)0.8164965809277;

   //private variables for the class

   private AckWindowList ack;
   private int expected_seqnum;

   public TFRCCongestion (int ps, int expecting) {
           packet_size = ps;
           expected_seqnum = expecting;
   }
```

```
public synchronized void update(long rtt, LinkedList LL)
{
        UpdateRTT(rtt);

        //time estimated that a packet is lost

        float lost_time= 0;

        // for keeping track of packets
        int i;
        int now  = 0;
        int past = 0;

        /* traverses through the linked list until it either
          finds the expected packet sequence number of or a lost packet */

        for (i = 0; i < 8; ++i) {

           past = now;
           now = ((TFRCPacketHeader)LL.get(i)).seq_no;

           if ( expected_seqnum == now )
                   break;
           if (i > 0 && ( now != past - 1))
                   break;
        }

        int curAckSeq = (((TFRCPacketHeader)LL.get(0)).seq_no);

        //if no packet lost, add the packets to loss_interval

        if (expected_seqnum == now){
           loss_intervals[curr_index] += 1 + curAckSeq - expected_seqnum;
        }

        // figure out if the lost packet is within the previous loss
        // event or a new one (within the RTT)

        else {

           //calculate the approximate lost time of the last lost packet
           lost_time = (((TFRCPacketHeader)(LL.get(i))).time +
                           ((TFRCPacketHeader) (LL.get(i-1))).time) / 2;

           //if within the round trip time
           if (last_packet_lost_time + round_trip_time >= lost_time) {
                   loss_intervals[curr_index] += 1 + curAckSeq - past;
           }

 // if not within RTT
           else {
                   last_packet_lost_time = lost_time;
                   last_packet_lost = past - 1;
           }
        }
```

```
        expected_seqnum =(((TFRCPacketHeader) LL.getFirst()).seq_no) + 1;
        updateLossEvent(past - 1, curAckSeq - past + 1);
 }

 //calculates the loss event rate

 private synchronized void updateLossEvent (int lost_seqnum, int length)
 {
        float avg_interval = 0;

        for (int i = 0; i<8; i++) {
           avg_interval += loss_intervals[(curr_index+i)%8] * WEIGHTS[i];
        }

        //loops the index of the loss intervals, so it imitates a circular list

        curr_index = (curr_index - 1);
        if (curr_index < 0)
           curr_index = 7;

        loss_intervals[curr_index] = length;

        avg_interval /= 8;
        loss_event_rate = 1 / avg_interval;
 }

 //calculates the sending rate using the TFRC throughput equation

 synchronized float sendingrate() {
        float den = (float) (round_trip_time *
           Math.sqrt((double)loss_event_rate) *
           SQRT_2thirds * ( 1 + 9 * loss_event_rate + 288 *
             loss_event_rate * loss_event_rate * loss_event_rate));
        return den;
 }

 //converts a the round trip time into a long

 private synchronized void UpdateRTT(long rrt) {
        round_trip_time = (float)rrt / (float)1000;
 }

 //returns the sending rate

 public int getRate()
 {
        return((int)(1000 * sendingrate()));
}

 //returns the bandwidth

 public int getBandwidth()
 {
        int band = (int)(packet_size / sendingrate());
        return(band);
```

```
    }

    //returns the packet size

    public int getPacketSize()
    {
            return(packet_size);
    }

    //returns the round trip time

    public float getRTT()
    {
            return(round_trip_time);
    }
}
```

TFRCInputStream.java

```
/*-----------------------------------------------------------------

TFRCInputStream in an application's connection to the input
buffer of TFRCSocket.  An application can read the buffer with
a read call.

  ----------------------------------------------------------------*/


public class TFRCInputStream
{
   private TFRCSocket sock;

   public TFRCInputStream(TFRCSocket s) {
            sock = s;
   }

   //reads specified number of bytes from the TFRCSocket

   public int read(byte[] bytes, int size) {
            if (sock.read(bytes, size) == -1)
               return -1;

            else
               return 1;
   }
}
```

TFRCOutputStream.java

```
/*-------------------------------------------------------------

An applicaiton's connection to the output buffer in TFRCSocket
for sending.

 -------------------------------------------------------------*/


public class TFRCOutputStream
{
   private TFRCSocket sock;

   public TFRCOutputStream(TFRCSocket s) {
            sock = s;
   }

   //writes the bytes to the TFRCSocket

   public void write(byte[] bytes) {
            sock.write(bytes);
   }

}
```

TFRCPacket.java

```
/*-------------------------------------------------------

TFRCPacket has a byte array for data and a TFRCPacketHeader.
There is a toObject and toByteArray functions.

 -------------------------------------------------------*/

import java.io.*;

public class TFRCPacket
{
   public TFRCPacketHeader header = null;
   public byte[] data;

   public TFRCPacket(int size)
   {
          data = new byte[size];
   }

   public TFRCPacket(TFRCPacketHeader h, byte[] Data)
   {
          header = h;
          data = Data;
   }

   public TFRCPacket(int s, long t, byte[] Data)
   {
          this(new TFRCPacketHeader(s,t), Data);
   }

   public TFRCPacket(TFRCPacketHeader h, int size)
   {
          this(h, new byte[size]);
   }

   //converts a TFRCPacket into a byte array

   public byte[] toByteArray()
   {
          byte[] Data = new byte[data.length + TFRCPacketHeader.SIZE];
          System.arraycopy(header.toByteArray(), 0, Data, 0,
                                TFRCPacketHeader.SIZE);
          System.arraycopy(data, 0, Data, TFRCPacketHeader.SIZE, data.length);
          return Data;
   }

   //converts a byte array to a TFRCPacket Object

   public static TFRCPacket toObject(byte Data[])
   {
          byte[] head_data = new byte[TFRCPacketHeader.SIZE];
```

```
            byte[] pack_data = new byte[Data.length - TFRCPacketHeader.SIZE];
            System.arraycopy(Data, 0, head_data, 0, TFRCPacketHeader.SIZE);
            System.arraycopy(Data, TFRCPacketHeader.SIZE, pack_data, 0,
                                    Data.length - TFRCPacketHeader.SIZE);
            return (new TFRCPacket(TFRCPacketHeader.toObject(head_data),
                                        pack_data));
    }
}
```

TFRCPacketHeader.java

```
/*-------------------------------------------------------------

A TFRCPacketHeader has a sequence number and time stamp.
Ther is a toObject and toByteArray functions.

 -------------------------------------------------------------*/

import java.io.*;

public class TFRCPacketHeader
{
   public int seq_no;  //helps in sorting the packets in order
   public long time;  //the time packet was sent
   public static short SIZE = 12; //header size in bytes

   public TFRCPacketHeader(int s, long t) {
            seq_no = s;
            time = t;
   }

   //takes this Object and converts it into an array of bytes

   public byte[] toByteArray()
   {
            ByteArrayOutputStream Ostream = new ByteArrayOutputStream();
            try {
               //converts the timestamp and sequence number into a byte array
               DataOutputStream dataOut = new DataOutputStream(Ostream);
               dataOut.writeInt(seq_no);
               dataOut.writeLong(time);
            }
            catch (IOException e) {
               System.out.println("Packet Header toByteArray error.");
               e.printStackTrace();
            }
            return (Ostream.toByteArray());
   }

   //converts the byte array into TFRCPacketHeader Object

   public static TFRCPacketHeader toObject(byte[] data)
   {
            int seq_no_in = 0;
            long time_in = 0;

            try {
               DataInputStream inData =
                        new DataInputStream(new ByteArrayInputStream(data));
               seq_no_in = inData.readInt();
               time_in  = inData.readLong();
            }
```

```
            catch (IOException e) {
               System.out.println("Packet Header toObject error.");
               e.printStackTrace();
            }
            return (new TFRCPacketHeader(seq_no_in, time_in));
      }
   }
```

TFRCServerSocket.java

```
/*-------------------------------------------------------------

Similar to ServerSocket it allows for an application to await
a connection from a TFRCSocket.
A call to accept awaits for a connection.

 -------------------------------------------------------------*/


import java.net.ServerSocket;
import java.net.Socket;
import java.io.IOException;

public class TFRCServerSocket
{
   private ServerSocket ss = null;
   private int dataPort;
   private int ctrlPort;

   public TFRCServerSocket(int port, int dp, int cp) throws IOException
   {
           dataPort = dp;
           ctrlPort = cp;
           ss = new ServerSocket(port);
   }

   //waits for a connection

   public TFRCSocket accept() throws IOException
   {
           Socket s = ss.accept();
           return(new TFRCSocket(s, dataPort, ctrlPort));
   }
}
```

TFRCSession.java

```
/* -------------------------------------------------------------

TFRCSession track TFRC Session variables for the
next sequence number to send, ACK expecting from
a remote machine, and an ACK from the local machine.

  -------------------------------------------------------------*/

public class TFRCSession
{
   public int NextSequenceNumToSend;

   //next data sequence to receive from remote
   public int AckSequenceNumWaitRemote;

   //next acknowledgment sequence to receive
   public int AckSequenceNumWaitLocal;

   public TFRCSession()
   {
          NextSequenceNumToSend    = 1;
          AckSequenceNumWaitLocal  = 1;
          AckSequenceNumWaitRemote = 1;
   }
}
```

TFRCSocket.java

```
/*-----------------------------------------------------------------

TFRCSocket is the application's way of connecting to a
TFRCServerSocket and have a two-way connection.
It has a datastream, controlstream, TFRCCongestion, input and
output buffers, and an ACKWindowList.
TFRCSocket have functions for the application to query about
status such as bandwidth, ports, addresses, and packet size.
There is a running thread that sends data at a sending rate
determined by calling TFRCCongestion.
Data is built by using data from the output buffer.
The input buffer has the data received by a sending TFRCSocket
through the datastream, which calls recvData.
Control stream calls recvControl with an ACKWindowList to
update TFRCCongestion.

 ----------------------------------------------------------------*/

import java.net.*;
import java.io.*;
import java.util.*;

public class TFRCSocket extends Thread
{
   private static final short MAX_SIZE = 32000;

   //variables used to implement the TFRC protocol

   private ControlStream  cstream     = null;
   private DataStream     dstream     = null;
   private TFRCSession    session     = null;
   private TFRCCongestion congestion  = null;
   private AckWindowList  AckWinLis   = null;

   //variables used to receive and send

   private byte[]        inbuffer    = new byte[MAX_SIZE];
   private byte[]        outbuffer   = new byte[MAX_SIZE];
   private short         inlength    = 0;
   private short         outlength   = 0;

   public TFRCSocket(Socket s, int dataPort, int ctrlPort) throws IOException
   {
          s.close();
          cstream = new ControlStream( new DatagramSocket (ctrlPort),
                                       s.getInetAddress(),
                                       ctrlPort,
                                       this);
          dstream = new DataStream(new DatagramSocket (dataPort),
                                   s.getInetAddress(),
                                   dataPort,
```

```
                                        this);

        session = new TFRCSession();
        AckWinLis = new AckWindowList(session.NextSequenceNumToSend);
        congestion = new TFRCCongestion(950, session.NextSequenceNumToSend);

        start();
}

public TFRCSocket(String host, int port, int dataPort, int ctrlPort)
        throws UnknownHostException, IOException
{
        this(new Socket(host,port), dataPort, ctrlPort);
}

// called by ControlStream when control data arrives

public synchronized void recvControl(AckWindowList AWL)
{

        LinkedList LL = AWL.LastAcks;
        TFRCPacketHeader first = (TFRCPacketHeader)LL.get(0);
        long rtt = (new Date()).getTime() - first.time;

        // temporary printout must be erased later
    int a;

        // Checks if ACK is late, discard late ACKS

        if ( (first.seq_no >=
            session.AckSequenceNumWaitLocal)) {


           /*           for (int i = 0; i < 8; ++i){
                    a = ((TFRCPacketHeader)LL.get(i)).seq_no;
                    System.out.print(a + " ");
           }
           */
           // Update next local ACK to wa it for

           session.AckSequenceNumWaitLocal = first.seq_no + 1;

           //every time valid control stream receive, update congestion
           congestion.update(rtt, LL);

        }
        //System.out.print(">> ");
        //System.out.println(rtt + "    " + congestion.getBandwidth());
}

// Insert TFRCPacketHeader into ACKWinList and send it

public synchronized void sendControl(TFRCPacketHeader TPH)
{
        AckWinLis.insert(TPH);
```

```
        cstream.send(AckWinLis);
}

// Called by DataStream when data arrives

public void recvData(TFRCPacket TP)
{

        // Checks is packet is out of sequence

        if (TP.header.seq_no >= session.AckSequenceNumWaitRemote) {
           sendControl(TP.header);

           // Update Ack from remote waiting for

           session.AckSequenceNumWaitRemo te = TP.header.seq_no + 1;

           // Call function to write data to buffer

           writeInput(TP.data, TP.data.length);
        }
}

// Sends a byte array of data through a TFRCPacket by calling DataStream
// send

public void sendData(byte data[])
{
        // Create TFRCPacket with data in it along with current time

        TFRCPacket TP = new TFRCPacket(data.length);
        TP.data = data;
        TFRCPacketHeader TPH = null;
        TP.header = new TFRCPacketHeader(session.NextSequenceNumToSend++,
                                         (new Date().getTime()));

        // send TFRCPacket to Datastream for sending

        dstream.write(TP);
}

// Get remote address

public InetAddress getRemoteAddress()
{
        return (dstream.getRemoteAddress());
}

// Get Local Address

public InetAddress getLocalAddress()
{
        return (dstream.getLocalAddress());
}
```

```
// Get remote control port

public int getRemotePortControl()
{
        return (cstream.getRemotePort());
}

// get local control port

public int getLocalPortControl()
{
        return (cstream.getLocalPort());
}

// get remote data port

public int getRemotePortData()
{
        return (dstream.getRemotePort());
}

// get local data port

public int getLocalPortData()
{
        return (dstream.getLocalPort());
}

// gets sending rate Bandwidth

public int getBandwidth()
{
        return (congestion.getBandwidth());
}

// gets packet size set to

public int packet_size()
{
        return (congestion.getPacketSize());
}

// gets current round trip time

public float round_trip_time()
{
        return(congestion.getRTT());
}

/* TFRCOutputStream calls this to write array bytes
   to TFRCSocket output buffer to send */

public synchronized int write(byte[] buff) {

        /* the TFRCOuputStream waits till there is room
```

```
        in the output buffer before it writes */

        while (buff.length > MAX_SIZE - outlength) {
          try {
                wait(10);
          }
          catch (InterruptedException e) {
                System.err.println("TFRCSocket - write: InterruptedException");
          }
        }

        // Copies bytes to output buffer

        System.arraycopy(buff, 0, outbuffer, outlength, buff.length);
        outlength += buff.length;

        return buff.length;
}

// recvData writes the data into the input buffer

private synchronized void writeInput(byte[] buff, int len)
{

        // wait till there is room in the input buffer

        while (inlength + len > MAX_SIZE) {
          try {
                wait(10);
          }
          catch (InterruptedException e) {
          }
        }

        // write data into input buffer

        System.arraycopy(buff, 0, inbuffer, inlength, buff.length);
        inlength += buff.length;
}

/* TFRCInputStream calls this to read data from
   the TFRCSocket input buffer */

public synchronized int read(byte[] buff, int len) {

        // Waits till the buffer if at least of size len

        while (len >= inlength) {
          try {
                wait(10);
          }
          catch (InterruptedException e) {
                System.err.println("TFRCSocket - write: InterruptedException");
          }
        }
```

```
        // Read the data and put it into buff

        System.arraycopy(inbuffer, 0, buff, 0, len);
        inlength -= len;
        System.arraycopy(inbuffer, len, inbuffer, 0, inlength);

        return len;
}

//TFRCSocket sends the data

public void run() {

        // Variables for time calculations

        long last_time_sent = (new Date().getTime());
        long current_time;
        int  rate;
        long wait;
        int pack_size = 950;

        while (true){

          // Checks for a sending rate and sends at that time

          try {
                rate = congestion.getRate();
                current_time = new Date().getTime();

                // Enforces the sending rate by waiting a calculated time
                // Waiting time is equal to the rate minus time already
                // spent doing other calculations

                wait = rate - (current_time - last_time_sent);

                // Wait for positive amount of time
                // Update the last time packet was sent

                if (wait > 0) {
                   Thread.sleep(wait);
                   last_time_sent = current_time + wait;
                }
                else last_time_sent = current_time;
          }
          catch (InterruptedException e) { e.printStackTrace(); }

          // After time waited is spent, check if there is data to send

          if (outlength >0) {

                // Check if data is enough to fill pack_size
                // If so send a packet of that size or else
                // send what you have
```

```
                    if (outlength >= pack_size) {

                       // create send data of pack_size

                       byte[] senddata = new byte[pack_size];
                       System.arraycopy(outbuffer, 0, senddata, 0,
                                        pack_size);
                       outlength -= pack_size;
                       System.arraycopy(outbuffer, pack_size, outbuffer, 0,
                                        outlength);

                       // Send data to sendData which sends it to DataStream
                       sendData(senddata);
                    }
                    else {

                       // create send data of buffer size

                       byte[] senddata = new byte[outlength];
                       System.arraycopy(outbuffer, 0, senddata, 0,
                                        outlength);
                       outlength = 0;

                       // Send data to sendData which sends to DataStream

                       sendData(senddata);
                    }
                 }
              }
           }

        // get a TRRCInputStream

        public TFRCInputStream getInputStream() {
                return new TFRCInputStream(this);
        }

        // get a TFRCOutputStream

        public TFRCOutputStream getOutputStream() {
                return new TFRCOutputStream(this);
        }

}
```

Appendix B
TFRC Audio Application

AudioAppClient.java
AudioAppServer.java
AudioGen.java

AudioAppClient.java

```
import java.io.*;
import java.util.*;

public class AudioAppClient {

   public static void main(String[] args) {

         String filename;
         PrintWriter writer = null;
         TFRCSocket sock = null;
         int BUFSIZE = 950;
         byte[] buffer = new byte[BUFSIZE];
         TFRCInputStream inStream;
         short freq = 0;
         short seqNum = 0;
         short lastSeqNum = 0;
         ByteArrayInputStream byteStr;
         DataInputStream dataStr;

         if (args.length < 1) {
            System.out.println("Usage: java AudioAppClient filename");
            System.exit(0);
         }

         filename = args[0];

         try {
            writer = new PrintWriter
                  (new BufferedWriter
                     (new FileWriter(filename)));
         }
         catch (java.io.IOException e) {
            System.err.println("AudioAppClient: opening file");
            e.printStackTrace();
            System.exit(0);
         }

         try {
            sock = new TFRCSocket("192.168.2.2", 2849, 1550, 5500);
         }

         catch (java.io.IOException e) {
            System.err.println("AudioAppClient: accepting connection");
            e.printStackTrace();
            System.exit(0);
         }

         inStream = sock.getInputStream();

         try {
            int skipbytes = 0;
```

```
long start = new Date().getTime();
long et = 0;
int loss = 0;

while(inStream.read(buffer, 950) != -1) {
        byteStr = new ByteArrayInputStream(buffer);
        dataStr = new DataInputStream(byteStr);

        freq = dataStr.readShort();
        seqNum = dataStr.readShort();

        /* Check to see if a packet was lost */
        if (seqNum - lastSeqNum > 1) {
           loss += (seqNum - lastSeqNum - 1);
           System.out.println("Loss: " + loss);
           byte tbuf[] = new byte[BUFSIZE];
           for (int i = 0; i < BUFSIZE; i++) {
                   tbuf[i] = 0;
           }
           for(int i = 1; i < (seqNum - lastSeqNum); i++) {
                   writer.print(new String(tbuf));
           }
        }

        if (freq >= 8820)
           skipbytes = 0;

        else if (freq >= 4410)
           skipbytes = 1;
        else if (freq >= 2740)
           skipbytes = 2;
        else if (freq >= 2205)
           skipbytes = 3;
        else if (freq >= 1764)
           skipbytes = 4;
        else if (freq >= 882)
           skipbytes = 9;
        else {
           System.out.println("Freq below 4410 bytes/sec/10");
           continue;
        }
        // System.out.println("Skipbytes = " + skipbytes);

        byte[] tempbuf = new byte[(BUFSIZE - 4) * (skipbytes + 1)];

        int k = 0;
        for (int i = 4; i < BUFSIZE - 4; i++) {
           tempbuf[k] = buffer[i];
           k++;
           for (int j = 0; j < skipbytes; j++) {
                   tempbuf[k] = 0;
                   k++;
           }
        }
        writer.print(new String(tempbuf));
```

```
                    lastSeqNum = seqNum;
                    et = (new Date().getTime() - start);
                    System.out.println("Elapsed Time: " + et + " ms");
                }
            }
            catch(Exception e) {
                System.err.println("AudioAppClient: Exception reading file");
                e.printStackTrace();
            }
        }
    }
```

AudioAppServer.java

```
import java.io.*;

public class AudioAppServer {

   public static void main(String[] args) {
          String filename;
          FileInputStream fileIn = null;
          TFRCServerSocket serversock = null;
          TFRCSocket sock = null;
          TFRCOutputStream outStream;
          final int BUFSIZE = 950;
          final int FILESIZE = 2646000;
          byte[] buffer = new byte[FILESIZE];
          short seqNum = 1;
          short freq = 0;
          ByteArrayOutputStream byteStr;
          DataOutputStream dataStr;
          int index = 0;

          if (args.length < 1) {
             System.out.println("Usage: java AudioAppServer filename");
             System.exit(0);
          }

          filename = args[0];

          try {
             fileIn = new FileInputStream(filename);
          }
          catch (java.io.IOException e) {
             System.err.println("AudioAppServer: reading file");
             e.printStackTrace();
             System.exit(0);
          }

          try {
             fileIn.read(buffer, 0, FILESIZE);
          }
          catch (IOException e) {
             System.err.println("Error reading input file");
             System.exit(1);
          }

          System.out.println("Ready to accept clients");

          try {
             serversock = new TFRCServerSocket(2849, 1550, 5500);
          }
          catch (java.io.IOException e) {
             System.err.println("AudioAppServer: creating socket");
             e.printStackTrace();
```

```
    System.exit(0);
}

try {
    sock = serversock.accept();
}
catch (java.io.IOException e) {
    System.err.println("AudioAppServer: accepting connection");
    e.printStackTrace();
    System.exit(0);
}

outStream = sock.getOutputStream();

try {
    int skipbytes = 0;
    int f;

    while (index < FILESIZE) {
            byte[] tempbuf = new byte[BUFSIZE];

            byteStr = new ByteArrayOutputStream(BUFSIZE);
            dataStr = new DataOutputStream(byteStr);

            f = sock.getBandwidth() / 10;
            if (f > 8820)
                freq = 8820;
            else
                freq = (short)f;

            if (freq >= 8820)
                skipbytes = 0;

            else if (freq >= 4410)
                skipbytes = 1;
            else if (freq >= 2740)
                skipbytes = 2;
            else if (freq >= 2205)
                skipbytes = 3;
            else if (freq >= 1764)
                skipbytes = 4;
            else if (freq >= 882)
                skipbytes = 9;
            else
                System.out.println("Freq below 441 bytes/sec/10");

            dataStr.writeShort(freq);
            dataStr.writeShort(seqNum);

            System.out.println("Freq: " + freq + "  Seq: " + seqNum +
                                "  Skipbytes: " + skipbytes);

            System.arraycopy(byteStr.toByteArray(), 0, tempbuf, 0, 4);

            for (int i = 4; i < BUFSIZE && index < FILESIZE; i++) {
```

```
                    tempbuf[i] = buffer[index++];
                    index += skipbytes;
                }
                outStream.write(tempbuf);
                seqNum++;

            }

        }
        catch(java.io.IOException e) {
            System.err.println("AudioAppServer: IOException reading file");
        }
    }
}
```

AudioGen.java

```
import java.io.*;

public class AudioGen {
   public static void main(String[] args) {
            final float PI = 3.1415926f;
            final int BUFSIZE = 4096;
            final int SPEED = 44100;
            FileOutputStream out = null;
            float omega;
            float freq = 440;
            float scale = 1.001f;
            byte[] buffer = new byte[BUFSIZE];
            int time;
            long n = 0;
            int i = 0;
            short samp;

            /* Check for right number of arguments */
            if (args.length < 2) {
               System.out.println("Usage: java AudioGen filename time");
               System.exit(0);
            }
            try {
               out = new FileOutputStream(args[0]);
            }
            catch (FileNotFoundException e) {
               e.printStackTrace();
            }

            time = Integer.parseInt(args[1]);

            while (n < time * SPEED) {
               omega = 2 * PI * freq / SPEED;
               samp = (short)(0x8000 * Math.sin(omega * n));
               buffer[i++] = (byte)(samp & 0xff);
               buffer[i++] = (byte)((samp >> 8) & 0xff);

               if (i == BUFSIZE) {
                        try {
                           out.write(buffer, 0, i);
                        }
                        catch (IOException e) {
                           System.err.println("AudioGen: Error writing file");
                           e.printStackTrace();
                        }

                        i = 0;
               }

               n++;
            }
```

```
            /* Write out any remaining bytes in buffer */
            if (i > 0) {
               try {
                        out.write(buffer, 0, i);
               }
               catch (IOException e) {
                        System.err.println("AudioGen: Error writing file");
                        e.printStackTrace();
               }
            }
      }
}
```

Appendix C
Test Run Data

Test 1 Data
Test 2 Data
Test 3 Data

Test 1 Data

Test Data for 0% packet loss for all three protocols.

Time	TCP	TFRC	UDP
1	91200	87542	358150
2	274828	151996	640300
3	214304	130832	659300
4	228784	92352	657400
5	220096	130832	659300
6	225888	138528	638400
7	228784	73112	659300
8	205616	130832	659300
9	222992	103896	657400
10	222992	94276	659300
11	222992	103896	638400
12	217200	90428	659300
13	208512	88504	649800
14	220096	111592	659300
15	225888	80808	657400
16	220096	109668	638400
17	222992	78884	659300
18	199824	103896	
19	228784	90428	
20	214304	101972	
21	228784	100048	
22	217200	82732	
23	205616	100048	
24	220096		
25	128332		
Average:	212648	103352.3	635885.3
Std. Dev.	34214.1	20731.91	72086.58

Test 2 Data

Test Data for 5% packet loss for all three protocols.

Time	TCP	TFRC	UDP
1	122466	32708	99750
2	121632	39442	288800
3	118736	52910	288800
4	117288	63492	291650
5	115840	20202	284050
6	111496	74074	287850
7	120184	41366	283100
8	115840	48100	290700
9	118736	45214	291650
10	107152	51948	287850
11	123080	36556	283100
12	110048	54834	290700
13	118736	48100	291650
14	123080	54834	288800
15	118736	39442	291650
16	114392	60606	284050
17	108600	37518	283100
18	86880	59644	
19	117288	39442	
20	118736	54834	
21	120184	46176	
22	101360		
23	123080		
24	117288		
25			
Average:	115452.4	47687.71	276897.1
Std. Dev.	8203.469	12071.36	45768.06

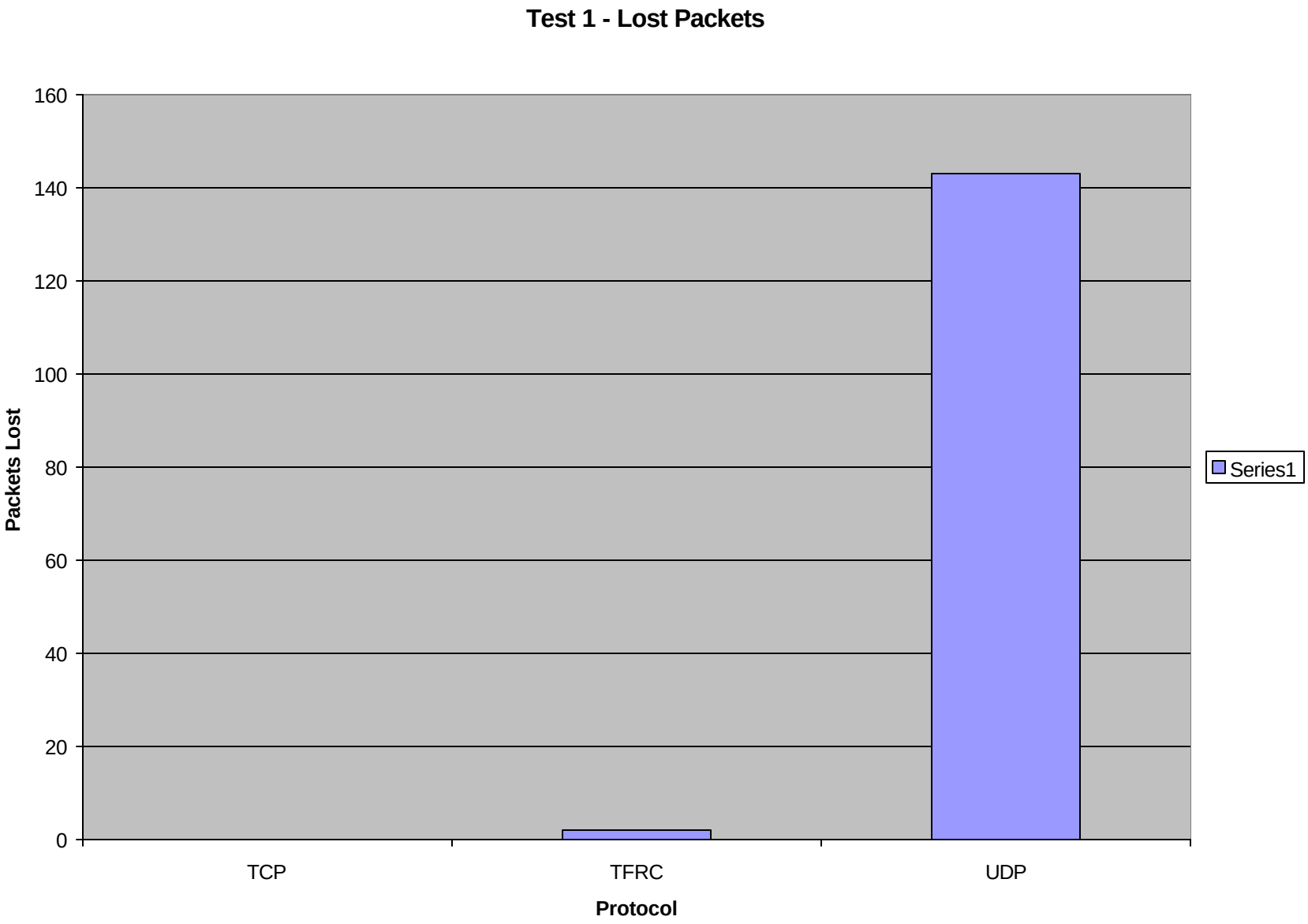
Test 3 Data

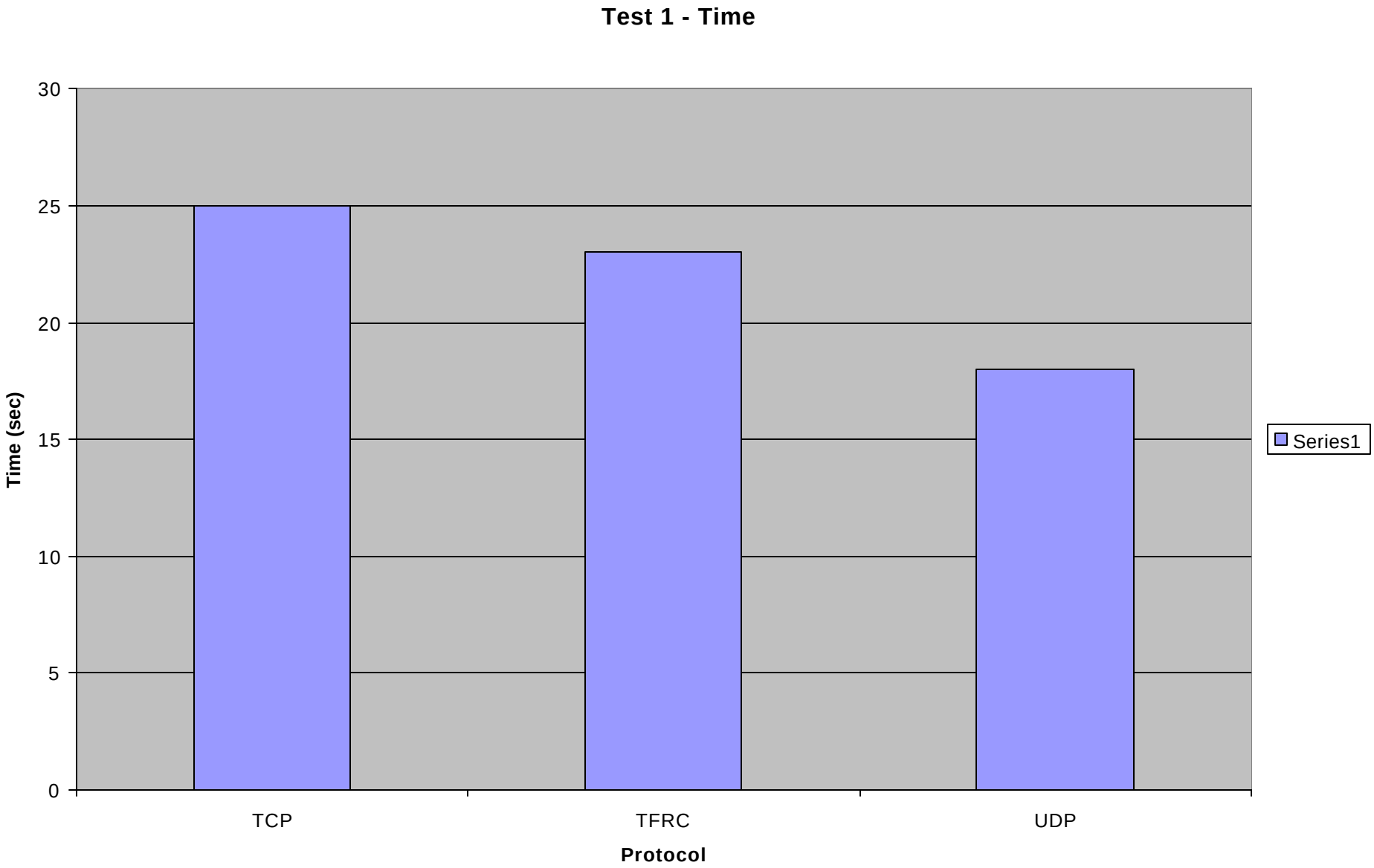
Test Data for 20% packet loss for all three protocols.

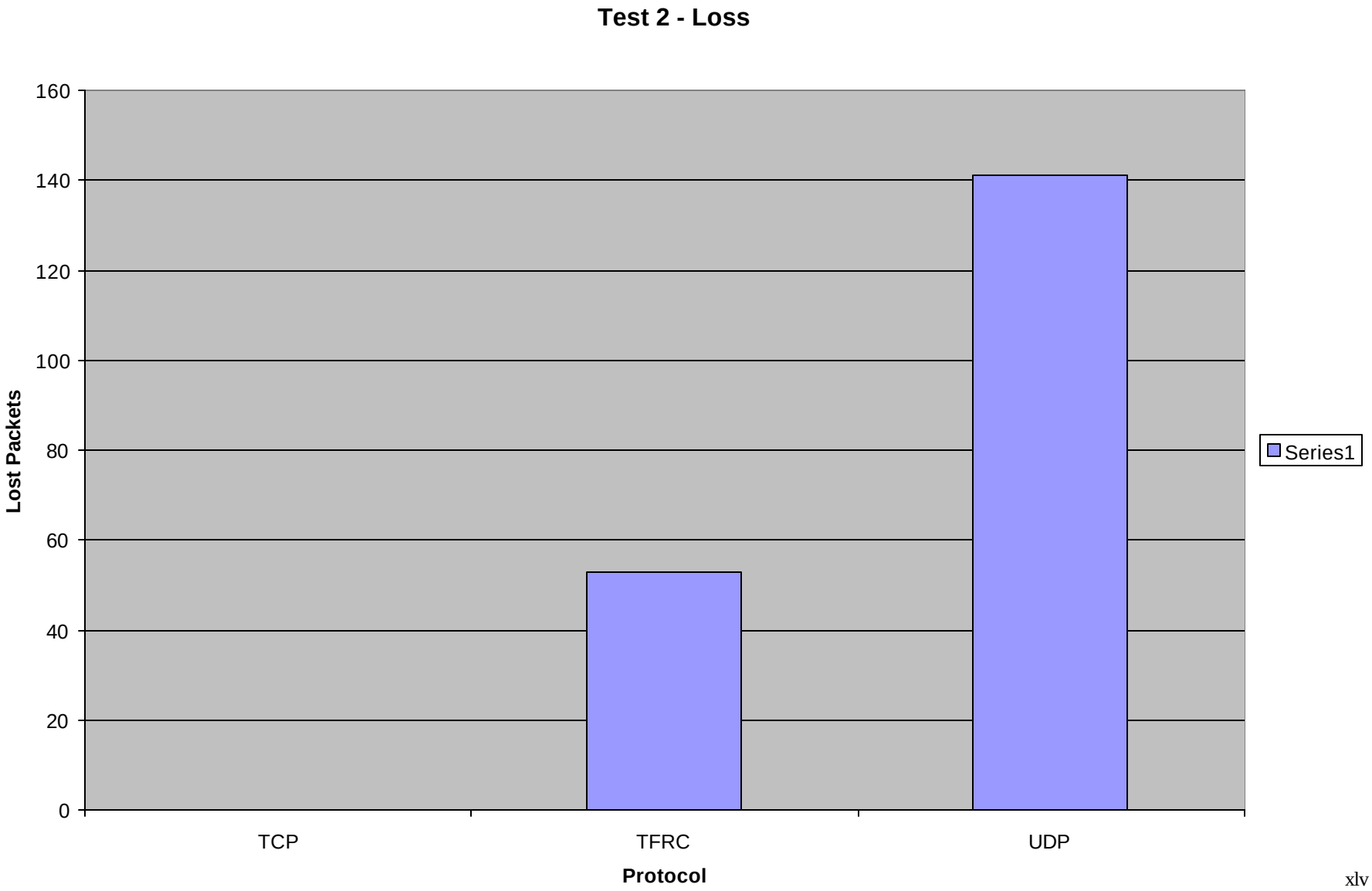
Time	TCP	TFRC	UDP
1	104470	0	73150
2	143352	42328	278350
3	75296	61568	285000
4	169416	62530	287850
5	130320	66378	285950
6	98464	71188	287850
7	95568	71188	280250
8	176656	73112	286900
9	127424	75998	285950
10	41992	51948	278350
11	41992	81770	287850
12	94120	80808	280250
13	52128	57720	286900
14	73848	67340	285950
15	118736	49062	278350
16	112944	50986	287850
17	141904	30784	285000
18	143352	67340	
19	108600	34632	
20	152040	32708	
21	136112	75036	
22	114392	40404	
23	126260		
24	78192		
25	21720		
26	56472		
27	4344		
28	85432		
29	72400		
30	162176		
31	146248		
Average:	99402.08	58802.25	283963.6
Std. Dev.	43983.53	20143.31	51337.63

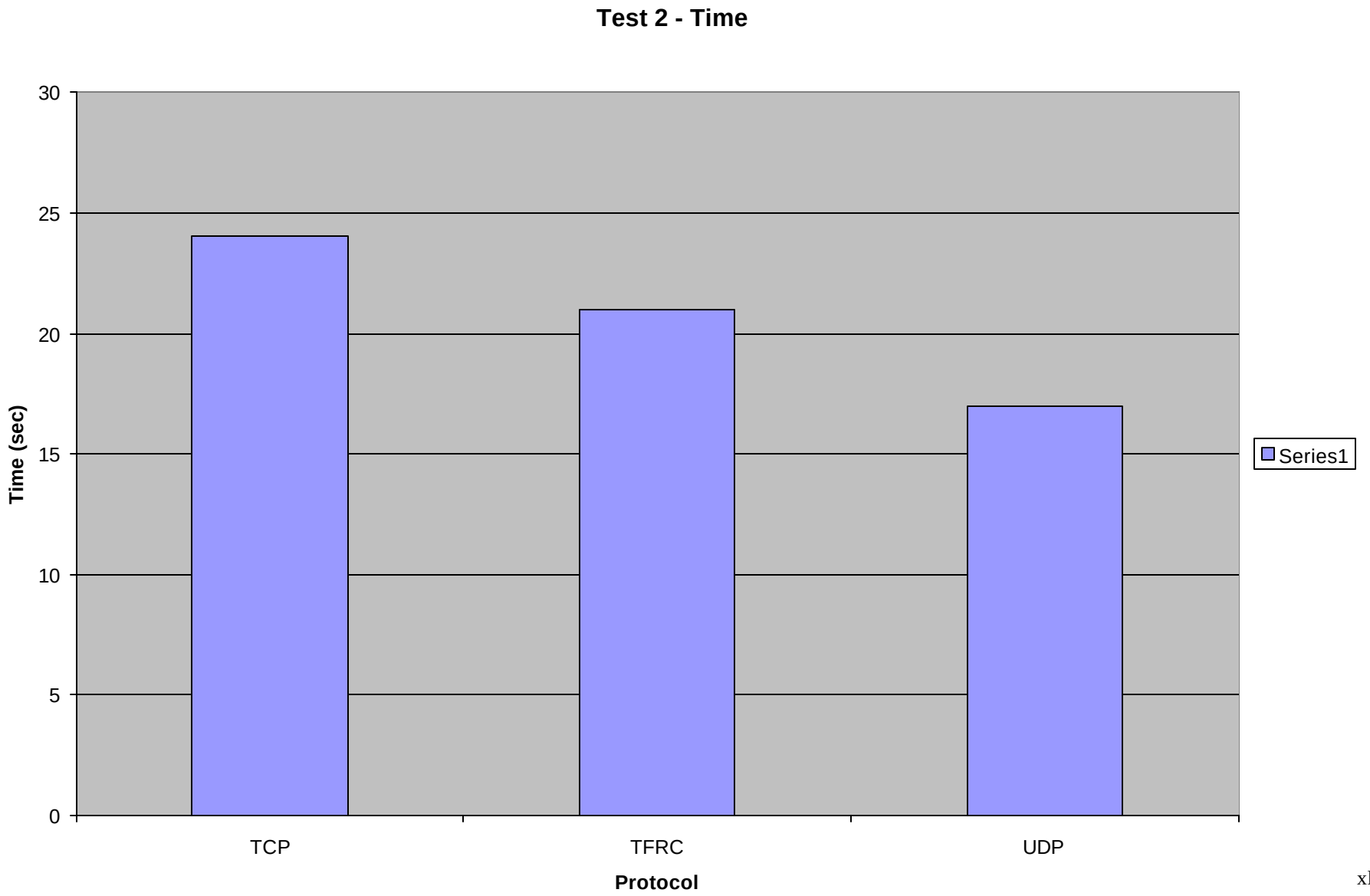
Appendix D
Test Run Data

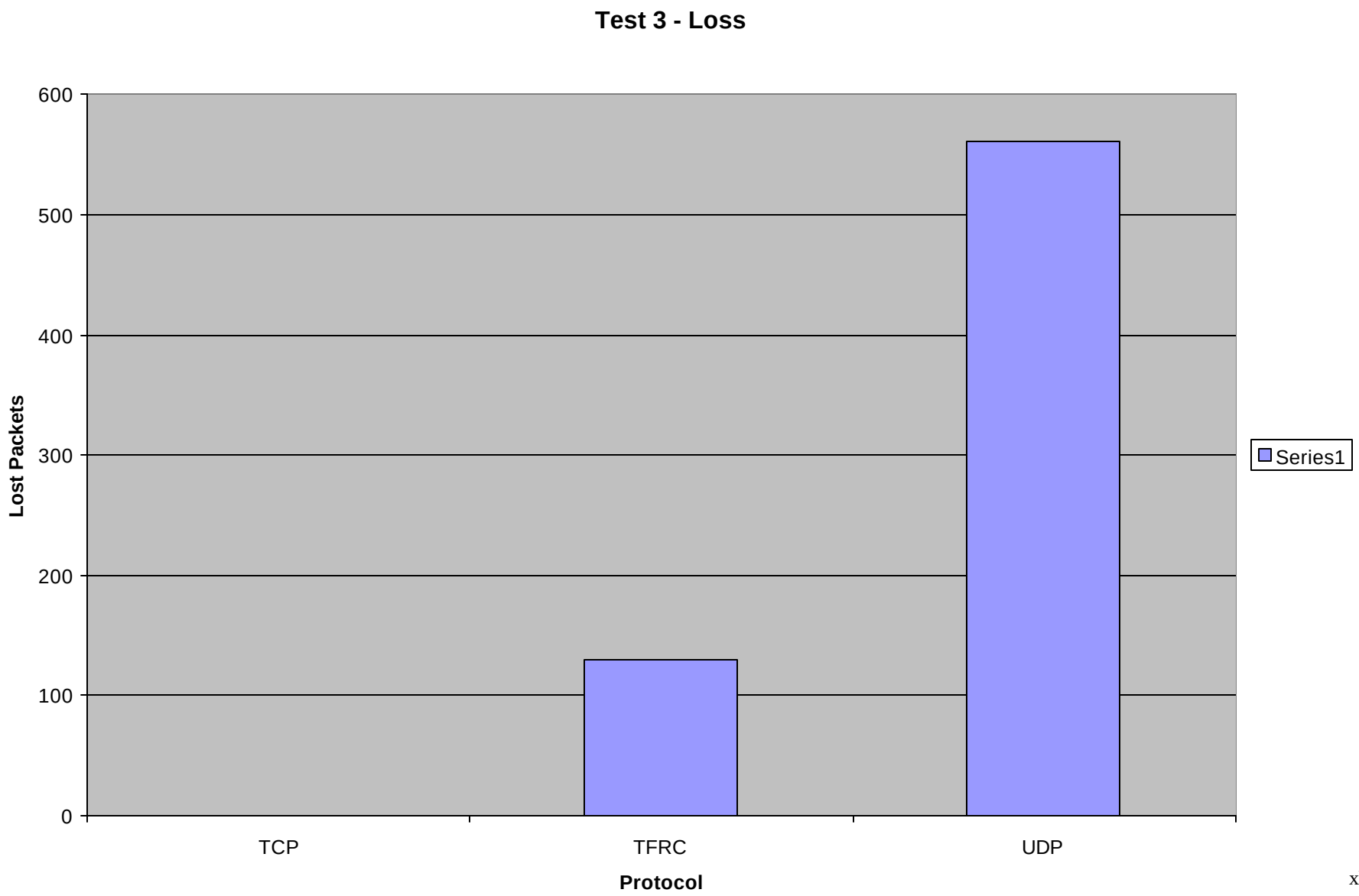
Test 1 Lost Packets
Test 1 Time
Test 2 Loss
Test 2 Time
Test 3 Loss
Test 3 Time

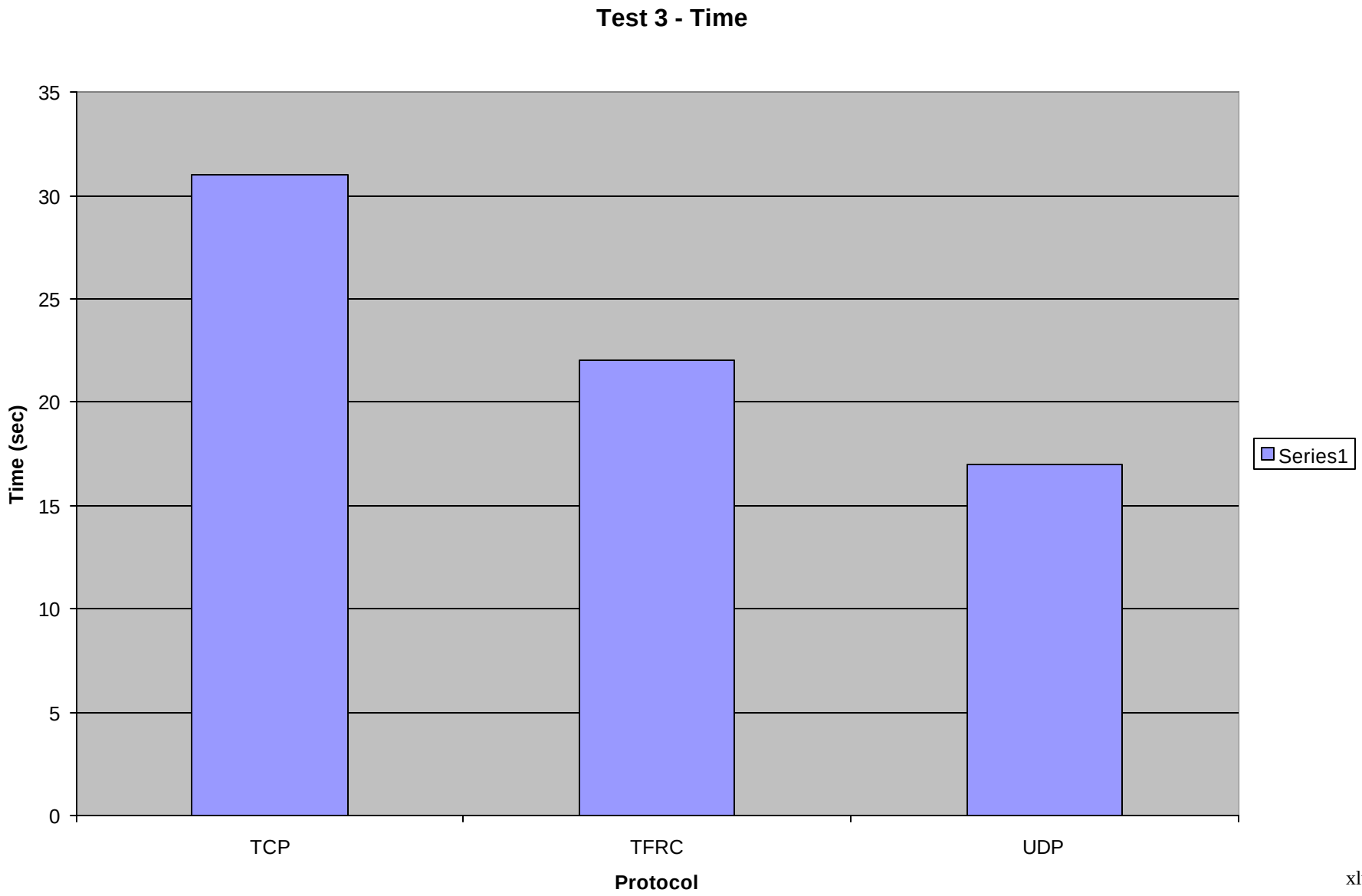

Test 1 - Lost Packets
160
140
120
100
80
60
40
20
0
Packets Lost
TCP
TFRC
UDP
Protocol
Series1


Test 1 - Time
30
25
20
15
10
5
0
Time (sec)
TCP
TFRC
UDP
Protocol
Series1


Test 2 - Loss
160
140
120
100
80
60
40
20
0
Lost Packets
TCP
TFRC
UDP
Protocol
Series1


Test 2 - Time
30
25
20
15
10
5
0
Time (sec)
TCP
TFRC
UDP
Protocol
Series1


Test 3 - Loss
600
500
400
300
200
100
0
Lost Packets
TCP
TFRC
UDP
Protocol
Series1


Test 3 - Time
Time (sec)
35
30
25
20
15
10
5
0
TCP
TFRC
UDP
Protocol
Series1