year/sem:III/VI department:CSE

```
System.out.println("Press Ctrl+C to quit");
while(true)
{
cs = ss.accept();
ps = new PrintStream(cs.getOutputStream());
Date d = new Date();
ps.println(d);
dis = new BufferedReader(new
InputStreamReader(cs.getInputStream()));
inet = dis.readLine();
System.out.println("Client System/IP address is :"+ inet);
ps.close();
dis.close();
}
}
catch(IOException e)
{
System.out.println("The exception is :" + e);
}
}
}
```

// TCP Date Client--tcpdateclient.java

```
import java.net.*;
import java.io.*;
class tcpdateclient
{
public static void main (String args[])
{
Socket soc;
BufferedReader dis;
String sdate;
PrintStream ps;
try
{
InetAddress ia = InetAddress.getLocalHost();
if (args.length == 0)
soc = new Socket(InetAddress.getLocalHost(),4444);
else
soc = new Socket(InetAddress.getByName(args[0]),
4444);
dis = new BufferedReader(new
InputStreamReader(soc.getInputStream()));
sdate=dis.readLine();
System.out.println("The date/time on server is : " +sdate);
ps = new PrintStream(soc.getOutputStream());
ps.println(ia);
ps.close();
}
catch(IOException e)
{
System.out.println("THE EXCEPTION is :" + e);
}
}
}
```

OUTPUT

Server:
```
$ javac tcpdateserver.java
$ java tcpdateserver
Press Ctrl+C to quit
Client System/IP address is : localhost.localdomain/127.0.0.1
Client System/IP address is : localhost.localdomain/127.0.0.1
```
Client:
```
$ javac tcpdateclient.java
$ java tcpdateclient
The date/time on server is: Wed Jul 06 07:12:03 GMT 2011
```

RESULT

Thus every time a client connects to the server, server’s date/time will be returned to the client for synchronization.

EX NO:1.b **CLIENT-SERVER APPLICATION FOR CHAT**

AIM:

To implement a chat server and client in java using TCP sockets.

ALGORITHM:

Server

1. Create a server socket and bind it to port.
2. Listen for new connection and when a connection arrives, accept it.
3. Read Client's message and display it
4. Get a message from user and send it to client
5. Repeat steps 3-4 until the client sends "end"
6. Close all streams
7. Close the server and client socket
8. Stop

Client

1. Create a client socket and connect it to the server’s port number
2. Get a message from user and send it to server
3. Read server's response and display it
4. Repeat steps 2-3 until chat is terminated with "end" message
5. Close all input/output streams
6. Close the client socket
7. Stop

PROGRAM:

// TCP Chat Server--tcpchatserver.java

```
import java.io.*;
import java.net.*;
class tcpchatserver
{
public static void main(String args[])throws Exception
{
PrintWriter toClient;
BufferedReader fromUser, fromClient;
try
{
ServerSocket Srv = new ServerSocket(5555);
System.out.print("\nServer started\n");
Socket Clt = Srv.accept();
System.out.println("Client connected");
toClient = new PrintWriter(new BufferedWriter(new
OutputStreamWriter(Clt.getOutputStream())), true);
fromClient = new BufferedReader(new
InputStreamReader(Clt.getInputStream()));
fromUser = new BufferedReader(new
InputStreamReader(System.in));
String CltMsg, SrvMsg;
while(true)
{
CltMsg= fromClient.readLine();
if(CltMsg.equals("end"))
break;
else
{
System.out.println("\nServer <<< " +CltMsg);
System.out.print("Message to Client : ");
SrvMsg = fromUser.readLine();
toClient.println(SrvMsg);
}
}
System.out.println("\nClient Disconnected");
fromClient.close();
toClient.close();
fromUser.close();
Clt.close();
Srv.close();
}
catch (Exception E)
{
System.out.println(E.getMessage());
}
}
}
```

// TCP Chat Client--tcpchatclient.java

```
import java.io.*;
import java.net.*;
class tcpchatclient
{
public static void main(String args[])throws Exception
{
Socket Clt;
PrintWriter toServer;
BufferedReader fromUser, fromServer;
try
{
if (args.length > 1)
{
System.out.println("Usage: java hostipaddr");
System.exit(-1);
}
if (args.length == 0)
Clt = new Socket(InetAddress.getLocalHost(),5555);
else
Clt = new Socket(InetAddress.getByName(args[0]),5555);
toServer = new PrintWriter(new BufferedWriter(new
OutputStreamWriter(Clt.getOutputStream())), true);
fromServer = new BufferedReader(new
InputStreamReader(Clt.getInputStream()));
fromUser = new BufferedReader(new
InputStreamReader(System.in));
String CltMsg, SrvMsg;
System.out.println("Type \"end\" to Quit");
while (true)
{
System.out.print("\nMessage to Server : ");
CltMsg = fromUser.readLine();
toServer.println(CltMsg);
if (CltMsg.equals("end"))
break;
SrvMsg = fromServer.readLine();
System.out.println("Client <<< " + SrvMsg);
}
}
catch(Exception E)
{
System.out.println(E.getMessage());
}
}
}
```

OUTPUT

Server:

```
$ javac tcpchatserver.java
$ java tcpchatserver
Server started
Client connected
Server <<< hi
Message to Client : hello
Server <<< how r u?
Message to Client : fine
Server <<< me too
Message to Client : bye
Client Disconnected
```

Client:

```
$ javac tcpchatclient.java
$ java tcpchatclient
Type "end" to Quit
Message to Server : hi
Client <<< hello
Message to Server : how r u?
Client <<< fine
Message to Server : me too
Client <<< bye
Message to Server : end
```

RESULT

Thus both the client and server exchange data using TCP socket programming.

EX NO: 1.c IMPLEMENTATION OF TCP/IP ECHO

AIM:

To implement echo server and client in java using TCP sockets.

ALGORITHM:

Server

1. Create a server socket and bind it to port.
2. Listen for new connection and when a connection arrives, accept it.
3. Read the data from client.
4. Echo the data back to the client.
5. Repeat steps 4-5 until ‘bye’ or ‘null’ is read.
6. Close all streams.
7. Close the server socket.
8. Stop.

Client

1. Create a client socket and connect it to the server’s port number.
2. Get input from user.
3. If equal to bye or null, then go to step 7.
4. Send user data to the server.
5. Display the data echoed by the server.
6. Repeat steps 2-4.
7. Close the input and output streams.
8. Close the client socket.
9. Stop.

PROGRAM:

// TCP Echo Server--tcpechoserver.java

```
import java.net.*;
import java.io.*;
public class tcpechoserver
{
public static void main(String[] arg) throws IOException
{
```

```
ServerSocket sock = null;
BufferedReader fromClient = null;
OutputStreamWriter toClient = null;
Socket client = null;
try
{
sock = new ServerSocket(4000);
System.out.println("Server Ready");
client = sock.accept();
System.out.println("Client Connected");
fromClient = new BufferedReader(new
InputStreamReader(client.getInputStream()));
toClient = new
OutputStreamWriter(client.getOutputStream());
String line;
while (true)
{
line = fromClient.readLine();
if ( (line == null) || line.equals("bye"))
break;
System.out.println ("Client [ " + line + " ]");
toClient.write("Server [ "+ line +" ]\n");
toClient.flush();
}
fromClient.close();
toClient.close();
client.close();
sock.close();
System.out.println("Client Disconnected");
}
catch (IOException ioe)
{
System.err.println(ioe);
}
}
}
```

//TCP Echo Client--tcpechoclient.java

```
import java.net.*;
import java.io.*;
public class tcpechoclient
{
public static void main(String[] args) throws IOException
{
BufferedReader fromServer = null, fromUser = null;
PrintWriter toServer = null;
Socket sock = null;
try
```

```
{
if (args.length == 0)
sock = new Socket(InetAddress.getLocalHost(),4000);
else
sock = new Socket(InetAddress.getByName(args[0]),4000);
fromServer = new BufferedReader(new
InputStreamReader(sock.getInputStream()));
fromUser = new BufferedReader(new
InputStreamReader(System.in));
toServer = new PrintWriter(sock.getOutputStream(),
true
String Usrmsg, Srvmsg;
System.out.println("Type \"bye\" to quit");
while (true)
{
System.out.print("Enter msg to server : ");
Usrmsg = fromUser.readLine();
if (Usrmsg==null || Usrmsg.equals("bye"))
{
toServer.println("bye");
break;
}
else
toServer.println(Usrmsg);
Srvmsg = fromServer.readLine();
System.out.println(Srvmsg);
}
fromUser.close();
fromServer.close();
toServer.close();
sock.close();
}
catch (IOException ioe)
{
System.err.println(ioe);
}
```

Output

Server:

```
$ javac tcpechoserver.java
$ java tcpechoserver
Server Ready
Client Connected
Client [ hello ]
Client [ how are you ]
Client [ i am fine ]
Client [ ok ]
```

```
        Client Disconnected
Client:
        $ javac tcpechoclient.java
        $ java tcpechoclient
        Type "bye" to quit
        Enter msg to server : hello
        Server [ hello ]
        Enter msg to server : how are you
        Server [ how are you ]
        Enter msg to server : i am fine
        Server [ i am fine ]
        Enter msg to server : ok
        Server [ ok ]
        Enter msg to server : bye
```

Result

Thus data from client to server is echoed back to the client to check reliability/noise level of the channel.

EX NO: 2.a PROGRAM USING UDP SOCKET

UDP CHAT SERVER/CLIENT

AIM:

To implement a chat server and client in java using UDP sockets.

ALGORITHM:

Server

1. Create two ports, server port and client port.
2. Create a datagram socket and bind it to client port.
3. Create a datagram packet to receive client message.
4. Wait for client's data and accept it.
5. Read Client's message.
6. Get data from user.
7. Create a datagram packet and send message through server port.
8. Repeat steps 3-7 until the client has something to send.
9. Close the server socket.
10. Stop.

Client

1. Create two ports, server port and client port.
2. Create a datagram socket and bind it to server port.
3. Get data from user.
4. Create a datagram packet and send data with server ip address and client port.
5. Create a datagram packet to receive server message.
6. Read server's response and display it.
7. Repeat steps 3-6 until there is some text to send.
8. Close the client socket.

9. Stop.

PROGRAM

// UDP Chat Server--udpchatserver.java

```
import java.io.*;
import java.net.*;
class udpchatserver
{
public static int clientport = 8040,serverport = 8050;
public static void main(String args[]) throws Exception
{
DatagramSocket SrvSoc = new DatagramSocket(clientport);
byte[] SData = new byte[1024];
BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
System.out.println("Server Ready");
while (true)
{
byte[] RData = new byte[1024];
DatagramPacket RPack = new DatagramPacket(RData,
RData.length);
SrvSoc.receive(RPack);
String Text = new String(RPack.getData());
if (Text.trim().length() == 0)
break;
System.out.println("\nFrom Client <<< " + Text );
System.out.print("Msg to Cleint : " );
String srvmsg = br.readLine();
InetAddress IPAddr = RPack.getAddress();
SData = srvmsg.getBytes();
DatagramPacket SPack = new DatagramPacket(SData,
SData.length, IPAddr, serverport);
SrvSoc.send(SPack);
}
System.out.println("\nClient Quits\n");
SrvSoc.close();
}
}
```

// UDP Chat Client--udpchatclient.java

```
import java.io.*;
import java.net.*;
class udpchatclient
{
public static int clientport = 8040,serverport = 8050;
public static void main(String args[]) throws Exception
{
BufferedReader br = new BufferedReader(new
InputStreamReader (System.in));
DatagramSocket CliSoc = new DatagramSocket(serverport);
InetAddress IPAddr;
String Text;
if (args.length == 0)
IPAddr = InetAddress.getLocalHost();
else
IPAddr = InetAddress.getByName(args[0]);
byte[] SData = new byte[1024];
System.out.println("Press Enter without text to quit");
while (true)
{
System.out.print("\nEnter text for server : ");
Text = br.readLine();
SData = Text.getBytes();
DatagramPacket SPack = new DatagramPacket(SData,
SData.length, IPAddr, clientport );
CliSoc.send(SPack);
if (Text.trim().length() == 0)
break;
byte[] RData = new byte[1024];
DatagramPacket RPack = new DatagramPacket(RData,
RData.length);
CliSoc.receive(RPack);
String Echo = new String(RPack.getData()) ;
Echo = Echo.trim();
System.out.println("From Server <<< " + Echo);
}
CliSoc.close();
}
}
```

OUTPUT

Server

```
$ javac udpchatserver.java
$ java udpchatserver
Server Ready
From Client <<< are u the SERVER
Msg to Cleint : yes
From Client <<< what do u have to serve
Msg to Cleint : no eatables
Client Quits
```

Client

```
$ javac udpchatclient.java
$ java udpchatclient
Press Enter without text to quit
Enter text for server : are u the SERVER
From Server <<< yes
Enter text for server : what do u have to serve
From Server <<< no eatables
Enter text for server :
```

RESULT

Thus both the client and server exchange data using UDP sockets.

EX NO: 2.b **UDP DNS SERVER/CLIENT**

AIM:

To implement a DNS server and client in java using UDP sockets**.**

ALGORITHM:

Server

1. Create an array of hosts and its ip address in another array
2. Create a datagram socket and bind it to a port
3. Create a datagram packet to receive client request
4. Read the domain name from client to be resolved
5. Lookup the host array for the domain name
6. If found then retrieve corresponding address
7. Create a datagram packet and send ip address to client
8. Repeat steps 3-7 to resolve further requests from clients
9. Close the server socket
10. Stop

Client

1. Create a datagram socket
2. Get domain name from user
3. Create a datagram packet and send domain name to the server
4. Create a datagram packet to receive server message

5. Read server's response
6. If ip address then display it else display "Domain does not exist"
7. Close the client socket
8. Stop

PROGRAM

// UDP DNS Server -- udpdnsserver.java

```
import java.io.*;
import java.net.*;
public class udpdnsserver
{
private static int indexOf(String[] array, String str)
{
str = str.trim();
for (int i=0; i < array.length; i++)
{
if (array[i].equals(str))
return i;
}
return -1;
}
public static void main(String arg[])throws IOException
{
String[] hosts = {"yahoo.com", "gmail.com",
"cricinfo.com", "facebook.com"};
String[] ip = {"68.180.206.184", "209.85.148.19",
"80.168.92.140", "69.63.189.16"};
System.out.println("Press Ctrl + C to Quit");
while (true)
{
DatagramSocket serversocket=new DatagramSocket(1362);
byte[] senddata = new byte[1021];
byte[] receivedata = new byte[1021];
DatagramPacket recvpack = new
DatagramPacket(receivedata, receivedata.length);
serversocket.receive(recvpack);
String sen = new String(recvpack.getData());
InetAddress ipaddress = recvpack.getAddress();
int port = recvpack.getPort();
String capsent;
System.out.println("Request for host " + sen);
if(indexOf (hosts, sen) != -1)
capsent = ip[indexOf (hosts, sen)];
else
capsent = "Host Not Found";
senddata = capsent.getBytes();
DatagramPacket pack = new DatagramPacket(senddata,
senddata.length,ipaddress,port);
serversocket.send(pack);
serversocket.close();
}
}
}
```

//UDP DNS Client -- udpdnsclient.java

```
import java.io.*;
import java.net.*;
public class udpdnsclient
{
public static void main(String args[])throws IOException
{
BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
DatagramSocket clientsocket = new DatagramSocket();
InetAddress ipaddress;
if (args.length == 0)
ipaddress = InetAddress.getLocalHost();
else
ipaddress = InetAddress.getByName(args[0]);
byte[] senddata = new byte[1024];
byte[] receivedata = new byte[1024];
int portaddr = 1362;
System.out.print("Enter the hostname : ");
String sentence = br.readLine();
Senddata = sentence.getBytes();
DatagramPacket pack = new DatagramPacket(senddata,
senddata.length, ipaddress,portaddr);
clientsocket.send(pack);
DatagramPacket recvpack =new DatagramPacket(receivedata,
receivedata.length);
clientsocket.receive(recvpack);
String modified = new String(recvpack.getData());
System.out.println("IP Address: " + modified);
clientsocket.close();
}
}
```

OUTPUT

Server

```
$ javac udpdnsserver.java
$ java udpdnsserver
Press Ctrl + C to Quit
Request for host yahoo.com
Request for host cricinfo.com
Request for host youtube.com
```

Client

```
$ javac udpdnsclient.java
$ java udpdnsclient
Enter the hostname : yahoo.com
IP Address: 68.180.206.184
$ java udpdnsclient
Enter the hostname : cricinfo.com
IP Address: 80.168.92.140
$ java udpdnsclient
Enter the hostname : youtube.com
IP Address: Host Not Found
```

RESULT

Thus domain name requests by the client are resolved into their respective logical address using lookup method.

EX NO 3: PROGRAMS USING RAW SOCKETS (LIKE PACKET CAPTURING AND FILTERING)

AIM:

To implement programs using raw sockets (like packet capturing and filtering).

ALGORITHM :

1. Start the program and to include the necessary header files.
2. To define the packet length.
3. To declare the IP header structure using TCP header.
4. Using simple checksum process to check the process.
5. Using TCP \IP communication protocol to execute the program.
6. And using TCP\IP communication to enter the Source IP and port number and Target IP
address and port number.
7. The Raw socket () is created and accept the Socket ( ) and Send to ( ), ACK
8. Stop the program

PROGRAM:

```
//---cat rawtcp.c---
        // Run as root or SUID 0, just datagram no data/payload
        #include <unistd.h>
        #include <stdio.h>
        #include <sys/socket.h>
        #include <netinet/ip.h>
        #include <netinet/tcp.h>
        // Packet length
        #define PCKT_LEN 8192
        // May create separate header file (.h) for all
        // headers' structures
        // IP header's structure
        struct ipheader {
        unsigned char iph_ihl:5, /* Little-endian */
        iph_ver:4;
        unsigned char iph_tos;
        unsigned short int iph_len;
        unsigned short int iph_ident;
        unsigned char iph_flags;
        unsigned short int iph_offset;
        unsigned char iph_ttl;
        unsigned char iph_protocol;
        unsigned short int iph_chksum;
        unsigned int iph_sourceip;
        unsigned int iph_destip;
        };
        /* Structure of a TCP header */
        struct tcpheader {
        unsigned short int tcph_srcport;
        unsigned short int tcph_destport;
        unsigned int tcph_seqnum;
        unsigned int tcph_acknum;
        unsigned char tcph_reserved:4, tcph_offset:4;
        // unsigned char tcph_flags;
        unsigned int
        tcp_res1:4, /*little-endian*/
        tcph_hlen:4, /*length of tcp header in 32-bit
        words*/
        tcph_fin:1, /*Finish flag "fin"*/
        tcph_syn:1, /*Synchronize sequence numbers to
        start a connection*/
        tcph_rst:1, /*Reset flag */
        tcph_psh:1, /*Push, sends data to the
        application*/
        tcph_ack:1, /*acknowledge*/
```

```
tcph_urg:1, /*urgent pointer*/
tcph_res2:2;
unsigned short int tcph_win;
unsigned short int tcph_chksum;
unsigned short int tcph_urgptr;
};
// Simple checksum function, may use others such as Cyclic
Redundancy Check, CRC
unsigned short csum(unsigned short *buf, int len)
{
unsigned long sum;
for(sum=0; len>0; len--)
sum += *buf++;
sum = (sum >> 16) + (sum &0xffff);
sum += (sum >> 16);
return (unsigned short)(~sum);
}
int main(int argc, char *argv[])
{
int sd;
// No data, just datagram
char buffer[PCKT_LEN];
// The size of the headers
struct ipheader *ip = (struct ipheader *) buffer;
struct tcpheader *tcp = (struct tcpheader *) (buffer +
sizeof(struct ipheader));
struct sockaddr_in sin, din;
int one = 1;
const int *val = &one;
memset(buffer, 0, PCKT_LEN);
if(argc != 5)
{
printf("- Invalid parameters!!!\n");
printf("- Usage: %s <source hostname/IP> <source port>
<target hostname/IP> <target port>\n", argv[0]);
exit(-1);
}
sd = socket(PF_INET, SOCK_RAW, IPPROTO_TCP);
if(sd < 0)
{
perror("socket() error");
exit(-1);
}
else
printf("socket()-SOCK_RAW and tcp protocol is OK.\n");
// The source is redundant, may be used later if needed
// Address family
sin.sin_family = AF_INET;
```

```
din.sin_family = AF_INET;
// Source port, can be any, modify as needed
sin.sin_port = htons(atoi(argv[2]));
din.sin_port = htons(atoi(argv[4]));
// Source IP, can be any, modify as needed
sin.sin_addr.s_addr = inet_addr(argv[1]);
din.sin_addr.s_addr = inet_addr(argv[3]);
// IP structure
ip->iph_ihl = 5;
ip->iph_ver = 4;
ip->iph_tos = 16;
ip->iph_len = sizeof(struct ipheader) + sizeof(struct
tcpheader);
ip->iph_ident = htons(54321);
ip->iph_offset = 0;
ip->iph_ttl = 64;
ip->iph_protocol = 6; // TCP
ip->iph_chksum = 0; // Done by kernel
// Source IP, modify as needed, spoofed, we accept through
command line argument
ip->iph_sourceip = inet_addr(argv[1]);
// Destination IP, modify as needed, but here we accept
through command line argument
ip->iph_destip = inet_addr(argv[3]);
// The TCP structure. The source port, spoofed, we accept
through the command line
tcp->tcph_srcport = htons(atoi(argv[2]));
// The destination port, we accept through command line
tcp->tcph_destport = htons(atoi(argv[4]));
tcp->tcph_seqnum = htonl(1);
tcp->tcph_acknum = 0;
tcp->tcph_offset = 5;
tcp->tcph_syn = 1;
tcp->tcph_ack = 0;
tcp->tcph_win = htons(32767);
tcp->tcph_chksum = 0; // Done by kernel
tcp->tcph_urgptr = 0;
// IP checksum calculation
ip->iph_chksum = csum((unsigned short *) buffer,
(sizeof(struct ipheader) + sizeof(struct tcpheader)));
// Inform the kernel do not fill up the headers' structure,
we fabricated our own
if(setsockopt(sd, IPPROTO_IP, IP_HDRINCL, val, sizeof(one))
< 0)
{
perror("setsockopt() error");
exit(-1);
}
```

```
else
printf("setsockopt() is OK\n");
printf("Using:::::Source IP: %s port: %u, Target IP: %s
port: %u.\n", argv[1], atoi(argv[2]), argv[3],
atoi(argv[4]));
// sendto() loop, send every 2 second for 50 counts
unsigned int count;
for(count = 0; count < 20; count++)
{
if(sendto(sd, buffer, ip->iph_len, 0, (struct sockaddr
*)&sin, sizeof(sin)) < 0)
// Verify
{
perror("sendto() error");
exit(-1);
}
else
printf("Count #%u - sendto() is OK\n", count);
sleep(2);
}
close(sd);
return 0;
}
```

RESULT:

Thus the above programs using raw sockets TCP \IP (like packet capturing and filtering) was executed and successfully.

EX NO: 4.a PROGRAMS USING RPC / RMI

SIMPLE CALCULATOR

AIM:

To implement simple calculator on a remote host and invoke operations from a client.

ALGORITHM:

Interface:

Declare server's remote interface for all calculator operation by extending *Remote* interface

Implementation:

Define basic calculator operations such as summation, difference, product, quotient and remainder by extending *UnicastRemoteObject*.

Server:

1.Create a calculator object

2.Register the object with the RMI registry on the server machine using *rebind* method

Client

1.Obtain operands from the user

2.Lookup for the Calculator service on the remote server

3.Call all arithmetic operations on the remote server

4.Display result of various arithmetic operations.

Procedure

1. Compile the four java files (Interface, Implementation, Server and Client)
2. Generate stub by compiling the implementation file using RMI compiler
3. Distribute the class files of Client, Interface and Stub to the clients
4. Start the RMI registry on the server
5. Start the server
6. Call procedures that exist on the remote host from client machine.

PROGRAM:

// Declares remote method interfaces--CalcInf.java

```
import java.rmi.*;
public interface CalcInf extends Remote
{
public long add(int a, int b) throws RemoteException;
public int sub(int a, int b) throws RemoteException;
public long mul(int a, int b) throws RemoteException;
public int div(int a, int b) throws RemoteException;
public int rem(int a, int b) throws RemoteException;
}
// Implement Remote behavior--CalcImpl.java
import java.rmi.*;
import java.rmi.server.*;
public class CalcImpl extends UnicastRemoteObject implements
CalcInf
{
public CalcImpl() throws RemoteException { }
public long add(int a, int b) throws RemoteException
{
return a + b;
}
public int sub(int a, int b) throws RemoteException
{
int c = a > b ? a - b : b - a;
return c;
}
public long mul(int a, int b) throws RemoteException
{
return a * b;
}
public int div(int a, int b) throws RemoteException
{
return a / b;
}
public int rem(int a, int b) throws RemoteException
{
return a % b;
}
}
// Server that names the service implemented--CalcServer.java
import java.rmi.*;
public class CalcServer
{
public static void main(String args[])
{
try
{
CalcInf C = new CalcImpl();
Naming.rebind("CalcService", C);
}
catch (Exception e)
{
System.out.println(e.getMessage());
}
}
}
```

// Client that invokes remote host methods--CalcClient.java

```
import java.rmi.*;
import java.net.*;
public class CalcClient
{
public static void main(String[] args) throws Exception
{
try
{
CalcInf C = (CalcInf) Naming.lookup("rmi://" +
args[0] + "/CalcService");
int a, b;
if (args.length != 3)
{
System.out.println("Usage: java CalcClient
<remoteip> <operand1> <operand2>");
System.exit(-1);
}
a = Integer.parseInt(args[1]);
b = Integer.parseInt(args[2]);
System.out.println( "\nBasic Remote Calc\n" );
System.out.println("Summation : " + C.add(a, b));
System.out.println("Difference : " + C.sub(a, b));
System.out.println("Product : " + C.mul(a, b));
System.out.println("Quotient : " + C.div(a, b));
System.out.println("Remainder : " + C.rem(a, b));
}
catch (Exception E)
{
System.out.println(E.getMessage());
}
}
}
```

OUTPUT

Server

```
C:\>javac CalcInf.java
C:\>javac CalcImpl.java
C:\>javac CalcServer.java
C:\>javac CalcClient.java
C:\>rmic CalcImpl
C:\>start rmiregistry
C:\>java CalcServer
```

Client

```
C:\>java CalcClient 172.16.6.45 6 8
Basic Remote Calc
Summation : 14
Difference : 2
Product : 48
Quotient : 0
Remainder : 6
```

RESULT

Thus remote procedure calls for basic operations of a calculator is executed using Java RMI.

Ex.no:4.b **FIBONACCI SERIES**

AIM

To implement fibonacci series on a remote host and generate terms onto a client.

ALGORITHM

Interface

Declare server's remote interface for Fibonacci series generation by extending *Remote* interface

Implementation

Define the procedure for fibonacci series (new term = sum of last two terms) that extends
UnicastRemoteObject.

Server

1. Create a interface object
2. Register the object with the RMI registry on the server machine using *rebind* method

Client

1. Obtain number of terms from the user
2. Lookup for the Fibonacci service on the remote server
3. Call fibonacci series
4. Display the generated fibonacci terms.

Procedure

1. Compile the four java files (Interface, Implementation, Server and Client)
2. Generate stub by compiling the implementation file using RMI compiler
3. Distribute the class files of Client, Interface and Stub to the client machines
4. Start the RMI registry on the server
5. Start the server
6. Call procedures that exist on the remote host from client machine.

PROGRAM

// remote method interface--FiboIntf.java

```
import java.rmi.*;
public interface FiboIntf extends Remote
{
int[] fiboseries(int n)throws RemoteException;
}
```

//Remote behaviour implementation--FiboImpl.java

```
import java.rmi.*;
import java.rmi.server.*;
public class FiboImpl extends UnicastRemoteObject implements
FiboIntf
{
public FiboImpl() throws RemoteException { }
public int[] fiboseries(int n)throws RemoteException
{
int f1 = 0, f2 = 1, f3, i;
int arr[]= new int[25];
arr[0] = f1;
arr[1] = f2;
for(i=2; i<n; i++)
{
f3 = f1 + f2;
arr[i] = f3;
f1 = f2;
f2 = f3;
}
return(arr);
}
}
```

//Server that registers the service--FiboServer.java

```
import java.rmi.*;
public class FiboServer
{
public static void main(String arg[])
{
try
{
FiboIntf Fi = new FiboImpl();
Naming.rebind("FiboGen", Fi);
}
catch(Exception e)
{
System.out.println(e.getMessage());
}
}
}
```

// Client that invokes remote host methods--FiboClient.java

```
import java.rmi.*;
import java.net.*;
public class FiboClient
{
public static void main(String args[])
{
try
{
FiboIntf Fi = (FiboIntf) Naming.lookup("rmi://" +
args[0] + "/FiboGen");
if (args.length != 2)
{
System.out.println("Usage: java FiboClient
<remoteip> <terms>");
System.exit(-1);
}
int n = Integer.parseInt(args[1]);
int a[]=Fi.fiboseries(n);
System.out.print("\nFibonacci Series for " + n +
" terms : ");
for(int i=0; i<n; i++)
System.out.print(a[i] + " ");
}
catch(Exception e)
{
System.out.println(e.getMessage());
}
}
}
```

OUTPUT

Server

```
C:\>javac FiboIntf.java
C:\>javac FiboImpl.java
C:\>javac FiboServer.java
C:\>javac FiboClient.java
C:\>rmic FiboImpl
C:\>start rmiregistry
C:\>java FiboServer
```

Client

```
C:\>java FiboClient 172.16.6.45 8
Fibonacci Series for 8 terms : 0 1 1 2 3 5 8 13
```

RESULT

Thus remote procedure call for fibonacci series generation is executed using Java RMI.

Ex.no: 4.C **FACTORIAL VALUE**

AIM

To implement factorial on a remote host and obtain its value from a client.

ALGORITHM

Interface

Declare server's remote interface for Factorial by extending *Remote* interface

Implementation

Define the procedure for factorial (n! = n × (n-1) × … × 1) that extends *UnicastRemoteObject*.

Server

1. Create a interface object
2. Register the object with the RMI registry on the server machine using *rebind* method

Client

1. Obtain number from the user
2. Lookup for the Factorial service on the remote server
3. Call the remote factorial method
4. Display factorial value.

Procedure

1. Compile the four java files (Interface, Implementation, Server and Client)
2. Generate stub by compiling the implementation file using RMI compiler
3. Distribute the class files of Client, Interface and Stub to the client machines
4. Start the RMI registry on the server
5. Start the server
6. Call procedures that exist on the remote host from client machine**.**

PROGRAM

// remote method interface--FactIntf.java

```
import java.rmi.*;
public interface FactIntf extends Remote
{
long factorial(int n)throws RemoteException;
}
```

//Remote behaviour implementation--FactImpl.java

```
import java.rmi.*;
import java.rmi.server.*;
public class FactImpl extends UnicastRemoteObject implements
FactIntf
{
public FactImpl() throws RemoteException { }
public long factorial(int n)throws RemoteException
{
long f = 1;
for(int i=n; i>0; i--)
f *= i;
return f;
}
}
```

//Server that registers the service--FactServer.java

```
import java.rmi.*;
public class FactServer
{
public static void main(String arg[])
{
try
{
FactIntf Fa = new FactImpl();
Naming.rebind("FactService", Fa);
}
catch(Exception e)
{
System.out.println(e.getMessage());
}
}
```

}

// Client that invokes remote host methods--FactClient.java

```
import java.rmi.*;
import java.net.*;
public class FactClient
{
public static void main(String args[])
{
try
{
FactIntf Fa = (FactIntf) Naming.lookup("rmi://" +
args[0] + "/FactService");
if (args.length != 2)
{
System.out.println("Usage: java FactClient
<remoteip> <number>");
System.exit(-1);
}
int n = Integer.parseInt(args[1]);
long factval = Fa.factorial(n);
System.out.print("\n" + n + " Factorial value is " +
factval);
}
catch(Exception e)
{
System.out.println(e.getMessage());
```

```
        }
    }
}
```

OUTPUT

Server

```
C:\>javac FactIntf.java
C:\>javac FactImpl.java
C:\>javac FactServer.java
C:\>javac FactClient.java
C:\>rmic FactImpl
C:\>start rmiregistry
C:\>java FactServer
```

Client

```
C:\>java FactClient 172.16.6.45 10
10 Factorial value is 3628800
C:\>java FactClient 172.16.6.45 0
0 Factorial value is 1
```

RESULT

Thus remote procedure call to determine factorial value is executed using Java RMI.

EX No: 05 SIMULATION OF SLIDING WINDOW PROTOCOL

AIM:

To write a C program to perform sliding window.

ALGORITHM:

1. Start the program.
2. Get the frame size from the user.
3. To create the frame based on the user request.
4. To send frames to server from the client side.
5. If your frames reach the server it will send ACK signal to client otherwise it will send NACK signal to client.
6. Stop the program.

PROGRAM:

// SLIDING WINDOW PROTOCOL

Client:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
struct mymsgbuf
{
long mtype;
char mtext[25];
};
FILE *fp;
int main()
{
struct mymsgbuf buf;
int msgid;
int i=0,s;
int count=0,frmsz;
int a[100];
char d;
if((msgid=msgget(89,IPC_CREAT|0666))==-1)
{
printf("\n ERROR IN MSGGET");
exit(0);
}
printf("\n Enter the frame size:");
scanf("%d",&frmsz);
if((fp=fopen("check","r"))==NULL)
printf("\n FILE NOT OPENED");
else
printf("\n FILE OPENED");
while(!feof(fp))
{
d=getc(fp);
a[i]=d;
i++;
}
s=i;
for(i=0;i<frmsz;i++) //print from the check file
printf("\t %c",a[i]);
for(i=0;i<frmsz;i++)
{ if((msgrcv(msgid,&buf,sizeof(buf),0,1))==-1)
{
printf("\n ERROR IN MSGRCV");
exit(0);
}
printf("\n RECEIVED FRAMES ARE:%c",buf.mtext[i]);
}
for(i=0;i<frmsz;i++)
{ if(a[i]==buf.mtext[i])
count++;
} if(count==0)
{
printf("\n FRAMES WERE NOT RECEIVED IN CORRECT SEQ");
exit(0);
} if(count==frmsz)
{
printf("\n FRAMES WERE RECEIVED IN CORRECT SEQ");
} else
{
printf("\n FRAMES WERE NOT RECEIVED IN CORRECT SEQ");
}}
```

**Sliding Window Protocol -
Server**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
struct mymsgbuf
{ long mtype;
char mtext[25];
};
FILE *fp;
int main()
{s
truct mymsgbuf buf;
int si,ei,sz;
int msgid;
int i=0,s;
int a[100];
char d;
if((fp=fopen("send","r"))==NULL)
printf("\n FILE NOT OPENED");
else
printf("\n FILE OPENED");
printf("\n Enter starting and ending index of frame array:");
scanf("%d%d",&si,&ei);
sz=ei-si;
if((msgid=msgget(89,IPC_CREAT|0666))==-1)
{
printf("\n ERROR IN MSGGET");
exit(0);
}
while(!feof(fp))
{
d=getc(fp);
a[i]=d;
i++;
}s
=i;
buf.mtype=1;
for(i=si;i<=ei;i++)
{
buf.mtext[i]=a[i];
}
for(i=si;i<=ei;i++) //the frames to be sent
printf("\t %c",buf.mtext[i]);
for(i=0;i<=sz;i++)
{ if((msgsnd(msgid,&buf,sizeof(buf),0))==-1)
{
printf("\n ERROR IN MSGSND");
exit(0);
}}
printf("\n FRAMES SENT");
return 0;
}
```

RESULT:

Thus the above program sliding window protocol was executed and successfully

Ex No:6 **ADDRESS RESOLUTION PROTOCOL**

AIM:

To get the MAC or Physical address of the system using Address Resolution Protocol.

ALGORITHM:

1. Include necessary header files.
2. Initialize the arpreq structure initially to zero.
3. Get the IPAddress of the system as command line argument.
4. Check whether the given IPAddress is valid.
5. Copy the IPAddress from sockaddr_in structure to arpreq structure using miscopy ()
system call.
6. Create a socket of type SOCK_DGRAM.
7. Calculate the MAC address for the given IPAddress using ioctl() system call.
8. Display the IPAddress and MAC address in the standard output.

Program:

```
#include<unistd.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<net/if_arp.h>
#include<stdlib.h>
#include<stdio.h>
#include<netdb.h>
#include<sys/ioctl.h>
#include<arpa/inet.h>
int main(int argc,char *argv[])
{ int sd;
unsigned char *ptr;
struct arpreq myarp={{0}};
struct sockaddr_in sin={0};
sin.sin_family=AF_INET;
if(inet_aton(argv[1],&sin.sin_addr)==0)
{
printf("IP address Entered%s is not valid\n",argv[1]);
exit(0);
}
memcpy(&myarp.arp_pa,&sin,sizeof(myarp.arp_pa));
strcpy(myarp.arp_dev,"eth0");
sd=socket(AF_INET,SOCK_DGRAM,0);
if(ioctl(sd,SIOCGARP,&myarp)==1)
{
printf("No entry in ARP cache for%s",argv[1]);
exit(0);
}
ptr=&myarp.arp_ha.sa_data[0];
printf("MAC address for%s",argv[1]);
printf("%x%x%x%x%x%x\n",*ptr,*(ptr+1),*(ptr+2),*(ptr+3),*(ptr+4),*(ptr+5));
return(0);
}
```

RESULT:

Thus the MAC address was generated for IP address using ARP protocol.

EX NO: 7 IMPLEMENTING ROUTING PROTOCOLS

AIM:

To simulate the Implementing Routing Protocols using border gateway protocol (BGP)

ALGORITHM:

1. Read the no. of nodes n.
2. Read the cost matrix for the path from each node to another node.
3. Initialize SOURCE to 1 and include 1.
4. Compute D of a node which is the distance from source to that corresponding node.
5. Repeat step 6 to step 8 for n-l nodes.
6. Choose the node that has not been included whose distance is minimum and include that node.
7. For every other node not included compare the distance directly from the source with the distance to reach the node using the newly included node.
8. Take the minimum value as the new distance.
9. Print all the nodes with shortest path cost from source node.

Program :

```
#include <stdio.h>
#include<conio.h>
int main()
{
int n;
int i,j,k;
int a[10][10],b[10][10];
```

```
printf("\n Enter the number of nodes:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
for(j=0;j<n;j++)
{
printf("\n Enter the distance between the host %d - %d:",i+1,j+1);
scanf("%d",&a[i][j]);
}
}
for(i=0;i<n;i++)
{
for(j=0;j<n;j++)
{
printf("%d\t",a[i][j]);
}
printf("\n");
}
for(k=0;k<n;k++)
{
for(i=0;i<n;i++)
{
for(j=0;j<n;j++)
{
if(a[i][j]>a[i][k]+a[k][j])
{
a[i][j]=a[i][k]+a[k][j];
}
}
}
}
for(i=0;i<n;i++)
{
for(j=0;j<n;j++)
{
b[i][j]=a[i][j];
if(i==j)
{
b[i][j]=0;
}
}}
printf("\n The output matrix:\n");
for(i=0;i<n;i++)
{
for(j=0;j<n;j++)
{
printf("%d\t",b[i][j]);
}
printf("\n");
}
getch();
}
```

RESULT:

Thus the above program to simulate the Implementing Routing Protocols using border gateway protocol was executed and successfully.

EX NO:8 OPEN SHORTEST PATH FIRST ROUTING PROTOCOL

AIM:

To simulate the OPEN SHORTEST PATH FIRST routing protocol based on the cost assigned to the path.

ALGORITHM:

1. Read the no. of nodes n.
2. Read the cost matrix for the path from each node to another node.
3. Initialize SOURCE to 1 and include 1.
4. Compute D of a node which is the distance from source to that corresponding node.
5. Repeat step 6 to step 8 for n-l nodes.
6. Choose the node that has not been included whose distance is minimum and include that node.
7. For every other node not included compare the distance directly from the source with the distance to reach the node using the newly included node.
8. Take the minimum value as the new distance.
9. Print all the nodes with shortest path cost from source node

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
int a[5][5],n,i,j;
void main()
```

```
{
void getdata();
void shortest();
void display();
clrscr();
printf("\n\n PROGRAM TO FIND SHORTEST PATH BETWEEN TWO
NODES\n");
getdata();
shortest();
display();
getch();
}
void getdata()
{
clrscr();
printf("\n\nENTER THE NUMBER OF HOST IN THE GRAPH\n");
scanf("%d",&n);
printf("\n\nIF THERE IS NO DIRECT PATH \n");
printf(" \n\nASSIGN THE HIGHEST DISTANCE VALUE 1000 \n");
for(i=0;i<n;i++)
{
a[i][j]=0;
for(j=0;j<n;j++)
{
if(i!=j)
{
printf("\n\nENTER THE DISTANCE BETWENN (%d,
%d): ",i+1,j+1);
scanf("%d",&a[i][j]);
if(a[i][j]==0)
a[i][j]=1000;
}
}
}
}
void shortest()
{
int i,j,k;
for(k=0;k<n;k++)
for(i=0;i<n;i++)
for(j=0;j<n;j++)
{
if(a[i][k]+a[k][j]<a[i][j])
a[i][j]=a[i][k]+a[k][j];
}
}
void display()
{
```

```
int i,j;
for(i=0;i<n;i++)
for(j=0;j<n;j++)
if(i!=j)
{
printf("\n SHORTEST PATH IS : (%d,%d)--%d\n",i+1,j+1,a[i][j]);
}
getch(); }
```

RESULT:

Thus the above program to simulate the Implementing Routing Protocols using open shortest path first (OSPF) was executed and successfully

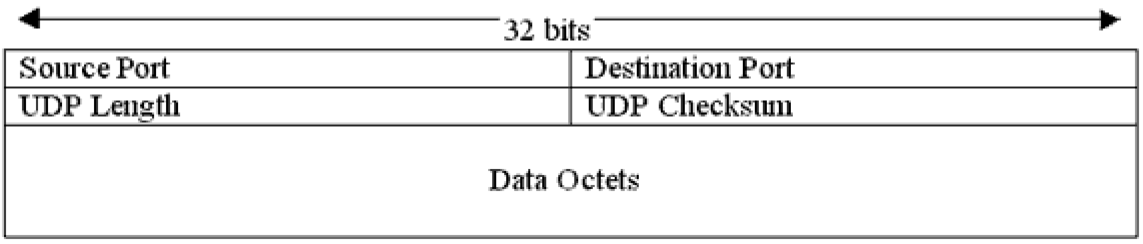
EX NO: 9 STUDY OF UDP PERFORMANCE

Introduction

Most network games use the User Datagram Protocol (UDP) as the underlying transport protocol. The Transport Control Protocol (TCP), which is what most Internet traffic relies on, is a reliable connection-oriented protocol that allows data streams coming from a machine connected to the Internet to be received without error by any other machine on the Internet. UDP however, is an unreliable connectionless protocol that does not guarantee accurate or unduplicated delivery of data.

Why do games use UDP?

TCP has proved too complex and too slow to sustain real-time game-play. UDP allows gaming application programs to send messages to other programs with the minimum of protocol mechanism. Games do not rely upon ordered reliable delivery of data streams. What is more important to gaming applications is the prompt delivery of data. UDP allows applications to send IP datagram to other applications without having to establish a connection and than having to release it later, which increases the speed of communication. UDP is described in RFC 768. The UDP segment shown above consists of an 8-byte header followed by the data octets.


32 bits

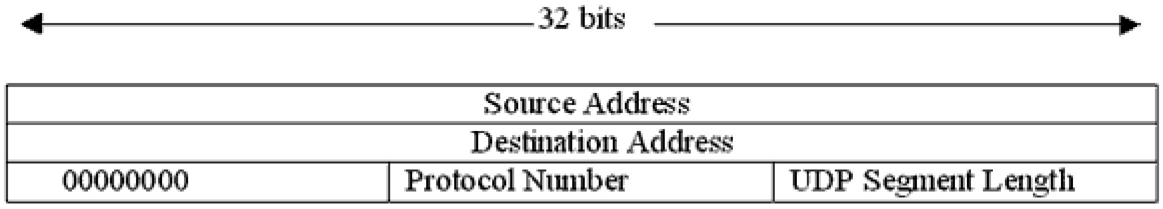
Source Port	Destination Port
UDP Length	UDP Checksum
Data Octets	

The UDP Header.

The UDP segment shown above consists of an 8-byte header followed by the data octets

Fields

The source and destination ports identify the end points within the source and destination machines. The source port indicates the port of the sending process and unless otherwise stated it is the port to which a reply should be sent to. A zero is inserted into it if it is not used. The UDP Length field shows the length of the datagram in octets. It includes the 8-byte header and the data to be sent.


32 bits

Source Address		
Destination Address		
00000000	Protocol Number	UDP Segment Length

The pseudo-header included in the UDP checksum.

The UDP checksum field contains the UDP header, UDP data and the pseudo-header shown above. The pseudo-header contains the 32-bit IP addresses of the source and destination machines, the UDP protocol number and the byte count for the UDP segment. The pseudo-header helps to find undelivered packets or packets that arrive at the wrong address. However the pseudo-header violates the protocol hierarchy because the IP addresses which are used in it belong to the IP layer and not to the UDP layer.

UDP Latency

While TCP implements a form of flow control to stop the network from flooding there is no such concept in UDP. This is because UDP does not rely on acknowledgements to signal successful delivery of data. Packets are simply transmitted one after another with complete disregard to event of the receiver being flooded.

The effects of UDP

As mentioned before the majority of the traffic on the Internet relies on TCP. With the explosive increase in the amount of gaming taking place on the Internet, and with most of these games using UDP, there are concerns about the effects that UDP will have on TCP traffic.

UDP Broadcast Flooding

A *broadcast* is a data packet that is destined for multiple hosts. Broadcasts can occur at the data link layer and the network layer. Data-link broadcasts are sent to all hosts attached to a particular physical network. Network layer broadcasts are sent to all hosts attached to a particular logical network. The Transmission Control Protocol/Internet Protocol (TCP/IP) supports the following types of broadcast packets:

- *All ones*—By setting the broadcast address to all ones (255.255.255.255), all hosts on the network receive the broadcast.
- *Network*—By setting the broadcast address to a specific network number in the network portion of the IP address and setting all ones in the host portion of the broadcast address, all hosts on the specified network receive the broadcast. For example, when a broadcast packet is sent with the broadcast address of 131.108.255.255, all hosts on network number 131.108 receive the broadcast.
- *Subnet*—By setting the broadcast address to a specific network number and a specific subnet number, all hosts on the specified subnet receive the broadcast. For example, when a broadcast packet is set with the broadcast address of 131.108.4.255, all hosts on subnet 4 of network 131.108 receive the broadcast. Because broadcasts are recognized by all hosts, a significant goal of router configuration is to control unnecessary proliferation of broadcast packets.

Cisco routers support two kinds of broadcasts:*directed* and *flooded*. A directed broadcast is a packet sent to a specific network or series of networks, whereas a flooded broadcast is a packet sent to every network. In IP internetworks, most broadcasts take the form of User Datagram Protocol (UDP) broadcasts. Although current IP implementations use a broadcast address of all ones, the first IP implementations used a broadcast address of all zeros. Many of the early implementations do not recognize broadcast addresses of all ones and fail to respond to the broadcast correctly. Other early implementations forward broadcasts of all ones, which causes a serious network overload known as a *broadcast storm*.

Implementations that exhibit these problems include systems based on versions of BSD UNIX prior to Version 4.3. In the brokerage community, applications use UDP broadcasts to transport market data to the desktops of traders on the trading floor. This case study gives examples of how brokerages have implemented both directed and flooding broadcast schemes in an environment that consists of Cisco routers and Sun workstations.Note that the addresses in this network use a 10-bit netmask of 255.255.255.192.

Internetworking Case Studies

UDP broadcasts must be forwarded from a source segment (Feed network) to many
destination segments that are connected redundantly. Financial market data, provided, for example, by Reuters, enters the network through the Sun workstations connected to the Feed network and is disseminated to the TIC servers. The TIC servers are Sun workstations running Teknekron Information Cluster software. The Sun workstations on the trader networks subscribe to the TIC servers for the delivery of certain market data, which the TIC servers deliver by means of UDP broadcasts. The two routers in this network provide redundancy so that if one router becomes unavailable, the other router can assume the load of the failed router without intervention from an
operator. The connection between each router and the Feed network is for network administration purposes only and does not carry user traffic.

Two different approaches can be used to configure Cisco routers for forwarding UDP broadcast traffic: IP helper addressing and UDP flooding. This case study analyzes the advantages and disadvantages of each approach.

Router A Router B

164.53.8.0 164.53.9.0 164.53.10.0

E1

E0 E0

E1

TIC server network 164.53.7.0

200.200.200.0

Feed Network

200.200.200.61 200.200.200.62

164.53.7.61 164.53.7.62

164.53.8.61

164.53.9.61

164.53.10.61

Trader Net 1 Trader Net 2 Trader Net 3

TIC TIC TIC TIC

164.53.9.62

164.53.10.62

E4 164.53.8.62

E2 E3

E4

E2 E3

Implementing IP Helper Addressing

IP helper addressing is a form of static addressing that uses directed broadcasts to forward local and all-nets broadcasts to desired destinations within the internetwork.

To configure helper addressing, you must specify the **ip helper-address** command on every interface on every router that receives a broadcast that needs to be forwarded. On Router A and Router B, IP helper addresses can be configured to move data from the TIC server network to the trader networks. IP helper addressing in not the optimal solution for this type of topology because each router receives unnecessary broadcasts from the other router

In this case, Router A receives each broadcast sent by Router B *three times*, one for each segment, and Router B receives each broadcast sent by Router A three times, one for each segment. When each broadcast is received, the router must analyze it and determine that

the broadcast does not need to be forwarded. As more segments are added to the network, the routers become overloaded with unnecessary traffic, which must be analyzed and discarded.
When IP helper addressing is used in this type of topology, no more than one router can be configured to forward UDP broadcasts (unless the receiving applications can handle duplicate broadcasts). This is because duplicate packets arrive on the trader network. This restriction limits redundancy in the design and can be undesirable in some implementations.
To send UDP broadcasts bidirectionally in this type of topology, a second **ip helper address** command must be applied to every router interface that receives UDP broadcasts. As more segments and devices are added to the network, more **ip helper address** commands are required to reach them, so the administration of these routers becomes more complex over time. Note, too, that bidirectional traffic in this topology significantly impacts router performance.
Router A Router B
164.53.8.0 164.53.9.0 164.53.10.0
E1
E0 E0
E1
TIC server network 164.53.7.0
200.200.200.0
Feed Network
200.200.200.61 200.200.200.62
164.53.7.61 164.53.7.62
164.53.8.61
164.53.9.61
164.53.10.61
Trader Net 1 Trader Net 2 Trader Net 3
TIC TIC TIC TIC
164.53.9.62
164.53.10.62
E4 164.53.8.62
E2 E3
E4

E2 E3
UDP packets

Although IP helper addressing is well-suited to nonredundant, nonparallel topologies that do not require a mechanism for controlling broadcast loops, in view of these drawbacks, IP helper addressing does not work well in this topology. To improve performance, network designers considered several other alternatives:

• *Setting the broadcast address on the TIC servers to all ones (255.255.255.255)*—This alternative was dismissed because the TIC servers have more than one interface, causing TIC broadcasts to be sent back onto the Feed network. In addition, some workstation implementations do not allow all ones broadcasts when multiple interfaces are present.

• *Setting the broadcast address of the TIC servers to the major net broadcast (164.53.0.0)*—This alternative was dismissed because the Sun TCP/IP implementation does not allow the use of major net broadcast addresses when the network is subnetted.

• *Eliminating the subnets and letting the workstations use Address Resolution Protocol (ARP) to learn addresses*—This alternative was dismissed because the TIC servers cannot quickly learn an alternative route in the event of a primary router failure.With alternatives eliminated, the network designers turned to a simpler implementation that supports redundancy without duplicating packets and that ensures fast convergence and minimal loss of data when a router fails: UDP flooding.

UDP flooding uses the spanning tree algorithm to forward packets in a controlled manner. Bridging is enabled on each router interface for the sole purpose of building the spanning tree.

The spanning tree prevents loops by stopping a broadcast from being forwarded out an interface on which the broadcast was received. The spanning tree also prevents packet duplication by placing certain interfaces in the blocked state (so that no packets are forwarded) and other interfaces in the forwarding state (so that packets that need to be forwarded are forwarded).

To enable UDP flooding, the router must be running software that supports transparent bridging and bridging must be configured on each interface that is to participate in the flooding. If bridging is not configured for an interface, the interface will receive broadcasts, but the router will not forward those broadcasts and will not use that interface as a destination for sending broadcasts received on a different interface. Releases prior to Cisco Internetwork Operating System (Cisco IOS) Software Release 10.2 do not support flooding subnet broadcasts.

When configured for UPD flooding, the router uses the destination address specified by the **ip broadcast-address** command on the output interface to assign a destination address to a flooded UDP datagram. Thus, the destination address might change as the datagram propagates through the network. The source address, however, does not change. With UDP flooding, both routers use a spanning tree to control the network topology for the purpose of forwarding broadcasts. The key commands for enabling UDP flooding are as follows:

bridge *group* protocol *protocol*ip forward-protocol spanning tree
bridge-group *group* input-type-list *access-list-number*

The **bridge protocol** command can specify either the **dec** keyword (for the DEC spanning-tree protocol) or the **ieee** keyword (for the IEEE Ethernet protocol). All routers in the network must enable the same spanning tree protocol. The **ip forward-protocol spanning tree** command uses the database created by the **bridge protocol** command. Only one broadcast packet arrives at each segment, and UDP broadcasts can traverse the network in both directions.

Because bridging is enabled only to build the spanning tree database, use access lists to prevent the spanning tree from forwarding non-UDP traffic. To determine which interface forwards or blocks packets, the router configuration specifies a path cost for each interface. The default path cost for Ethernet is 100. Setting the path cost for each interface on Router B to 50 causes the spanning tree algorithm to place the interfaces in Router B in forwarding state. Given the higher path cost (100) for the interfaces in Router A, the interfaces in Router A are in the blocked state and do not forward the broadcasts.

With these interface states, broadcast traffic flows through Router B. If Router B fails, the spanning tree algorithm will place the interfaces in Router A in the forwarding state, and Router A will forward broadcast traffic. With one router forwarding broadcast traffic from the TIC server network to the trader networks, it is desirable to have the other forward uncast traffic. For that reason, each router enables the ICMP Router Discovery Protocol (IRDP), and each workstation on the trader networks runs the **irdp** daemon.

On Router A, the **preference** keyword sets a higher IRDP preference than does the configuration for Router B, which causes each **irdp** daemon to use Router A as its preferred default gateway for unicast traffic forwarding. Users of those workstations can use **netstat -rn** to see how the routers are being used. On the routers, the **holdtime**, **maxadvertinterval**, and **minadvertinterval** keywords reduce the advertising interval from the default so that the **irdp** daemons running on the hosts expect to see advertisements more frequently. With the advertising interval reduced, the workstations will adopt Router B more quickly if Router A becomes unavailable. With this configuration, when a router becomes unavailable, IRDP offers a convergence time of less than one minute. IRDP is preferred over the Routing Information Protocol (RIP) and default gateways for the following reasons:

• RIP takes longer to converge, typically from one to two minutes.

• Configuration of Router A as the default gateway on each Sun workstation on the trader networks would allow those Sun workstations to send unicast traffic to Router A, but would not provide an alternative route if Router A becomes unavailable.

EX.NO: 10 **Study of TCP performance**

Introduction:

The *Transmission Control Protocol* (TCP) and the User Datagram Protocol (UDP) are both IP transport-layer protocols. UDP is a lightweight protocol that allows applications to make direct use of the unreliable datagram service provided by the underlying IP service. UDP is commonly used to support applications that use simple query/response transactions, or applications that support real-time communications. TCP provides a reliable data-transfer service, and is used for both bulk data transfer and interactive data applications. TCP is the major transport protocol in use in most IP networks, and supports the transfer of over 90 percent of all traffic across the public Internet today. Given this major role for TCP, the performance of this protocol forms a significant part of the total picture of service performance for IP networks. In this article we examine TCP in further detail, looking at what makes a TCP session perform reliably and well. This article draws on material published in the *Internet Performance Survival Guide*.

Overview of TCP

TCP is the embodiment of reliable end-to-end transmission functionality in the overall Internet architecture. All the functionality required to take a simple base of IP datagram delivery and build upon this a control model that implements reliability, sequencing, flow control, and data streaming is embedded within TCP .

TCP provides a communication channel between processes on each host system. The channel is reliable, full-duplex, and streaming. To achieve this functionality, the TCP drivers break up the session data stream into discrete segments, and attach a TCP header to each segment. An IP header is attached to this TCP packet, and the composite packet is then passed to the network for delivery. This TCP header has numerous fields that are used to support the intended TCP functionality. TCP has the following functional characteristics:

- *Unicast protocol*: TCP is based on a unicast network model, and supports data exchange between precisely two parties. It does not support broadcast or multicast network models.
- *Connection state*: Rather than impose a state within the network to support the connection, TCP uses synchronized state between the two endpoints. This

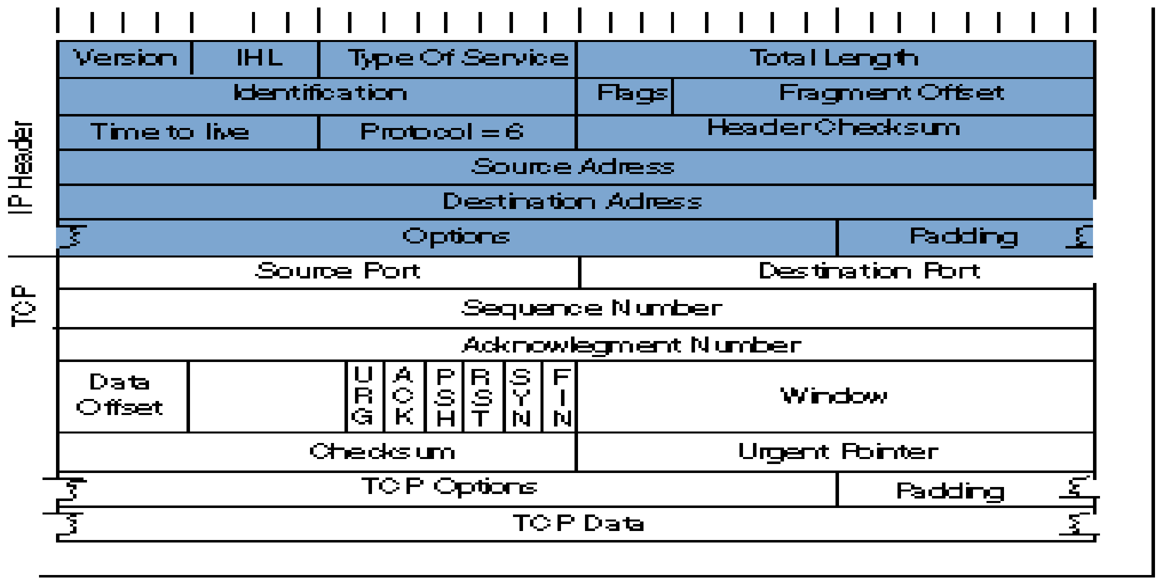
synchronized state is set up as part of an initial connection process, so TCP can be regarded as a connection-oriented protocol. Much of the protocol design is intended to ensure that each local state transition is communicated to, and acknowledged by, the remote party.

- *Reliable* : Reliability implies that the stream of octets passed to the TCP driver at one end of the connection will be transmitted across the network so that the stream is presented to the remote process as the same sequence of octets, in the same order as that generated by the sender.This implies that the protocol detects when segments of the data stream have been discarded by the network, reordered, duplicated, or corrupted. Where necessary, the sender will retransmit damaged segments so as to allow the receiver to reconstruct the original data stream. This implies that a TCP sender must maintain a local copy of all transmitted data until it receives an indication that the receiver has completed an accurate transfer of the data.
- *Full duplex* : TCP is a full-duplex protocol; it allows both parties to send and receive data within the context of the single TCP connection.
- *Streaming* : Although TCP uses a packet structure for network transmission, TCP is a true streaming protocol, and application-level network operations are not transparent. Some protocols explicitly encapsulate each application transaction; for every *write* , there must be a matching *read* . In this manner, the application-derived segmentation of the data stream into a logical record structure is preserved across the network. TCP does not preserve such an implicit structure imposed on the data stream, so that there is no pairing between *write* and *read* operations within the network protocol. For example, a TCP application may *write* three data blocks in sequence into the network connection, which may be collected by the remote reader in a single *read* operation. The size of the data blocks (segments) used in a TCP session is negotiated at the start of the session. The sender attempts to use the largest segment size it can for the data transfer, within the constraints of the maximum segment size of the receiver, the maximum segment size of the configured sender, and the maxi-mum supportable non-fragmented packet size of the network path (path *Maximum Transmission Unit* [MTU]). The path MTU is refreshed periodically to adjust to any changes that may occur within the network while the TCP connection is active.

- *Rate adaptation* : TCP is also a rate-adaptive protocol, in that the rate of data transfer is intended to adapt to the prevailing load conditions within the network and adapt to the processing capacity of the receiver. There is no predetermined TCP data-transfer rate; if the network and the receiver both have additional available capacity, a TCP sender will attempt to inject more data into the network to take up this available space. Conversely, if there is congestion, a TCP sender will reduce its sending rate to allow the network to recover. This adaptation function attempts to achieve the highest possible data-transfer rate without triggering consistent data loss.

The TCP Protocal Header

The TCP header structure, shown in Figure 1, uses a pair of 16-bit source and destination Port addresses. The next field is a 32-bit sequence number, which identifies the sequence number of the first data octet in this packet. The sequence number does not start at an initial value of 1 for each new TCP connection; the selection of an initial value is critical, because the initial value is intended to prevent delayed data from an old connection from being incorrectly interpreted as being valid within a current connection. The sequence number is necessary to ensure that arriving packets can be ordered in the sender?s original order. This field is also used within the flow-control structure to allow the association of a data packet with its corresponding acknowledgement, allowing a sender to estimate the current round-trip time across the network. Figure 1: The TCP/IP Datagram



The *acknowledgment sequence number* is used to inform the remote end of the data that has been successfully received. The acknowledgment sequence number is actually one greater than that of the last octet correctly received at the local end of the connection. The *data offset* field indicates the number of four-octet words within the TCP header. Six single *bit flags* are used to indicate various conditions. URG is used to indicate whether the urgent pointer is valid. ACK is used to indicate whether the *acknowledgment* field is valid. PSH is set when the sender wants the remote application to push this data to the remote application. RST is used to reset the connection. SYN (for *synchronize*) is used within the connection startup phase, and FIN (for *finish*) is used to close the connection in an orderly fashion. The *window* field is a 16-bit count of available buffer space. It is added to the acknowledgment sequence number to indicate the highest sequence number the receiver can accept. The TCP *checksum* is applied to a synthesized header that includes the source and destination addresses from the outer IP datagram. The final field in the TCP header is the urgent pointer, which, when added to the sequence number, indicates the sequence number of the final octet of urgent data if the urgent flag is set.

Many options can be carried in a TCP header. Those relevant to TCP performance include:

- *Maximum-receive-segment-size option* : This option is used when the connection is being opened. It is intended to inform the remote end of the maximum segment size, measured in octets, that the sender is willing to receive on the TCP connection. This option is used only in the initial SYN packet (the initial packet exchange that opens a TCP connection). It sets both the maximum receive segment size and the maximum size of the advertised TCP window, passed to the remote end of the connection. In a robust implementation of TCP, this option should be used with path MTU discovery to establish a segment size that can be passed across the connection without fragmentation, an essential attribute of a high-performance data flow.
- *Window-scale option* : This option is intended to address the issue of the maximum window size in the face of paths that exhibit a high-delay bandwidth product. This option allows the window size advertisement to be right-shifted by the amount specified (in binary arithmetic, a right-shift corresponds to a multiplication by 2). Without this option, the maximum window size that can be advertised is 65,535 bytes (the maximum value obtainable in a 16-bit field). The limit of TCP transfer speed is

effectively one window size in transit between the sender and the receiver. For high-speed, long-delay networks, this performance limitation is a significant factor, because it limits the transfer rate to at most 65,535 bytes per round-trip interval, regardless of available network capacity. Use of the window-scale option allows the TCP sender to effectively adapt to high-band-width, high-delay network paths, by allowing more data to be held in flight. The maximum window size with this option is 2^{30} bytes. This option is negotiated at the start of the TCP connection, and can be sent in a packet only with the SYN flag. Note that while an MTU discovery process allows optimal setting of the maximum-receive-segment-size option, no corresponding bandwidth delay product discovery allows the reliable automated setting of the window-scale option.

- SACK-*permitted option and* SACK *option* : This option alters the acknowledgment behavior of TCP. SACK is an acronym for *selective acknowledgment* . The SACK-permitted option is offered to the remote end during TCP setup as an option to an opening SYN packet. The SACK option permits selective acknowledgment of permitted data. The default TCP acknowledgment behavior is to acknowledge the highest sequence number of in-order bytes. This default behavior is prone to cause unnecessary retransmission of data, which can exacerbate a congestion condition that may have been the cause of the original packet loss. The SACK option allows the receiver to modify the acknowledgment field to describe noncontinuous blocks of received data, so that the sender can retransmit only what is missing at the receiver's end.

Any robust high-performance implementation of TCP should negotiate these parameters at the start of the TCP session, ensuring the following: that the session is using the largest possible IP packet size that can be carried without fragmentation, that the window sizes used in the transfer are adequate for the bandwidth-delay product of the network path, and that selective acknowledgment can be used for rapid recovery from line-error conditions or from short periods of marginally degraded network performance.

TCP Operation

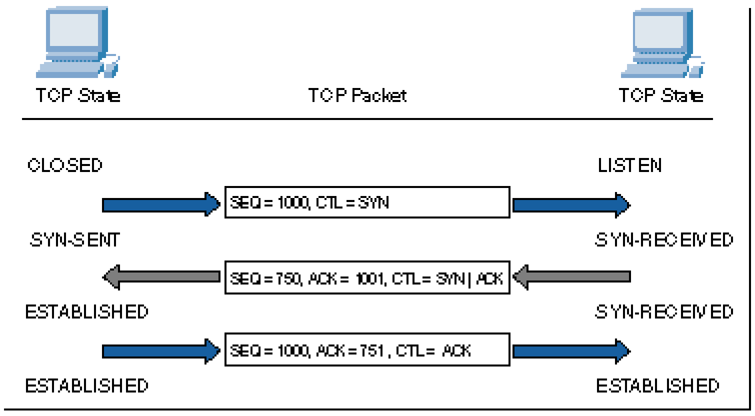
The first phase of a TCP session is establishment of the connection. This requires a three-way handshake, ensuring that both sides of the connection have an unambiguous understanding of the sequence number space of the remote side for this session. The operation of the connection is as follows:

- The local system sends the remote end an initial sequence number to the remote port, using a SYN packet.
- The remote system responds with an ACK of the initial sequence number and the initial sequence number of the remote end in a response SYN packet.
- The local end responds with an ACK of this remote sequence number.

The connection is opened.

The operation of this algorithm is shown in Figure 2. The performance implication of this protocol exchange is that it takes one and a half *round-trip times* (RTTs) for the two systems to synchronize state before any data can be sent.

Figure 2 : TCP Connection Handshake



After the connection has been established, the TCP protocol manages the reliable exchange of data between the two systems. The algorithms that determine the various retransmission timers have been redefined numerous times. TCP is a sliding-window protocol, and the general principle of flow control is based on the management of the advertised window size and the management of retransmission timeouts, attempting to optimize protocol performance within the observed delay and loss parameters of the connection. Tuning a TCP protocol stack for optimal performance over a very low-delay, high-bandwidth LAN requires different settings to obtain optimal performance over a dialup Internet connection, which in turn is different for the requirements of a high-speed wide-area network. Although TCP attempts to discover the delay bandwidth product of the connection, and attempts to automatically optimize its flow rates within the estimated parameters of the network path, some estimates will not be accurate, and the corresponding efforts by TCP to optimize behavior may not be completely successful.

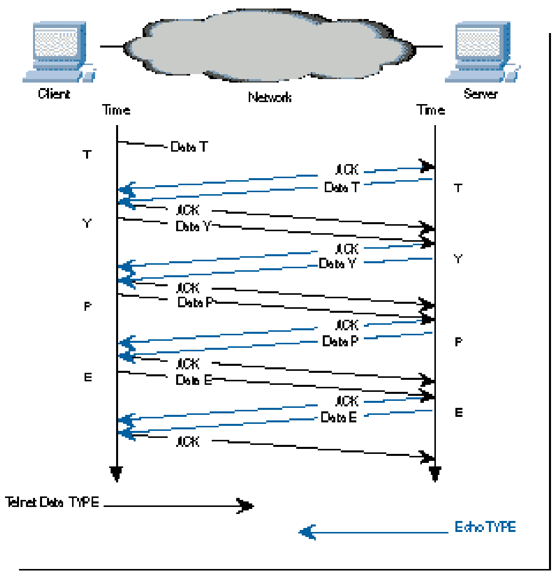
Another critical aspect is that TCP is an adaptive flow-control protocol. TCP uses a basic flow-control algorithm of increasing the data-flow rate until the network signals that some form of saturation level has been reached (normally indicated by data loss). When the sender receives an indication of data loss, the TCP flow rate is reduced; when reliable transmission is reestablished, the flow rate slowly increases again.If no reliable flow is reestablished, the flow rate backs further off to an initial probe of a single packet, and the entire adaptive flow-control process starts again.

This process has numerous results relevant to service quality. First, TCP behaves *adaptively* , rather than *predictively* . The flow-control algorithms are intended to increase the data-flow rate to fill all available network path capacity, but they are also intended to quickly back off if the available capacity changes because of interaction with other traffic, or if a dynamic change occurs in the end-to-end network path. For example, a single TCP flow across an otherwise idle network attempts to fill the network path with data, optimizing the flow rate within the available network capacity. If a second TCP flow opens up across the same path, the two flow-control algorithms will interact so that both flows will stabilize to use approximately half of the available capacity per flow. The objective of the TCP algorithms is to adapt so that the network is fully used whenever one or more data flows are present. In design, tension always exists between the efficiency of network use and the enforcement of predictable session performance. With TCP, you give up predictable throughput but gain a highly utilized, efficient network.

Interactive TCP

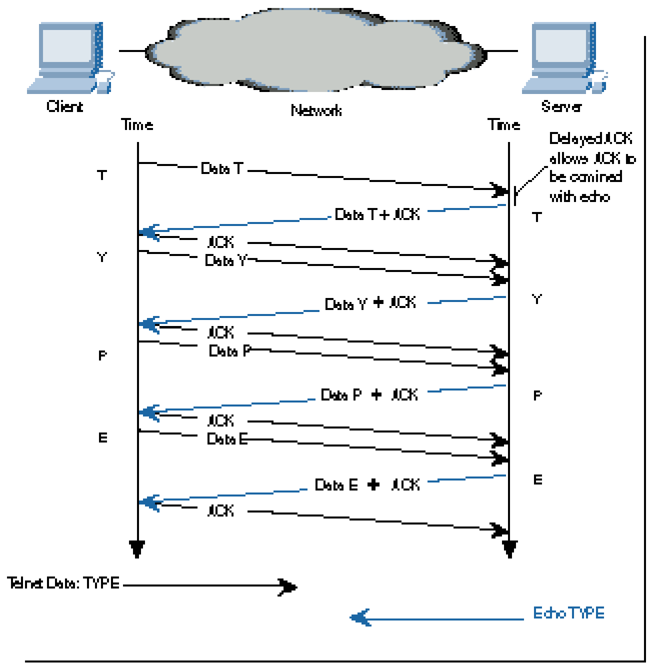
Interactive protocols are typically directed at supporting single character interactions, where each character is carried in a single packet, as is its echo. The protocol interaction to support this is indicated in Figure 3.

Figure 3: Interactive Exchange



These 2 bytes of data generate four TCP/IP packets, or 160 bytes of protocol overhead. TCP makes some small improvement in this exchange through the use of *piggybacking* , where an ACK is carried in the same packet as the data, and *delayed acknowledgment* , where an ACK is delayed up to 200 ms before sending, to give the server application the opportunity to generate data that the ACK can piggyback. The resultant protocol exchange is indicated in Figure 4.

Figure 4: Intereactive Exchange with Delayed ACK



For short-delay LANs, this protocol exchange offers acceptable performance. This protocol exchange for a single data character and its echo occurs within about 16 ms on an Ethernet LAN, corresponding to an interactive rate of 60 characters per second. When the network delay is increased in a WAN, these small packets can be a source of congestion load. The TCP mechanism to address this small-packet congestion was described by John Nagle in RFC 896 [5]. Commonly referred to as the *Nagle Algorithm* , this mechanism inhibits a sender from transmitting any additional small segments while the TCP connection has outstanding unacknowledged small segments. On a LAN, this modification to the algorithm has a negligible effect; in contrast, on a WAN, it has a dramatic effect in reducing the number of small packets in direct correlation to the network path congestion level (as shown in Figures 5 and 6). The cost is an increase in session jitter by up to a round-trip time interval. Applications that are jitter-sensitive typically disable this control algorithm.

Figure 5: Wan Interactive Exchange

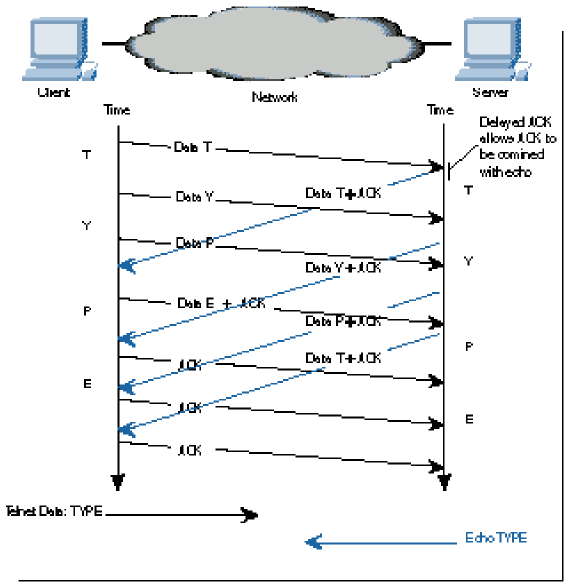
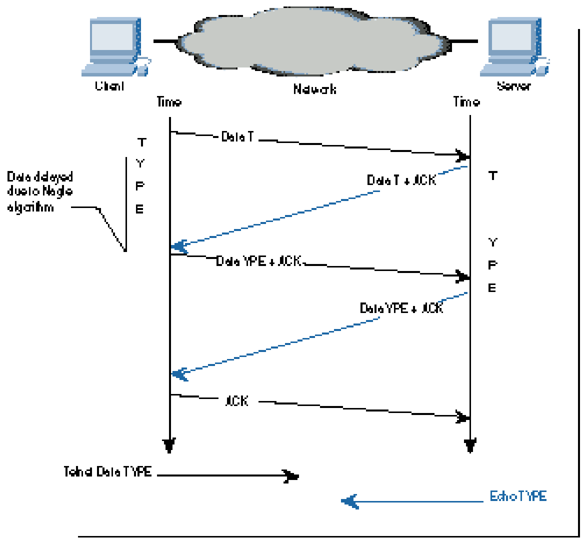


Figure 6: Wan Interactive Exchange with Nagle Algorithm



TCP is not a highly efficient protocol for the transmission of interactive traffic. The typical carriage efficiency of the protocol across a LAN is 2 bytes of payload and 120 bytes of protocol overhead. Across a WAN, the Nagle algorithm may improve this carriage efficiency slightly by increasing the number of bytes of payload for each payload transaction, although it will do so at the expense of increased session jitter.

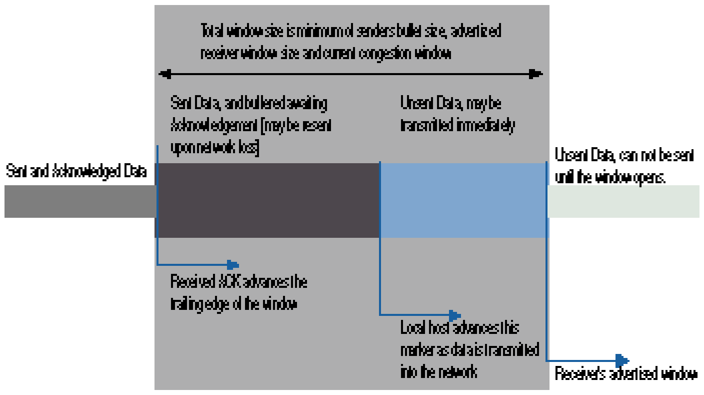
TCP Volume Transfer

The objective for this application is to maximize the efficiency of the data transfer, implying that TCP should endeavor to locate the point of dynamic equilibrium of maximum network efficiency, where the sending data rate is maximized just prior to the onset of sustained packet loss.

Further increasing the sending rate from such a point will run the risk of generating a congestion condition within the network, with rapidly increasing packet-loss levels. This, in turn, will force the TCP protocol to retransmit the lost data, resulting in reduced data-transfer efficiency. On the other hand, attempting to completely eliminate packet-loss rates implies that the sender must reduce the sending rate of data into the network so as not to create transient congestion conditions along the path to the receiver. Such an action will, in all probability, leave the network with idle capacity, resulting in inefficient use of available network resources.

The notion of a point of equilibrium is an important one. The objective of TCP is to coordinate the actions of the sender, the network, and the receiver so that the network path has sufficient data such that the network is not idle, but it is not so overloaded that a congestion backlog builds up and data loss occurs. Maintaining this point of equilibrium requires the sender and receiver to be synchronized so that the sender passes a packet into the network at precisely the same time as the receiver removes a packet from the network. If the sender attempts to exceed this equilibrium rate, network congestion will occur. If the sender attempts to reduce its rate, the efficiency of the network will drop. TCP uses a sliding-window protocol to support bulk data transfer (Figure 7).

Figure 7: TCP Sliding Window


Total window size is minimum of senders buffer size, advertized receiver window size and current congestion window
Sent Data, and buffered awaiting Acknowledgement [may be resent upon network loss]
Unsent Data, may be transmitted immediately
Sent and Acknowledged Data
Unsent Data, can not be sent until the window opens.
Received ACK advances the trailing edge of the window
Local host advances this marker as data is transmitted into the network
Receiver's advertized window

The receiver advertises to the sender the available buffer space at the receiver. The sender can transmit up to this amount of data before having to await a further buffer update from the receiver. The sender should have no more than this amount of data in transit in the network. The sender must also buffer sent data until it has been ACKed by the receiver. The send window is the minimum of the sender's buffer size and the advertised receiver window. Each time an ACK is received, the trailing edge of the send window is advanced. The minimum of the sender's buffer and the advertised receiver's window is used to calculate a new leading edge. If this send window encompasses unsent data, this data can be sent immediately.

The size of TCP buffers in each host is a critical limitation to performance in WANs. The protocol is capable of transferring one send window of data per round-trip interval. For example, with a send window of 4096 bytes and a transmission path with an RTT of 600 ms, a TCP session is capable of sustaining a maximum transfer rate of 48 Kbps, regardless of the bandwidth of the network path. Maximum efficiency of the transfer is obtained only if the sender is capable of completely filling the network path with data. Because the sender will have an amount of data in forward transit and an equivalent amount of data awaiting reception of an ACK signal, both the sender's buffer and the receiver's advertised window should be no smaller than the Delay-Bandwidth Product of the network path.

The 16-bit field within the TCP header can contain values up to 65,535, imposing an upper limit on the available window size of 65,535 bytes. This imposes an upper limit on TCP performance of some 64 KB per RTT, even when both end systems have arbitrarily large send and receive buffers. This limit can be modified by the use of a window-scale option, described in RFC 1323, effectively increasing the size of the window to a 30-bit field, but transmitting only the most significant 16 bits of the value. This allows the sender and receiver to use buffer sizes that can operate efficiently at speeds that encompass most of the current very-high-speed network transmission technologies across distances of the scale of the terrestrial intercontinental cable systems.

Although the maximum window size and the RTT together determine the maximum achievable data-transfer rate, there is an additional element of flow control required for TCP. If a TCP session commenced by injecting a full window of data into the network, then there is a strong probability that much of the initial burst of data would be lost because of transient congestion, particularly if a large window is being used. Instead, TCP adopts a more conservative approach by starting with a modest amount of data that has a high probability of successful transmission, and then probing the network with increasing amounts of data for as long as the network does not show signs of congestion. When congestion is experienced, the sending rate is dropped and the probing for additional capacity is resumed.

The dynamic operation of the window is a critical component of TCP performance for volume transfer. The mechanics of the protocol involve an additional overriding modifier of the sender's window, the *congestion window* , referred to as *cwnd* . The objective of the window-management algorithm is to start transmitting at a rate that has a very low probability of packet loss, then to increase the rate (by increasing the *cwnd* size) until the sender receives an indication, through the detection of packet loss, that the rate has exceeded the available capacity of the network. The sender then immediately halves its sending rate by reducing the value of *cwnd* , and resumes a gradual increase of the sending rate. The goal is to continually modify the sending rate such that it oscillates around the true value of available network capacity. This oscillation enables a dynamic adjustment that automatically senses any increase or decrease in available capacity through the lifetime of the data flow.

The intended outcome is that of a dynamically adjusting cooperative data flow, where a combination of such flows behaves fairly, in that each flow obtains essentially a fair share of the network, and so that close to maximal use of available network resources is made. This flow-control functionality is achieved through a combination of *cwnd* value management and packet-loss and retransmission algorithms. TCP flow control has three major parts: the flow-control modes of *Slow Start* and Congestion Avoidance, and the response to packet loss that determines how TCP switches between these two modes of operation.

TCP Slow Start

The starting value of the *cwnd* window (the Initial Window, or IW) is set to that of the Sender Maximum Segment Size (SMSS) value. This SMSS value is based on the receiver's maximum segment size, obtained during the SYN handshake, the discovered path MTU (if used), the MTU of the sending interface, or, in the absence of other information, 536 bytes. The sender then enters a flow-control mode termed *Slow Start* .

The sender sends a single data segment, and because the window is now full, it then awaits the corresponding ACK. When the ACK is received, the sender increases its window by increasing the value of *cwnd* by the value of SMSS. This then allows the sender to transmit two segments; at that point, the congestion window is again full, and the sender must await the corresponding ACKs for these segments. This algorithm continues by increasing the value of *cwnd* (and, correspondingly, opening the size of the congestion window) by one SMSS for every ACK received that acknowledges new data.

If the receiver is sending an ACK for every packet, the effect of this algorithm is that the data rate of the sender doubles every round-trip time interval. If the receiver supports delayed ACKs, the rate of increase will be slightly lower, but nevertheless the rate will increase by a minimum of one SMSS each round-trip time. Obviously, this cannot be sustained indefinitely. Either the value of *cwnd* will exceed the advertised receive window or the sender's window, or the capacity of the network will be exceeded, in which case packets will be lost.

There is another limit to the slow-start rate increase, maintained in a variable termed *ssthresh* , or *Slow-Start Threshold* . If the value of *cwnd* increases past the value of ssthresh, the TCP flow-control mode is changed from *Slow Start* to congestion avoidance. Initially the value of ssthresh is set to the receiver's maximum window size. However, when congestion is noted, ssthresh is set to half the current window size, providing TCP with a memory of the point where the onset of network congestion may be anticipated in future.

One aspect to highlight concerns the interaction of the slow-start algorithm with high-capacity long-delay networks, the so-called Long Fat Networks (or LFNs, pronounced "elephants"). The behavior of the slow-start algorithm is to send a single packet, await an ACK, then send two packets, and await the corresponding ACKs, and so on. The TCP activity on LFNs tends to cluster at each epoch of the round-trip time, with a quiet period that follows after the available window of data has been transmitted. The received ACKs arrive back at the sender with an inter-ACK spacing that is equivalent to the data rate of the bottleneck point on the network path. During *Slow Start* , the sender transmits at a rate equal to twice this bottleneck rate. The rate adaptation function that must occur within the network takes place in the router at the entrance to the bottleneck point. The sender's packets arrive at this router at twice the rate of egress from the router, and the router stores the overflow within its internal buffer. When this buffer overflows, packets will be dropped, and the slow-start phase is over. The important conclusion is that the sender will stop increasing its data rate when there is buffer exhaustion, a condition that may not be the same as reaching the true available data rate. If the router has a buffer capacity considerably less than the delay-bandwidth product of the egress circuit, the two values are certainly not the same.

In this case, the TCP slow-start algorithm will finish with a sending rate that is well below the actual available capacity. The efficient operation of TCP, particularly in LFNs, is critically reliant on adequately large buffers within the network routers.

Another aspect of *Slow Start* is the choice of a single segment as the initial sending window. Experimentation indicates that an initial value of up to four segments can allow for a more efficient session startup, particularly for those short-duration TCP sessions so prevalent with Web fetches. Observation of Web traffic indicates an average Web data transfer of 17 segments. A *slow start* from one segment will take five RTT intervals to transfer this data, while using an initial value of four will reduce the transfer time to three RTT intervals. However, four segments may be too many when using low-speed links with limited buffers, so a more robust approach is to use an initial value of no more than two segments to commence *Slow Start* .

Packet Loss

Slow Start attempts to start a TCP session at a rate the network can support and then continually increase the rate. How does TCP know when to stop this increase? This slow-start rate increase stops when the congestion window exceeds the receiver's advertised window, when the rate exceeds the remembered value of the onset of congestion as recorded in ssthresh, or when the rate is greater than the network can sustain. Addressing the last condition, how does a TCP sender know that it is sending at a rate greater than the network can sustain? The answer is that this is shown by data packets being dropped by the network. In this case, TCP has to undertake many functions:

- The packet loss has to be detected by the sender.
- The missing data has to be retransmitted.
- The sending data rate should be adjusted to reduce the probability of further packet loss.

TCP can detect packet loss in two ways. First, if a single packet is lost within a sequence of packets, the successful delivery packets following the lost packet will cause the receiver to generate a *duplicate* ACK for each successive packet The reception of these duplicate ACKs is a signal of such packet loss. Second, if a packet is lost at the end of a sequence of sent packets, there are no following packets to generate duplicate ACKs. In this case, there are no corresponding ACKs for this packet, and the sender's retransmit timer will expire and the sender will assume packet loss.

A single duplicate ACK is not a reliable signal of packet loss. When a TCP receiver gets a data packet with an out-of-order TCP sequence value, the receiver must generate an immediate ACK of the highest in-order data byte received. This will be a duplicate of an earlier transmitted ACK. Where a single packet is lost from a sequence of packets, all subsequent packets will generate a duplicate ACK packet.

On the other hand, where a packet is rerouted with an additional incremental delay, the reordering of the packet stream at the receiver's end will generate a small number of duplicate ACKs, followed by an ACK of the entire data sequence, after the errant packet is received. The sender distinguishes between these cases by using three duplicate ACK packets as a signal of packet loss.

The third duplicate ACK triggers the sender to immediately send the segment referenced by the duplicate ACK value (*fast retransmit*) and commence a sequence termed *Fast Recovery* . In fast recovery, the value of *ssthresh* is set to half the current send window size (the send window is the amount of unacknowledged data outstanding). The congestion window, *cwnd* , is set three segments greater than *ssthresh* to allow for three segments already buffered at the receiver. If this allows additional data to be sent, then this is done. Each additional duplicate ACK inflates *cwnd* by a further segment size, allowing more data to be sent. When an ACK arrives that encompasses new data, the value of*cwnd* is set back to ssthresh, and TCP enters congestion-avoidance mode. Fast Recovery is intended to rapidly repair single packet loss, allowing the sender to continue to maintain the ACK-clocked data rate for new data while the packet loss repair is being undertaken. This is because there is still a sequence of ACKs arriving at the sender, so that the network is continuing to pass timing signals to the sender indicating the rate at which packets are arriving at the receiver. Only when the repair has been completed does the sender drop its window to the *ssthresh* value as part of the transition to congestion-avoidance mode.

The other signal of packet loss is a complete cessation of any ACK packets arriving to the sender. The sender cannot wait indefinitely for a delayed ACK, but must make the assumption at some point in time that the next unacknowledged data segment must be retransmitted. This is managed by the sender maintaining a *Retransmission Timer* . The maintenance of this timer has performance and efficiency implications. If the timer triggers too early, the sender will push duplicate data into the network unnecessarily. If the timer triggers too slowly, the sender will remain idle for too long, unnecessarily slowing down the flow of data. The TCP sender uses a timer to measure the elapsed time between sending a data segment and receiving the corresponding acknowledgment. Individual measurements of this time interval will exhibit significant variance, and implementations of TCP use a smoothing function when updating the retransmission timer of the flow with each measurement. The commonly used algorithm was originally described by Van Jacobson [9], modified so that the retransmission timer is set to the smoothed round-trip-time value, plus four times a smoothed mean deviation factor .

When the retransmission timer expires, the actions are similar to that of duplicate ACK packets, in that the sender must reduce its sending rate in response to congestion. The threshold value, *ssthresh* , is set to half of the current value of outstanding unacknowledged data, as in the duplicate ACK case. However, the sender cannot make any valid assumptions about the current state of the network, given that no useful information has been provided to the sender for more than one RTT interval. In this case, the sender closes the congestion window back to one segment, and restarts the flow in *slow start* -mode by sending a single segment. The difference from the initial *slow start* is that, in this case, the *ssthresh* value is set so that the sender will probe the congestion area more slowly using a linear sending rate increase when the congestion window reaches the remembered ssthresh value.

Congestion Avoidance

Compared to *Slow Start* , congestion avoidance is a more tentative probing of the network to discover the point of threshold of packet loss. Where *Slow Start* uses an exponential increase in the sending rate to find a first-level approximation of the loss threshold, congestion avoidance uses a linear growth function.

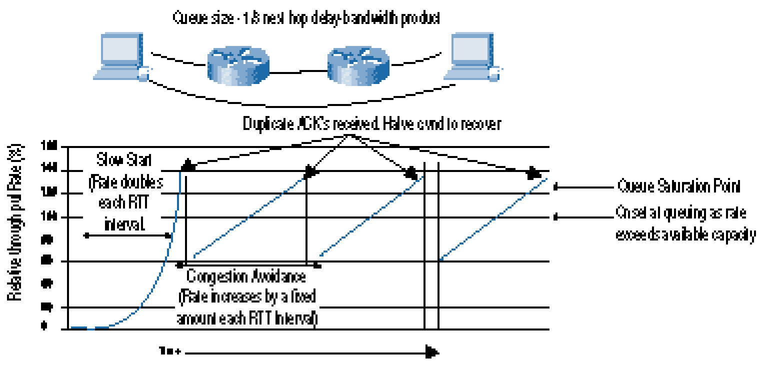
When the value of *cwnd* is greater than *ssthresh* , the sender increments the value of *cwnd* by the value *SMSS X SMSS/cwnd* , in response to each received nonduplicate ACK, ensuring that the congestion window opens by one segment within each RTT time interval.

The congestion window continues to open in this fashion until packet loss occurs. If the packet loss is isolated to a single packet within a packet sequence, the resultant duplicate

ACKs will trigger the sender to halve the sending rate and continue a linear growth of the congestion window from this new point, as described above in fast recovery.

The behavior of *cwnd* in an idealized configuration is shown in Figure 8,

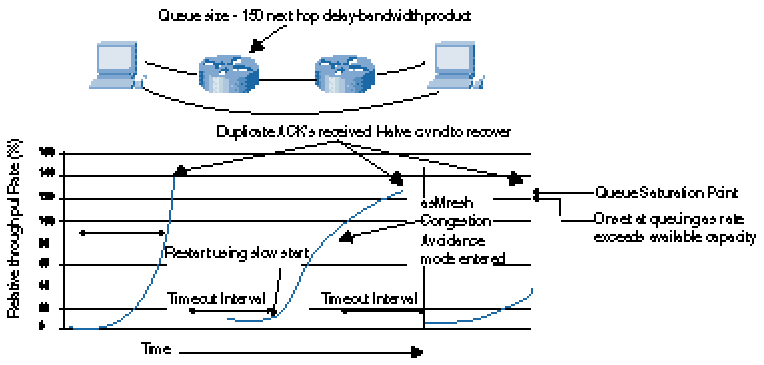
Figure 8: Simulation of Single TCP Transfer


Queue size - 1.8 next hop delay-bandwidth product
Duplicate ACK's received. Halve cwnd to recover
Slow Start (Rate doubles each RTT interval
Congestion Avoidance (Rate increases by a fixed amount each RTT Interval)
Queue Saturation Point
Onset of queuing as rate exceeds available capacity
Relative throughput Rate (%)
Time

along with the corresponding data-flow rates. The overall characteristics of the TCP algorithm are an initial relatively fast scan of the network capacity to establish the approximate bounds of maximal efficiency, followed by a cyclic mode of adaptive behavior that reacts quickly to congestion, and then slowly increases the sending rate across the area of maximal transfer efficiency.

Packet loss, as signaled by the triggering of the retransmission timer, causes the sender to recommence slow-start mode, following a timeout interval. The corresponding data-flow rates are indicated in Figure 9.

Figure 9: Simulation of TCP Transfer with Tail Drop Queue


Queue size - 150 next hop delay-bandwidth product
Duplicate ACK's received. Halve cwnd to recover
ssthresh
Congestion Avoidance mode entered
Restart using slow start
Timeout Interval
Timeout Interval
Queue Saturation Point
Onset of queuing as rate exceeds available capacity
Relative throughput Rate (%)
Time

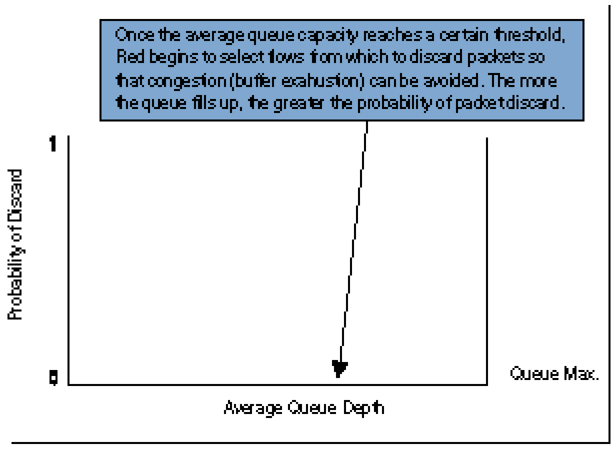
The inefficiency of this mode of performance is caused by the complete cessation of any form of flow signaling from the receiver to the sender. In the absence of any information, the sender can only assume that the network is heavily congested, and so must restart its probing of the network capacity with an initial congestion window of a single segment. This leads to the performance observation that any form of packet-drop management that tends to discard the trailing end of a sequence of data packets may cause significant TCP performance degradation, because such drop behavior forces the TCP session to continually time out and restart the flow from a single segment again.

Assisting TCP Performance Network-RED and ECN

Although TCP is an end-to-end protocol, it is possible for the network to assist TCP in optimizing performance. One approach is to alter the queue behaviour of the network through the use of *Random Early Detection* (RED). RED permits a network router to discard a packet even when there is additional space in the queue. Although this may sound inefficient, the interaction between this early packet-drop behaviour and TCP is very effective.

RED uses a the weighted average queue length as the probability factor for packet drop. As the average queue length increases, the probability of a packet being dropped, rather than being queued, increases. As the queue length decreases, so does the packet-drop probability. (See Figure 10). Small packet bursts can pass through a RED filter relatively intact, while larger packet bursts will experience increasingly higher packet-discard rates. Sustained load will further increase the packet-discard rates. This implies that the TCP sessions with the largest open windows will have a higher probability of experiencing packet drop, causing a back-off in the window size.

Figure 10: Red Behavior

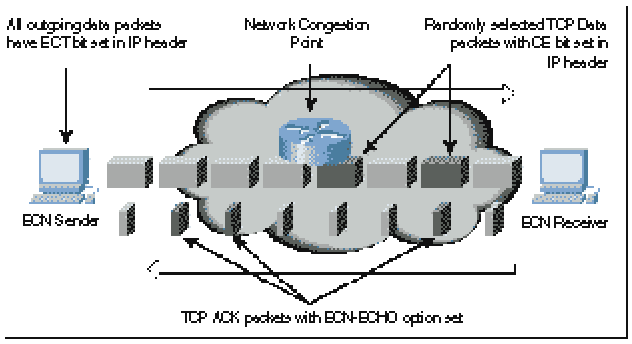


A major goal of RED is to avoid a situation in which all TCP flows experience congestion at the same time, all then back off and resume at the same rate, and tend to synchronize their behaviour . With RED, the larger bursting flows experience a higher probability of packet drop, while flows with smaller burst rates can continue without undue impact. RED is also intended to reduce the incidence of complete loss of ACK signals, leading to timeout and session restart in slow-start mode. The intent is to signal the heaviest bursting TCP sessions the likelihood of pending queue saturation and tail drop before the onset of such a tail-drop congestion condition, allowing the TCP session to undertake a fast retransmit recovery under conditions of congestion avoidance. Another objective of RED is to allow the queue to operate efficiently, with the queue depth ranging across the entire queue size within a timescale of queue depth oscillation the same order as the average RTT of the traffic flows.

Behind RED is the observation that TCP sets very few assumptions about the networks over which it must operate, and that it cannot count on any consistent performance feedback signal being generated by the network. As a minimal approach, TCP uses packet loss as its performance signal, interpreting small-scale packet-loss events as peak load congestion events and extended packet loss events as a sign of more critical congestion load. RED attempts to increase the number of small-scale congestion signals, and in so doing avoid long-period sustained congestion conditions.

It is not necessary for RED to discard the randomly selected packet. The intent of RED is to signal the sender that there is the potential for queue exhaustion, and that the sender should adapt to this condition. An alternative mechanism is for the router experiencing the load to mark packets with an explicit *Congestion Experienced* (CE) bit flag, on the assumption that the sender will see and react to this flag setting in a manner comparable to its response to single packet drop. This mechanism, *Explicit Congestion Notification* (ECN), uses a 2-bit scheme, claiming bits 6 and 7 of the IP Version 4 Type-of-Service (ToS) field (or the two Currently Unused [CU] bits of the IP *Differentiated Services* field). Bit 6 is set by the sender to indicate that it is an ECN-capable transport system (the ECT bit). Bit 7 is the CE bit, and is set by a router when the average queue length exceeds configured threshold levels. The ECN algorithm is that an active router will perform RED, as described. After a packet has been selected, the router may mark the CE bit of the packet if the ECT bit is set; otherwise, it will discard the selected packet. (See Figure 11).

Figure 11: Operation of Explicit Congestion Notification


All outgoing data packets have ECT bit set in IP header
Network Congestion Point
Randomly selected TCP Data packets with CE bit set in IP header
ECN Sender
ECN Receiver
TCP ACK packets with ECN-ECHO option set

The TCP interaction is slightly more involved. The initial TCP SYN handshake includes the addition of ECN-echo capability and *Congestion Window Reduced* (CWR) capability flags to allow each system to negotiate with its peer as to whether it will properly handle packets with the CE bit set during the data transfer. The sender sets the ECT bit in all packets sent. If the sender receives a TCP packet with the ECN-echo flag set in the TCP header, the sender will adjust its congestion window as if it had undergone fast recovery from a single lost packet.

The next sent packet will set the TCP CWR flag, to indicate to the receiver that it has reacted to the congestion. The additional caveat is that the sender will react in this way at most once every RTT interval. Further, TCP packets with the ECN-echo flag set will have no further effect on the sender within the same RTT interval. The receiver will set the ECN-echo flag in all packets when it receives a packet with the CE bit set. This will continue until it receives a packet with the CWR bit set, indicating that the sender has reacted to the congestion. The ECT flag is set only in packets that contain a data payload. TCP ACK packets that contain no data payload should be sent with the ECT bit clear.

The connection does not have to await the reception of three duplicate ACKs to detect the congestion condition. Instead, the receiver is notified of the incipient congestion condition through the explicit setting of a notification bit, which is in turn echoed back to the sender in the corresponding ACK. Simulations of ECN using a RED marking function indicate slightly superior throughput in comparison to configuring RED as a packet-discard function.

However, widespread deployment of ECN is not considered likely in the near future, at least in the context of Version 4 of IP. At this stage, there has been no explicit standardization of the field within the IPv4 header to carry this information, and the deployment base of IP is now so wide that any modifications to the semantics of fields in the IPv4 header would need to be very carefully considered to ensure that the changed field interpretation did not exercise some malformed behavior in older versions of the TCP stack or in older router software implementations.

ECN provides some level of performance improvement over a packet-drop RED scheme. With large bulk data transfers, the improvement is moderate, based on the difference between the packet retransmission and congestion-window adjustment of RED and the congestion-window adjustment of ECN. The most notable improvements indicated in ECN simulation experiments occur with short TCP transactions (commonly seen in Web transactions), where a RED packet drop of the initial data packet may cause a six-second retransmit delay. Comparatively, the ECN approach allows the transfer to proceed without this lengthy delay.

The major issue with ECN is the need to change the operation of both the routers and the TCP software stacks to accommodate the operation of ECN. While the ECN proposal is carefully constructed to allow an essentially uncoordinated introduction into the Internet without negative side effects, the effectiveness of ECN in improving overall network throughput will be apparent only after this approach has been widely adopted. As the Internet grows, its inertial mass generates a natural resistance to further technological change; therefore, it may be some years before ECN is widely adopted in both host software and Internet routing systems. RED, on the other hand, has had a more rapid introduction to the Internet, because it requires only a local modification to router behavior, and relies on existing TCP behavior to react to the packet drop.

Tuning TCP

How can the host optimize its TCP stack for optimum performance? Many recommendations can be considered. The following suggestions are a combination of those measures that have been well studied and are known to improve TCP performance, and those that appear to be highly productive areas of further research and investigation .

- *Use a good TCP protocol stack* : Many of the performance pathologies that exist in the network today are not necessarily the byproduct of oversubscribed networks and consequent congestion. Many of these performance pathologies exist because of poor implementations of TCP flow-control algorithms; inadequate buffers within the receiver; poor (or no) use of path-MTU discovery; no support for fast-retransmit flow recovery, no use of window scaling and SACK, imprecise use of protocol-required timers, and very coarse-grained timers. It is unclear whether network ingress-imposed Quality-of-Service (QoS) structures will adequately compensate for such implementation deficiencies. The conclusion is that attempting to address the symptoms is not the same as curing the disease. A good protocol stack can produce even better results in the right environment.

- *Implement a TCP Selective Acknowledgment (SACK) mechanism* : SACK, combined with a selective repeat-transmission policy, can help overcome the limitation that traditional TCP experiences when a sender can learn only about a single lost packet per RTT.
- *Implement larger buffers with TCP window-scaling options* : The TCP flow algorithm attempts to work at a data rate that is the minimum of the delay-bandwidth product of the end-to-end network path and the available buffer space of the sender. Larger buffers at the sender and the receiver assist the sender in adapting more efficiently to a wider diversity of network paths by permitting a larger volume of traffic to be placed in flight across the end-to-end path.
- *Support TCP ECN negotiation* : ECN enables the host to be explicitly informed of conditions relating to the onset of congestion without having to infer such a condition from the reserve stream of ACK packets from the receiver. The host can react to such a condition promptly and effectively with a data flow-control response without having to invoke packet retransmission.
- *Use a higher initial TCP slow-start rate than the current 1 MSS (Maximum Segment Size) per RTT* . A size that seems feasible is an initial burst of 2 MSS segments. The assumption is that there will be adequate queuing capability to manage this initial packet burst; the provision to back off the send window to 1 MSS segment should remain intact to allow stable operation if the initial choice was too large for the path. A robust initial choice is two segments, although simulations have indicated that four initial segments is also highly effective in many situations.
- *Use a host platform that has sufficient processor and memory capacity to drive the network* . The highest-quality service network and optimally provisioned access

circuits cannot compensate for a host system that does not have sufficient capacity to drive the service load. This is a condition that can be observed in large or very popular public Web servers, where the peak application load on the server drives the platform into a state of memory and processor exhaustion, even though the network itself has adequate resources to manage the traffic load.

All these actions have one thing in common: They can be deployed incrementally at the edge of the network and can be deployed individually. This allows end systems to obtain superior performance even in the absence of the network provider tuning the network's service response with various internal QoS mechanisms.

Conclusion

TCP is not a predictive protocol. It is an adaptive protocol that attempts to operate the network at the point of greatest efficiency. Tuning TCP is not a case of making TCP pass more packets into the network. Tuning TCP involves recognizing how TCP senses current network load conditions, working through the inevitable compromise between making TCP highly sensitive to transient network conditions, and making TCP resilient to what can be regarded as noise signals.

If the performance of end-to-end TCP is the perceived problem, the most effective answer is not necessarily to add QoS service differentiation into the network. Often, the greatest performance improvement can be made by upgrading the way that hosts and the network interact through the appropriate configuration of the host TCP stacks.