V 2.1

Lab Practices on Computer Networks and Distributed Systems

Stream Sockets and the Client/Server model (C language, WIP)

This practice illustrates basic concepts about the TCP protocol by creating a simple C/S pair of applications and by observing the C/S protocol with a packet sniffer. It also introduces the Client/Server computing model and the relevant APIs.

A brief recollection of TCP/IP and Berkeley sockets

The TCP protocol module in an operating system of today offers some essential services to user programs such as reliable transmission, flow and congestion control. This variety of objectives makes TCP complex, thus, in order to isolate the application programmer from the complexity of TCP, operating systems offer the Sockets service interface; programming languages incorporate APIs that allow programs to call TCP services for communicating over the Internet. The two languages used in this course, C and Java, offer the Sockets API. In this practice we write a simple C/S pair in C. In a forthcoming practice, we will use Java for Sockets-based C/S programming.

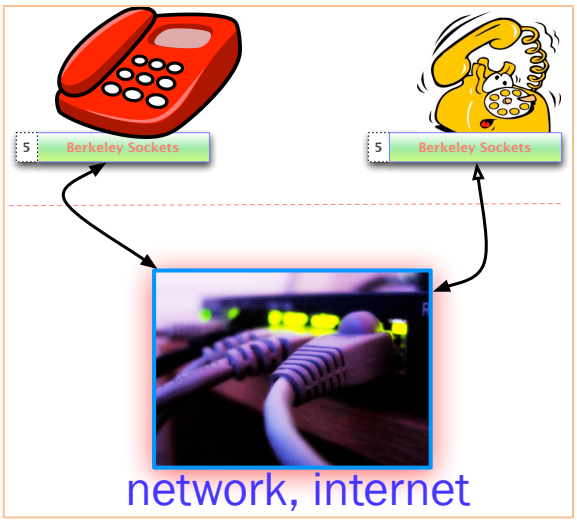

Fig. 1. TCP Connection analogy to telephone dialing

TCP is a connection-oriented protocol, that means that every time we want to transmit information reliably by using it, the protocol module will have to establish a connection with the remote computer, the result of this connection is that the communicating computers share state, a series of data structures that permit the execution of the sliding window, flow control and congestion control algorithms of TCP.

The Berkeley Socket API, originally programmed in C has been exported to other languages where the similarities are clear, this was so that interoperability could be guaranteed, for example, when a sockets-based server program written in Java communicates with a client written in Python, we want both programs to interoperate correctly at the sockets level. The programmer's task in this case consists of implementing the

application-level protocol correctly by sending the relevant protocol messages over the client and server sockets. The whole process, from connection setup through connection teardown is illustrated in Fig. 2.

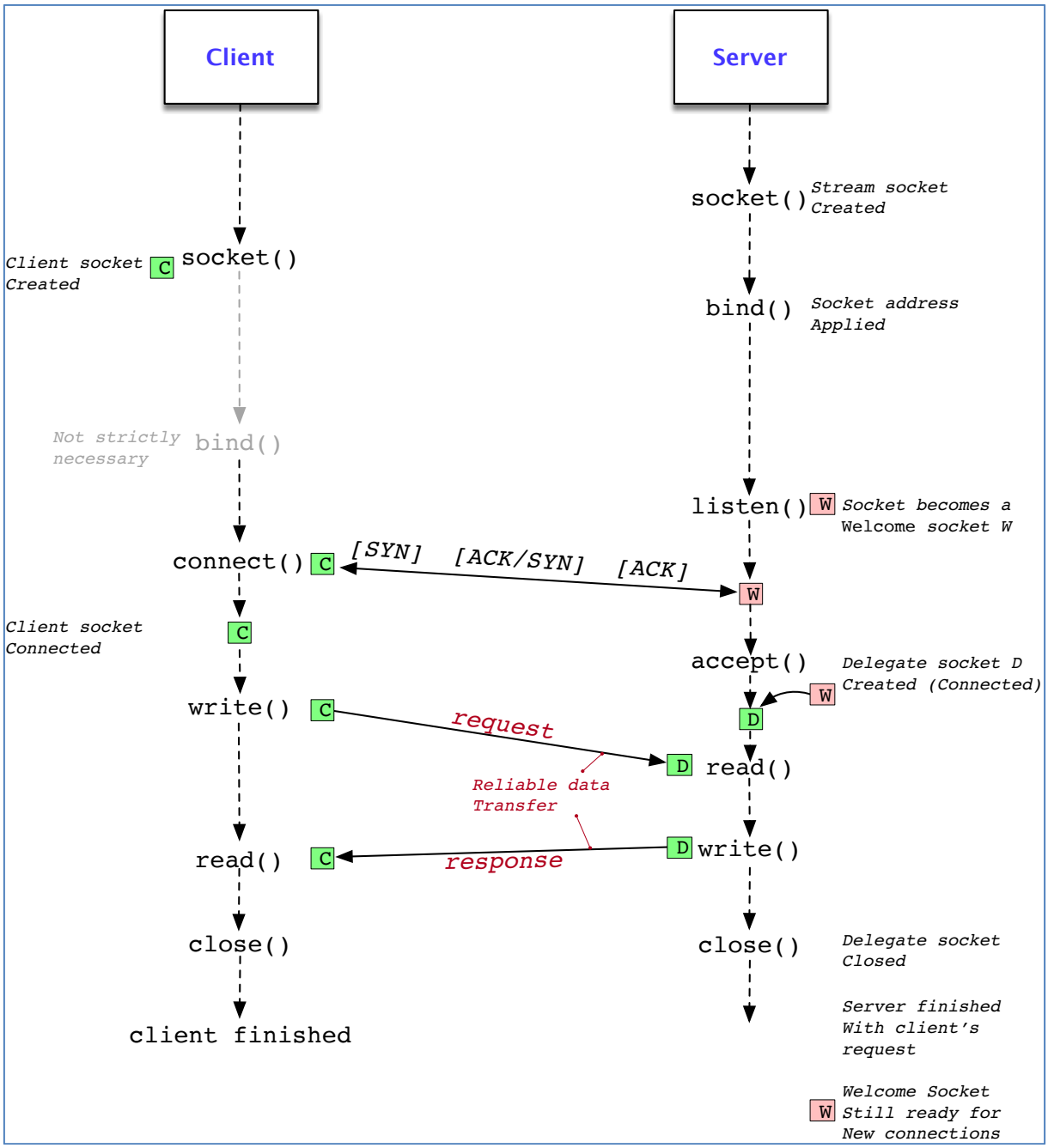

Fig. 2. Berkeley socket connection, data transfer and connection teardown phases

Exercise 0: Check a simple TCP connection created with the nc (netcat) command on both ends of the connection

1. Open a remote session to your paloalto.unileon.es/50500 account. We'll refer this session as session A.

```
$ ssh -p 50500 <'login name'>@paloalto.unileon.es
```

2. Check out the man page to the nc command to make sure that it fits our aim to create a C/S connection.

3. Open a two new sessions in paloalto for observing the C/S protocol with tcpdump and the stack state with netstat, if necessary. We'll refer to these ancillary sessions as B and C, respectively.

4. Check TCP port availability at paloalto by using session C; execute the following netstat command, and check man netstat if you need more detail about the used options:

```
$ netstat –a –tcp –n –c
```

5. The TCP ports that are currently open at paloalto for access from the Internet are these: 60001..60008. Before executing the nc command, you should choose a TCP port that does not appear in the preceding stack state listing. For example, assuming that port 60001 is not in use, create a simple server that listens on that port. Do this at session A. If necessary, check the man page to the nc command:

```
$ nc –l –p 60001
```

6. After you press return to execute the preceding command, check that a new TCP connection to paloalto.unileon.es port 60001 now does appear in the stack listing above. If the repetitive netstat display (option -c) is not of help now, get rid of that option, in which case, you will have to repeat the command whenever you wish to update the listing.

   The Lab B6 internal IP address to paloalto.unileon.es/50500 is 192.168.1.88. This IP address is translated to paloalto's public address (193.146.101.46) whenever packets are sent from 192.168.1.88 onto the Internet. Lab B6 router does that translation which is known as NAPT (Network and Port Translation). NAPT is a valuable solution to the problem of public IP-address scarcity.

7. On the client side, at your home or at some of the University libraries or hearing classrooms, run the nc (netcat) command to create a TCP connection with paloalto.unileon.es/60001:

```
$ nc paloalto.unileon.es 60001
```

8. Compose a few messages at the client and have them sent by pressing return, then, observe that the server terminal displays the message that you sent. At the same time, observe that the TCP socket at port 60001 at the server has changed from the LISTEN state to the ESTABLISHED state:

```
$ nc paloalto.unileon.es 60001
```
   Hello! *(Sent C -> S)*
   Bye *(Sent C -> S)*
   <Compose the **ctrl-D** combination to have the client program close the TCP connection to the server>

9. Repeat the same steps above and monitor the C/S traffic now by running a tcpdump sniffer. Run a packet sniffer and have it capture traffic on your active network interface (Execute the $ ifconfig command and choose an interface that has Internet access in your own Linux system). Assuming that the interface your system is using is enp1s0, compose the following command:

```
$ tcpdump –i enp1s0 –X –vvv tcp port 60001
```

10. Summary about the lifecycle of a TCP connection. A typical TCP connection goes through the following three phases:

    a. **Connection establishment**
        i. Three-way handshake: SYN; SYN/ACK; ACK
        ii. Announcing connection options
        iii. Setting initial Sequence Numbers

    b. **Data transfer**
        i. Send simple, short character strings and observe their Sequence Numbers and the corresponding Acknowledgement Numbers received over the other channel
        ii. TCP segments may carry data which is indexed by the contents of the SN field; the contents of the ACK SN field represents

    c. **Connection close**
        i. Observe which of the two connection ends initiates connection close. In our case, if you've followed the practice script above, it's the client the side that sends the connection close TCP segment.

11. Document all the relevant results and have them included in your LabBooks.

Exercise 1. The following text is an excerpt from the Linux manual page of function listen(); it explains the purpose of its second formal parameter: the length of the pending connection backlog. When the server receives a burst of connection requests, it services each one in turn, consequently, while one connection is being established, the others must wait in the backlog. If the burst of simultaneous connection requests is larger than the backlog size, the connections that cannot be completed will undergo a delay in receiving their ACK-SYN response. The client TCP stack will retry the connection establishment by sending a new SYN packet to the server. If this situation persists, the client will repeat sending the SYN packet a few times and separating them with exponentially increasing times. In this exercise we wish to establish this behavior experimentally in different ways:

a. Start your tcpdump protocol analyzer and observe the separation in time of each two successive connection requests (SYN segments sent by the client). You can use either the nc command or the client program included in this practice (http://paloalto.unileon.es/ds/lab/TCP/TCP-Client-Base.c). In either case, the IP address that your client program should attempt to connect with **should point to an Internet host that is unconnected**, in fact; otherwise, if the target host is up and is connected to the network, then it will send back RST segments to our connecting client which will ultimately prevent it from retrying sending anymore SYN segments. Neither should you attempt connecting your client to any host within your local network, either connected or unconnected. Now, do the experiment and document your results including the name of the operating system you're using (Below you can see the results obtained in a Linux system running the client):

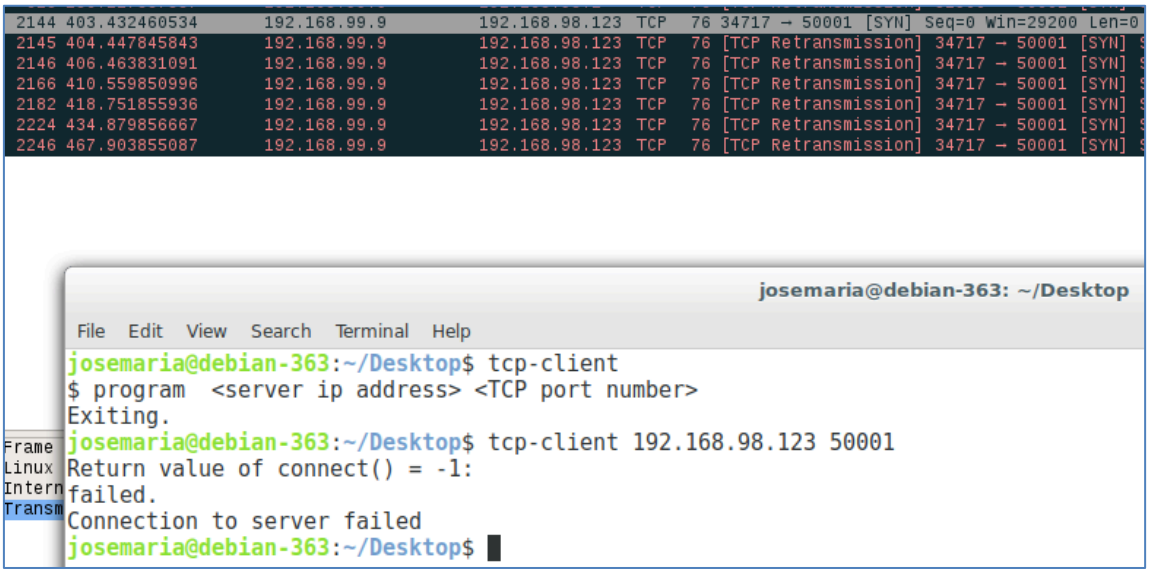

Fig. 3. A Linux TCP client sending SYN segments with an exponentially distributed time separation

b. Can you explain why we are providing the advice at the end of the previous paragraph?

c. Modify the client program so that it prints the return value of the connect() call to see whether it reports an error or it does retransmissions (Carefully read the man page excerpt below).

```
The backlog argument defines the maximum length to which the queue of
pending connections for sockfd may grow. If a connection request
arrives when the queue is full, the client may receive an error with an
indication of ECONNREFUSED or, if the underlying protocol supports
retransmission, the request may be ignored so that a later reattempt at
connection succeeds.
```

d. In this section, we will modify the Server program (http://paloalto.unileon.es/ds/Lab/TCP/TCP-Server-Base.c). Other ways of having the server not respond to the client's SYN could be achieved by having it not call listen() or call listen and not call accept() and others. Test these different ways of having the server not respond to the SYN sent by the client, to that end, make the relevant modifications to the server program and then test each one in turn. Explain the results that you obtained in this exercise compared to the results obtained in the previous one.

The TCP connection protocol, known as *three-way handshake*, is initiated when the client program creates a client socket and connects it with a listening server socket that is accepting connections. The three-way handshake implies the exchange of three messages between client and server and is normally started by the client in what is known as an *active open* operation. When the 3-way handshake has finished, the server socket and the client socket can begin to exchange C/S protocol messages reliably.

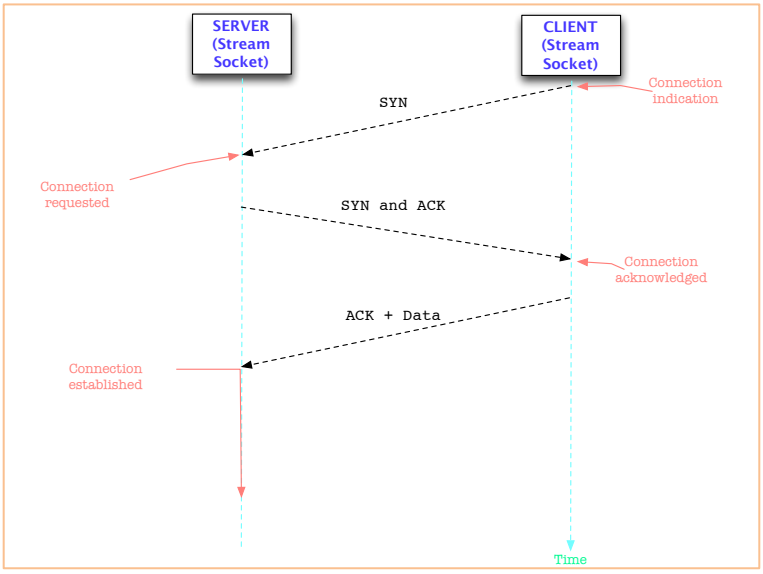

Fig. 4. Three-way handshake started by the client (Active open)

The welcome socket, created by the server program waits for connection requests coming from clients right after the server has called listen() on the welcome socket. When a new connection request is received (SYN segment), it is stored in a pending connection queue. The length of this queue is set with second parameter to the listen() call. The semantics of the backlog queue is highly dependent on the specific operating system type and version. From Linux kernel version 2.2 and on, **the backlog contains established connections**, *i.e.,* those connections that successfully finished the 3-way handshake (Before kernel 2.2, the backlog contained pending connections and established connections). Though those connections are in the established state, the sockets API will not make them **available for bidirectional data transfers until the server calls accept()** and this call returns the delegate socket corresponding to the connection request laying on the queue head. If the backlog queue fills up, due to a burst of connection requests being received, the ensuing connection requests will not be completed by the server IP stack, thereby causing the clients connect() calls timeout to elapse which will consequently make it retry the connection request. This is in direct relation to our consideration about the backlog in Exercise no. 1, above.

The central document specifying TCP is RFC 793. This document explains in detail the state transitions undergone by each stream socket involved in a TCP connection. You can find the state diagram in fig. 3.1 within this document where you'll readily identify the closed state of a socket, the connected state where reliable data transfer occurs, etc. Grasping the TCP state diagram might seem challenging since it has so many states and many more state transitions, however, that might not be so by carefully studying the significance of its elements and its dynamics.

States are represented by boxes and state transitions are represented by arrows. Each state transition is caused by the relevant socket receiving some specific protocol message, which causes the socket to send back some response message and by ultimately changing its state. For example, when the Welcome Socket, which must be in the Listen state, receives a legitimate TCP segment with the SYN flag activated, TCP changes its state from LISTEN to SYN RCVD and sends back a TCP segment having flags "SYN, ACK" set.

As you see, the state changes represented in the TCP state diagram affect all the different sockets involved in the process of connection setup, data transfer and connection teardown, all of which constitute the connection lifecycle.

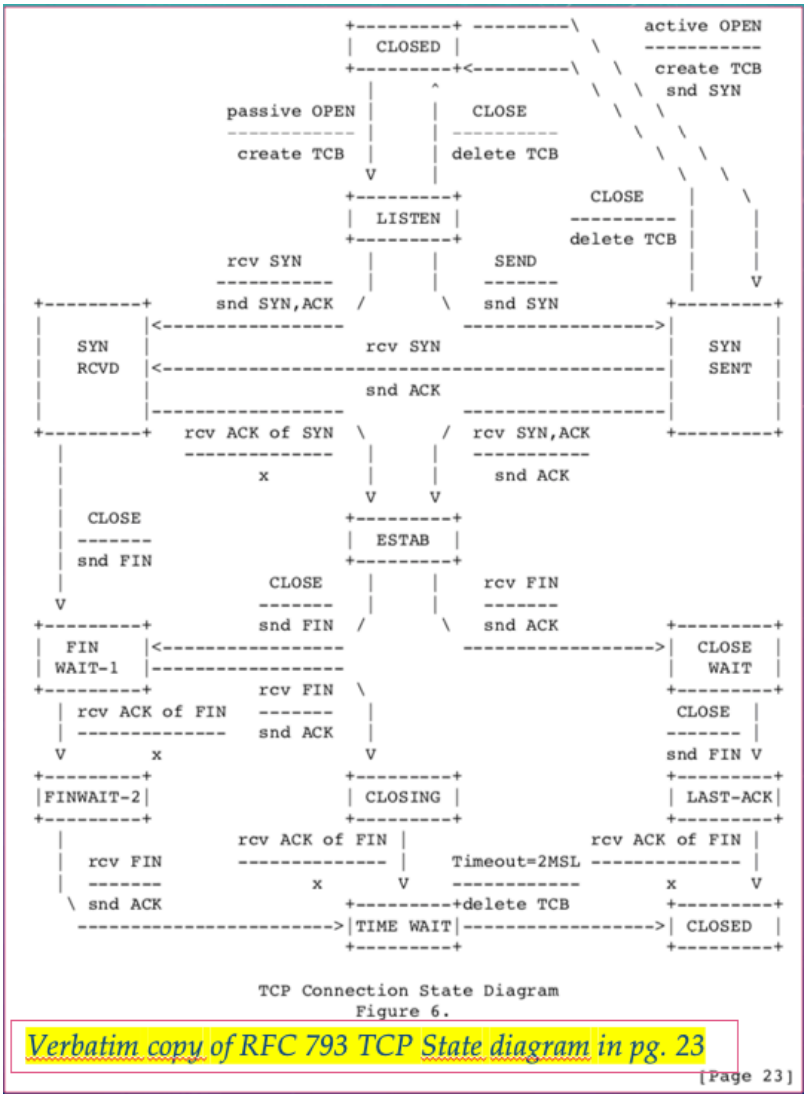

Fig. 3.1 TCP state diagram taken from RFC 793

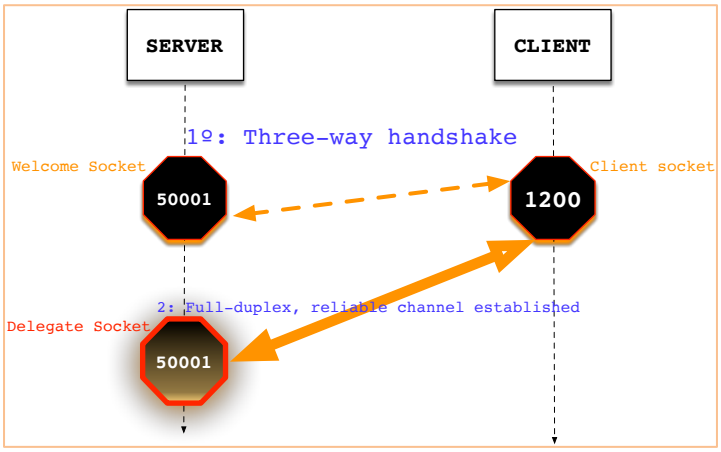

Fig. 4. After the three-way handshake, the delegate socket and the client socket begin the data transfer

Sockets as names of processes

A socket is created at the request of an application program executing as a process, such that, once a socket is created, it effectively identifies the program that created it, that is, if we know the socket we will be able to identify the unique process that created it. This process is the receiving end of the information conveyed through the socket. Since a socket gives a process a *name,* what is, then, the difference between any two sockets?

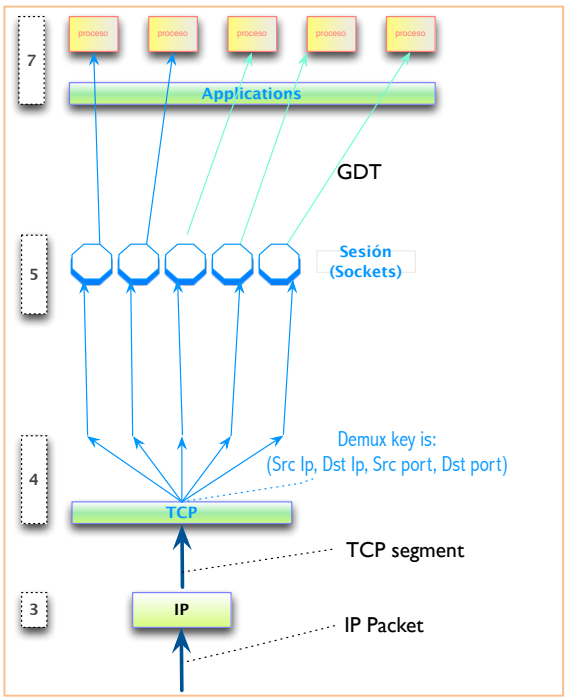

Fig. 5. Layer-4 demultiplexing keys in TCP

Two sockets within an operating system differentiate by their TCP port numbers, these are 16-bit integers constitute one of the components of the multiplexing keys at the transport layer, TCP. For example, when a web server performs a *passive open* to establish its Welcome Socket it specifies port number 80, the well-known (Standardized) port number for accessing the web server from the internet. Well-known ports and their corresponding services are related in file /etc/services in Unix and Linux and in a similar file in Windows. All in all, the port number applied to a socket is one of the four components of the TCP mux key (Src Ip, src, port, dst Ip, dst port) turning the socket interface into a *name space* for processes within a system, this namespace will allow client sockets to connect with server sockets and thereby with their creator processes, end-to-end. Figures 5 and 6 illustrate this concept of sockets as a namespace for processes.

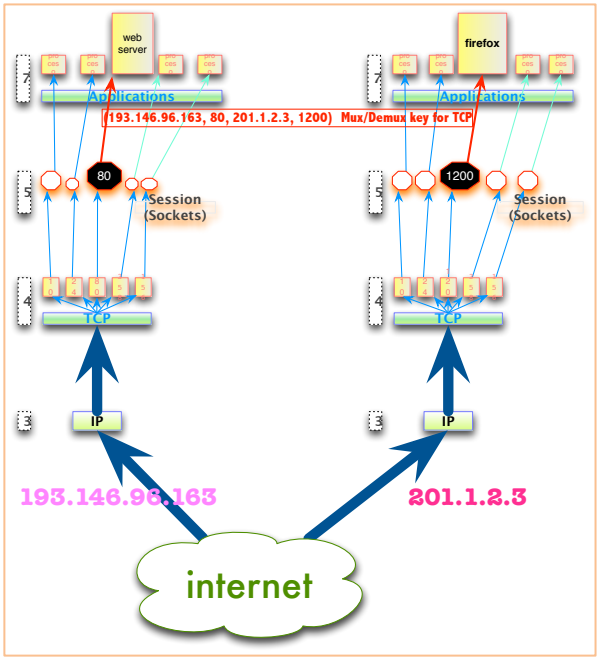

Fig. 6. Example of a demultiplexing key value that identifies a Web client-to-server TCP connection

Exercise 2. Programming a simple C/S protocol based on stream sockets. We provide you the C source code to a basic client and server for a simple C/S application. We start by downloading and compiling the server and leave the client for exercises 10 and the ensuing ones.

a. Request a terminal and download the server's source code:

```
$ wget http://paloalto.unileon.es/ds/lab/TCP-Server-Base.c
```

TCP-Server-Base.c is also available at paloalto.unileon.es/50500 /home directory

b. Compile the server program:

```
$ gcc –o server TCP-Server-Base.c
```

c. Run the server (No need to sudo since this practical uses Stream sockets which require no particular permissions to be created and used)

```
$ ./server
Server loop restarted
```

d. Skim the server code, notice where the welcome and delegate sockets are created.

Exercise 3. Have a class mate at a host other than yours run the $ nc command to connect with your server. This command allows you to communicate with a server by using your choice of TCP or UDP and send it character strings that you type; it results handy when checking simple clients and servers. It can also function as a server. Before running $ nc, you'll have to find out your IP address (The default TCP port used in the server program is 50001):

```
$ nc <server IP address> 50001
```

(At this point, you can type any string you want here; attentively read the server's source code and test

all the possibilities)

Exercise 4. Now, send the "Shutdown server" string to the server (Type it verbatim, otherwise the server program will not be able to recognize it); it will close the server socket and finish the program. Observe that the client sends request strings to the server which will be subsequently interpreted by the latter so it can fulfil the request and send back the results to the client.

Exercise 5. Restart the server program and notice that when the server socket (Welcome socket) is created, no traffic is generated. Check this is true by observing the Wireshark trace as you start execution of the server program; also, check that the Welcome Socket is successfully created after the server starts execution by issuing the command below. Make sure you understand the full output produced by the command:

```
$ netstat –a –tcp –n | grep 50001
```

Exercise 6. Observe the messages exchanged by C and S as the TCP connection is being created between (3-way handshake). Make sure Wireshark is acquiring a trace and that the filter tcp.port==50001 is active.

a. Start the server program

b. Connect with your server by using $ nc <Server IP address> <50001>

c. Observe the SYN segment sent from the Client to the Server and tell what the Initial Sequence Number is. Note that Wireshark does not present the real sequence numbers which were generated by each end system, but *relative sequence numbers.* These sequence numbers result easy to handle by the engineer conducting the analysis and, if necessary, Wireshark makes available the real sequence numbers in the TCP segment SN and ACK SN fields.

d. Did the SYN segment carry any options? List the options that you saw.

e. Probably you did see the MSS option activated in the SYN segment, is that true? Tell us the value of MSS under the following two conditions: The server is running in the same host where the server is running and, second, the client is run at a host other than that where the server is running. Have you observed that the values of MSS might be vastly different? Provide a detailed explanation about this fact.

f. Observe the response sent by the server (ACK-SYN); again, list the options that were included in the segment. Observe whether the ACK number sent in the ACK-SYN segment is consistent with the received Initial Sequence Number sent by the Client. Recall that this ACK sequence number in the response message should be 1 plus the sequence number received in the SYN sequence number.

g. Finally, you'll see an ACK segment sent from the Client to the Server. When that segment arrives at S, the connection reaches the ESTABLISHED state (C and S are in the ESTABLISHED state). Check that you saw the segment having the expected ACK.

Exercise 7. The Sliding Window algorithm functions when the server and the client are both in the ESTABLISHED STATE; this TCP state machine will govern all the transfers between Client and Server and you'll be able to watch them in your Wireshark:

a. Check that the sequence numbers and acknowledgement numbers are the expected ones according to

the TCP protocol. For example, when the nc command is running and the connection with the server is established, send the message "Hello world" and check the sequence numbers included in the segment sent by TCP when you press the return key.

b. Identify the segment that contains the ACK corresponding to the data sent in the preceding segment containing "Hello world". What ACK sequence number does this segment contain?

c. Wireshark offers a convenient facility for observing a TCP C/S conversation which name is Flow Graph. Apply this the Flow Graph facility to the interaction between $ nc and the base TCP server. The Flow Graph menu can be found at Wireshark Statistics | Flow Graph menu. Make sure that tick the box "Limit to display filter" and set the flow type to TCP flow:

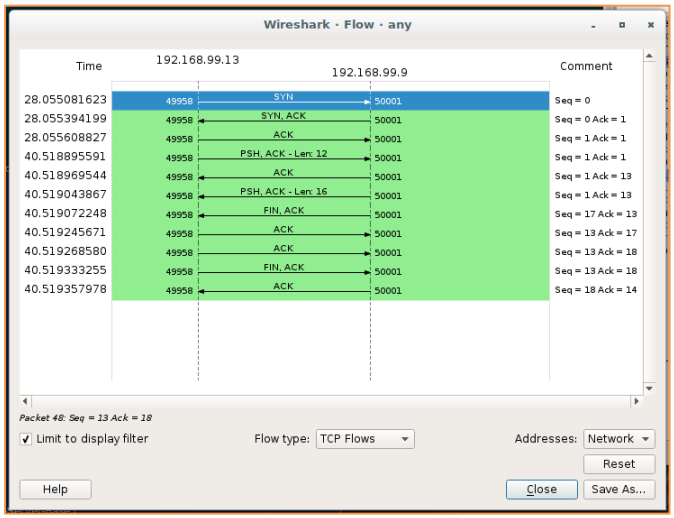

Fig. 6.1. Applying Wireshark's Flow Graph to the dialog between Client and Server

Exercise 8. Have the server stop its execution by connecting to it with $ nc command and then sending the literal string "Shutdown server". When received, this message will cause the server to close the active delegate socket and also the server socket (Welcome socket).

a. Carefully observe the segments exchanged by C and S upon connection close. In this case the close operation is initiated by the server, however, it could also have been generated by the client; check that this statement is true by observing the TCP State Diagram included in this practice (RFC 793).

b. Issue now a netstat command on the Client side and also on the Server side and check that the relevant Client, Welcome and Delegate sockets are in the process of closing.

Exercise 9. Equipped with the experience gained by using command $ nc with our server, we set out to write a client program that is compatible with it. We provide you the base client code in the following file:

```
$ wget http://paloalto.unileon.es/ds/Lab/TCP/TCP-Client-Base.c
```

a. This base client program sends a message to the server that is unknown to it. Check this by running it

and passing it the correct command line arguments. Read the source code in function main().

b. Observe the full TCP connection lifecycle with Wireshark.

Exercise 10. Study the computeResponse() server function and according to it, add the client the capability to send some of the relevant messages that function is able to handle, maybe by using some new command line argument. One of the messages the server is able to handle is the "Send the date" message, which, when received should cause the server to send the current system's current wall clock time.

a. The implementation of this functionality in function computeResponse() uses two library functions: time() and ctime_r(). Speculate why we selected ctime_r() function and not ctime().

b. If you had to use synchronize other host's clock with this host's, what result would you send to the other host, the string resulting from ctime_r() or the time_t resulting from time()?

Exercise 11. The server program is able to print the IP and the port used by the client when the connection is established. Variable clientAddress in function server() represents an empty socket address which address (A pointer to it) is passed to function accept(), so that, when it returns the delegate socket, we can find out the IP address and TCP port used by the client.

a. Study the code and read the relevant man pages if necessary to explain why it is required that a pointer to the variable containing the size of the clientAddress be passed to function accept().

b. What happens if we pass a size other than the real size of variable clientAddress. Try this in your own PC and in some of the lab PC's and contrast the observed behavior.

Exercise 12. Using Stream sockets write an echo server and a client for testing it, or maybe test it with **nc**.

