V 4.0

Lab Practicals on Computer Networks and Distributed Systems

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

This practical aims to illustrate basic concepts about the TCP protocol by creating a simple client/server application that will be used for observing the implemented application protocol with the tcpdump packet sniffer. We begin by recollecting the Client/Server computing model and the relevant service interfaces to TCP (Stream sockets).

What is *sockets*?

Nowadays, the TCP protocol module in a mainstream operating system offers essential services to application programs such as reliable transmission, flow and congestion control. This variety of objectives makes TCP complex, thus, in an attempt to isolate application programmers from the complexity of TCP, operating systems offer the Sockets service interface; programming languages include APIs that allow programs to call TCP services for communicating over the Internet. The two languages used in this course, C and Java, do offer the Sockets API; in this practical we'll complete a simple C/S pair written in the C language.

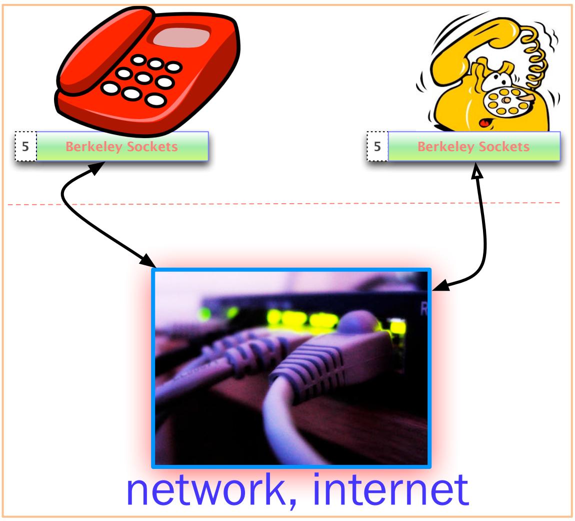

Fig. 1. TCP Connection analogy to telephone dialing

TCP is a connection-oriented protocol. Whenever we want to transmit information reliably by using it, the TCP protocol will setup a connection with the remote computer. What is the purpose of that so-called connection? During the connection setup stage, the two communicating hosts share a few communication parameters that foster a better use of the networking resources that stand in between the client host and the server host. The sharing of these parameters will also permit the cooperative execution of key TCP algorithms such as Sliding Window, Flow Control and Congestion Control. These algorithms constitute the essence of TCP.

The Berkeley Socket API, originally programmed in C, has been ported to other languages, thereby fostering interoperability between clients and servers written in whatever language that implements the API. For example, a sockets-based server program written in Java will interoperate with a client written in Python to the Sockets API. The programmer's task in this case consists of implementing the application-level protocol correctly by sending the relevant protocol messages over the c/s socket pair. The process that will eventually lead to the creation of the c/s connection is illustrated in Fig. 2.

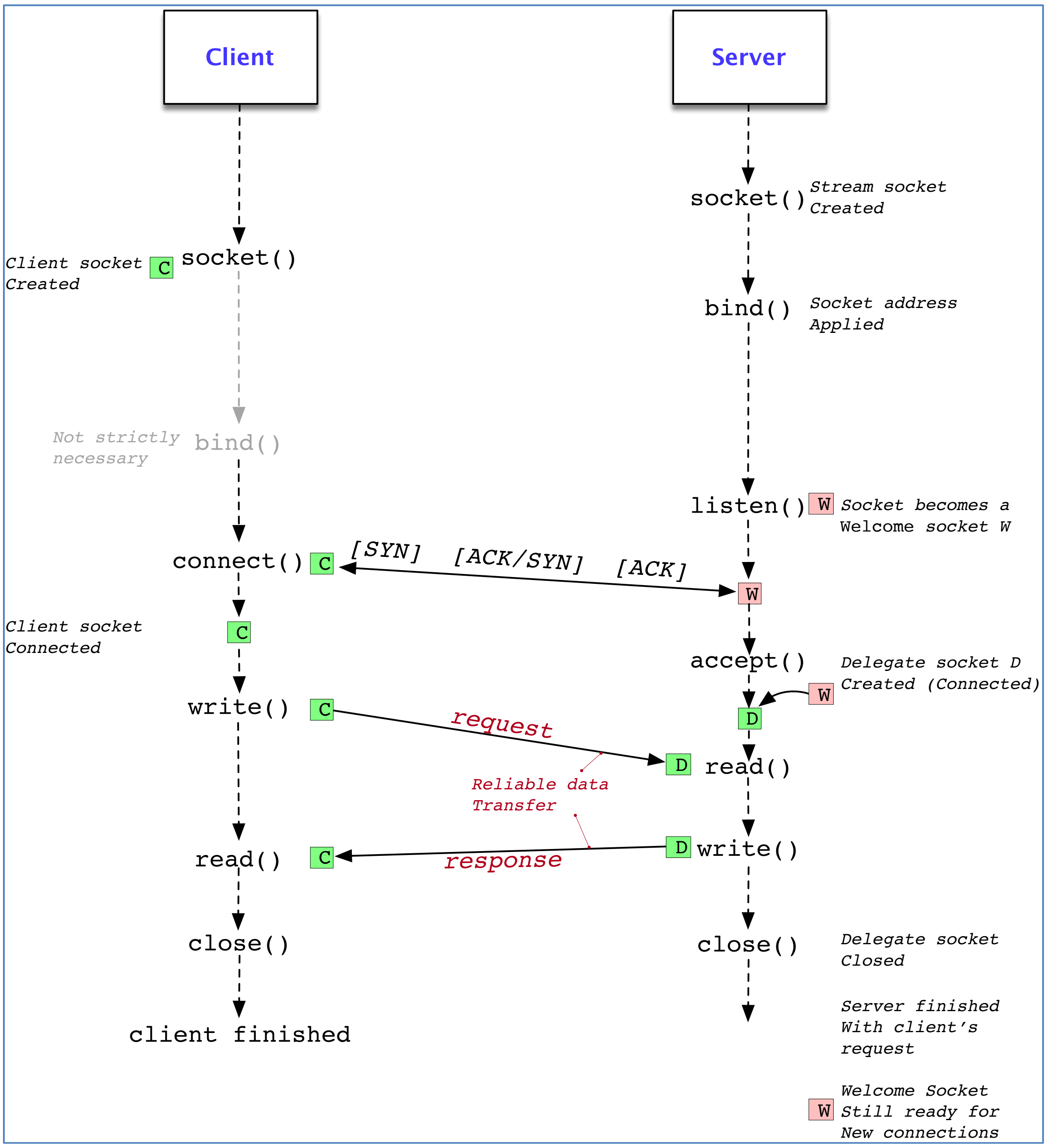

Fig. 2. Berkeley socket c/s stages: connection setup, data transfer and connection teardown.

Exercise 1. Create a TCP connection from an nc client to an nc server. Use tcpdump to observe the TCP segments exchanged. Finally, explain their semantics by interpreting them according to the relevant RFC alongside the lecture's slides (nc refers to the netcat Linux program used in previous practicals).

1. Log on to one of Lab B6 hosts. Below, we'll refer to this host as host H_0.

   In the present example, the IP address of host H_0 is 192.168.1.89. In your specific case, that IP address could be any of the IP addresses in range 192.168.1.150 to 192.168.1.240.

2. At host H_0, log on to a remote host (H_1) of Lab B6 by opening an ssh session to it. In this specific example, host H_1 has IP address 192.168.1.210. In your case, that IP address might be any of the IP addresses mentioned above.

   ```
   $ ssh administrator@192.168.1.210
   ```

3. Open two additional ssh sessions to host H_1. These two new sessions will allow us to observe the messages that make up the C/S application protocol. As usual, we'll run tcpdump as for tracing the TCP connection along with the TCP/IP protocol stack state with command netstat. We'll refer to these ancillary ssh sessions as B and C, respectively. In summary, so far, we should have the following sessions active at the time:

   - SSH Session A: For running the server program and for monitoring it
   - SSH Session B: For tracing the C/S server connection life cycle with tcpdump
   - SSH Session C: For monitoring the state of the server's stack with command netstat

4. Check TCP port availability at H_1 in session C. To that end, execute the following netstat command. Check out man netstat if you need more detail about the used options:

   ```
   $ netstat -a -tcp -n -c
   ```

5. You should choose a TCP port number that does not appear in the preceding stack state listing, i.e., one that is available at the time. **For example**, assuming port 60000 doesn't appear on the listing, which means that it is available at the time, use the **nc command** to create a simple TCP server that listens on that port (The nc server simply prints out each character received). Do this at ssh session A.

   If necessary, check out the man page to the nc command. The options necessary for doing this example are these:

   ```
   -l nc is to be run as a TCP server
   -q 1: On the server side, after ^D the program will close the
   connection and stop after 1 second
   -k After completing one connection, wait for the next connection
   request by using a stream socket in LISTEN mode
   -p 60000: Have the server listen on TCP port 60000
   ```

   ```
   [In Session A] $ nc -l -q 1 -k -p 60000
   ```

6. After you press return to execute the preceding command, check that a new TCP connection to host H_1 at port 60000 does appear now in the stack state listing produced by netstat in session C.

```
[In Session C] $ netstat -anc --tcp
```

If the repetitive netstat display (option -c) is not of help now, get rid of that option and repeat the command whenever you need to update the stack state listing.

7. In session B, run tcpdump to trace the connection progress; make sure you specify the correct network interface. In this example, it is eno1 as reported by command ifconfig. Notice that the TCP connection hasn't been made yet. We'll do that below, soon.

   ```
   [In Session B] $ su
   Password: …

   [In Session B] $ whereis tcpdump
   /usr/sbin/tcpdump


   [In Session B] $ /usr/sbin/tcpdump -i eno1 -nvvv -X tcp port 60000
   ```

   The full tcpdump trace has been included in Appendix 1, at the end of this document.

8. On the client side, at host H_0, in a new terminal, run the nc (netcat) command that will create the TCP connection with H_1 at port 60000

   ```
   [In host H0, nc client] $ nc 192.168.1.210 60000
   ```

9. Compose a few messages at the terminal where the nc client is running and have them sent by clicking the enter key, then, observe that the server terminal displays each message that you sent from the client. At the same time, observe that the TCP socket at port 60000 at the server has changed states from the LISTEN state to the ESTABLISHED state:

   ```
   [In host H0, nc client]
   Hello world!
   Hello, again!
   ```

10. Switch to Session A, where the nc server is running. Compose the control combination ^D (^D means that you must press the CONTROL key at the same time that you press the D key. By default, your console assigns the TERMINATE action to that control combination, which will indicate the server to close the TCP connection and stop running):

   ```
   [In Session A]
   ^D
   ```

   Soon, the server will start the connection close stage by sending back a segment that has the F (Finalize) flag set and an ACK that is 1 more than the last data byte received from the client, in this case. This indicates that the server successfully accepts all the received data up to the ACK value minus 1 and that the Server-to-Client channel is now closed.

   You can examine what actually happens on the stream by studying the tcpdump trace included in Appendix 1.

Exercise 2. Study the trace obtained with tcpdump (Appendix 1, below) which should contain the frames involved in the full TCP connection lifecycle that was created in the previous exercise. Label the frames that comprise the three TCP-connection stages:

a. **Connection establishment**
   - Three-way handshake
   - Announcing connection options
   - Setting initial Sequence Numbers

b. **Data transfer**
   - Send simple, short character strings and observe their Sequence Numbers and the corresponding Acknowledgement Numbers received over the other channel
   - TCP segments may carry data which is indexed by the contents of the SN field; the contents of the ACK SN field represent the sequence number to the next data byte that should be sent by the other side in-order

c. **Connection close**
   - Observe which of the two connection ends initiates connection close. In our case, if you've followed the practice script above, it's the server that sends the connection close TCP segment after you type ^D

Exercise 3. Study more deeply the TCP-connection trace, again. Respond to these questions:

1. Observe the TCP segment sent from the client, which starts the 3-way handshake procedure against the server and which has the S (SYN) flag set. Write down the SN set by TCP in that segment.

2. Probably you saw the MSS (Maximum Segment Size) option activated in the preceding SYN segment, is that true? Tell us the actual value of MSS that you saw included in that segment. That value represents the maximum payload size, in Bytes, that can be encapsulated into a TCP segment such that the IP packet that will in turn encapsulate that TCP segment, honors the underlying datalink MTU.

3. Prove that the value of MSS is correct by doing the necessary calculations by hand. Command ifconfig will tell you the relevant Ethernet interface's MTU, so you can do those calculations as we did in CN, by computing every PDU size from the protocol stack (App -> TCP -> IP -> Ethernet).

4. Notice that SN (Sequence Numbers), after the two initial steps in the 3-way handshake, are *logical sequence numbers.* These begin at 0, and are much easier to grasp when we are analyzing traces than the actual ones. Nevertheless, the **actual** sequence numbers (SN) will always be available in the hexadecimal dump section of the trace, that is, despite the fact that the textual area of tcpdump prints out the **logical** version of those SN.

5. Check that the ACK SN sent back by the server after it receives the "Hello world!" message from the client is correct.

Appendix 1. TCP connection traces from Exercise 1.

```
root@debian:/home/administrator# tcpdump -i eno1 -nvvv -X tcp port 60000
tcpdump: listening on eno1, link-type EN10MB (Ethernet), snapshot length 262144 bytes
```

Connection Setup Stage

```
11:24:53.777987 IP (tos 0x0, ttl 64, id 1430, offset 0, flags [DF], proto
TCP (6), length 60)
    192.168.1.89.33536 > 192.168.1.210.60000: Flags [S], cksum 0x16c9
(correct), seq 924685319, win 64240, options [mss 1460,sackOK,TS val
664162734 ecr 0,nop,wscale 7], length 0
      0x0000:  4500 003c 0596 4000 4006 b0aa c0a8 0159  E..<..@.@......Y
      0x0010:  c0a8 01d2 8300 ea60 371d 9407 0000 0000  .......`7.......
      0x0020:  a002 faf0 16c9 0000 0204 05b4 0402 080a  ................
      0x0030:  2796 51ae 0000 0000 0103 0307            '.Q.........

11:24:53.777995 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP
(6), length 60)
    192.168.1.210.60000 > 192.168.1.89.33536: Flags [S.], cksum 0x84aa
(incorrect -> 0x85c7), seq 2625965332, ack 924685320, win 65160, options
[mss 1460,sackOK,TS val 2432651967 ecr 664162734,nop,wscale 7], length 0
      0x0000:  4500 003c 0000 4000 4006 b640 c0a8 01d2  E..<..@.@..@....
      0x0010:  c0a8 0159 ea60 8300 9c85 0d14 371d 9408  ...Y.`......7...
      0x0020:  a012 fe88 84aa 0000 0204 05b4 0402 080a  ................
      0x0030:  90ff 52bf 2796 51ae 0103 0307            ..R.'.Q.....

11:24:53.778363 IP (tos 0x0, ttl 64, id 1431, offset 0, flags [DF], proto
TCP (6), length 52)
    192.168.1.89.33536 > 192.168.1.210.60000: Flags [.], cksum 0xb126
(correct), seq 1, ack 1, win 502, options [nop,nop,TS val 664162734 ecr
2432651967], length 0
      0x0000:  4500 0034 0597 4000 4006 b0b1 c0a8 0159  E..4..@.@......Y
      0x0010:  c0a8 01d2 8300 ea60 371d 9408 9c85 0d15  .......`7.......
      0x0020:  8010 01f6 b126 0000 0101 080a 2796 51ae  .....&......'.Q.
      0x0030:  90ff 52bf                                ..R.
```

Data Transfer Stage

```
11:25:00.372914 IP (tos 0x0, ttl 64, id 1432, offset 0, flags [DF], proto
TCP (6), length 65)
    192.168.1.89.33536 > 192.168.1.210.60000: Flags [P.], cksum 0x1b5f
(correct), seq 1:14, ack 1, win 502, options [nop,nop,TS val 664169329 ecr
2432651967], length 13
      0x0000:  4500 0041 0598 4000 4006 b0a3 c0a8 0159  E..A..@.@......Y
      0x0010:  c0a8 01d2 8300 ea60 371d 9408 9c85 0d15  .......`7.......
      0x0020:  8018 01f6 1b5f 0000 0101 080a 2796 6b71  ....._......'.kq
      0x0030:  90ff 52bf 4865 6c6c 6f20 776f 726c 6421  ..R.Hello.world!
      0x0040:  0a                                       .

11:25:00.372918 IP (tos 0x0, ttl 64, id 29408, offset 0, flags [DF], proto
TCP (6), length 52)
    192.168.1.210.60000 > 192.168.1.89.33536: Flags [.], cksum 0x84a2
(incorrect -> 0x7d8c), seq 1, ack 14, win 509, options [nop,nop,TS val
2432658562 ecr 664169329], length 0
      0x0000:  4500 0034 72e0 4000 4006 4368 c0a8 01d2  E..4r.@.@.Ch....
      0x0010:  c0a8 0159 ea60 8300 9c85 0d15 371d 9415  ...Y.`......7...
      0x0020:  8010 01fd 84a2 0000 0101 080a 90ff 6c82  ..............l.
      0x0030:  2796 6b71                                '.kq

11:25:05.082909 IP (tos 0x0, ttl 64, id 1433, offset 0, flags [DF], proto
TCP (6), length 66)
    192.168.1.89.33536 > 192.168.1.210.60000: Flags [P.], cksum 0x34de
(correct), seq 14:28, ack 1, win 502, options [nop,nop,TS val 664174039 ecr
2432658562], length 14
      0x0000:  4500 0042 0599 4000 4006 b0a1 c0a8 0159  E..B..@.@......Y
      0x0010:  c0a8 01d2 8300 ea60 371d 9415 9c85 0d15  .......`7.......
      0x0020:  8018 01f6 34de 0000 0101 080a 2796 7dd7  ....4.......'.}.
      0x0030:  90ff 6c82 4865 6c6c 6f2c 2061 6761 696e  ..l.Hello,.again
      0x0040:  210a                                     !.

11:25:05.082913 IP (tos 0x0, ttl 64, id 29409, offset 0, flags [DF], proto
TCP (6), length 52)
    192.168.1.210.60000 > 192.168.1.89.33536: Flags [.], cksum 0x84a2
(incorrect -> 0x58b2), seq 1, ack 28, win 509, options [nop,nop,TS val
2432663272 ecr 664174039], length 0
      0x0000:  4500 0034 72e1 4000 4006 4367 c0a8 01d2  E..4r.@.@.Cg....
      0x0010:  c0a8 0159 ea60 8300 9c85 0d15 371d 9423  ...Y.`......7..#
      0x0020:  8010 01fd 84a2 0000 0101 080a 90ff 7ee8  ..............~.
      0x0030:  2796 7dd7                                '.}.
```

Connection teardown (Initiated at server side)

```
11:25:12.497527 IP (tos 0x0, ttl 64, id 29410, offset 0, flags [DF], proto
TCP (6), length 52)
    192.168.1.210.60000 > 192.168.1.89.33536: Flags [F.], cksum 0x84a2
(incorrect -> 0x3bba), seq 1, ack 28, win 509, options [nop,nop,TS val
2432670687 ecr 664174039], length 0
      0x0000:  4500 0034 72e2 4000 4006 4366 c0a8 01d2  E..4r.@.@.Cf....
      0x0010:  c0a8 0159 ea60 8300 9c85 0d15 371d 9423  ...Y.`......7..#
      0x0020:  8011 01fd 84a2 0000 0101 080a 90ff 9bdf  ................
      0x0030:  2796 7dd7                                '.}.

11:25:12.500625 IP (tos 0x0, ttl 64, id 1434, offset 0, flags [DF], proto
TCP (6), length 52)
    192.168.1.89.33536 > 192.168.1.210.60000: Flags [.], cksum 0x1ec7
(correct), seq 28, ack 2, win 502, options [nop,nop,TS val 664181457 ecr
2432670687], length 0
      0x0000:  4500 0034 059a 4000 4006 b0ae c0a8 0159  E..4..@.@......Y
      0x0010:  c0a8 01d2 8300 ea60 371d 9423 9c85 0d16  .......`7..#....
      0x0020:  8010 01f6 1ec7 0000 0101 080a 2796 9ad1  ............'...
      0x0030:  90ff 9bdf                                ....

11:25:19.381477 IP (tos 0x0, ttl 64, id 1435, offset 0, flags [DF], proto
TCP (6), length 52)
    192.168.1.89.33536 > 192.168.1.210.60000: Flags [F.], cksum 0x03e6
(correct), seq 28, ack 2, win 502, options [nop,nop,TS val 664188337 ecr
2432670687], length 0
      0x0000:  4500 0034 059b 4000 4006 b0ad c0a8 0159  E..4..@.@......Y
      0x0010:  c0a8 01d2 8300 ea60 371d 9423 9c85 0d16  .......`7..#....
      0x0020:  8011 01f6 03e6 0000 0101 080a 2796 b5b1  ............'...
      0x0030:  90ff 9bdf                                ....

11:25:19.381483 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP
(6), length 52)
    192.168.1.210.60000 > 192.168.1.89.33536: Flags [.], cksum 0xe8fa
(correct), seq 2, ack 29, win 509, options [nop,nop,TS val 2432677571 ecr
664188337], length 0
      0x0000:  4500 0034 0000 4000 4006 b648 c0a8 01d2  E..4..@.@..H....
      0x0010:  c0a8 0159 ea60 8300 9c85 0d16 371d 9424  ...Y.`......7..$
      0x0020:  8010 01fd e8fa 0000 0101 080a 90ff b6c3  ................
      0x0030:  2796 b5b1                                '...
```

```
^C
11 packets captured
11 packets received by filter
0 packets dropped by kernel
root@debian:/home/administrator#
```