KHALIFA UNIVERSITY

Spring 2014
Computer Networks
CMPE323

Laboratory Experiment No. 6: Introduction to TCP

Aims and Objectives:

- Introducing Transmission Control Protocol (TCP) and the mechanisms of connection establishment, data transmission, and connection termination.

Materials Required:

- IP routers,
- Ethernet switches,
- PCs with Ethernet adapters,
- and straight-through/crossover/rollover cables.

Change Log:

- 8-4-2014: original document – mkhonji.
- 9-4-2014: expanded section 1.4 with added explanation – mkhonji.

1 Introduction

So far, we have covered the following topics:

- **Layer 1:** Common types of physical mediums that allow the transmission of bits to and from connected nodes. This was achieved in the lab via twisted-pairs of copper wires using Cat6e cables and RJ45 connectors. We've also explored how RX and TX pins can be connected accordingly via cross-over and straight-through cables, as well as Auto-MDIX for automatic pin re-arrangement within Ethernet network adapters.
- **Layer 2:** Broadcast domains (e.g. Ethernet), scalability issues that we face if the broadcast domains grow too large, and how to reduce their size. This was achieved by physical separation as well as logical separation of broadcast domains using VLANs and IEEE 802.1Q tags over trunking ports.
- **Layer 3:** A scalable architecture that allows nodes within *different* broadcast domains to communicate with each other. This introduced IP and CIDR which allowed end-to-end reachability.
- **Layer 4:** A mechanism by which applications on nodes can specify not only which destination node they wish to communicate to, but also specify which application or service they wish to receive the sent messages. So far, we have only covered UDP — a simple minimalist protocol that does just that.

In this lab, we will study another *layer 4* protocol, namely: Transmission Control Protocol (TCP). However, before we describe what TCP is, let's first see the advantages and the disadvantages of UDP.

1.1 Why is UDP good?

Before we look at the limitations of UDP, it is important to note that, due to the simplicity of UDP, it is a pretty good choice for scenarios where low-delay transmission of data is needed.

For example, if you are playing some multi-player first-person shooter game over the network and that you press the trigger to fire a bullet towards some moving object (e.g. running enemy vehicle).

In the example above, it is very important to ensure that the message (that signals the firing of the bullet to the server) is sent as soon as possible to the server. If there is any transmission delay, then by the time the message is delivered to the server, the object may have moved too far away.

This is why time-sensitive applications use UDP. This includes applications such as the Domain Name System (DNS) where transmission delay can cause significant effects against the overall end-user experience.

1.2 Why is UDP not enough?

UDP does not provide the following by itself:

- **Reliable data transmission:** e.g. when sender S sends packet P to receiver R, and if P was dropped along the path before it reached R, then UDP does not provide (by itself) any mechanism to identify such packet loss and thus P will not be retransmitted by UDP.
- **Ordered data delivery:** UDP does not provide any mechanism by itself by which the receiver can ensure that the order of reception is the same as the order of transmission. In other words, if the sender S sent packets (P_1, P_2) in this order to some receiver R, and if R received the packets in a different order (P_2, P_1), then UDP provides no mechanism by itself for R to identify the correct order.
- **Flow control:** e.g. when sender S sends some packet P to receiver R, then there is no mechanism that is provided by UDP itself to let S know how much data R is able to receive/process. This way, S may overwhelm R with excessive amounts of data and yet S does not know that R is under too much load.
- **Congestion control:** e.g. when some network congestion accrues, there is no mechanism that is provided by UDP to automatically reduce the transmission rate in order to reduce the congestion.

However, it is important to note that while UDP does not provide the above mechanisms by itself, there is nothing that forbids UDP applications to implement their own application-layer mechanisms that facilitate the above. For example, a UDP application may implement reliable data transmission in its application layer by sending acknowledgements of received data.

1.3 Why do we need TCP?

There are many applications that require reliable data transmission, flow control and congestion control, while (at the same time) are not delay-sensitive. Examples are HTTP, FTP, SMTP, IMAP, POP3, Telnet, SSH, ... Since there are many applications that require such features, then having each application implement its own set of mechanisms is redundant and time-consuming.

Due to the reason above, it is clear that it would be useful if a layer 4 protocol provided the above mechanisms by itself, so that all application layer protocols can *transparently* take advantage of them without the need of coding them into their application layer.

This is exactly what TCP does: a layer 4 protocol that provides the following additional mechanisms: reliable data transmission, ordered data delivery, flow control, congestion control. UDP provides error detection and so does TCP. Therefore, if your application is not delay-sensitive and needs the mechanisms above, then TCP is a good choice.

1.4 How does TCP work?

In this lab, we will cover the basic behaviour of TCP, which is a subset of what is defined in RFC793.

However, it should be noted that changes beyond RFC793 exist. This is due to a number of reasons, one of which is that TCP allows for extensions via variable length options, as well as the existence of tweakable parameters in TCP's operation (e.g. multiple timer parameters and finding the right timers is a problem that is an optimization problem. For example, RFC7298 was proposed as late as 2011 to present a newer method for calculating the re-transmission timer).

Simply put, the most visible principles of TCP that achieve that are:

- Use of sequence numbers: all transmitted data are given sequence numbers that receivers must acknowledge by sending back the incremented sequence number by one. I.e. if node R receives an octet with the sequence number `SEQ` $= 1$, then R must acknowledge it by sending an acknowledgement number `ACK` $=$ `SEQ` $+ 1 = 1+1 = 2$ back to the sender S.

- Use of timers: if an acknowledgement is not received within some time interval, then actions will be taken accordingly. For example, if the sender node S sent some data to the receiver node R and S did not receive any acknowledgements back from R for some time T, then S will assume that the data was lost will then retransmit it again. Additionally, R will also slow-down its transmission speed (by reducing its transmission window size) in an attempt to avoid possible causes of the assumed data loss (e.g. network congestion, or if R was too busy processing previous data).

In this lab, we will focus on: connection establishment, data transmission, and connection termination.

1.4.1 Connection establishment

TCP requires that, prior to the transmission of any data, a connection to be established first. Such connection establishment happens via what's known as *TCP's 3-way hand-shake*. The goal of this stage is simple: identifying the sequence number the first bytes of both of the communicating peers.

An example is depicted in Figure 1 by which the peers A and B agreed at sequence numbers 101 and 301 respectively.

```
   TCP A                                                TCP B
1. CLOSED                                               LISTEN
2. SYN-SENT    --> <SEQ=100><CTL=SYN>               --> SYN-RECEIVED
3. ESTABLISHED <-- <SEQ=300><ACK=101><CTL=SYN,ACK>  <-- SYN-RECEIVED
4. ESTABLISHED --> <SEQ=101><ACK=301><CTL=ACK>      --> ESTABLISHED
```

Figure 1: TCP's three-way hand-shake for connection establishment — Source RFC793.

1.4.2 Data transmission

Once the TCP 3-way hand-shake is successfully performed to obtain the sequence numbers, it is possible for the involved peers to transmit data as depicted in Figure 2. Notice how the transmitted data "`Hello`" is composed of 5 octets with its first octet having the sequence 101, which resulted in the peer B to acknowledge its reception by sending an acknowledgement of 106 = 101 + 5. In other words, the acknowledgements are the sequence number of the first octet + the total number of received octets since the first one.

```
   TCP A                                                     TCP B
1. ESTABLISHED --> <SEQ=101><ACK=301><CTL=PSH><Hello>  --> ESTABLISHED
4. ESTABLISHED --> <SEQ=301><ACK=106><CTL=ACK>         --> ESTABLISHED
```

Figure 2: TCP's data communication.

1.4.3 Connection termination

Similar to the connection establishment, except for using the `FIN` flag instead of the `SYN` flag. As depicted in Figure 3, note how the sequence numbers are continuations of the last sequence numbers.

```
   TCP A                                                  TCP B
1. ESTABLISHED                                            ESTABLISHED
2. FIN-WAIT    --> <SEQ=106><CTL=FIN>                 --> CLOSE-WAIT
3. TIME-WAIT   <-- <SEQ=301><ACK=107><CTL=FIN,ACK>   <-- LAST-ACK
4. CLOSED      --> <SEQ=107><ACK=302><CTL=ACK>        --> CLOSED
```

Figure 3: TCP's three-way hand-shake for connection termination. Note that this is a simplification for the original 4-way hand-shake that is described in RF793. I.e. instead of having peer `B` send two TCP packets (one packet `ACK`ing peer A's `FIN` and another indicating its own desire by terminate the connection by setting the `FIN` flag), it combines them in a single packet (one packet that has both `ACK` and `FIN` flags set.

1.4.4 Header format

TCP's fields are (fields that are beyond the scope are marked as gray):

- Source and destination port numbers: each is 16 bits.
- Sequence and acknowledgement numbers: each is 32bits.
- Data offset: this is essentially TCP's header length in units of 32bits. Thus the minimal size is 5 (assuming no option fields are used).
- Future reserved bits: 5 bits that are not used according to RFC793, however, later RFCs that are beyond the scope of this lab have proposed methods to utilize those reserved bits.
- Flags: 6 on/off flags (each is 1 bit). The flags that are covered in this lab are ACK, PSH, SYN and FIN.
- Window: 16 bits that indicate receivers buffer. The unit is in octets by default (however, TCP allows for changing the unit via the use of TCP option headers, which is beyond the scope of this lab).

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4: TCP's header format — Source RFC793.

- Checksum: 16 bits of one's complement of one's complement sum of all 16 bit words of the TCP header, payload and the pseudo IP header as depicted in Figure 5.
- Urgent pointer: 16 bits number that indicates an offset from the current byte sequence down to some byte sequence. Bytes that fall within this range are considered urgent and will be processed first by the receiver (beyond the scope of the lab).
- Options: variable length, used to extend TCP (beyond the scope of this lab).
- Data: variable length sequence of bits to accommodate user data.

```
+--------+--------+--------+--------+
|           Source Address          |
+--------+--------+--------+--------+
|         Destination Address       |
+--------+--------+--------+--------+
|  zero  |  PTCL  |    TCP Length   |
+--------+--------+--------+--------+
```

Figure 5: Pseudo IP header format as used in TCP's checksum calculation, where `PTCL` is protocol ID — Source RFC793.

2 Lab Preparation

1. Connect a PC to the routers console port using a rollover cable[1].
2. Erase the configuration of the routers[2].
3. Connect a PC to the switches console ports using rollover cables.
4. Erase the configuration of the switch[3].
5. Physically connect the lab as depicted in Figure 6.

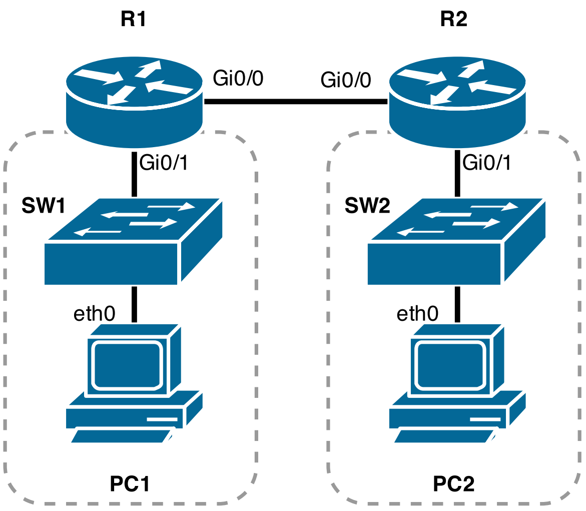

Figure 6: Physical lab topology.

6. Run Wireshark on all involved lab PCs as depicted in Figure 6.
7. Configure the interfaces:
   - Configure[4] `R1`'s interfaces as follows:
     - GigabitEthernet 0/0: IPv4 address 10.0.12.1, subnet mask 255.255.255.0.
     - GigabitEthernet 0/1: IPv4 address 10.0.1.1, subnet mask 255.255.255.0.
   - Configure `R2`'s interfaces as follows:
     - GigabitEthernet 0/0: IPv4 address 10.0.12.2, subnet mask 255.255.255.0.
     - GigabitEthernet 0/1: IPv4 address 10.0.2.1, subnet mask 255.255.255.0.
   - Configure[5] `PC1`'s interfaces as follows:

[1]If using Linux: `screen /dev/ttySx` were `x` is the serial interfaces ID that is connected to the console cable. If using Windows: Use Hyperterminal or Putty to connect to `COMx` ports.

[2]`enable`, `erase startup-config`, `reload`, and make sure to answer `no` to all yes/no questions while hitting *enter* for all `confirm` prompts.

[3]`enable`, `erase startup-config`, `delete vlan.dat`, `reload`, and answer `no` to all yes/no questions while hitting *enter* for all `confirm` prompts.

[4]`enable`, `configure terminal`, `interface Gi0/0`, `ip address 10.0.0.1 255.255.255.0`.

[5]`ifconfig eth0 10.0.1.2/24`

  - eth0: IPv4 address 10.0.1.2, subnet mask 255.255.255.0.
- Configure `PC2`'s interfaces as follows:
  - eth0: IPv4 address 10.0.2.2, subnet mask 255.255.255.0.

8. Then add the IP routing tables as follows:
   - For R1: `ip route 10.0.2.0 255.255.255.0 10.0.12.2`
   - For R2: `ip route 10.0.1.0 255.255.255.0 10.0.12.1`
   - For PC1: `route add -net 0.0.0.0/0 gw 10.0.1.1`
   - For PC2: `route add -net 0.0.0.0/0 gw 10.0.2.1`
9. Apply the following modifications to the TCP behaviour of the involved PCs:
   - Disable TCP `RST` messages[6]. This due to the fact that you will be using raw sockets which do not register connections in the kernel-maintained connections table, which causes the respond with `RST` packets as it thinks that there is no corresponding connection for the received TCP packets.
   - Extend the `SYN-RECEIVED` period[7] on the remote peer. This is simply to give you more time to figure out what should the response be before the remote peer decides to give up the connection and free its its resources.
10. Test connectivity using the `ping` command to send ICMP Echo messages to PCs in other networks and receive the respective ICMP Echo-Reply messages back.

3 Lab experiments

3.1 Connection establishment and data transmission

[100 points]

Note: When done, show your work to the lab engineer for grading purposes.

Using the `nc`[8] command run three applications on PC1 that listen on TCP ports as follows:

- Application 1: listen on IP 10.0.1.2 and TCP port 100.
- Application 2: listen on IP 10.0.1.2 and TCP port 200.
- Application 3: listen on IP 10.0.1.2 and TCP port 300

Then, using the provided tools (`macframesender.c`) send the following messages by using TCP:

- Message data: "Hello".
- Source: PC2.
- Destination: to PC1's `nc` process that is listening on port 100.

[6] `iptables -A OUTPUT -p tcp --tcp-flags RST RST -j DROP`
[7] `sysctl -w net.ipv4.tcp_synack_retries=9999`
[8] `nc -l -p 100 -s 10.0.1.2`

3.1.1 Notes

- Appendix A contains a description of protocol formats.
- Comment your `macframesender.c`. This can help you identify the problems much faster.
- You can save your time by organizing your code by copying `macframesender.c` into multiple files such as `syn.c`, `ack.c`, `psh.c`. This way you can repeat previous tasks much easier (except for updating the `ACK` number and TCP's checksum.
- The tool `nc` will not accept subsequent connections from *other* source IP addresses and source port numbers after it is used by some source IP and source port. Note that this is only a limitation in `nc` and it is easy to code TCP servers that accept connections from multiple clients.

3.2 Connection termination (extra question; not graded)

Terminate the TCP connection above using the FIN 3-way hand-shake.

A Protocol header formats and IDs

Protocol IDs: IP = 0x0800, TCP = 0x06.

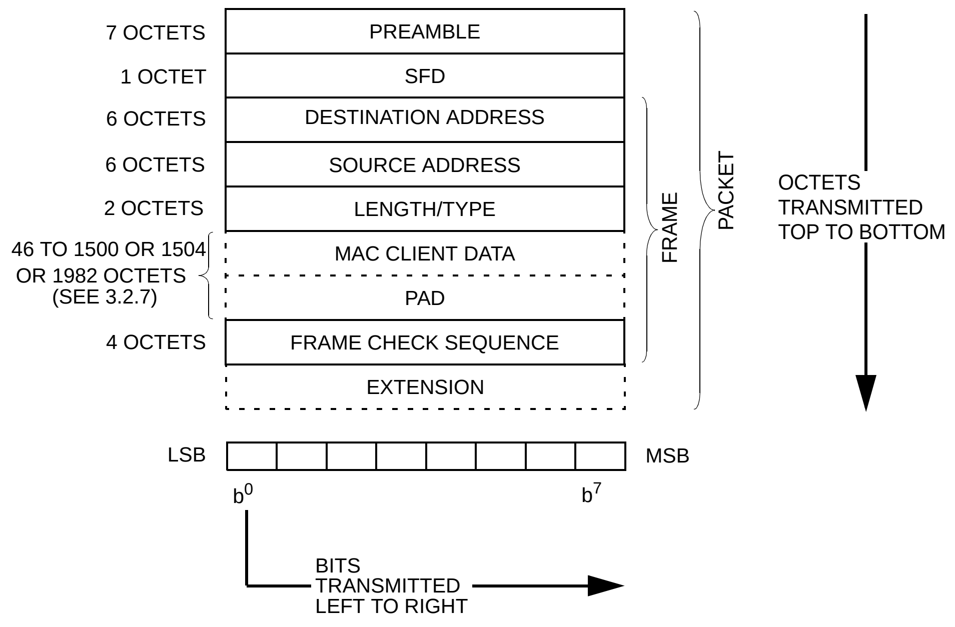

Figure 7: Ethernet Media Access Control (MAC) packet (note that an octet is 8 bits) — Source: IEEE Std. 802.3-2012.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Identification        |Flags|      Fragment Offset    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Time to Live |    Protocol   |         Header Checksum       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Source Address                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 8: Internet Protocol (IP) version 4 header format — Source RFC791.