NEW RELEASE

SAMPLE CHAPTER

CHAPTER 7: NETWORKS

About the author

Klaus Elk graduated as Master of Science in electronics from the Danish Technical University in Copenhagen in 1984, with a thesis in digital signal processing. Since then, he has worked in the private industry within the domains of telecommunication, medical electronics, and sound and vibration. He also holds a Bachelor's degree in Marketing. In a period of 10 years, Klaus—besides his R&D job—taught at the Danish Technical University. The subjects were initially object oriented programming (C++ and Java), and later the Internet Protocol Stack. Today he is R&D Manager in Instrumentation at Brüel & Kjær Sound and Vibration.

Acknowledgments

I am very grateful to Stuart Douglas for his scouting—bringing this book into the De Gruyter family. Thanks to my editor, Jeffrey Pepper, for his patient scrutinizing work—my overuse of "initial caps" and hyphens alone did not make this easy. Jeffrey has provoked many improvements to the text and figures that enhances the reading experience. Likewise, I want to thank Paul Cohen for his good in-depth technical review. Finally, thanks to my family for listening patiently and for their acceptance of my many hours by the PC.

Klaus Elk

Contents

7 Networks

7.1 Introduction to the internet protocols

"The good thing about standards is that there are so many to choose from."

The above quote is amusing because people agree that using standards is great, but nevertheless new standards keep popping up all of the time. This is especially true when it comes to protocols. However, most of these protocols are "application layer" protocols. The "internet protocol stack" has proven victorious—and it is the core of IoT. Instead of going through a myriad of application protocols, we will focus on the internet protocol stack, especially TCP. Look at any IoT application protocol and you will find TCP just beneath it (on few occasions UDP).

If TCP does not work seamlessly, the application protocol won't work either. Unfortunately, a badly designed application protocol *can* slow a system down; see Section 7.20. This chapter also introduces REST in Section 7.12, an important concept used in many new protocols.

7.2 Cerf and Kahn-internet as net of nets

We honor Vinton Cerf and Robert Kahn as the inventors of the internet, which goes back to the 1960s, long before the world wide web. The simple beauty of their concept is the realization that instead of fighting over which local network is best, we should embrace them all, and build a "virtual" net on top of these local networks. Hence the term "inter-net."

The internet protocol stack is shown in Figure 7.1.

The term "protocol stack" means that a number of protocols are running on top of each other—using the layering pattern (see Section 4.3). The layers are normally numbered 1–5 from the bottom. Thanks to Cerf and Kahn, we have the "virtual" IP addresses on layer 3. The application is layer 5. Figure 7.1 here says "your code" as a typical application. However, the application layer can be very busy, consisting of

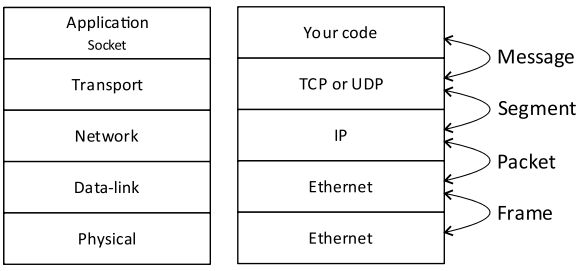


Figure 7.1: The generic layers, typical variant and PDU names.

https://doi.org/10.1515/9781547401024-007

many layers by itself. In a web server, you will find HTTP as one of the lower layers in the application, but nevertheless in layer 5 in this model. Devices that run applications that are communicating are called "hosts." It doesn't matter whether they act as client or server at the application layer.

Typically, we have the Ethernet at the lowest two layers, with the physical 48-bit MAC address at the data-link layer, and the copper or fiber cable at the physical layer. The MAC address is typically fixed for a given physical *interface, not device*, and is written as six bytes in hexadecimal, separated by ":". This address is like a social security number; it does not change even though the device, for example, a laptop, is moved from home to work. Inside a "subnet," effectively behind a router, the devices are communicating with each other using local-net addresses on layer 2. Today, in most cases these are Ethernet MAC addresses. Devices communicating on layer 2 are called "nodes." A PC and a smartphone are both nodes and hosts. A router remains a node only, until we start communicating directly to an embedded web server inside it. Then it also becomes a host.

7.3 Life of a packet

In this section, we follow a packet from a web browser to a web server. The routing described, is no different from the communication between a cloud server and an embedded device in the field. The term "packet" is not precise, but very common. As shown in Figure 7.1, the various layers each have their own name for the packet, as it grows on its way down through the stack, while each layer adds a header, and in case of the Ethernet layer also a tail. The correct generic term for a "packet" on any layer is *PDU* (protocol data unit).

Figure 7.2 can be seen as a generic setup, demonstrating hosts, switches and routers. It can also be seen as very specific. If you have 4 to 6 ports on the switch and put it in the same box as a two-port router, you have a standard SOHO (small office home office) "router." The LAN connecting the switch to the host might even be wireless. This is actually the setup used for the capture we will dig into now. The relevant interfaces are marked with circles. In Chapter 9 we will look more into the differences between wired and wireless LAN, but in this chapter there is no difference. When a host on a local network needs to contact a host outside the network, it will first send the packet to its "gateway" router, using the router's MAC address. In the case of Figure 7.2, the router interface towards the LAN has IPv4 address 192.168.0.1 and MAC address 00:18:e7:8b:ee:b6.

Figure 7.2 shows the web browser's request from a web client to a web server (from right to left). These are both termed hosts, because they "terminate" the path on the application level, as well as on the transport level (TCP). This is shown as the dashed lines between the two applications, and between the two transport layers. The messages seem to go from application to application, even though they in reality pass down the stack at the sender side and up at the receiver side. Likewise, the two TCP

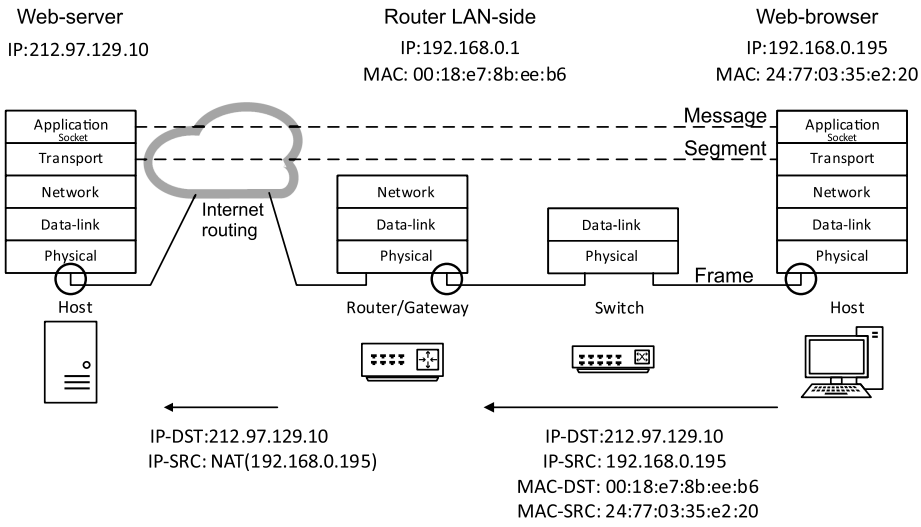


Figure 7.2: Internet with main network devices.

processes seem to communicate segments. The addresses of the interfaces are noted at the top. The packet follows the solid line, and is "stored and forwarded" by the switch as well as by the router/gateway. On the other side of the router we could have more LAN, but in this case we move into the actual internet. The web server is also connected to a gateway, but this is not shown. The router has its hands on several things in the packet from the client:

- The network-layer source containing the client's IP address.
  IP addresses remain unchanged for the lifetime of the packet—*unless* the router contains a NAT, in which case the source IP will be changed when going towards the Internet—see Section 7.9. Most SOHO routers does contain a NAT, and this is no exception.
- The network-layer destination containing the web server's IP address.
  IP addresses remain unchanged for the lifetime of the packet—*unless* the router contains a NAT, in which case the destination IP will be changed when going towards the local LAN—see Section 7.9. Our packet going from left to right will thus keep its destination IP address, but the answer coming back from right to left will be changed here. Generally a router looks up the remote IP address in its router table in order to decide which interface to transmit the package onto. In the case of a SOHO router there is only one choice.
- The link-layer source containing the client's MAC address.
  If the router is connected to LAN on the outgoing interface, the source MAC address is replaced with the router's own MAC address on this interface, as it is now the new link-layer source. In our case the router is an Internet gateway and thus uses Internet routing, which is not the subject of this book.

- The link-layer destination containing the routers MAC address.
  If the router is connected to LAN on the outgoing interface, the destination MAC address is replaced with the web server's MAC address—or the next router's MAC address—whichever comes first. In our case the router is an Internet gateway and thus uses Internet routing, which is not the subject of this book.
- The hop count.
  If IPv4 is used, the so-called "hop-count" is decremented. When it reaches zero, the packet is thrown away. This is to stop packets from circling forever.
- The checksum.
  Changing hop count or an IP address (due to NAT) introduces the need to change the IPv4 checksum.

The bottom of Figure 7.2 shows the source and destination addresses in the packet, as it moves from right to left. The data is a simple GET request. IP4 addresses are used for simplicity, and the 192.168.x.y address is a typical NAT address; see Section 7.9.

The switch is completely transparent, changing nothing in the packet. The basic switch has no IP or MAC address. The only way you may recognize its presence, is by the "store-and-forward" delay it introduces when it first waits for the whole packet to be "clocked in," and then "clocked out" on another port. This is actually a little confusing. One might think that since the router, a layer 3 device, changes the address on layer 2 (the link-layer), so does the switch which *is* a layer 2 device, but it doesn't. This transparency is what makes a switch a fantastic plug'n play device, contrary to a router, which requires a lot of configuration.

When the web server responds to the request from the client, it simply swaps sources with destinations on both levels, and the whole thing is repeated, now going from left to right. It actually swaps one more source/destination pair: the TCP ports. These are not mentioned above, as the router doesn't care about them. They are on a layer higher than what a router understands (also here an exception is made for the NAT case). An official web server is always port 80, while the client's TCP port is carefully randomly chosen. This we will get back to.

Figure 7.3 shows the scenario from Figure 7.2, now captured on Wireshark.

Each line in the top window in Wireshark is a "frame," which is the transmitted unit on the Ethernet layer. The relation between frames (in the Ethernet layer), packets (in the IP layer), segments (in the transport layer), and messages (in the application layer) can be tricky. Several small messages may be added into one segment, while large messages may be split over several segments. With TCP, the boundaries between messages from the application layer disappear due to the "stream" nature of TCP. A TCP segment may theoretically be up to 64 kBytes. However, TCP has a "Maximum Segment Size" parameter. This is normally set so that a single segment can fit into an Ethernet frame. When this is the case, as it is in most samples in this book, there is a 1:1 correspondence between a segment in TCP and a frame in the Ethernet.

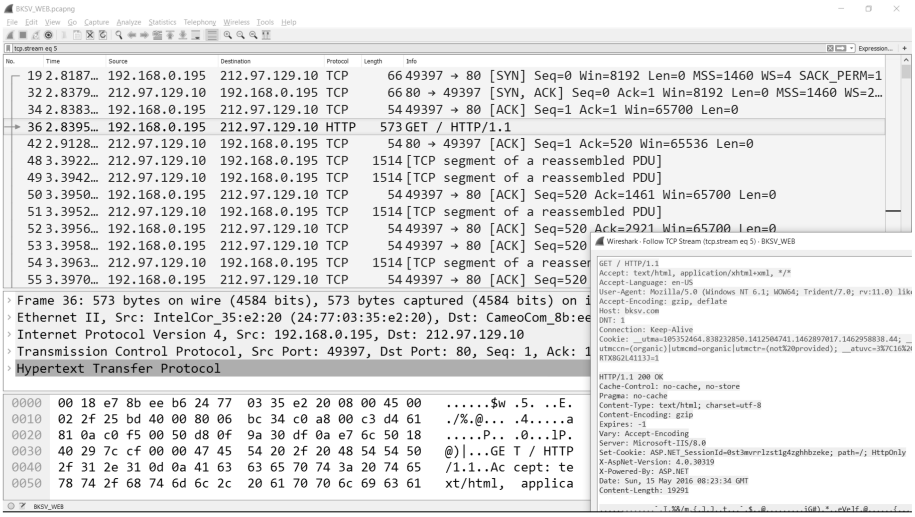


Figure 7.3: Transmission with HTTP using "follow TCP-Stream."

We only see the relevant conversation in Figure 7.3, which is why many frame numbers are missing in the left column in the top window. The filter for the conversation was easily created by right-clicking on the frame with the HTTP request (no. 36), and selecting "Follow TCP-Stream" in the context-sensitive menu. This again was easy as Wireshark by default fills the "Protocol" field with relevant data from the "highest protocol" it knows, in this case HTTP. You can sort on this column by clicking the header, and thus quickly find a good place to start analyzing.

Thus finding the "GET" request was simple. Another result of the "Follow TCP-stream," is the overlaid window at the right, showing the HTTP communication in ASCII by default. It even colors the client part red and the server part blue. This is very nice, but can be a little confusing as we see the full conversation, not just the selected frame. Thus the dialog includes the following frames. In the info-field, these contain the text "[TCP segment of reassembled PDU]."

The middle window shows the selected frame (no. 36). The bottom window shows headers and data in hexadecimal. A nice feature is that if you select something in the middle window, the corresponding binary data is selected in the bottom window. Notice that Wireshark shows the internet protocol stack "bottom up" with the Ethernet on top and HTTP at the bottom. Each of these can be expanded, as we shall see later.

Each of the nonexpanded lines in the middle window still show the most important information: TCP ports, IP addresses, and MAC addresses. As the latter are handed out in ranges, Wireshark often recognizes the leftmost 3 bytes, and inserts a vendor name in one of the two versions of the same MAC address it writes. This makes it easier to guess which device you are actually looking at. In this case, the client is a PC, with an Intel MAC.

HTTP doesn't just fly out of the PC in frame no. 36. Notice frames 19, 32, and 34. Together they form the *three-way handshake* of TCP that initiates a TCP connection. You can start a Wireshark capture at any time during a transmission, but it's best to catch the three-way handshake first—it contains some important information, as we shall soon see.

Let's get some terms right: the *TCP client* is the host that sends the first frame (19) with only the SYN flag set, and the *TCP server* is the one that responds with both SYN and ACK set (frame 32). The terms client and server are related to TCP, but they correspond to the web browser and the web server. In theory, a communication could open on another TCP connection in the opposite direction, but this is not the case here.

The terms "sender" (or transmitter) and "receiver" are *not* the same as server and client, they are more dynamic. You may argue that most information is going from the server to the browser, but certainly not all. In a TCP connection, application data may (and typically will) flow in both directions. It is up to the application layer to manage this. However, let's see what we may learn from the initial handshake, via the "info" field on frame 19 in Wireshark in Figure 7.3:

- *SYN Flag*
  This flag (bit) is set by the TCP client, only in the initial opening request for a TCP. The answer from the TCP server on this particular packet, also contains the SYN flag.
- *ACK Flag* [not in frame 19]
  Contained in all frames except the client's opening SYN. This allows you to tell the TCP client from the TCP server.
- *Seq=0*
  The 32-bit *sequence number* of the first byte in the segment sent. Many protocols number their packets, but TCP numbers its bytes. This allows for smarter retransmissions—concatenating smaller packets that were initially sent one-by-one as they were handed to TCP from the application layer. Note that the SYN flag and the closing FIN flag (outside the screen), count as bytes in this sense. Because we got the transmission from the initial three-way-handshake, Wireshark is nice, giving us relative sequence numbers (starting from 0). If you open the bottom hexadecimal view, you will see that sequence numbers do not start from 0. In fact, this is an important part of the security. Sequence numbers are unsigned and simply wrap.
- *Ack* [not in frame 19]
  The sequence number that this client or server expects to see next from the other side. As either side will sometimes send ACKs with no data, there is nothing wrong in finding several frames with the same Seq or Ack no. Wireshark will help you and tell you if either side, or Wireshark itself, has missed a frame.
- *Win=8192*
  This 16-bit *window size*, used by both sides, tells the other side how much space it currently has in its receive buffer. Thus the given sender knows when to stop trans-

mitting, and wait for the receiver to pass the data up to the application above it. As the receiver guarantees its application that data is delivered exactly once, without gaps, and in order, this buffer may easily become full if an early packet is lost. Not until the gap is filled by a retransmission, will the data go to the application. This is the reason why you sometimes will see packets that are data-less, but are marked by Wireshark as a "Window Update." This means that the receiver (either client or server) wishes to tell the other side: "I finally got rid of some data to my application, and I now have room for more."

- *WS=4*

  This is the *window scale*. TCP is old, and originally a 16-bit number was thought to be enough for the window size. But today's "long-fat-pipes" are only utilized decently if we allow for a lot of data on-the-fly. The backward-compatible fix was to introduce a *window scale* factor. This is an option in the opening three-way-handshake. If the client uses this option, the window scale contains the number of bits the future window size numbers *could* be left-shifted from the client. If the server responds by also using this option, this is an acknowledgment that the client's suggestion is accepted. Furthermore, the window scale sent from the server, is the scale it will use in its window size. Thus the two window scales do not have to be the same. All this explains why Wireshark often reports a window size larger than 65535, even though the window size is a 16-bit number. This is the main reason why you should always try to get the handshake included in the capture.
- *SACK_PERM=1*

  SACK_PERM means *selective acknowledge permitted*. The original implementation of TCP is rather simple. The ACK number can be understood as "This is how much data I have received from you without gaps." The transmitter may realize it has sent something, not seen by the receiver, and it will retransmit all bytes from this number and forward. However, with the "big-fat-pipes," the sender may need to resend a lot of data already caught by the receiver. With SACK_PERM, the host says it is capable of understanding a later addendum, the selective acknowledge. This allows the receiver to be more specific about what is has received and what not. The ACK no still numbers the first not-seen byte and *all* bytes before this are safely received, but with the help of SACK, the receiver can tell the transmitter about some "well-received" newer data blocks after a gap. This means that less data is retransmitted, saving time at both ends, as well as utilizing the network better.
- *MSS=1460*

  MSS means *maximum segment size* and is the maximum size of the payload in the TCP segment (the data from the application) in bytes. This is related to the MTU; see Section 7.17.

If segments longer than MSS are sent, they may get there, but the overhead will affect performance. This is where packets on the IP[1] layer become "fragmented" into more frames in the Ethernet layer, as discussed in Section 7.14.

– *49397->80*
These are the port numbers used with the sender's first.

Amazing how much can be learned from studying the Wireshark "info" on a single frame. Wireshark shows us the TCP connection as well as its initiation and end. All this is clearly visible on the network between two hosts—and known by both ends.

A "TCP socket" is an OS construction offered to the application programmer, handling the TCP details on one end of a TCP connection. In Chapter 2, we saw a central part of a state-machine implementation of a socket. The socket does the book-keeping on sequence-numbers, the numbers of ACKs received, time left before next timeout and much more.

In our case, the client's port number was 49397. This is the *ephemeral* port number—a random number selected by the operating system. It is a part of the security on plain connections (non-SSL) that this port number really *is* random. The socket is registered in the OS by the two IP addresses, the two port numbers and the TCP protocol. Together these five numbers constitute a *5-tuple*. Whenever the OS receives a packet, this 5-tuple is extracted from the packet and used to lookup the correct socket in the OS. The communicating hosts network interfaces dictate the IP addresses, and as the server is a public web server, the protocol is TCP and the server's port number is 80. The client port number is thus the only of these five numbers not "written in stone." If nobody knows the next port number, it makes the system somewhat less sensitive to "injection attacks," where bandits send fake packets.

A real and a fake packet only differ in the payload part—the data from the application layer. The first packet received "wins," and the other one is thrown away as an unneeded retransmission. A bandit on the network path might for example try to reroute the client to another website. If the client port number can be guessed, the bandit has good time to generate the fake packet and win the race. Clearly, this is not rock-solid security, but a small part of the picture. We will meet the much safer SSL in Section 10.12.

7.4 Life before the packet

When for example, a PC on a local Ethernet is told to talk to another PC on the same local net, the PC is given an IP address to talk to. However, the two nodes[2] need to communicate via their MAC addresses. Typically, the PC knows the MAC address of the gateway router from the original DHCP, but what about other hosts?

1 This only happens in version 4 of IP, not in version 6.

2 We use the term "host" on the application layer, but devices communicating on the Ethernet layer are called "nodes."

This is where ARP (address resolution protocol) comes into play. A host keeps a list of corresponding IP and MAC addresses, known as an ARP cache. If the IP address of the destination is not in the ARP cache, the host will issue an ARP request, effectively shouting: "Who has IP address xx?" When another node replies, the information is stored in the ARP cache. The replying host may also put the requesting node in its ARP table, as there apparently is going to be a communication.

Listing 7.1 shows a PC's ARP cache. All the entries marked as "dynamic" are generated as just described, and they all belong to the subnet with IP addresses 192.168.0.x. The ones marked "static" are all (except one) in the IP range from 224.x.y.z to 239.x.y.z. This is a special range reserved for "multicasting," and this is emphasized by their MAC addresses, all starting with 01:00:5e and ending with a bit pattern, equal to the last part of the IP address.[3] The last entry is the "broadcast" where all bits in both MAC and IP address are 1. Such a broadcast will never get past a router. Note that in Listing 7.1 there is no entry for 192.168.0.198.

Listing 7.1: ARP cache with router at top

```
C:\Users\kelk>arp -a

Interface: 192.168.0.195 --- 0xe
  Internet Address      Physical Address      Type
  192.168.0.1           00-18-e7-8b-ee-b6     dynamic
  192.168.0.193         00-0e-58-a6-6c-8a     dynamic
  192.168.0.194         00-0e-58-dd-bf-36     dynamic
  192.168.0.196         00-0e-58-f1-f3-f0     dynamic
  192.168.0.255         ff-ff-ff-ff-ff-ff     static
  224.0.0.2             01-00-5e-00-00-02     static
  224.0.0.22            01-00-5e-00-00-16     static
  224.0.0.251           01-00-5e-00-00-fb     static
  224.0.0.252           01-00-5e-00-00-fc     static
  239.255.0.1           01-00-5e-7f-00-01     static
  239.255.255.250       01-00-5e-7f-ff-fa     static
  255.255.255.255       ff-ff-ff-ff-ff-ff     static
```

Listing 7.2 shows an ICMP "ping" request, the low level "are you there?" to the IP address 192.168.0.198, followed by a display of the ARP cache (in the same listing), now with an entry for 192.168.0.198.

Listing 7.2: ARP cache with new entry

```
C:\Users\kelk>ping 192.168.0.198

Pinging 192.168.0.198 with 32 bytes of data:
Reply from 192.168.0.198: bytes=32 time=177ms TTL=128
Reply from 192.168.0.198: bytes=32 time=4ms TTL=128
```

3 This is not so easy to spot as IPv4 addresses are given in decimal and MAC addresses in hexadecimal.

```
Reply from 192.168.0.198: bytes=32 time=3ms TTL=128
Reply from 192.168.0.198: bytes=32 time=5ms TTL=128

Ping statistics for 192.168.0.198:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 3ms, Maximum = 177ms, Average = 47ms

C:\Users\kelk>arp -a

Interface: 192.168.0.195 --- 0xe
  Internet Address      Physical Address      Type
  192.168.0.1           00-18-e7-8b-ee-b6     dynamic
  192.168.0.193         00-0e-58-a6-6c-8a     dynamic
  192.168.0.194         00-0e-58-dd-bf-36     dynamic
  192.168.0.196         00-0e-58-f1-f3-f0     dynamic
  192.168.0.198         10-7b-ef-cd-08-13     dynamic
  192.168.0.255         ff-ff-ff-ff-ff-ff     static
  224.0.0.2             01-00-5e-00-00-02     static
  224.0.0.22            01-00-5e-00-00-16     static
  224.0.0.251           01-00-5e-00-00-fb     static
  224.0.0.252           01-00-5e-00-00-fc     static
  239.255.0.1           01-00-5e-7f-00-01     static
  239.255.255.250       01-00-5e-7f-ff-fa     static
  255.255.255.255       ff-ff-ff-ff-ff-ff     static
```

Figure 7.4 shows a Wireshark capture of the above scenario.

The first of three ping requests is sent in frame 42, with a reply in frame 45. Before this, however, we see the ARP request from the pinging PC in frame 40, and the

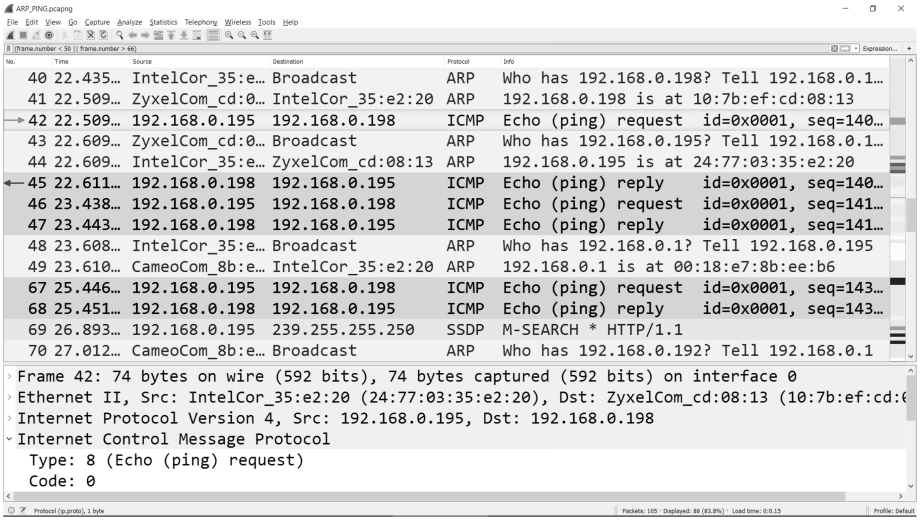


Figure 7.4: A ping that provokes ARP.

response in frame 41. Notice that the source and destination addresses here are not IP addresses, but Ethernet addresses (partly filled with the vendor names of the MACs). Also note that the request naturally is a broadcast, while the reply is a "unicast." The response could also have been a broadcast, but a unicast disturbs less.[4]

Here, we see an explanation for the extended time for the first ping. The responding Windows 7 PC, decided to do its own ARP, 100 ms after having answered the original ARP. This is a defense against "ARP poisoning": If we see more than one device answering this second ARP, it is a sign that someone might be trying to take over one side of the communication. As a side note, the display filter used in Wireshark is somewhat nontraditional. To avoid a swarm of unrelated traffic in the figure, it was easy to filter based on frame numbers.

7.5 Getting an IP address

Network interfaces are born with a unique 48-bit MAC address. This is like a person's social security number, following the interface for life. Not so with IP addresses. IP addresses are hierarchical to facilitate routing. This is similar to letters addressed to <country>-<ZIP>-<street and number>-<floor>. This means that IP addresses may change. With mobile equipment they change all the time. There are a number of ways to get an IP address:

- *Statically configured*
  This can be good in a, surprise, static system as the address is ready from power-up and does not change. Many systems use static addresses in one of the private IP ranges (see Section 7.8) in "islands" not connected to the internet, but to a dedicated network interface card (NIC) on a PC.
  Like many others, my company provides large systems with racks, and modules that go into slots in the racks, see Section 4.9. In our case, each module is IP addressable. I introduced a "best practice" addressing scheme:
  192.168.<rackno>.<slotno>, using network mask 255.255.0.0.
  This makes it easy to find a module from its address.
- *DHCP*
  Dynamic Host Configuration Protocol. This is practical with laptops and phones moving from home to work to school, etc. When the device joins the network, it gets a relevant IP address and more.
- *Reserved DHCP*
  Via their web page most *SOHO* (small office/home office) routers allow you to fix the DHCP address handed out to a device, by linking it to the MAC address of the device. This is very practical as your PC, phone or whatever, can remain a

4 Unicasts, multicasts and broadcasts are explained in Section 7.18.

DHCP client wherever you go, and still you can be sure to always have the same IP address in your home. This is practical for, for example, development servers on a laptop.

– *Link Local*
  When a device is setup as a DHCP client, but cannot "see" a DHCP server, it waits some time and finally selects an address in the range 169.254.x.y. First, it tests that the relevant candidate address is free, using ARP, and then it announces the claim with a "Gratuitous ARP." See Section 7.4. This is a nice fall-back solution, allowing two hosts configured as DHCP clients, to communicate even without a DHCP server—*if* you can get past all the security in firewalls and other security devices. Microsoft calls link-local for "Auto-IP."
  Should the DHCP server become visible after a link-local address is selected, the device will change IP address to the one given from the server. This can be very upsetting.
– *Link-Local IP6*
  IPv6 includes another version of link-local that makes the address unique, based on the MAC address on the interface.

7.6 DHCP

Listing 7.3 shows how the DHCP process is easily provoked on a Windows PC.

Listing 7.3: DHCP release and renew

```
1 C:\Users\kelk>ipconfig /release
2 ...skipped...
3 C:\Users\kelk>ipconfig /renew
```

Figure 7.5 shows the Wireshark capture corresponding to Listing 7.3. This time no filter was applied. Instead Wireshark was told to sort on the "protocol" by clicking on this column. This organizes the frames alphabetically by the name of the protocol, and secondarily by frame number. Frame by frame, we see:

– Frame 6: "DHCP Release" from the PC caused by line 1 in Listing 7.3. Here, the PC has the same address as before: 192.168.0.195. After this frame, it is dropped.
– Frame 108: "DHCP Discover" from the PC. This is a broadcast on the IP level, which can be seen by the address: 255.255.255.255. It is also a broadcast on the Ethernet level with the address: ff:ff:ff:ff:ff:ff. This is a request for an IP address from any DHCP server. Note the "Transaction ID": 0x1eae232a, not the same as in the release in frame 6.
– Frame 120: "DHCP Offer" from the server. This is sent as a broadcast since the PC has restarted the process and, therefore, has no address yet. However, this offer contains the MAC address of the PC as well as the original Transaction ID. It cannot be received by mistake by another client. The offer contains the future IP address

and mask, as well as the IP address of the DNS server (see Section 7.10) and the gateway router. It also includes the "lease time," which is how long the PC may wait before it asks for a renewal.

- Frame 121: "DHCP Request" from the PC. Why, we just had the whole thing handed on a silver plate. The explanation is that another DHCP server might simultaneously have offered us an address. This packet contains more or less the same as the offer, but now from the PC to the server. There is one extra thing: the "client fully qualified domain name" which is the "full computer name" also found in the "system" group in the control panel on Windows PCs. This explains why my home router identifies my company laptop with my company's domain, even though I am currently not on this domain.
- Frame 123: "DHCP ACK" from the server. This means that finally, the deal is sealed.
- Frame 779: "DHCP Inform" from the PC. This is a new message in relation to the standards. According to IETF, it was introduced to enable hosts with a static IP address to get some of the other information passed with DHCP (DNS, etc.). Also according to IETF, this has been seen to be kind of misused. This appears to be one such case.
- Frame 780: "DHCP ACK" from the server. Answer to the above.

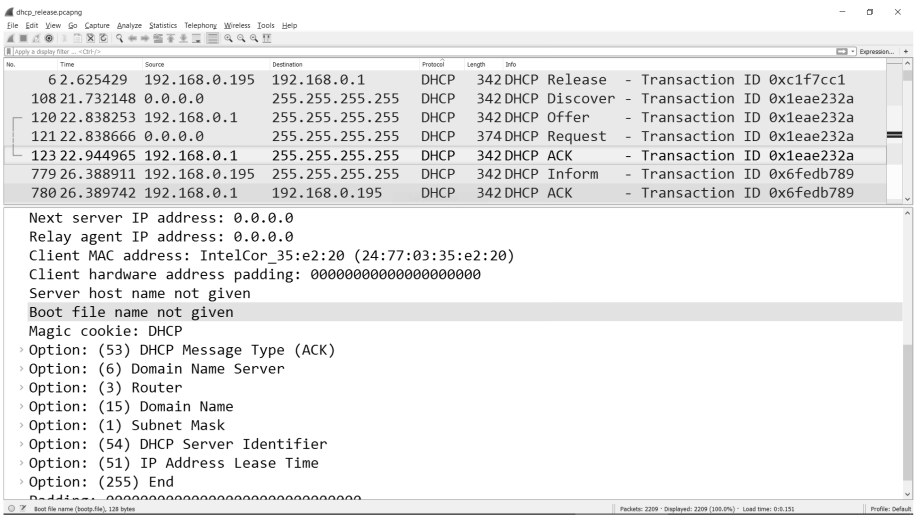


Figure 7.5: DHCP commands and options.

When a DHCP release is about to run out, or "ipconfig /renew" is used without the "release," only the DHCP Request from the PC and the DHCP ACK from the server is seen. In this case, they are unicast. In other words, the PC does not "forget" its address but uses it while it is renewed. If you would like to try some experiments, Section 5.9 contains a Python script for simulating the client side of DHCP.

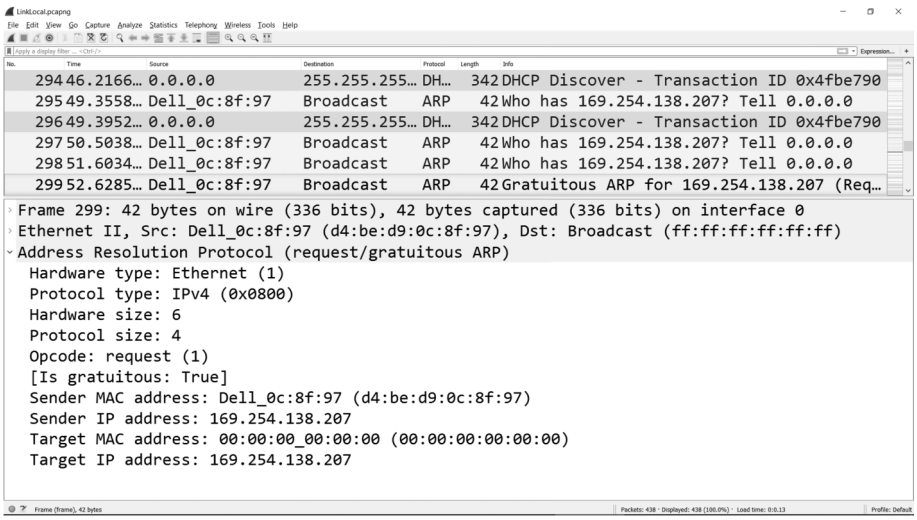


Figure 7.6: Link-local with gratuitous ARP.

Figure 7.6 shows the link-local scenario where the DHCP client cannot see any server. Frames 294 and 296 are the last attempts on DHCP. With frame 295, the PC starts ARP'ing for 197.254.138.207. This is what the protocol stack inside the PC is planning on using, but only if its not already in use. After the third unanswered ARP, we see a "Gratuitous ARP." This is the PC using the ARP protocol, not to ask for anyone else with the link-local address, but instead informing its surroundings that the address is about to be taken—unless someone answers to it. Frame 299 is the "selected" and in the window below we see that "Sender Address" is 169.254.138.207. In other words, the PC is asking for this address, and saying it has it. Hence the term "gratuitous."

7.7 Network masks, CIDR, and special ranges

We have touched briefly on network masks, and many people working with computers have a basic understanding of these. However, when it comes to the term "subnet" it gets more fluffy. It might help to do a backward definition and say that a subnet is the network island you have behind a router. As discussed in Section 7.3, frames inside such an island are sent by their Ethernet address, using ARP cache and ARP protocol. Inside the island, all IP addresses share the same first "n" bits, defined by the mask.

A typical mask in a SOHO (small office/home office) installation is 255.255.255.0. This basically means that the first 24 bits are common, defining the network ID, this being the subnet. The last 8 bits are what separates the hosts from each other—their host ID. Since we do not use addresses ending with "0", and the host ID address with all bits

set is for broadcasts within the subnet, this leaves 254 possible host IDs in the above example. Network masks used to be either 255.0.0.0 or 255.255.0.0 or 255.255.255.0—and the networks were respectively named class A, B, and C. These terms are still used way too much. In many companies, class C was too little, while class B was too much, and who needs class A? A lot of address space was wasted this way, as corporations were given a class B range to use at will. While everybody was waiting for IPv6 to help us out, CIDR and NAT were born.

CIDR is *classless inter-domain routing*. It allows subnet definitions to "cross the byte border." To help, a new notation was invented where "/n" means that the first n bits of the given address is the subnet, and thus defines the subnet mask as the leftmost n bits, leaving 32 minus n bits for the host ID. The IP address 192.168.0.56/24 is the same as a classic class C mask, and thus the host ID is 56.

However, we can also have 192.168.0.66/26, meaning that the mask is 255.255.255.192, the network ID is 192.168.0.64 and the host ID is 2. Relatively easy numbers are used in these examples, but it quickly becomes complicated, which is probably why the old masks are die hards. There are however many apps and homepages offering assistance with this.

7.8 Reserved IP ranges

We have seen addresses like 192.168.0.x numerous times. Now, let us take a look at what is known as "reserved IP ranges" (Table 7.1). Packets destined for an IP address in a "private range" will not get past a router. This makes them perfect for the home or company network, as they are not blocking existing global addresses. They can be reused again and again in various homes and institutions. Another advantage of these addresses not being routed, is that they cannot be addressed directly from the outside of the home or office, thus improving security. The "multicast range" is also known as "class D." There are more ranges, but the ones shown in Table 7.1 are the most relevant.

Table 7.1: The most important reserved IP address ranges.

CIDR	Range	Usage
10.0.0.0/8	10.0.0.0–10.255.255.255	Private
169.254.1.0/16	169.254.1.0–169.254.255.255	Link-local
172.16.0.0/12	172.16.00–172.31.255.255	Private
192.168.0.0/16	192.168.0.0–192.168.255.255	Private
224.0.0.0/4	224.0.0.0–239.255.255.255	Multicast
240.0.0.0/4	240.0.0.0–255.255.255.254	Future use
255.255.255.255	Broadcast	Subnet only

7.9 NAT

NAT (network address translation) has been extremely successful, at least measured by how it has saved IP addresses. Most households need to be able to act as TCP clients against a lot of different servers, but very few households have a server which others need to be client against. These households are in fact mostly happy (or ignorant) about not being addressable directly from the internet. A NAT assures that we on the inside can live with our private IP addresses, while on the outside we have a single IP address. This external address may even change once-in-awhile without causing problems, again because we have no servers. So what does it do?

Say that the external address of a company or home is 203.14.15.23 and on the inside we have a number of PCs, tablets, and phones in the 192.168.0.0/16 subnet. Now the phone with address 192.168.12.13 uses a web browser to contact an external web server (port 80)—its source port number being 4603. The NAT opens the packet and swaps the internal IP address with the external. It also swaps the source port number with something of its own choice. In this case, port 9000; see Table 7.2. This scheme allows multiple internal IP addresses to be mapped to one external IP address by using some of the port number space. The packet moves to the web server where the request is answered and source and destinations of ports and IP addresses are swapped as usual. An answer is routed back to the external address where the NAT switches the destination IP and port back to the original (private) source IP and port. If the NAT originally also stored the external address (rightmost column in table), it can even verify that this in-going packet indeed is an answer to something we sent. The NAT is said to be stateful in this case.

Table 7.2: Sample NAT table.

Intranet	NAT	Internet
192.168.12.13:4603	203.14.15.23:9000	148.76.24.7:80
192.168.12.13:4604	203.14.15.23:9001	148.76.24.7:80
192.168.12.10:3210	203.14.15.23:9002	101.23.11.4:25
192.168.12.10:3211	203.14.15.23:9003	101.23.11.4:25
192.168.12.28:7654	203.14.15.23:9004	145.87.22.6:80
Nothing	Nothing	205.97.64.6:80

Table 7.2 is using the standard notation <IP>:<Port>. To make it easy to see what is happening, the NAT ports are consecutive, which they are not in real life. All rows, except the last, show packets generated internally, going out to web servers or mail servers and being answered. The last row shows how an external bandit attempts to fake a packet as a response to a web request. As there is no match it is dropped.

The basic NAT functionality only requires the NAT to store the two left columns (thus being stateless). By looking at the incoming TCP segments from the internet, the NAT can throw away any segment that hasn't got the "ACK" bit set. Such a segment can only be a client opening "SYN." As the installation has no servers, we disallow all SYNs from the outside; see Section 7.13.

By storing the third column as well, security is improved. The NAT concept can also be used on server parks for load balancing when the protocol is stateless; see Section 7.12. The mappings shown in the table are *dynamic*, as they are generated from traffic. It is possible to create static mappings. If we do want a PC on the intranet to act as a company/household web server, the static mapping will send all packets towards port 80 to this PC. This way we are able to have a server inside our network. In this case, we now *will* prefer that the external IP address is fixed, as this is what we tell DNS servers is our address.

Not everyone is happy about NAT. It conflicts with the concept of layering, see Section 4.3. If an application embeds information about ports and IP addresses in the application protocol, this is not translated, and will cause problems.

7.10 DNS

The domain name system was invented because people are not really good at remembering long numbers. You may be able to remember an IPv4 address, but probably not an IPv6 address. With the DNS we can remember a simple name like "google.com" instead of long numbers.

This is not the only feature. DNS also gives us a first-hand scaling ability assistance. Listing 7.4 was created on a Windows 10 PC.

Listing 7.4: Simple nslookup

```
C:\Users\kelk>nslookup google.com
Server:  UnKnown
Address:  192.168.0.1

Non-authoritative answer:
Name:    google.com
Addresses:  2a00:1450:4005:80a::200e
          195.249.145.118
          195.249.145.114
          195.249.145.98
          195.249.145.88
          195.249.145.99
          195.249.145.103
          195.249.145.113
          195.249.145.109
          195.249.145.119
          195.249.145.84
```

```
18              195.249.145.123
19              195.249.145.93
20              195.249.145.94
21              195.249.145.89
22              195.249.145.108
23              195.249.145.104
```

When looking up google.com, we get 16 IPv4 answers (and one answer with an IPv6 address). A small server park could use this to have 16 different servers, with separate IP addresses for the same URL. Most programmers are lazy and are simply using the first answer, and for this reason the next call will show that the answer is "cycled." This is a simple way to "spread out" clients to different physical servers. Google does not need this kind of help, they are masters in server parks, but for many smaller companies, this means that they can have a scalable system with a few web servers, simply by registering them to the same name.

In order for a URL to be translated to an IP address by the DNS, it needs to be registered. This can be done via many organizations, as a simple search will show. Once registered, the URL and the belonging IP addresses are programmed into the 13 root DNS servers. When your program asks its local DNS resolver about a URL, the request will be sent upwards through the hierarchy of DNS servers until it is answered. Answers are cached for some time in the servers they pass. This time is the TTL (time-to-live) which is set by the administrator.

The actual DNS request is using UDP. This is the fastest way, with no "social throttling" (see Section 7.18), and these requests are so small they can easily fit into a single UDP segment—no matter which protocol is beneath. DNS requests come in two forms, *recursive* or *iterative*. In the recursive case, the DNS server takes over the job and queries further up the hierarchy, and eventually comes back with a result. In the iterative case, the local resolver must ask the next DNS server in the hierarchy itself, using an IP address for this which it got in the first answer, etc.

Figure 7.7 is a Wireshark caption of an older "googling," with some completely different answers (compared to earlier).

The selected frame is no. 2, containing the answer. We see that the request asked for, and got, a recursive query. We also see that in this case we got four answers. In frame 3, we see the web browser using the first (64.233.183.103) for its HTTP request. The request is of type "A" which is when we ask for an IPv4 address for a given URL. Had it been "AAAA" we would have received IPv6 addresses. There are many more types, including "PTR" used for reverse lookups.

Many smaller embedded systems operate directly on IP addresses, making the whole DNS concept irrelevant. A typical IoT system will have its cloud server registered in the DNS but normally not the devices. Alternatively, *mDNS* (multicast DNS) is used. This is used in *Bonjour* invented by Apple, and the Linux implementation of the same—called *Avahi*.

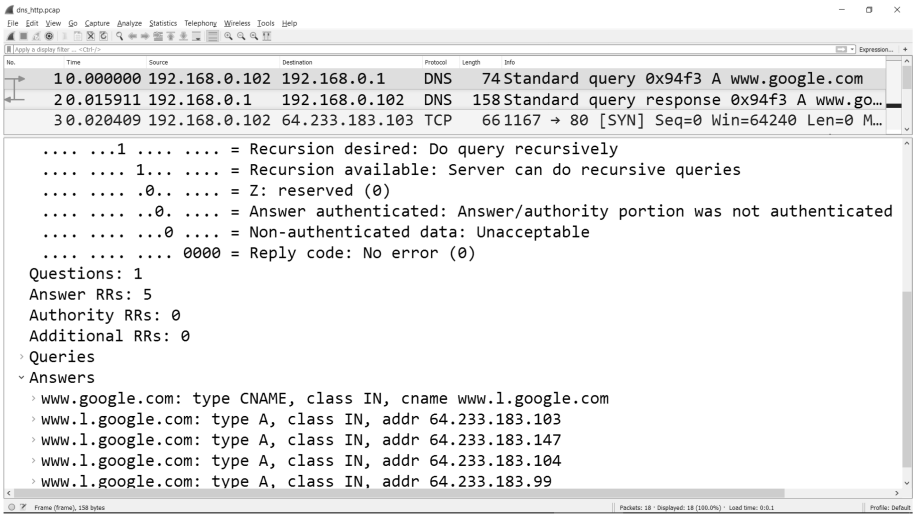


Figure 7.7: A DNS request for google.com.

7.11 Introducing HTTP

Telnet is not used very much anymore; there is absolutely no security in it. Nevertheless, it is great for showing a few basics on HTTP. Listing 7.5 shows a telnet session from a Linux PC. In modern Windows, Telnet is hidden. You need to enable it via "Control Panel"—"Programs and Features"—"Turn Windows features on or off."

Listing 7.5: Telnet as webbrowser

```
kelk@debianBK:~$ telnet www.bksv.com 80
Trying 212.97.129.10...
Connected to www.bksv.com.
Escape character is '^]'.
GET / HTTP/1.1
Host: www.bksv.com

HTTP/1.1 200 OK
Cache-Control: no-cache, no-store
Pragma: no-cache
Content-Type: text/html; charset=utf-8
Expires: -1
Server: Microsoft-IIS/8.0
Set-Cookie: ASP.NET_SessionId=nvyxdt5svi5x0fovm5ibxykq;
            path=/; HttpOnly
X-AspNet-Version: 4.0.30319
X-Powered-By: ASP.NET
Date: Sun, 15 May 2016 11:46:29 GMT
```

```
19 Content-Length: 95603
20
21
22 <!DOCTYPE html PUBLIC "-//W3C//DTD␣XHTML␣1.0␣Strict//EN"
23         "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
24 <html id="htmlStart" xmlns="http://www.w3.org/1999/xhtml"
25           lang="en" xml:lang="en">
26 <head><META http-equiv="Content-Type" content="text/html;
27 ␣␣␣␣␣␣␣charset=utf-8">
28 <title>Home - Bruel &amp;amp; Kjaer</title>
29 <meta http-equiv="X-UA-Compatible" content="IE=edge">
30 <meta name="DC.Title" content=".......”
```

We are now basically doing exactly the same thing as was done before with a browser: a HTTP-GET request in the root at www.bksv.com. We specifically need telnet to abandon its default port (23) and use port 80 instead. Once connected to the server, we can send the same lines we can see in the Wireshark communication in clear text in Figure 7.3. However, this time only the two mandatory headers are sent; see Listing 7.5, lines 5 and 6. Line 5 is no big surprise; with “www.bksv.com” written originally in the browser, and nothing more, we go to the root— “/” —and HTTP/1.1 is the standard supported by the browser.

But how come we need to send “Host: www.bksv.com?” After all, it was just written already in line 1: “telnet www.bksv.com 80.” The reason is that the URL in line 1 is substituted with an IP address before leaving the PC at all, thanks to the DNS. A modern web server installation can host many “sites” with different URLs, but with the same IP address. We need line 6 to tell the server which site we want to talk to.

Now we are done with the headers. Following the HTTP standard, we then send two line terminations (CR-LF), and then any “body.” Since this is a GET there is no body, and we therefore see the response starting in line 8. The response does have some body, which we see starting in line 22, after the two times CR-LF. This body is HTML—no surprise there. Also note line 19, stating the “Content-Length,” the number of bytes in the body, pretty much the same as a file-length.

Table 7.3 shows the well-known HTTP commands. Note how SQL-database statements suddenly popped up. In the database world, we operate with the acronym CRUD, for “Create,” “Retrieve,” “Update,” and “Delete.” This is basically enough to manage, expose and utilize information, and isn’t that exactly what IoT is about?

Table 7.3: The main HTTP commands, and what they do.

HTTP	Usage	SQL
POST	**C**reates Information	INSERT
GET	**R**etrieves information	SELECT
PUT	**U**pdates information	UPDATE
DELETE	**D**eletes information	DELETE

7.12 REST

While many of us were struggling to comprehend SOAP, the architectural concept from Microsoft and others, a dissertation for a doctorate in computer science was handed in by Roy Fielding in the year 2000. In this, he introduced an architectural concept called REST—representational state transfer. In short, this was a concept for managing a device, or resources, be it small as a sensor or huge as Amazon. He gave an important set of ground rules for REST; see Table 7.4.

Table 7.4: REST rules.

Rule	Explanation
Addressable	Everything is seen as a resource that can be addressed, for example, with an URL
Stateless	Client and server cannot get out of synch
Safe	Information may be retrieved without side-effects
IdemPotent	The same action may be performed more than once without side-effects
Uniform	Use simple and well-known idioms

Surely Roy Fielding already had his eyes on HTTP, although the concept does not have to be implemented using HTTP. There are many reasons why HTTP has been victorious. Simplicity is one-fitting the "Uniform" Criteria. However, the most important fact about HTTP is that it basically *is* stateless.

With FTP, client and server need to agree on "where are we now?" In other words, which directory. This is very bad for scalability; you cannot have one FTP server answer one request (e. g., changing directory), and then let another in the server park handle the next. This is exactly what HTTP does allow. You can fetch a page from a web server, and when opening it up, it is full of links to embedded figures, etc. In the following requests for these, it can be other servers in the park that deliver them, independent of each other, because every resource has a unique URL (addressable rule).

HTTP also lives up to the "Safe" criteria: there are no side-effects of forcing refetching of a page many times (apart from burning CPU cycles). So, out of the box HTTP delivers. An obvious trap, to avoid, is to have a command such as "Increment x." This suddenly adds state on the application level. So if you have received eight of something, and you want the ninth, then ask for the ninth, not the "next." This allows an impatient user (or application) to resend the command without side effects. It also means that the numbering of items must be well-defined.

Maybe REST is inspired by some hard learned experiences in the database world, where there are "client-side cursors" and "server-side cursors." A database system with a server-side cursor, forces the database to hold state for every client, just like FTP. With a client-side cursor, you put the load of state-keeping on the client. Clients are often smaller machines, but they are many. Thus the solution scales much better.

There are situations where a common idea of state cannot be avoided. If you are remote-controlling a car, you need to start the engine before you can drive. However, you may be able to derive that if the user wants to drive and the engine is not running, we better start it.

It is pretty clear what GET and DELETE does, but what are the differences between PUT and POST? It turns out that PUT is very symmetric to GET. A GET on a specific URL gives you the resource for this URL, and a PUT updates it. POST handles the rest.

Typically, you cannot create an object by doing something on the not-yet-existing URL. Instead you ask the "parent-to-be" in the containment tree, using POST, to create a child, and it typically gives you the exact URL of the new child. This all fits with the table spelling CRUD. The final thing we use POST for, is performing actions. These are a little bit on the "edge" of REST, but hard to do without.

Listing 7.6: Sample REST GET

```
1 http://10.116.1.45/os/time?showas=sincepowerup
```

Listing 7.6 can be written directly into a browser's address bar and demonstrates another quality in REST: it is human-readable. In fact, it is not hard to create small test scripts in your favorite language, however, only GET can be written directly in the browser. As a consequence, REST is easier to debug in Wireshark than most other application protocols. Once you have decided to use REST on HTTP, there are still a few decisions to take:

- The data-format of the body. Typical choices are XML or JSON. XML may be the one best known in your organization, while JSON is somewhat simpler.
- How to organize the resources. This is basically the "object model"; see Section 4.5. If you can create an object model, the whole development team understands and relates to, you are very far. If you can create an object model which third party developers, or customers, can understand and program against, you are even better off. The fact that you can piece together the URL by following the object model from its root is extremely simple, and very powerful.
- It is possible to address every node in the object tree specifically, reading or writing data. It might also be possible to go to a specific level in the object tree and from there handle the containment hierarchy in JSON or XML. Which concept to choose? This typically depends on how you want to safeguard the system against unwanted changes, or the actions you need to do on a change.
  You may plan on having different user roles, each with a specific set of what they may read or change, and what not. In this case, it's a good idea to address the nodes individually, so that it is the plug-in in the web browser that uniformly manages user access, instead of the individual objects. However, as discussed in Section 4.1, there is a huge performance boost in setting a large subtree in one swoop. You might end up enabling the addressing of subnodes, as well as delivering/fetching whole subtrees in JSON or XML.

7.13 TCP sockets on IPv4 under Windows

In Section 7.3, we went through the "opening ceremony" for TCP—the three-way handshake. Now is the time to go more in detail with TCP (transmission control protocol)—RFC793. The standard has a rough state diagram, which is redrawn in Figure 7.8.

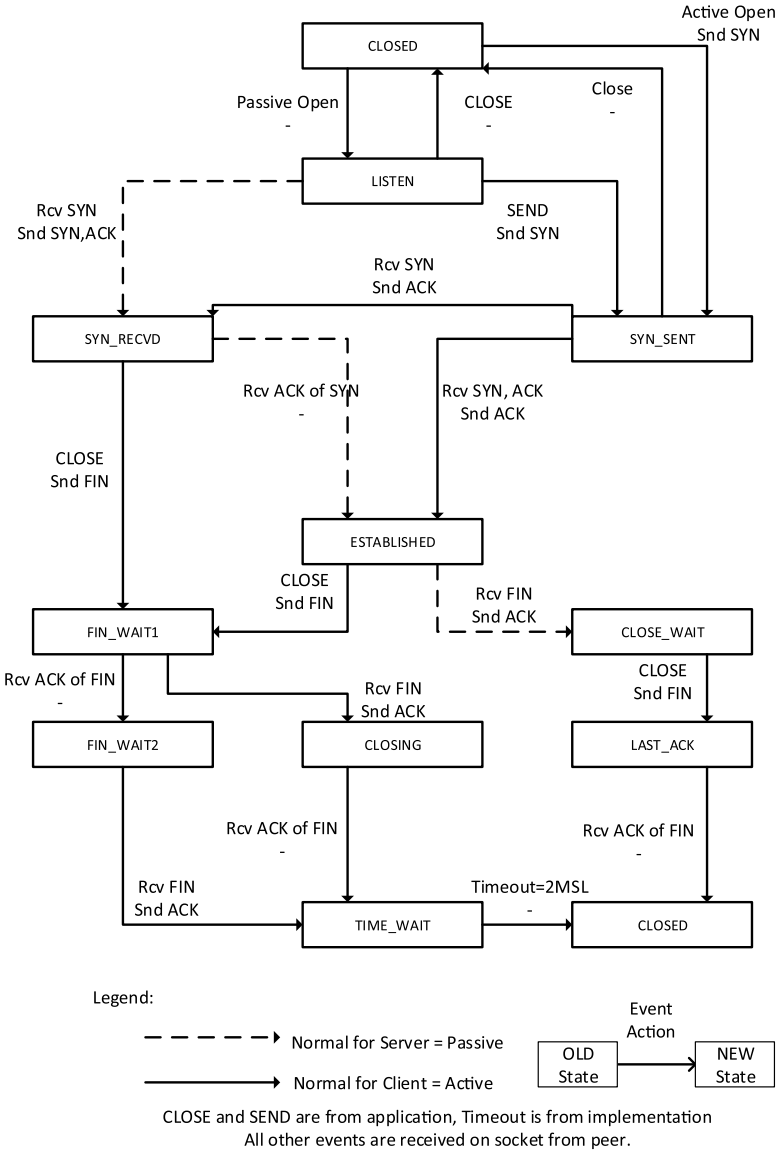


Figure 7.8: TCP state-event diagram.

The state where we want to be is "established." This is where data is transferred. The rest is just building up or tearing down the connection. Similar to the opening three-way handshake, closing is normally done using two two-way handshakes. This is because the connection can send data in both directions. Both transmitters can independently state "I have no more to say," which the other side can ACK. Older Windows stacks are lazy, they typically skip the two closing handshakes, instead sending an RST—Reset. This is bad practice.

The client is called the "active" part, as it sends the first SYN and almost always also the first FIN. The server is "passive," responding to the first SYN with (SYN, ACK)—meaning that both flags are set. We have seen this earlier. When the client application is done with its job, it calls `shutdown()` on its socket. This causes TCP (still on the client) to send a FIN on the connection. The server TCP at the other end will answer this with an ACK when all outstanding retransmissions are done.

In the server application, `recv()` now unblocks. The server application sends any reply it might have, and now in turn calls `shutdown()` on its socket. This causes the TCP on the server to send its FIN, which the client then answers with an ACK.

Now comes some waiting, to assure that all retransmissions are gone, and any wild packets will have reached their 0 hop-count, and finally the sockets are closed on both sides (not simultaneously). This long sequence is to assure that when the given socket is reused there will be no interference from packets belonging to the old socket.

It is easy to see all the sockets existing on your PC by using the "netstat" command, which works in Windows as well as Linux. With the "-a" option it shows "all"—meaning that listening servers are included, while the "-b" option will show the programs or processes to which the socket belongs. Listing 7.7 shows an example—with a lot of lines cut out to save space. Note the many sockets in "TIME_WAIT."

Listing 7.7: Extracts from running netstat

```
C:\Users\kelk>netstat -a -b -p TCP

Active Connections

  Proto  Local Address          Foreign Address        State
  TCP    127.0.0.1:843          DK-W7-63FD6R1:0        LISTENING
 [Dropbox.exe]
  TCP    127.0.0.1:2559         DK-W7-63FD6R1:0        LISTENING
 [daemonu.exe]
  TCP    127.0.0.1:4370         DK-W7-63FD6R1:0        LISTENING
 [SpotifyWebHelper.exe]
  TCP    127.0.0.1:4664         DK-W7-63FD6R1:0        LISTENING
 [GoogleDesktop.exe]
  TCP    127.0.0.1:5354         DK-W7-63FD6R1:0        LISTENING
 [mDNSResponder.exe]
  TCP    127.0.0.1:5354         DK-W7-63FD6R1:49156    ESTABLISHED
 [mDNSResponder.exe]
```

```
  TCP    127.0.0.1:17600        DK-W7-63FD6R1:0        LISTENING
[Dropbox.exe]
  TCP    127.0.0.1:49386        DK-W7-63FD6R1:49387    ESTABLISHED
[Dropbox.exe]
  TCP    127.0.0.1:49387        DK-W7-63FD6R1:49386    ESTABLISHED
[Dropbox.exe]
  TCP    192.168.0.195:2869     192.168.0.1:49704      TIME_WAIT
  TCP    192.168.0.195:2869     192.168.0.1:49705      TIME_WAIT
  TCP    192.168.0.195:2869     192.168.0.1:49706      TIME_WAIT
  TCP    192.168.0.195:52628    192.168.0.193:1400     TIME_WAIT
  TCP    192.168.0.195:52632    192.168.0.193:1400     TIME_WAIT
  TCP    192.168.0.195:52640    192.168.0.194:1400     TIME_WAIT
```

Socket handling in C can seem a little cumbersome, especially the server side, and on Windows it requires a few lines more than on Linux. After a few repetitions, it gets easier, and the client side is a lot simpler. Listings 7.8 and 7.9 together form the skeleton of a web server on a Windows PC.[5] If you compile and run it, you can fire up a browser and write: `http://localhost:4567` —this will give a response. "localhost" is the same as IP address 127.0.0.1, which is always your local PC or other device. This code is not waiting at the usual port 80 but at port 4567, which we tell the web browser by adding ":4567." Note that if you refresh your browser view (typically with CTRL-F5), the port number will change as the operating system provides new ephemeral ports.

Listing 7.8: Windows server socket LISTEN

```
#include "stdafx.h"
#include <winsock2.h>
#include <process.h>
#pragma comment(lib, "Ws2_32.lib")

static int threadCount = 0;
static SOCKET helloSock;
void myThread(void* thePtr);

int main(int argc, char* argv[])
{
   WSADATA wsaData; int err;
   if ((err = WSAStartup(MAKEWORD(2, 2), &wsaData)) != 0)
   {
      printf("Error in Winsock");
      return err;
   }

   helloSock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
   if (helloSock == INVALID_SOCKET)
   {
```

5 Please note that source code can be downloaded from https://klauselk.com

```
22        printf("Invalid␣Socket␣-␣error:␣%d", WSAGetLastError());
23        return 0;
24     }
25
26     sockaddr_in hello_in;
27     hello_in.sin_family = AF_INET;
28     hello_in.sin_addr.s_addr = inet_addr("0.0.0.0");  // wildcard
29     hello_in.sin_port = htons(4567);
30     memset(hello_in.sin_zero, 0, sizeof(hello_in.sin_zero));
31
32     if ((err = bind(helloSock, (SOCKADDR*)&hello_in,
33           sizeof (hello_in))) != 0)
34     {
35        printf("Error␣in␣bind");
36        return err;
37     }
38
39     if ((err = listen(helloSock, 5)) != 0)
40     {
41        printf("Error␣in␣listen");
42        return err;
43     }
44     sockaddr_in  remote;
45     int  remote_len = sizeof(remote);
46
47     while (true)
48     {
49        SOCKET sock = accept(helloSock, (SOCKADDR*)&remote,
50                                        &remote_len);
51        if (sock == INVALID_SOCKET)
52        {
53           printf("Invalid␣Socket␣-␣err:␣%d\n", WSAGetLastError());
54           break;
55        }
56
57        printf("Connected␣to␣IP:␣%s,␣port:␣%d\n",
58               inet_ntoa(remote.sin_addr), remote.sin_port);
59
60        threadCount++;
61        _beginthread(myThread, 0, (void *)sock);
62     }
63
64     while (threadCount)
65        Sleep(1000);
66
67     printf("End␣of␣the␣line\n");
68     WSACleanup();
69     return 0;
70  }
```

The following is a walk-through of the most interesting lines in Listing 7.8:

- Lines 1–8. Includes the socket library inclusion and global static variables.
- Line 13. Unlike Linux, Windows needs the WSAStartup to use sockets at all.
- Line 19. The basic *socket* for the server is created. AF_INET means "address family internet"—there are also pipes and other kinds of sockets. "SOCK_STREAM" is TCP. UDP is called "SOCK_DGRAM" (datagram). "IPPROTO_TCP" is redundant, and if you write 0 it still works. The socket call returns an integer which is used as a handle in all future calls on the socket.
- Lines 26–30. Here, we define which IP address and port the server will wait on. Typically, IP is set to 0—the "wildcard address," partly because this will work even if you change IP address, partly because it will listen to all your NICs. The port number must be specific. The macro `htons` means host-to-network-short, and is used to convert 16-bit **`short`**s, like a port number, from the host's "endian-ness" to the network's (see Section 3.3). The `inet_addr()` function outputs data in network order, so we do not need `htonl` here, which converts **`long`**s. Note that the macros change nothing if the CPU is big-endian.
- Line 32. We now handover the result of the above struggle to the socket in a `bind()` call.
- Line 39. We tell the server to actually `listen()` to this address and port. The parameter, here 5, is the number of sockets we ask the OS to have ready in stock for incoming connections. Our listening server is not a complete 5-tuple[6] as there is no remote IP address or port. `netstat -a` will show the status as "LISTENING."
- Line 49. The `accept()` call is the real beauty of the Berkeley TCP server. This call blocks until a client performs a SYN (connect). When this happens, the `accept()` unblocks. The original listening server socket keeps listening for new customers, and the new socket returned from the `accept()` call is a full-blown 5-tuple. We have used one of the 5 "stock" sockets, so the OS will create a new one in the background. `netstat -a` will show the status of this new socket as "ESTABLISHED."
- Line 61. A thread is spawned to handle the established socket, while the original thread loops back and blocks, waiting for the next customer.
- Line 68. When we break out of the loop, we need to clean-up.

Listing 7.9: Windows server socket ESTABLISHED

```
1  void myThread(void* theSock)
2  {
3     SOCKET sock = (SOCKET)theSock;
4     char rcv[1000]; rcv[0] = '\0'; // Allocate buffer
5     int offset = 0; int got;
6
7     sockaddr_in  remote;
```

6 5-Tuple: (Protocol, Src IP, Src Port, Dst IP, Dst Port).

```
8     int  remote_len = sizeof(remote);
9     getpeername(sock, (SOCKADDR*)&remote, &remote_len);
10
11    do // Build whole message if split in stream
12    {  // 0: Untimely FIN received, <0: Error
13       if ((got = recv(sock, &rcv[offset],
14                       sizeof(rcv)-1-offset, 0)) <= 0)
15          break;
16       offset += got;
17       rcv[offset] = '\0'; // Terminate the string
18       printf("Total␣String:␣%s\n", rcv);
19    } while (!strstr(rcv, "\r\n\r\n")); // No body in GET
20    // Create a HTML Message
21    char msg[10000];
22
23    int msglen = _snprintf_s(msg, sizeof(msg)-1, sizeof(msg),
24    "<html><title>ElkHome</title><body>"
25    "<h1>Welcome␣to␣Klaus␣Elk's␣Server</h1>"
26    "<h2>You␣are:␣IP:␣%s,␣port:␣%d␣-␣%d'th␣thread,␣and␣you␣sent:"
27    "<p>%s␣</p></h2>"
28    "</body></html>",
29    inet_ntoa(remote.sin_addr),remote.sin_port,threadCount,rcv);
30
31    // Create a new header and send it before the message
32    char header[1000]; int headerlen =
33       _snprintf_s(header, sizeof(header)-1, sizeof(header),
34       "HTTP/1.1␣200␣OK\r\nContent-Length:␣%d\r\nContent-Type:␣"
35       "text/html\r\n\r\n", msglen);
36    send(sock, header, headerlen, 0);
37
38    // Now send the message
39    send(sock, msg, msglen, 0);
40
41    shutdown(sock, SD_SEND);
42    closesocket(sock);
43    threadCount--;
44    if (strstr(rcv, "quit"))
45       closesocket(helloSock);
46 }
```

Continuing with Listing 7.9—again per line:

- Line 9. We use `getpeername()` so that we can write out IP address and port number of the remote client.
- Lines 13-17. The `recv()` call. If there is no data this statement blocks, otherwise it returns the number of bytes received so far. As this is a stream, there is absolutely no guarantee that we will read chunks of data in the same size as they were sent—we need to accumulate "manually." This is a side-effect in TCP that confuses many. If `recv()` returns 0 it means the client at the other end has sent it's FIN, and our

socket has ACK'ed it. It can also return a negative number, which will be an error-code that a serious program should handle.

- Line 19. We need to loop the `recv()` until the full request is received. As this "web server" only supports HTTP-GET there is no body. Once we see two pairs of CR-LF, we are good.
- Line 23. We create the body of our HTML first. This is practical as it gives us the length of the body, which we will need in the header.
- Line 29. For printout we convert the IP address from a 32-bit number to the well--known x.y.z.v format in ascii using `inet_ntoa()`.
- Line 33. The header is generated—including the length of the body. Note that it ends with two CR-LF pairs.
- Line 36. We send the header.
- Line 39. Finally, we send the body. This again demonstrates the stream nature of TCP. The other side does not see that there were two sends. It *may* need to `recv()` 1, 2, or more times. We could have concatenated the header and the body before sending them. This would require an extra copy, but on the other hand save an OS call. You may want to experiment to find what is best for your embedded system.
- Line 41. We do a `shutdown()` of our sending side. This generates a FIN to the remote.
- Line 42. We do a `closesocket()`.
- Lines 44-45. As a small finesse, we close the parent socket if we have received "quit," for example, "http://localhost:4567/quit."

Figure 7.9 shows the output from the "webserver."

Note the very last header-line written: "Connection: Keep-Alive." This is the web browsers part of a negotiation, and as the tiny web server's reply does not contain this header, it does not happen. If both parties agree on "Keep-Alive," we have the so-called "pipelining," where multiple sequential HTTP-requests and responses can happen on the same open TCP socket. The default behavior is to close the socket af-

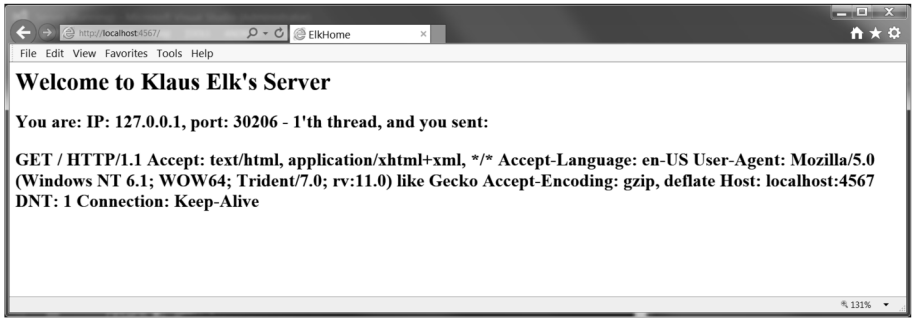


Figure 7.9: Webbrowser with response from simple server.

ter every request-response pair, but this creates a lot of overhead. Much faster to stay on the same socket. This gives the programmer some work in finding the "borders" between requests in one direction and responses in the other. This is where a communication library can be handy.

A browser can also open parallel sockets. When you access a homepage your browser retrieves the "basic" HTML page. Embedded in this may be many elements—pictures, logos, banners, etc. The first page contains the URL to all these. This means, that the basic page is retrieved alone in the first round-trip, but in the next, the browser will retrieve 5 figures if it has 5 parallel sockets and there are at least 5 figures. Since the URLs are unique, the embedded pages may be delivered by different servers in a server park. This is the beauty of HTTP, no state—each element can be retrieved "as is."

7.14 IP fragmentation

Version 4 of IP is responsible for "adapting" the size of segments coming from "above" to what the underlying layer can handle. This is a little weird, since we already have seen how TCP breaks up large messages from the application layer to segments fitting into Ethernet frames further down. Why this adapter function?

Normally TCP MSS (maximum segment size) is created so that even after adding the necessary headers, it still fits within an Ethernet frame. But the packet is routed, sometimes over many different links, and one of these may not allow frames this big.[7] The router "looking into" such a link, must "fragment" the packet, in its IPv4 layer. The original packet stays fragmented for the rest of the journey until the fragments reach the destination IPv4 layer in the receiving host. Here, they are re-assembled, and not until all fragments are together, is the packet delivered to the TCP layer above.

This is easily demonstrated with the help of the *ping* command. We tend to think of UDP and TCP as the only clients to the IP layer, but there is also ICMP. Listing 7.10 shows a normal ping command with the "-l" parameter that dictates the length of the data sent.

Listing 7.10: ICMP ping with 5000 bytes

```
1 C:\Users\kelk>ping -l 5000 192.168.0.1
2
3 Pinging 192.168.0.1 with 5000 bytes of data:
4 Reply from 192.168.0.1: bytes=5000 time=12ms TTL=64
5 Reply from 192.168.0.1: bytes=5000 time=6ms TTL=64
6 Reply from 192.168.0.1: bytes=5000 time=7ms TTL=64
7 Reply from 192.168.0.1: bytes=5000 time=6ms TTL=64
8
9 Ping statistics for 192.168.0.1:
```

7 It may not even be an Ethernet link.

```
10      Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
11  Approximate round trip times in milli-seconds:
12      Minimum = 6ms, Maximum = 12ms, Average = 7ms
```

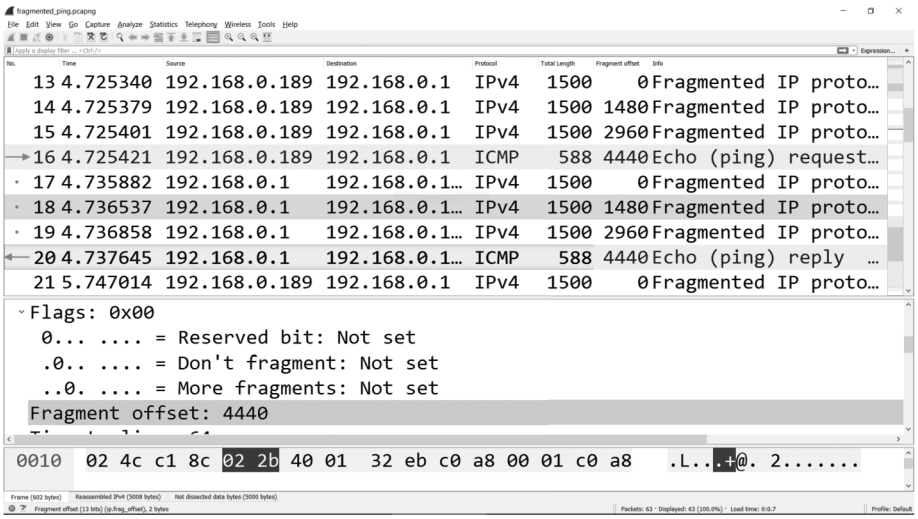


Figure 7.10: Fragmented ICMP ping with 5000 bytes of data.

Let's examine the Wireshark capture of the ping in Figure 7.10:

- The "total length" column contains the payload in the IP layer. If we add the numbers, we get $3 * 1500 + 588 = 5088$. This is the 5000 databytes plus 4 IP headers of 20 bytes each, plus one ICMP header of 8 bytes.
- The "fragment offset" is, as the name says, the offset of the first byte in the original non-fragmented packet. Wireshark is helping us here. The offset in the binary header is shifted three bits to the right to save bits, but Wireshark shifts them back for us in the dissection. In order to make this work, fragmentation always occurs at 8-byte boundaries. In the figure, the offset is selected in the middle window, with the value 4440. This selects the same field in the bottom hex view. Here, it is 0x22b = 555 decimal—one-eighth of 4440.
- The info field contains an ID (outside the view) generated by the transmitting IPv4 layer. This is the same for all fragments from the same package, and incremented for the next.

Fragmentation is removed from IPv6. To simplify routers and improve performance, the router facing a link with a smaller maximum frame size than incoming frames, now sends an error back to the originator, so that the problem can be dealt with at the origin.

If you experience fragmentation, you should ask why and probably make sure it does not happen in the future, as it degrades performance.

7.15 Introducing IPv6 addresses

The main reason for introducing IPv6 was the shrinking IPv4 address space, partly because of waste in the old class system. Moving from 32 bits to 128 bits certainly cures that. While the good people on the standard committees were in the machine room, they also fixed a number of minor irritation points in IPv4.

Quite opposite to the old telephone system, the Internet is supposed to be based on intelligent equipment at either end of conversations (the hosts), and "stupid" and, therefore, stable, infrastructure in between. The removal of fragmentation in IPv6 simplifies routers. IPv4's dynamic header size requires routers to be more complex than otherwise necessary. Similarly, the IPv4 checksum needs to be recalculated in every router, as the TTL (hopcount) is decremented. This is not optimal.

IPv6 still has the hopcount, now even called so, but the checksum is gone and the header size is static. There is still a much better checksum in the Ethernet, and the 1's complement checksum is still found in UDP and TCP.[8]

Table 7.5: How IPv4 and IPv6 notation differs.

Concept	IPv4	IPv6
Bit-width	32	128
Groupsize	1 Byte	2 Bytes
Notation	Decimal	Hexadecimal
Separator	.	:
Skip front zero	Yes	Yes
Skip zero groups	No	Yes
Mixed Notation	No	Yes

As Table 7.5 states: IPv6 addresses are grouped in 16-bit chunks, written as 4-digit hexadecimal numbers. We are permitted to skip zeroes in front of each group and we can also skip *one* consecutive streak of zeros, using "::" notation for this. Here are some examples:

`fc00:a903:0890:89ab:0789:0076:1891:0123` can be written as:
`fc00:a903:890:89ab:789:76:1891:123`

`fc00:0000:0000:0000:0000:98ab:9812:0b02` can be written as:
`fc00::98ab:9812:b02`

`::102.56.23.67` is an IPv4 address used in an IPv6 program.

8 It has been suggested by Evan Jones that switches and routers simply recalculate the CRC and, therefore, might hide an internal malfunction. The actual case uncovered a Linux kernel error. Integrity checks as in Chapter 10 is a way to detect such problems.

Table 7.6: Functional differences between IPv4 and IPv6.

Function	IPv4	IPv6
Checksum	Yes	No
Variable Header Size	Yes	No
Fragmenting	Yes	No
Hopcount Name	TTL	Hopcount
Flow Label	No	Yes
Scope & Zone ID	No	Yes

Table 7.6 shows the major differences between the two versions of the internet protocol.

IPv6 has introduced two new fields:

1. *Flow Label*
   RFC 6437 says "From the viewpoint of the network layer, a flow is a sequence of packets sent from a particular source to a particular unicast, anycast, or multicast destination that a node desires to label as a flow." This is, for example, used in Link Aggregation.[9] The standard says that all packets in a given 5-tuple SHOULD be assigned the same 20-bit unique number—preferably a hash[10] of the 5-tuple. Traditionally, the 5-tuple has been used directly by network devices, but as some of these fields are encrypted, the Flow label is a help. If NOT used it MUST be set to 0, which is what is done in this book.
2. *Scope ID*
   Defined in RFC 4007 and RFC 6874. RFC 4007 allows for a scope ID for link-local addresses that can be the device, the subnet, or global. RFC 6874 defines a special case where a "zone ID" is used to help the stack use the right interface. Apparently, this special case is the only one used yet. It is not a field in the IPv6 header but a part of the address string in link-local scenarios written after a "%", helping the stack to use the right interface. On Windows this is the interface number (seen with "netstat -nr"), on Linux it could be, for example, "eth0."

Table 7.7 gives some examples on addresses in the two systems.

Table 7.7: Examples on addresses in the two systems.

Address	IPv4	IPv6
Localhost	127.0.0.1	::1
Link-local	169.254.0.0/16	fe80::/64
Unspecified	0.0.0.0	::0 or ::
Private	192.168.0.0/16 etc	fc00::/7

9 Link-Aggregation aka "Teaming" is a practical concept that allows the use of several parallel cables—typically between a multiport NIC and a switch.

10 Hashes are discussed in Section 10.4.

7.16 TCP Sockets on IPv6 under Linux

Now that we have looked at Windows, let's see something similar on Linux. In the following, we will examine a small test program, running as server in one instance, and as client in another. In our case, both run on a single Linux PC. This time we will try IPv6—it might as well have been the other way around.

Listing 7.11: Linux test main

```
....main...
   int sock_comm;
   int sock1 = do_socket();

   if (is_client)
   {
      do_connect(sock1, ip, port);
      sock_comm = sock1;
   }
   else // server
   {
      do_bind(sock1, 0, port);
      do_listen(sock1);
      sock_comm = do_accept(sock1);
   }

   TestNames(sock_comm);
   gettimeofday(&start, NULL);
```

The basic workflow is seen in Listing 7.11. The `main()` shows the overall flow in a client versus a server on a single page. In both cases, we start with a `socket()` call. On a client, we then need a `connect()`, and are then in business for `recv()` and `send()`. On a server, we still need the `bind()`, `listen()`, and `accept()` before we can transfer data. This is exactly like the Windows server, and the client scenarios are also equal.

The functions called in Listing 7.11 are shown in Listings 7.12 and 7.13. The error-handling and other printout code is removed, as focus is on the IPv6 socket handling. Even though this is Linux and IPv6, it looks a lot like the Windows version using IPv4 we saw with the web server. The most notable difference between the OSs is that on Linux we do not need WSAStartup() or anything similar. This is nice, but not really a big deal.

What I find much more interesting in the Linux version, is very subtle and only shows itself in the `close()` call on the socket. On Windows, this is called `closesocket()`. Add to this that on Linux you don't need to use `send()` and `recv()`. It is perfectly all right to instead use `write()` and `read()`. The consequence is that you can open a socket and pass it on as a file handle to *any* process using files, for example, using `fprintf()`. This certainly can make life a lot easier for an application programmer.

Listing 7.12: Linux sockets part I

```
int do_connect(int socket, const char *ip, const bool port)
{
...
   struct sockaddr_in6 dest_addr;      // for destination addr
   dest_addr.sin6_family = AF_INET6;   // IPv6
   dest_addr.sin6_port = htons(port);  // network byte order

   int err = inet_pton(AF_INET6, ip, &dest_addr.sin6_addr);
   ...
   int retval = connect(socket, (struct sockaddr *)&dest_addr,
                        sizeof(dest_addr));
   ...
}

int do_bind(int socket, const char *ip, int port)
{
   struct sockaddr_in6 src_addr;       // for source addr
   src_addr.sin6_family = AF_INET6;    // IPv6
   src_addr.sin6_port = htons(port);   // network byte order
   if (ip)
   {
      int err = inet_pton(AF_INET6, ip, &src_addr.sin6_addr);
      {
         printf("Illegal address.\n");
         exit(err);
      }
   }
   else
      src_addr.sin6_addr = in6addr_any;

   int retval = bind(socket, (struct sockaddr *)&src_addr,
                     sizeof(src_addr));
   ...
}

int do_listen(int socket)
{
   int retval = listen(socket, 5); // Backlog of 5 sockets
    ...
}

int do_socket()
{
   int sock = socket(AF_INET6, SOCK_STREAM, 0);
   ...
}

int do_close(int socket)
```

```
{
   ...
   int retval = close(socket);
   ...
}
```

Looking at the code there are a number of differences between IPv4 and IPv6. Almost all relate to the functions not part of the actual socket handling:

- Instead of "AF_INET," we use "AF_INET6."
- The address structures used end with "_in6" instead of "_in."
- `inet_ntoa()` and `inet_aton()` are replaced with `inet_ntop()` and `inet_pton()`. The new functions work with both IPv4 and IPv6 addresses.
- We still use the `ntohs()` and `htons()` macros, as they are used on the 16-bit ports in the unchanged TCP. However, we do not use `ntohl()` and `htonl()` on IP addresses, as they are too big for 32-bit words.
- There are new macros as `in6addr_any`.

It can be a pain to get the address structures right at first. The most tricky parameter is the one inside a ***`sizeof()`****, with the size of the structure which is normally the last parameter in several calls, e. g.* `connect()` *and* `bind()`*. The compiler will not complain as long as you write something that evaluates to an integer. But it certainly makes a difference what is written. I find that I make less mistakes if I use the name of the variable in the* ***`sizeof()`*** *instead of the type, since I have typically just populated the variable, and the compiler will complain here if I try to put e. g.* `in6addr_any` *inside a 32-bit IP address.*

Listing 7.13: Linux sockets part II

```
int do_accept(int socketLocal)
{
   ...
   struct sockaddr_in6 remote;
   socklen_t adr_len = sizeof(remote);

   int retval = accept(socketLocal,
                       (struct sockaddr *) &remote,&adr_len);
   ...
   char ipstr[100];
   inet_ntop(AF_INET6,(void*) &remote.sin6_addr,
             ipstr, sizeof(ipstr));
   printf("Got␣accept␣on␣socket␣%d␣with:"
          "␣%s␣port␣%d␣-␣new␣socket␣%d\n",
           socketLocal,ipstr,ntohs(remote.sin6_port),retval);
}

int do_send(int socket, int bytes)
```

```
{
   ...
   if ((err = send(socket, numbers, bytes, 0)) < 0)
   ...
   return 0;
}

int do_recv(int socket, int bytes)
{
   int received = recv(socket, nzbuf,bytes-total, 0);
   ...
}

void TestNames(int socketLocal)
{
   struct sockaddr_in6 sock_addr;
   socklen_t adr_len = sizeof(sock_addr);
   char ipstr[100];

   getpeername(socketLocal,
               (struct sockaddr *) &sock_addr,&adr_len);

   inet_ntop(AF_INET6, (void*) &sock_addr.sin6_addr, ipstr,
             sizeof(ipstr));

   printf("getpeername:␣IP=␣%s,␣port=␣%d,␣adr_len=␣%d␣\n", ipstr,
          ntohs(sock_addr.sin6_port), adr_len);
}
```

Listing 7.14 shows the execution.

Listing 7.14: Running Linux test

```
kelk@debianBK:~/workspace/C/ss6$ ./socktest -r --bytes=10000 &
[1] 1555
kelk@debianBK:~/workspace/C/ss6$ Will now open a socket
Binding to ip: localhost, port: 12000 on socket: 3
Listen to socket 3
Accept on socket 3

kelk@debianBK:~/workspace/C/ss6$ ./socktest -c -s --bytes=10000
Will now open a socket
Connecting to ip: ::1, port: 12000 on socket: 3
Got accept on socket 3 with: ::1 port 43171 - new socket 4
getpeername: IP= ::1, port= 43171, adr_len= 28
Loop     0
Plan to receive 10000 bytes on socket 4
getpeername: IP= ::1, port= 12000, adr_len= 28
Loop     0
Plan to send 10000 bytes on socket 3
```

```
Received 10000 bytes at address 0x7ffcb3b34e30:
   0    1    2    3    4    5    6    7    8    9
......
  7c8  7c9  7ca  7cb  7cc  7cd  7ce  7cf
Total sent: 10000 in 1 loops in 0.001910 seconds = 41.9 Mb/s
....
[1]+  Done                    ./socktest -r --bytes=10000
kelk@debianBK:~/workspace/C/ss6$
```

First, the server is created, this is the default. It is told to receive 10000 bytes. It creates the socket, binds it to the default port (12000) and blocks in the `accept()` call. The "&" makes it run in the background. Now the client is started with the "-c" option and asked to send 10000 bytes. It receives an ephemeral port number from Linux: 43171. This unblocks the waiting server and we are done. The speed given is affected by writing it all to the screen.

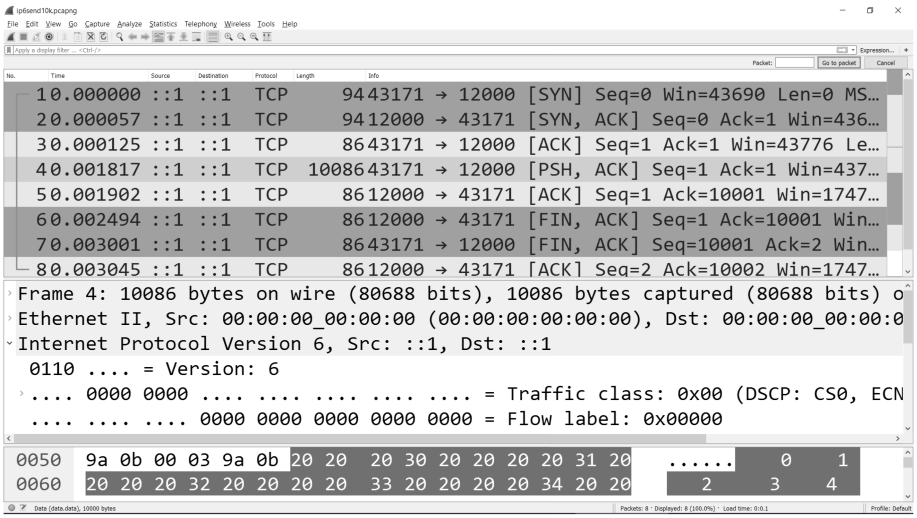


Figure 7.11: TCP on Linux localhost with IPv6.

Figure 7.11 shows the same scenario from a Wireshark point-of-view. We recognize the port numbers as well as the "::1" localhost IPv6 address. You may ask how the marked frame 4 can send 10000+ bytes as we normally have a maximum of 1460 bytes (making room for TCP and IP headers). The explanation is that the normal Maximum Segment Size stems from the Ethernet which we are skipping here, because everything happens on the same PC. Interestingly, Wireshark still shows Ethernet in the stack, but with zero addresses at both ends.

Note that it is easy to capture a localhost communication on Linux, as the "lo" interface is readily available. Not so on Windows. To do a capture on the localhost

on Windows you need to change your route table to send all frames to the gateway and back again. Now you see them twice, and timing is quite obscured. And don't forget to reset the route table. An alternative is to install the "Windows Loopback Adapter."

7.17 Data transmission

Until now, our focus on the TCP related stuff has been about how to get the transmission going. Now is the time to look at actual transmissions. When looking at TCP it makes little difference whether we have IPv4 or IPv6 beneath it—there is no "TCPv6." Figure 7.12 shows a Wireshark capture of a network device (10.116.120.155) transmitting data at high speed (for an embedded device) to a PC (10.116.121.100). All frames are TCP segments between the two devices. To save space, only the Source column is shown.

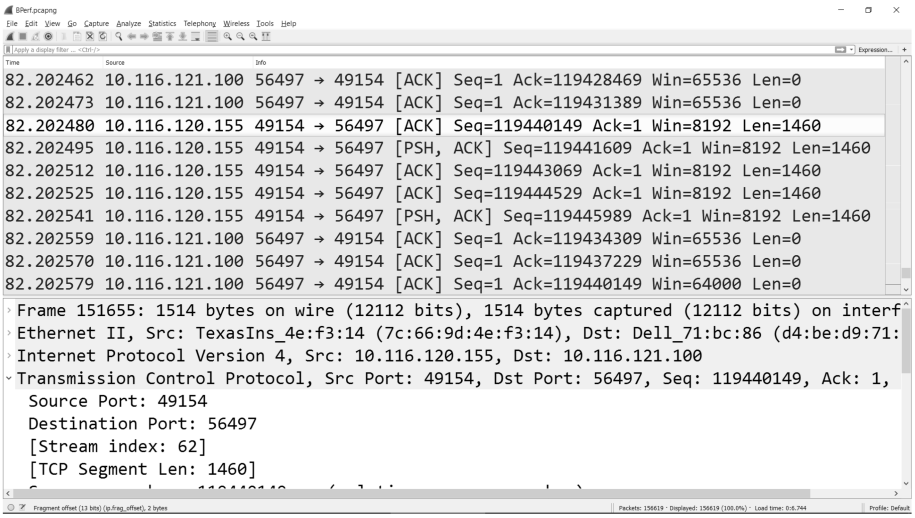


Figure 7.12: Transmitting many bytes.

All frames with data has length 1460. This fits nicely with an Ethernet MTU (maximum transmission unit), or payload, of 1500 bytes minus 20 bytes per each of the TCP and IP headers. The total Ethernet frames here are 1514 bytes according to Wireshark. The 14 bytes difference is the Source and Destination MAC addresses, each 6 bytes, plus the 2-byte "Type" Field with the value "0x0800" (not visible on the figure) for IPv4.

This is a pattern seen again and again. An incoming frame is received at the lowest layer. This layer needs to look at a given field to know where to deliver it. Here, Ethernet

delivers it to IPv4, which, based on it's "Protocol" field, delivers it to TCP, which again delivers it to the application process, based on the port number.

Interestingly, the PC's sequence number is constantly "1" while the device is racing toward higher numbers. This means that the PC hasn't sent anything since its SYN flag. This is indeed a one-sided conversation. Outside the figure, at the bottom line in the "dissection" of the selected frame there is a line saying "validation disabled" for the checksum. If checksum validation is enabled, Wireshark will, mistakenly, flag all outgoing frames as having a bad checksum, because modern systems no longer calculate the checksum in the stack. They leave it to the Network Interface Card, and as Wireshark is between these, it will see wrong checksums. It makes no sense to check in-going frames either, as the NIC only lets frames through if they have correct checksum.

If we select one of the frames with data (as in the figure), then select "Statistics" in the top menu, then "TCP Stream Graphs" and finally "Time Sequence (Stevens)" we get a great view of how data is sent over time; see Figure 7.13.

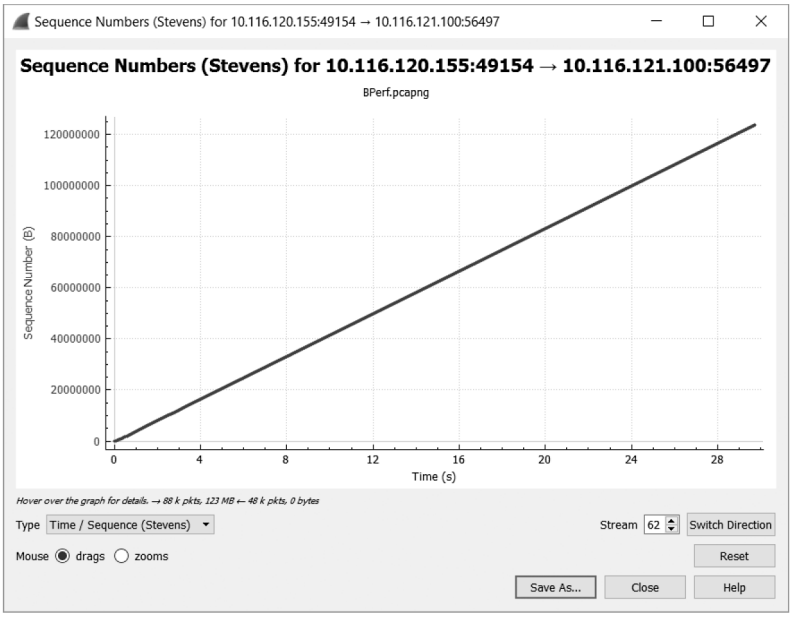


Figure 7.13: Nice Stevens graph of sequence number versus time.

The Stevens graph, named after the author of the famous *TCP/IP Illustrated* network books, shows the sequence numbers over time. This is the same as the accumulated number of bytes sent. Seldom do you see a line this straight. By reading the point in, for example, X = 20 s with corresponding Y = 80 MBytes, we easily calculate the

data rate to be 80 MByte/20 s = 4 MByte/s or 32 Mbps. The biggest challenge is counting the number of zeros. Inside the same window, you can also select, for example, "Roundtrip-Time." Another menu-selection can give an I/O graph, but in this case they are as boring as Figure 7.13.

Figure 7.14 shows a similar measurement from some years ago. If you click with a mouse on the flat part of the graph, Wireshark goes to the frame in question. Here, you should, for example, look for retransmissions, as these will occupy bandwidth without progressing sequence numbers.

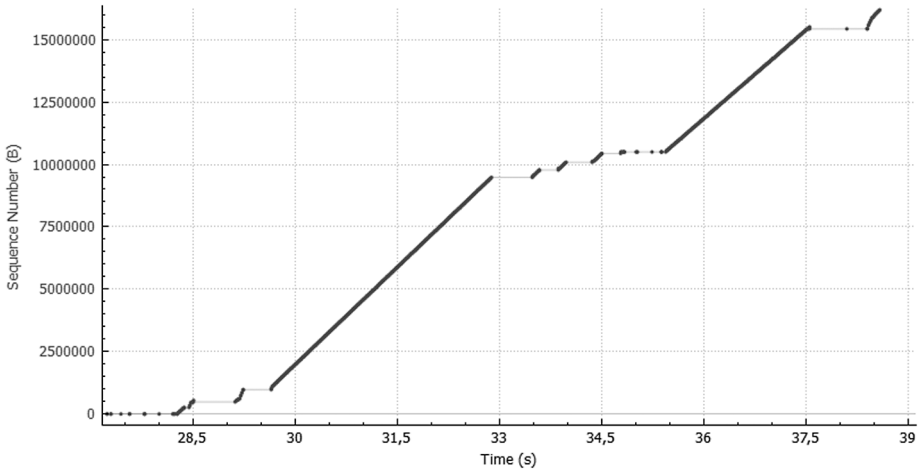


Figure 7.14: Stevens graph with many retransmissions.

You can also try "analyze" in the main menu, and then "expert information." This is shown in Figure 7.15.

Here, the PC sends up to 20 Duplicate ACKs for the same frame, effectively repeating the phrase: "The last Sequence Number I was expecting to see from you was xx and you are now much further ahead, please go back and start from xx."

If you click in this window, you will be at the guilty frame in the main window. Since Wireshark hasn't lost any packages, and reports *suspected* retransmissions, it hasn't seen these packages either. Apparently, the device is sending packages that nobody sees. This was exactly what was happening. We had a case of "crosstalk" between an oscillator and the PHY in this prototype. We will get back to this in Section 7.23.

Note that when you capture transmissions like this, and not just random frames, it is a good idea to turn off the live update of the screen. Wireshark is very good at capturing, but the screen cannot always keep up. Very often you do not need to capture the actual data, just the headers, so you can ask Wireshark to only capture, for example, 60 bytes per frame. This will allow the PC to "punch above its weight-class."

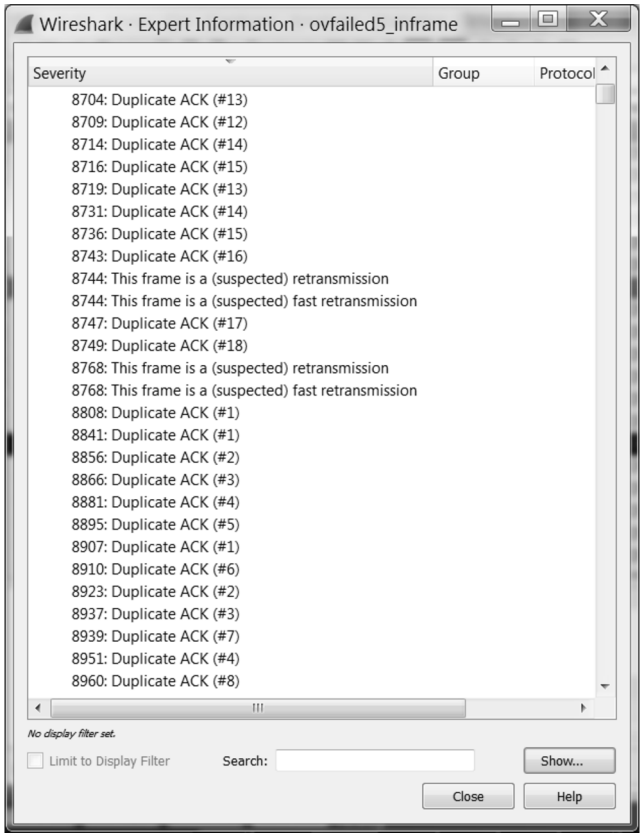


Figure 7.15: Expert Information with lost data.

7.18 UDP sockets

We have used the terms broadcast (aka anycast), unicast, and multicast without really going into what the meaning is. Table 7.8 compares the three concepts.

A unicast is sent from one process on one host to another process on another (or the same) host. A broadcast is sent from one process on one host to all other hosts, restricted to the original subnet. A broadcast will never get through a router.

Table 7.8: The three types of -casts.

-cast	Meaning	Protocols
Unicast	1:1	TCP, UDP
Multicast	1:Members	UDP
Broadcast	1:All	UDP

A multicast looks pretty much like a broadcast, but as stated it is only for members. What does that mean? A router can typically be configured to simply "flood" multicasts coming in on one port to all other ports, like broadcasts into a switch. Alternatively, it can support "group membership" and "spanning trees" and all sorts of advanced stuff, helping routers to only forward multicasts to ports that somewhere further out has a *subscriber* for the given address.

A multicast address is something special. In IPv4, it is all addresses in the "Class D" range 224.0.0.0-239.255.255.255. Take, for instance, PTP—Precision Time Protocol. It uses the IP addresses 224.0.1.129, 224.0.1.130, 224.0.1.131 and 224.0.1.132 as well as 224.0.0.107. Anyone who wants these messages must listen to one or more of these IP addresses, as well as their own.

Now what about ARP, if several hosts have the same IP address? ARP is not used in the class D range. Instead the MAC must listen to some programmed, specific addresses on top of the one it is "hardcoded" for. The MAC addresses are simply 01:00:5e:00:00:00 OR'ed with the lower 23-bits of the Multicast IP address. This means that MAC hardware has extra programmable "filter-addresses" apart from the static one (which might be programmable in, e. g., EEPROM).

It seems UDP can do everything—why use TCP at all? Table 7.9 compares TCP and UDP on some of their main features.

Table 7.9: TCP versus UDP.

Feature	TCP	UDP
Flow	Stream	Datagram
Retransmit	Yes	No
Guaranteed Order	Yes	No
Social Throttle	Yes	No
Maximum Size	No (stream)	(536) bytes
Latency 1'st Byte (RTT)	1.5	0.5

We have seen the stream nature of TCP in action. It is a two-edged sword giving programmers headaches trying to refind the "borders" in what was sent, but it is also an extremely powerful concept, as retransmissions can be much more advanced than simply retransmitting frames. The retransmit, the guarantee that every byte makes it all the way (if in any way possible), that it comes only once, and in the right order compared to the other bytes, are all important features in serious applications as web or file transfer.

"Social Throttle" relates to TCP's "digestion" handling, where all sockets will lower their speed in case of network problems. UDP has none of these advanced concepts. It is merely a thin skin on top of IP packets. DNS is a good example on using UDP (see Section 7.10). These packets are rather small and can be resent by the application. If there is digestion trouble due to a DNS problem, it makes sense that they are faster,

non-throttled. When it comes to the maximum size, an UDP packet can be 64k, but it is recommended not to allow it to "IP fragment." See Section 7.14. A size less than 536 bytes will guarantee this.

Finally, how long does it take before the first byte of a message is received? With TCP, we need the three-way handshake. Data may be embedded in the last of the three "ways." This corresponds to 1.5 Roundtrips, as a full roundtrip is "there and back again." UDP only takes a single one-way datagram, corresponding to 0.5 RTT. TCP actually has to do the closing dance afterwards. This is overhead, but not latency. Note that if the TCP socket is kept alive, the next message only has a latency of 0.5 RTT, as good as UDP.

7.19 Case: UDP on IPv6

Listing 7.15 shows a UDP transmission using IPv6. As we saw with TCP on IPv6: Instead of "AF_INET," "AF_INET*6*" is used. Some sources use "PF_INET6." It makes no difference as one is defined as the other. Since we are using UDP, the socket is opened with "SOCK_DGRAM" as the second parameter. Also here the IPv4 "sockaddr_in" structure has been replaced by its cousin with "in" replaced by "in6." The same rule applies to the members of the **`struct`**. We still use the macro `htons` on the port number, as this is UDP—not IP—and the transport layer is unchanged.

Listing 7.15: IPv6 UDP send on Linux

```
1  void sendit()
2  {
3     int sock = socket(AF_INET6, SOCK_DGRAM, 0);
4     if (sock <= 0)
5     {
6        printf("Opening␣socket␣gave␣error:␣%d\n",sock);
7        exit(sock);
8     }
9
10    int err;
11    char *mystring = "The␣center␣of␣the␣storm\n"; // Message
12
13    struct sockaddr_in6 dst_addr;
14    memset(&dst_addr, 0, sizeof(dst_addr)); // Clear scope & flow
15    dst_addr.sin6_family = AF_INET6;
16    dst_addr.sin6_port = htons(2000);       // TCP as usual
17
18    if ((err=inet_pton(AF_INET6, "::1",&dst_addr.sin6_addr)) == 0)
19    {
20       printf("Illegal␣address.\n");
21       exit(err);
22    }
23
```

```
24     if ((err = sendto(sock, mystring, strlen(mystring)+1, 0,
25     (struct sockaddr *)&dst_addr, sizeof(dst_addr))) < 0)
26     {
27        printf("Could␣not␣send.␣Error:␣%d\n", err);
28        exit(err);
29     }
30  }
```

In line 18, we see the IPv6 address "::1." This is the "localhost" address, where in IP4 we normally use "127.0.0.1." We also see the function `inet_pton()` which is the recommended function to use as it supports IPv4 as well as IPv6. Finally, in line 24 we see the UDP `sendto()` which is unchanged from IPv4, but called with "in6" `structs`. In TCP, we use the command `send()` without a "to", because in TCP the destination is given already in the `connect()` call. Here, there is no `connect()` because there is no connection.[11] Our next message may be `sendto()` someone else.

Just as we may call `bind()` before `connect()` in TCP, we may also do it in UDP before `sendto()`, but we normally don't in either case. One reason is that it's more work, another reason is that if we try to bind to an already used port we run into problems. Therefore, we normally go with the ephemeral port number the OS hands out, since we normally do not care about the client/active socket's port number.

Listing 7.16 shows the code for a receiver, waiting for the data sent in the previous listing. Here, we set up the listener for "in6addr_any," the same as "::0."

Listing 7.16: IPv6 UDP receive

```
1  void recvit()
2  {
3     int sock = socket(AF_INET6, SOCK_DGRAM, 0);
4     if (sock <= 0)
5     {
6        printf("Opening␣socket␣gave␣error:␣%d\n",sock);
7        exit(sock);
8     }
9
10    struct sockaddr_in6 listen_addr;
11    memset(&listen_addr, 0, sizeof(listen_addr));
12    listen_addr.sin6_family = AF_INET6 ;
13    listen_addr.sin6_port = htons(2000);
14    listen_addr.sin6_addr = in6addr_any;
15
16    int on=1;
17    if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR,
18                        (char *)&on,sizeof(on)) < 0)
```

11 Confusingly, `connect()` *is* possible. It does not make a connection but allows the use of `send()` instead of `sendto()`, etc.

```
19      {
20          printf("setsockopt(SO_REUSEADDR)␣failed");
21          exit(0);
22      }
23
24      int err;
25      err = bind(sock, (struct sockaddr *) &listen_addr,
26                 sizeof(struct sockaddr_in6));
27      if (err)
28      {
29          printf("Binding␣gone␣wrong␣-␣error␣%d\n", err);
30          exit(err);
31      }
32
33      char mystring[100];
34      struct sockaddr_in6 remote_addr;
35      socklen_t addr_len = (socklen_t) sizeof(remote_addr);
36
37      err = recvfrom(sock, mystring, sizeof(mystring),  0,
38           (struct sockaddr *) &remote_addr, &addr_len);
39      if (err < 0)
40      {
41          printf("Could␣not␣recv.␣Error:␣%d\n", err);
42          exit(err);
43      }
44      else
45          printf("Received:␣%s", mystring);
46  }
```

The strange code in lines 17–22 asks the operating system to create this socket, even though there may already be a socket from a previous run hanging in "TIME-WAIT." This is not needed if the code is only run once, but somehow it never is.

Now we need a bind for the listening socket. Naturally, it must be listening at the right port, and the "any" IP address is practical. If a packet makes it this far it must be to one of our interfaces, so why not wait for them all and do away with the need to find out which IP addresses we have where? Then we declare a "holder" for the remote address, which we could print out (but don't), and finally we call `recvfrom()`. This blocks until a message is received. When that happens it is written to the terminal.

Listing 7.17 shows the test of the two programs. First, `udprecv` is run in the background. We then use `netstat`[12] to show that we have a udp6 listener on port 2000. Note that it also gives us the process number and the name of the program. When the sender program is run, the receiver unblocks and terminates.

12 slimmed to fit on the page.

Listing 7.17: Running on localhost

```
kelk@debianBK:~/workspace/C/ss6$ ./udprecv &
[1] 2734
kelk@debianBK:~/workspace/C/ss6$ netstat -au -pn -6
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address Foreign Address PID/Prog name
udp6       0      0 :::48725      :::*            -
udp6       0      0 :::34455      :::*            -
udp6       0      0 :::10154      :::*            -
udp6       0      0 :::961        :::*            -
udp6       0      0 :::2000       :::*            2734/udprecv
udp6       0      0 :::111        :::*            -
udp6       0      0 :::5353       :::*            -
kelk@debianBK:~/workspace/C/ss6$ ./udpsend
Received: The center of the storm
[1]+  Done                    ./udprecv
```

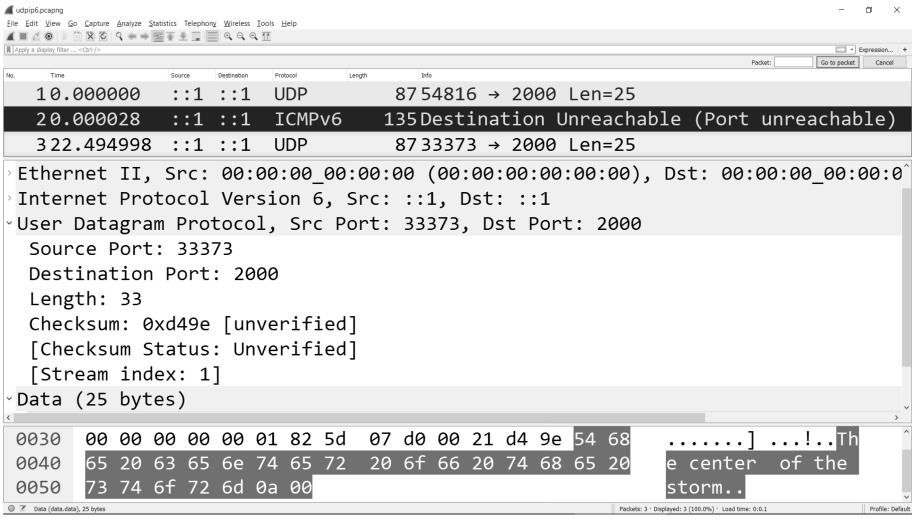


Figure 7.16: UDP on IP6 with ICMPv6 error.

Figure 7.16 shows a Wireshark capture of the scenario.

At first, only the "udpsend" program is run in frame 1. Interestingly, this generates an ICMPv6 "Port Unreachable" error in frame 2 as there is no receiver. This also happens on communication outside the localhost and enables us to write code that actually *can* handle retransmissions, etc. of UDP packets.

Twenty-two seconds later, the full scenario is run as in Listing 7.17, and this time there is no ICMP error as the waiting socket accepts it.

7.20 Application layer protocols

As stated in the Introduction to this chapter, there are myriads of application layer protocols. It is impossible to go through them all, so instead we will look at the criteria for choosing one or the other. An overview is given in Table 7.10.

Table 7.10: Parameters on application layer protocols.

Important Parameters for Protocols	
Standard	Domain or company
Documentation	Good or bad
Flow	Pipelining versus stop & go
State	Stateless or Stateful
Low level	Binary or Textual
Flexibility	Forgiving like XML—or not
Compatibility	Versionized
Dependencies	Requiring special OS, language, etc.
Power	Designed for power savings

- *Standard*
  Is the protocol already a standard in your company or in the application domain, or both? If this is the case, very good arguments are needed to pick another protocol. After all, protocols tie equipment together. Do you really want to be the person responsible, if the new product doesn't work in a system with the old product, or if the software developers are delayed another half year to support your new fancy protocol?[13]
- *Documentation*
  If different companies, in a common application domain, are using the same protocol, it will almost certainly be well documented. This is not a given truth when a single company has many products using the same protocol. A change in technology may force you to reimplement existing protocols, or start all over. In this case, it is important how well documented the old protocols are.
- *Flow*
  You may have a very fast connection and yet not be able to utilize it. If there is a long distance between the communicating parties, this means a delay. We typically talk about the roundtrip-time which is the time it takes to send a packet from A to B plus an answer back to A. If A and B are close geographically, the round trip time is dominated by the time it takes to "clock out" the packet and the time it takes to go up through the stack on A and B.

13 There are times when the answer is yes.

If A and B are on either side of the globe, the round trip time is dominated by the time it takes a packet to travel through the wires and intermediate routers (or the satellite link). The term "long fat pipes" means fast (fat) but also long wires. A *stop & go* protocol completely kills any speed in this as a stop & go protocol requires a response from the receiver to every packet sent, before the next packet can be sent. The opposite of this is "pipelining" where you can have lots of data in the pipe. TCP has gone through a lot of changes to support "long fat pipes," mainly the "Receive Window" can be very big, allowing a *lot* of data to be in "transit."
You might argue that HTTP actually does require a response to every request, and thus is a bad protocol. However, as we saw in Section 7.11, HTTP does not require a response to request "n" before sending request "n+1." When you load a new page in your browser, the first request for this page has to be answered, but when you have this answer you may simultaneously request all referenced pictures, banners, etc. See Section 7.11.

- *State*
  As discussed in Section 7.12 on REST, there is a lot to be gained by using stateless protocols. The main benefit in an IoT device is probably the possibility of pipelining as described above. One of the other main benefits with REST, is the ability to use server parks, but when you see the IoT device as a server, as you sometimes do, it is rarely part of a "park." There may be many almost identical IoT devices, but each is typically connected to unique sensors, or in other ways providing data unique to this particular device.
- *Low Level*
  Many embedded developers prefer binary protocols as they are compressed compared to handling, for example, the same numbers as hex ascii. A 16-bit number does not need more than 16-bits in a binary protocol, but in hex ascii it takes a full byte for each nibble – thus 32-bits for the same 16-bit number. If data is represented in, for example, JSON or XML, it takes a lot more space as, for example, parameter names induces overhead, and in the case of single variables take more space than the data. The PC programmer, at the other end, often prefers XML, as this is a well-known technology and he or she has a lot of tools available for parsing data. Between the two parties, we have the wire. Watching a custom binary protocol in Wireshark can be a pain, whereas the same data in JSON or XML is close to self-explanatory.
  The choice becomes a trade-off between CPU time, wire speed and which developers you listen to. It is true that connections are getting faster and that yesterday's PC technology often becomes tomorrow's embedded technology, meaning that there is a trend toward textual representations in the general embedded world. However, many IoT applications will weigh low energy consumption higher than any of the previously mentioned parameters. This will move the balance somewhat back toward the binary protocols as each bit sent has a cost in Joules.

- *Flexibility*
  Many protocols are rigid and cannot handle missing data or several versions. This is one of the reasons why XML is so popular. There is no requirement to send all defined parameters. This, however, introduces the need for a more or less advanced default handling.
- *Compatibility*
  It is easy to forget a version number in a protocol. This results in some very clumsy code once changes are made. Never create or use a protocol without a version number. A lot can be learned from the TCP/IP stack. It is amazing how many ways you can implement TCP within the standard—and how it can be extended in a backward compatible manner.
- *Dependencies*
  Not so long ago, DCOM was popular for remote control. A major problem with DCOM was the 100 % tie to the Windows OS. Before this, CORBA was very popular in the Unix world. It required expensive tools that you could also get for PCs, but it never really caught on here. Both DCOM and CORBA were based on remote procedure calls. Neither performed very well, if one end of the communication suddenly wasn't available.
  A major reason for the success of REST on top of HTTP (see Section 7.12) is that it is easy to understand, runs without complex tools and on any OS. In the IoT world, we need a loosely coupled concept with clients and servers instead of masters and slaves. We also need something not tied to a specific OS or language. There are "cross-overs" like JSON which is a "lean and mean" replacement for XML. It was made for Java, but was easily moved to other languages.
- *Power*
  This subject was already touched upon in the discussion on binary versus textual. Section 4.1 goes into more details on protocol performance which is closely linked to power consumption.

7.21 Alternatives to the socket API

The socket concept used in this book is generally known as Berkeley sockets. The basic API is the same on all major platforms, with many variations in the implementation of the options and the underlying TCP/IP stack. By now, it is clear that programming directly on the socket interface can be a little frustrating sometimes. Especially, the fact that TCP is streamed and, therefore, does not keep the boundaries from the transmitting hosts `send()` calls to the receiving hosts `recv()` calls, can be annoying.

"Raw sockets" is *not* the same as Berkeley sockets, although many programmers believe so. With a real "raw socket" things are even more primitive, as you are also responsible for all the headers, for example, generating checksums, hopcount, type

fields, etc. This is only relevant in test scenarios where you want to test error-handling. We saw this in Section 5.9.

In order to tweak the various socket options, you often end up using Berkeley sockets directly. Nevertheless, there are situations where you can get very far with a high-level library, and these are getting better all the time. The far most used is "libcurl." It is open source licensed under the MIT/X license and states that you may freely use it in any program. It supports a very long list of application protocols—among these HTTP and HTTPS (which basically is HTTP on top of secure sockets, see Section 10.12), on almost any platform known to mankind. In any case, the libcurl site is a very good starting point as it has a page with a long list of its competitors and their license types.

If scripting is acceptable, Python with and without ScaPy is a great tool; see Section 5.9. PHP on top of libcurl is also very effective. You may argue that Python and PHP are for web servers, but a larger IoT device may act as a server, and if you are using REST, you are probably implementing a web server. Another library is the "Libmicrohttpd" library which is based on the "GLPL" license; see Section 2.7.

You may prefer a full web server. If Apache or Nginx is too big for your device, the Go-Ahead web server may be interesting. Not so many years ago it was very small and primitive, but it seems to have come a long way since then. Go-Ahead comes with a GPL-license as well as a commercial royalty-free license.

An alternative to the various more or less independent libraries, is to use C# and mono if your platform is Linux based. C# has a large number of great libraries—including some for handling HTTP. With this solution, you get a lot of functionality that you may be spoiled with if you are used to working on the Windows platform. This is a drastic solution to a minor problem. One of the main advantages of the Linux platform is the huge user society available that may help in many cases. If you put C# and mono between your application and the OS, it may not be so simple to find someone in the same situation. Naturally, if you are on the Windows platform then C# is mainstream.

7.22 Ethernet cabling

When it comes to Ethernet cables, there is a lot of baggage from the old days and names and terms that are not completely correct. The most used are given in Table 7.11.

One of the confusing things with Ethernet cables is when to use the EIA-568A wiring scheme and when to use the EIA-568B wiring scheme. Since most cables today are the "straight-over patch-type" it makes little difference. The really important thing is that all cables go straight through *and* that the pairs are maintained, so we do get them twisted pairwise, thus maintaining a high immunity to induced "common mode" noise. The pairs are easily identified by their colors. For example, one wire is green and the other wire in the pair is white with a green stripe.

Table 7.11: Names and terms used with Ethernet cables.

Term	Explanation
Coax	Old standard for multidrop. Not used anymore.
UTP	Unshielded Twisted Pair. This is the most normal type today—used in star configuration.
Fully Mounted	UTP normally has 4 pairs of two wires. Cheaper versions with less wires may work. Do not use them.
F/UTP S/UTP	Two variants with foil shield. This must be connected throughout the network.
CAT 5	Cables supporting 100 Mbps up to 100 m.
CAT 5e	Cables supporting 1 Gbps up to 100 m.
CAT 6	Cables supporting 1 Gbps up to 100 m.
CAT 6a	Cables supporting 10 Gbps/s up to 100 m
8P8C	Real name for connector—8 Positions and 8 Contacts
RJ45	Name almost always used for the connector on Ethernet cables—although not completely correct.
Patch Cable	Normal cables where all connections go straight through. Male connectors at both ends.
Cross-over	Older cables for connecting, for example, two PCs.
Auto M-Dix	Part of std. for 1 Gbps ports that negotiates automatic cross-over when needed.
EIA-568A/B	A and B are two different standards for which wire pair goes where. In a patch cable it really makes no difference. The important thing is that the pairs are maintained.

If you look at a male "RJ45" Ethernet connector from the connector end (cable going away from you) with the small tap/hook downwards and the contacts upwards, the pins are numbered 1–8 from the right side. Table 7.12 shows the connections in the EIA-568A setup. The EIA-568B is the same, except that the orange/white and the green/white pairs are swapped, but as stated, it makes little difference which is chosen. The strange layout is historic, enabling backwards compatibility with 4-pin connectors. It may be easier to understand when explained: There is a pair at the center

Table 7.12: EIA-568A wiring scheme.

Pair	Wire	Pin
1	Blue	4
Blue+White	White/Blue	5
2	Orange	6
Orange+White	White/Orange	3
3	Green	2
Green+White	White/Green	1
4	Brown	8
Brown+White	White/Brown	7

pins (4,5), then another pair on either side of this (3,6) and then one pair at one side (1,2) and another pair at the other side (7,8).

The initial wiring scheme, where pairs are symmetrically added at either side of the center was clever, until it wasn't. The growing space between the wires in a pair became a problem due to noise, and was abandoned. Anyway, if the above instruction is followed, a patch cable will work.

If you are making your own Ethernet cables, I recommend that you buy an Ethernet tester. Figure 7.17 shows a tester costing less than $10. The tester simply uses its battery to test one wire at a time, so if you see a "walking light" from 1 through 8 at the remote end, you are okay.

Figure 7.17: Low-cost Ethernet tester.

7.23 Physical layer problems

In Section 7.17, we saw that a problem on the physical layer showed symptoms in a Wireshark capture, but wasn't really nailed down. Hardware engineers apply something called an eye diagram in these cases. A good sample is shown in Figure 7.18.

Basically, a test program is run that sends all "symbols" in many sequences. The symbols are overlaid on top of each other and variations in time and/or amplitude stands out. Thus, even though various parameters are measured, the diagram is very informative by itself.

There is, however, a huge threshold for an embedded programmer to fire up a test like this, even if the equipment is available, which very often it is not.

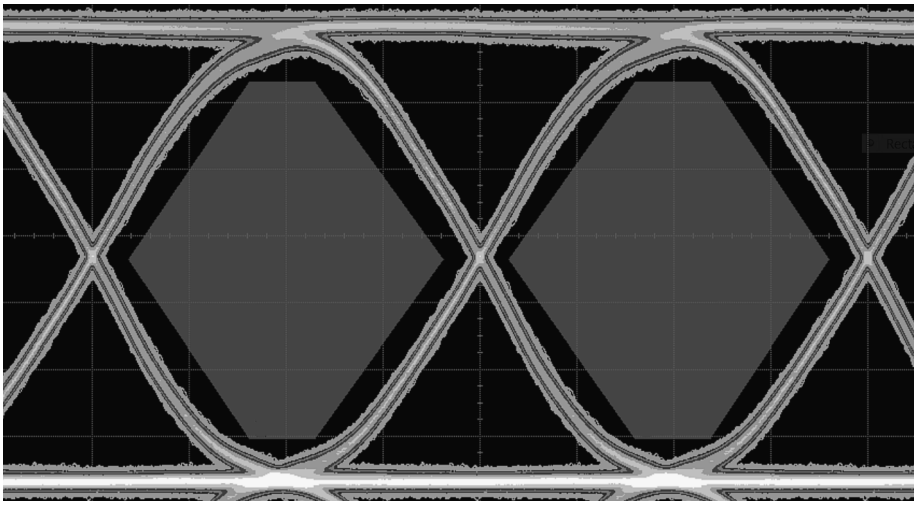


Figure 7.18: A good eye diagram.

My colleagues and I saw the problem with the physical layer in a prototype which we could split open, but what if you want to measure on a closed box? And what if a customer far away has a problem? I have had cases where I could ask a customer to do a Wireshark capture and send it to me, but I have yet to meet the guy who can convince his customer that he should do an eye diagram. There should be a better way to diagnose physical layer problems. Once such a problem is diagnosed, however, it is a good idea to fire up the heavy tools to understand and fix the problem.

The main reason you don't see these things in Wireshark is that a corrupt Ethernet frame also has a bad checksum. It has been stated earlier that modern network interface cards (NICs) have built-in checksum calculations. However, that statement is about the IP and TCP checksums. Even the oldest NICs have a hardware-based check of the Ethernet CRC, which is much more advanced than a checksum. On the Ethernet level, there are no retransmissions. If a "bad" frame is received, it is simply thrown away. This normally happens so rarely that a TCP retransmit is a fine and simple solution. If you are using UDP, it's just bad luck and, therefore, up to your application to handle.

Even though there are no actions taken in the protocol stack, there might be an error counter, and normally there is. In reality, your faithful NIC on the PC, and very often also in an embedded system is counting CRC errors. All you need to do is to read them, but how?

This is where SNMP (simple network management protocol) comes in. Most operating systems actually support this. All you need to do is to download an SNMP client,

fire it up, key in your PC or your device IP address, and you are in business. See Section 8.4.

You can get libraries for SNMP, and thus build this kind of test into your own diagnostics. This will allow your customers to run the test and mail the result.

Inexperienced software developers often jump to the conclusion that the bug is not in their software, it must be a hardware problem. More senior developers have experienced so many software bugs that they always look for the bug in software first, and then they check and check again. But one place where you actually should be prepared for hardware problems is the PHY, the device sitting between the MAC and the magnetics (and the connector), responsible for converting the more or less[14] analog waveform on the wire to a bit pattern. Many types of transients[15] can occur at the wire. They will go through the connector and the magnetics to the PHY, where they may kill the signal. In some scenarios, they may even kill the PHY. The correct protection circuitry is thus important here.

14 Depending on the "coding scheme" the bits are almost sent "as is" or coded into softer waveforms—unrecognizable on an oscilloscope.

15 Short peaks in currents and/or voltages, for example, electromagnetically induced.

7.24 Further reading

- Kurose and Ross: *Computer Networking: A Top-Down Approach*
  I used to teach at the Danish Technical University based on this book. The top-down approach is good when the whole concept of networks is new. As teaching material for 101 on Computer Networks it cannot and should not dive deep, but it gives a great overview. It also includes a very good chapter on security: ciphers, symmetric versus public, and private keys, etc.
- Laura Chappel: *Wireshark Network Analysis*
  An extremely detailed guide to Wireshark. This is a book filled with information and tricks.
- Stevens: *TCP/IP Illustrated Volume 1*
  This is *the* book on the internet protocol stack. It is old now and sadly will not be updated by Stevens. Still it is extremely well-written. If you are interested in networking, volume 1 is a must. Volume 2 is about implementing the stack, and volume 3 is about application layer protocols such as HTTP, and newer sources are recommended on these subjects.
- Richardson and Ruby: *RESTFul Web Services*
  This book explains REST very well. It is not really targeted for the embedded world but all the basic principles are the same as on big web servers.
- tcpipguide.com
  A very informative website with many good figures.

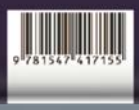
Even the most conservative estimates show the Internet of Things will be an enormous market where innovation is necessary and where programmers are and will be in great demand. The challenge is that programming the IoT spans many knowledge domains. This book will provide the fundamentals of embedded programming to programmers and enable them to reach their full potential.

Embedded Software for the IoT helps the reader understand the details in the technologies behind the devices used in the Internet of Things. It provides an overview of IoT, parameters of designing an embedded system, and good practice concerning code, version control and defect-tracking needed to build and maintain a connected embedded system.

After presenting a discussion on the history of the internet and the world wide web, the book introduces modern CPUs and operating systems. The author then delves into an in-depth view of core IoT domains including:

- Wired and wireless networking
- Digital filters
- Security in embedded and networked systems
- Statistical process control for Industry 4.0

This book will benefit software developers moving into the embedded realm as well as developers already working with embedded systems.



www.degruyter.com
ISBN 978-1-5474-1715-5

For more information, contact jaya.dalal@degruyter.com