Socket Programming

Unit 13 - Hands-On Networking - 2018

Prof. Dr.-Ing. Thorsten Herfet, Andreas Schmidt, Pablo Gil Pereira

Telecommunications Lab, Saarland Informatics Campus, 23rd Feb. 2018

Recap

- IP addresses identify **hosts**.
- Ports identify **processes** on a host.
- TCP and UDP are **transport layer** protocols.
- **Services** to the user are provided by the **application layer**.
- **Client-Server** and **Peer-to-Peer** architectures can be used.

What is a Socket?

Intuition: A in the one could with.

- Wall: Operating System
- Hole: API
- Do Stuff: Communicate Information (Send/Recv)
- Socket: OS Object

Source

Sockets are **the** network programming interface.

- Originally developed for UNIX, now everywhere.
- Well-defined network API for common network operations.
- Developing application layer protocols requires socket programming.

Lazy Programmers: As TCP sockets are a bit complicated, devs tend to use HTTP for writing endpoints. It is easier, provides messages and there are frameworks.

Socket API

Abstractions

- API provides primitives (`bind`, `send`).
- Sockets abstract nasty details away:
    - Buffers incoming data.
    - Reliably transmit data.
    - Establish / teardown connections.
- Concepts shared amongst implementations.

Language Support

- Socket syntax differs between languages.
- We are using Python Low-Level Socket API.
- Python syntax similar to `glibc`.
    - Nice to use, as we can deal with objects in Python.
    - Don't have to mess around with file descriptors.

Learning Objectives

- How to use UDP, TCP, and raw sockets.

Why low-level API and no frameworks?

Dealing directly with socket API lets you understand...

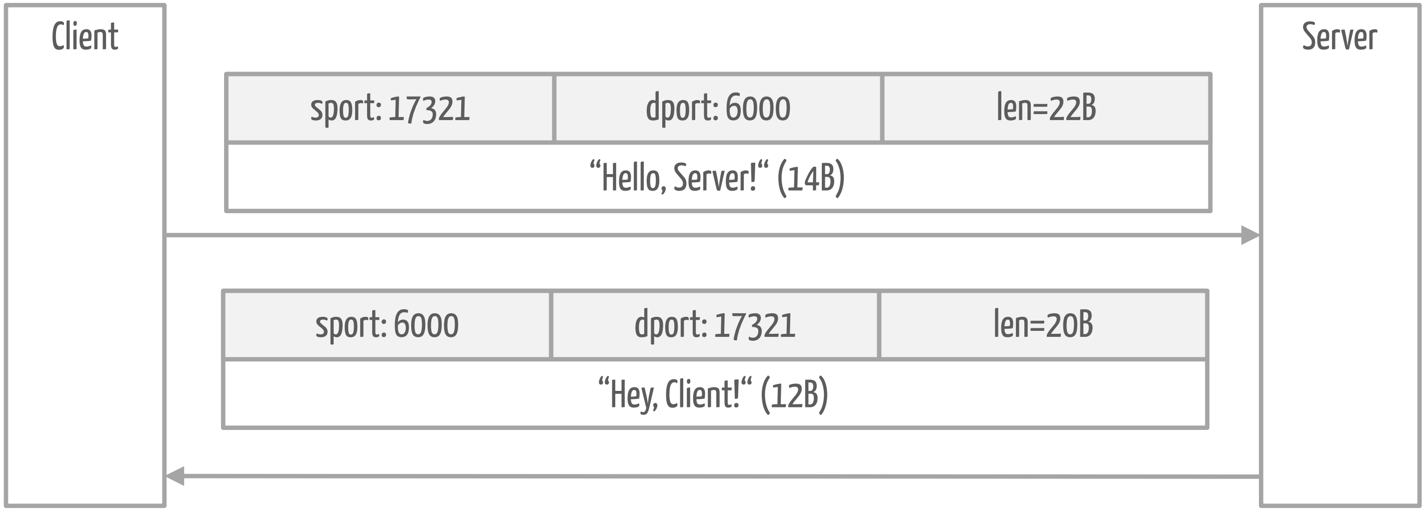
- ... the different states a socket can be in.
- ... how different parameters of sockets can be manipulated.
- ... what frameworks take over for you.
- ... what may be the reason when things don't work.

UDP Sockets

Motivating Example

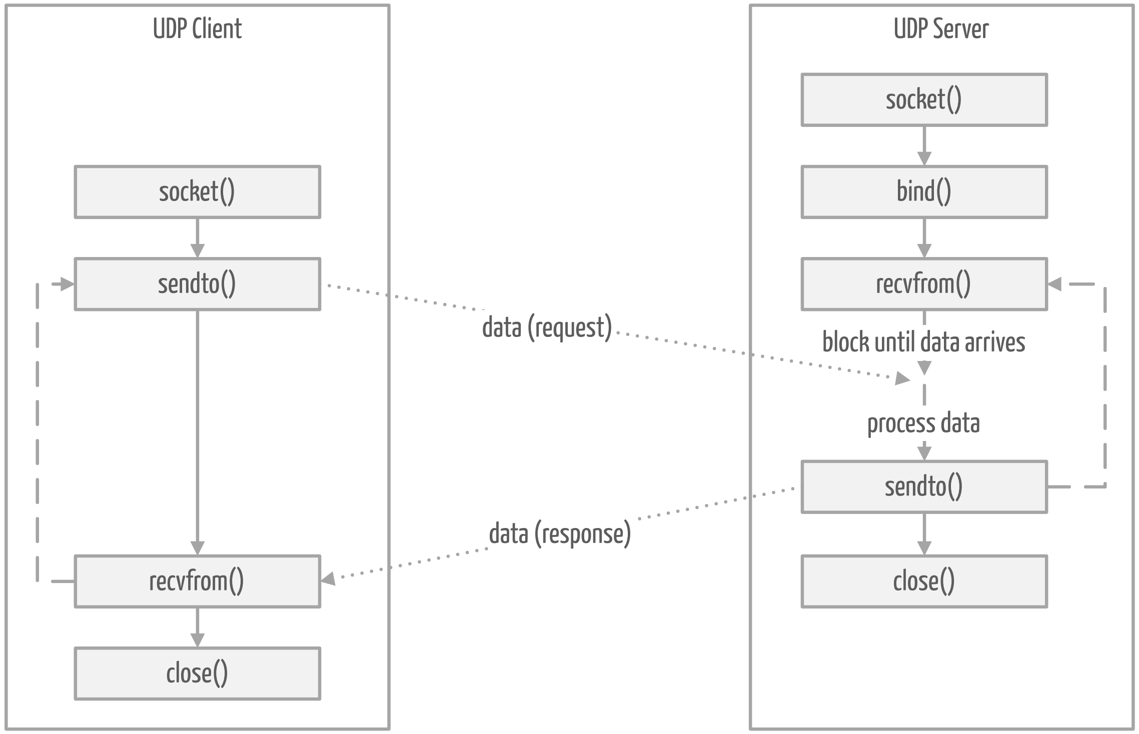
Goal: Simple UDP client/server programs.

- Client sends data (request).
- Server waits for request and responds (reply).



HON

UDP | State Graph



UDP | Socket Creation (Step 1)

`socket.socket(family, type, proto)`

- Client and server create sockets.
- Returns: socket object
- `family`:
  - `AF_INET`: IPv4 socket
  - `AF_INET6`: IPv6 socket
  - `AF_PACKET`: device-level socket
    (receive raw packets)
  - `AF_UNIX`: UNIX socket
- `type`:
  - `SOCK_STREAM`: TCP socket
  - `SOCK_DGRAM`: UDP socket
  - `SOCK_RAW`: Raw socket
- `proto`: usually zero (0)

```
import socket
s = socket.socket(
        socket.AF_INET,          # family (IPv4)
        socket.SOCK_DGRAM,       # type (UDP)
        0)                       # proto (default)
```

Protocols

- The default of 0 lets the OS pick the appropriate **protocol**.
  See IP protocol field in U10.
- Examples:
  - `IPPROTO_TCP`: 6
  - `IPPROTO_UDP`: 17
- All protocols can be found in `/etc/procotols` on UNIX.

UDP | Binding (Step 2)

`s.bind(address_tuple)`

- Server determines service port.
  (mandatory step)
- Client can chose source port, otherwise random
  (optional step)
- `address_tuple`: Address and port:
  - IP address: specific IP, or `0.0.0.0` to bind to all IPs.
  - Port: specific port, or 0 for random port assignment.
- `bind()` can only be called once per socket.
- Binding to ports <= 1023 requires root privileges.

```python
import socket
s = socket.socket(
        socket.AF_INET,         # family (IPv4)
        socket.SOCK_DGRAM,      # type (UDP)
        0)                      # proto (default)

s.bind(("192.168.0.1", 4567)) # (IP, port)
```

UDP | Receiving (Step 3a)

`s.recvfrom(bufsize[, flags])`

- Returns: `(buffer, address)` tuple
    - `buffer`: byte string with received data
    - `address`: address of sender (tuple of IP and port)
- `bufsize`: Maximum number of bytes to receive.
- `flags`: options (see later)

```python
import socket
s = socket.socket(
        socket.AF_INET,                # family
        socket.SOCK_DGRAM,             # type
        0)                             # proto

s.bind(("192.168.0.1", 4567))         # (IP, port)

(data, address) = s.recvfrom(65536) # bufsize

print(f"Received {len(data)} bytes")
```

- Blocking, unless otherwise specified.
- Returns entire single UDP segment.
- Specify maximum UDP payload size as `bufsize` (round up to next 2^x)
- `bufsize` = 2^{16} = 65536
- or: pass `MSG_TRUNC` flag option

- Segment payload larger than `bufsize` is discarded.
- Example: `recvfrom(1024)` with payload of 2024 bytes will return first 1024B and discard remaining 1000B.

HON

UDP | Send (Step 3b)

`s.sendto(buf, flags, addr)`

- Returns: number of bytes sent (`len(buf)`)
    - `sendto` atomic for UDP: entire buffer is sent (but no guarantee on reception)
- `buf`: byte string to be sent.
    - Has to fit into UDP payload of single IP packet.
    - Maximum size (IPv4): `65507`
- `flags`: optional control flags.
- `addr`: address tuple of recipient.

```
import socket
s = socket.socket(
        socket.AF_INET,             # family
        socket.SOCK_DGRAM,          # type
        0)                          # proto

s.bind(("192.168.0.1", 4567))       # (IP, port)

(data, address) = s.recvfrom(65536) # bufsize

data = data.decode("utf-8").toupper()

c = s.sendto(data.encode("utf-8"),  # buf
             0,                     # flags
             address)               # addr
```

Multicast: `addr` is specified when sending data and returned when receiving data, making UDP sockets usable for **one-to-many** transmissions.

HON

UDP | Maximum Segment Size (IPv4)

UDP Length field is 16 bit large, hence maximum theoretical length is: $UDP_{max} = 2^{16} - 1 = 65535$.

IP Total Length field is 16 bit large, hence maximum theoretical length is: $IP_{max} = 2^{16} - 1 = 65535$ (too).

UDP datagram must fit into one IP packet. Probably fragmented, but maximum length $IPmax$ holds.

Operating systems (Linux, etc.) know that you are putting UDP segments into IP packets.

IP ($IP_{hdr} \geq 20$) and UDP ($UDP_{hdr} = 8$) have a minimal / fixed header size.

Consequently, the practical maximum size of a UDP segment is: $IP_{max} - IP_{hdr} - UDP_{hdr} = 65535 - 20 - 8 = 65507$

UDP | Close (Step 4)

`s.close()`

- Flushes all data if pending.
- Closing UDP sockets:
    - Free system resources.
    - Undo binding.
    - No explicit notification of remote side.

```
import socket
s = socket.socket(
        socket.AF_INET,             # family
        socket.SOCK_DRAM,           # type
        0)                          # proto

s.bind(("192.168.0.1", 4567))       # (IP, port)

(data, address) = s.recvfrom(65536) # bufsize

data = data.decode("utf-8").toupper()

c = s.sendto(data.encode("utf-8"),  # buf
            0,                      # flags
            address)                # addr

s.close()
```

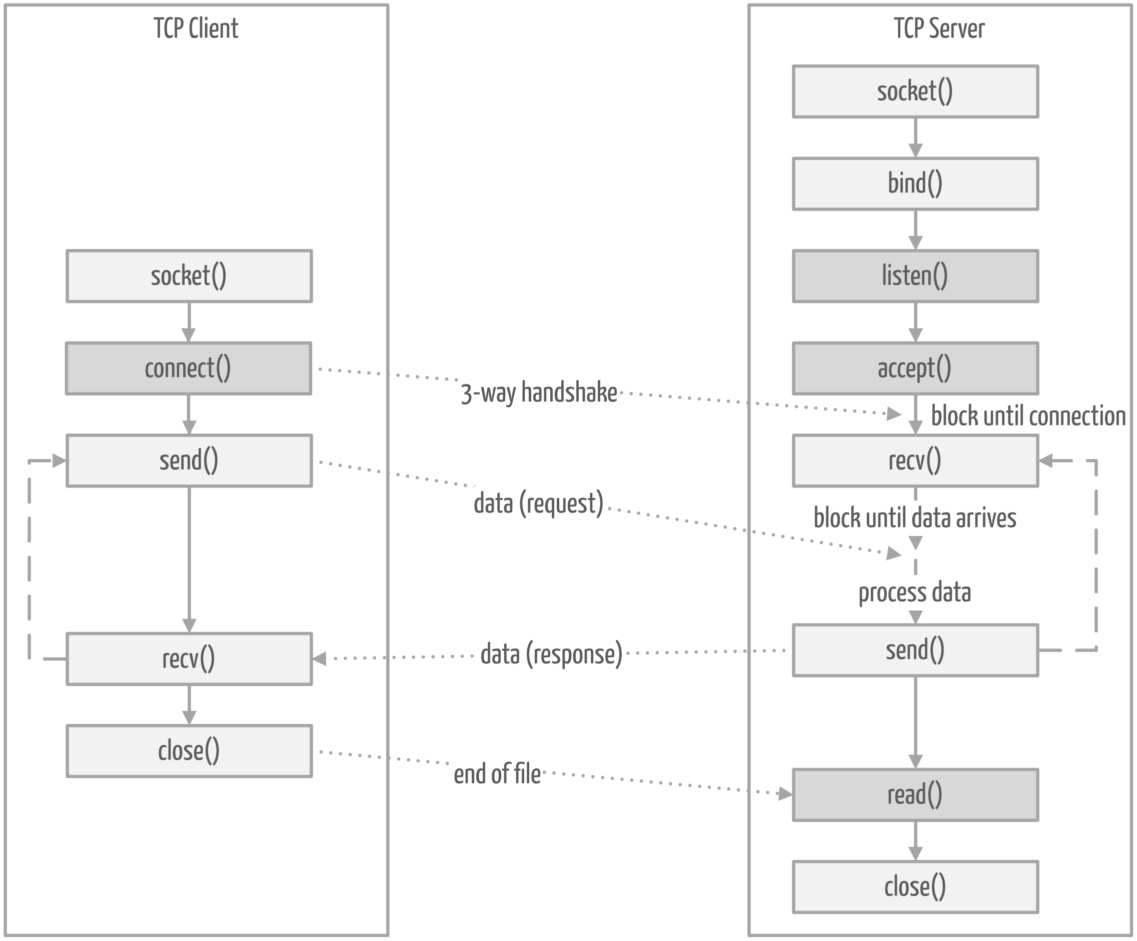
TCP Sockets

TCP | Motivation

- TCP is
    - connection-oriented.
    - stream-oriented.
    - reliable.

- TCP sockets provide
    - similar functions to UDP.
    - additional connection management.
    - additionally message segmentation.

TCP | State Graph



TCP Server | Preparation (#1)

A) Create socket

- Command: `socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0)`

B) Bind socket to port (similar to UDP)

- Command: `s.bind(address_tuple)`
- `address_tuple`: IP and port at which to wait for clients.

C) Listen for connections made to socket

- Starting at this command, kernel queues connections
- Command: `s.listen(backlog)`
- `backlog`: maximum number of queued connections

TCP Server | Connection Backlog

Concept

- After `listen`, kernel automatically completes TCP handshakes.
- OS kernel has two queues:
    - Incomplete connections: SYN received.
    - Completed connections: Handshake completed.
- `backlog`: maximum length of total entries (sum of both queues).
- Additional incoming TCP connections are ignored.

Design Decision

- Backlog of 0 is allowed, but can trigger unspecified behaviour.
- Backlog of 5 is a common value, but only suitable for small servers.
- Modern TCP-based servers have much larger values.

TCP Server | Accepting Connections (#2)

`s.accept()`

- Remove connection from completed `listen` queue.
    - First in, first out (FIFO) order.
- Returns: (conn, address) tuple.
    - `conn`: socket object of TCP connection.
    - `address`: address of sender (tuple of IP and port).
- `accept` is a blocking call:
    - Wait until client connects.
    - Blocks forever if nobody connects.

```python
import socket
s = socket.socket(
        socket.AF_INET,      # family
        socket.SOCK_STREAM,  # type
        0)                   # proto

s.bind(("0.0.0.0", 4567))    # (IP, port)

s.listen(10)                 # backlog

(conn, addr) = s.accept()    # block until connect

(src_ip, src_port) = addr    # connection accepted

print(f"Connection from {src_ip}:{src_port}")
```

Quiz

Assume a socket has a backlog of 5 connections. What is the state after the following sequence of events?

- SYNs from 3 sources.
- Handshake completed with additional 2 sources.
- 1 socket `accept()`ed.

A: Backlog is full.

B: One slot free.

C: Two slots free.

D: Three slots free.

Answer:

A: Wrong, `accept()` removes one.

C: Completed connections are not automagically removed.

B: True, because 3 incompletes remain and one completed remains.

D: Why? Off-by-two, maybe?

TCP Client | Establish Connection

A) Create Socket

- Command: `socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0)`

B) Connect to server

- Command: `s.connect(address_tuple)`
- `address_tuple`: IP and port of destination server
- Socket connected when `connect` returns.

```python
import socket
s = socket.socket(
        socket.AF_INET,          # family
        socket.SOCK_STREAM,      # type
        0)                       # proto

s.connect(("127.0.0.1", 4567))   # (IP, port)

print("Successfully connected.")
```

TCP Sockets | Recv / Send (#3a/b)

Receiving

- Command: `s.recv(bufsize[, flags])`
- Returns: Buffer that was received (up to `bufsize` bytes).
  - Empty buffer indicates other side **terminated** the connection.
- `bufsize`: Maximum number of bytes to receive.
- `flags`: options (see later)
- Blocking, unless otherwise specified.

Sending

- Command: `s.send(buf[, flags])`
- Returns: Number of bytes sent.
  - `send` is **not** atomic for TCP: partial data might be sent.
- `buf`: Byte string to be sent.
  - Can be of any size, will be fragmented accordingly.
  - If too large, `send()` will only send **partial data**.
- `flags`: options (see later)

❓ **For UDP, `recvfrom()` returned source address tuple. For TCP `recv()` does not. Why?**

TCP Send/Recv | Correct Usage

- Issues with `recv` and `send`:
  - May return **less than** `bufsize` bytes.
  - `recv` may return multiple messages/packets.
  - `send` may not send all data passed as argument.
- Corner Cases:
  - You may need multiple `send` calls to transmit all data.
  - Multiple `send` calls may be handeled by a single `recv` call.
  - A single `send` call may need multiple calls to `recv`.
- Cases may not be encountered under perfect conditions (e.g. testing), but it is quite common under high load.

Reliably *Send* all data

```
def send_all(s, buf):
    buf_len = len(buf)
    bytes_sent = 0
    while bytes_sent < buf_len:
        c = s.send(buf[bytes_sent:])
        if c == 0:
            raise RuntimeError("Socket " +
                "Connection Broken.")
        bytes_sent += c
```

- `sendall()` in Python returns only if all data was sent (more convenient than C calls).

Reliably *Recv* all data

- Not trivial.
- Depends on message separation (e.g. `\n`, `\0` or `\n\r`).

TCP Send/Recv | Correct Usage II

Idea: Reliably receive all data by using fixed message length.

Approach: Use `MSG_WAITALL`.

- Optional flag for `recv()`.
- `recv()` waits until `bufsize` bytes have been received.
- (… or a signal was caught.)
- (… or the connection was terminated)

Problem: Protocols with fixed length are usually bad.

Why is fixed length bad?

A: Sending larger messages than predefined size is not easy.

B: Fixed size messages waste space, when the message is shorter than the fixed length.

C: Sender and recipient have to agree on length, but might have different opinions.

✓ A,B,C: True!

TCP Message Separation Options

Sender and receiver have to agree on message separation.

A) Fixed message length (e.g 1024 bytes).

- Easy to program, but lots of padding overhead.
- Large messages may not fit. **Bad design** in most cases.

B) Only one message per socket.

- Connect, send, recv, disconnect (e.g. basic HTTP used this).
- Connection establishment overhead.

C) Delimit messages by well-defined string/character (e.g. `\n`).

- Need escaping (or well-defined delimiter forbidden in message).
- Unclear how much needs to be read to reach delimiter.

D) Message length indicated at start of message.

- Acceptable overhead, very flexible.
- Message length field must have fixed size (therefore limits overall size).

TCP Sockets | Closing (#4)

```
s.close()
```

- Flushes all data if pending.
- Closing TCP sockets:
  - Send `FIN` segment ("I am done sending data.")
  - Other side calling `recv()` will get empty buffer.
  - At later point, remote will also close and send `FIN`.

Raw Sockets

Raw | Motivation

- TCP, UDP sockets:
    - Do most of the work for you (put data into a socket, give an address and the rest is figured out by the OS).
    - Required for transport layer networking.
- Sometimes, it is necessary to work with protocol on lowest layers:
    - ICMP: directly in IP without UDP/TCP.
    - ARP: Ethernet frame, without IP.
    - ...
- Security testing requires you to work on raw layer, so that packets can be crafted that would never be created in such a way by an OS.
- For more control and low-level networking: Use raw sockets!
    - Sender **can** generate arbitrary packets (including IP and Ethernet).
    - Sender **must** specify every detail of packets (including checksums).

Raw | Sending Packets

A) Create a socket with `socket.SOCK_RAW`.

- `socket.socket(socket.AF_PACKET, socket.SOCK_RAW, 0)`
- Note: Requires root/admin privileges (as we are circumventing the kernel/OS).

B) Bind to an interface (instead of IPv4 address).

- `s.bind(("eth0", 0))`

C) Generate and send packet.

- `s.send("A" * 20)` (malformed packet)
- Include Ethernet and IP header if needed.

Raw | Sending Packets (Convenient)

Generate packets using third-party library `dpkt`.

```
import dpkt

# create Ethernet, IP and UDP objects
eth = dpkt.ethernet.Ethernet(src="08:00:27:bf:17:b4", dst="ff:ff:ff:ff:ff:ff")
ip = dpkt.ip.IP(src="127.0.0.1", dst="127.6.6.6")
udp = dpkt.udp.UDP(sport=1234, dport=5678)

eth.data = ip         # set IP pkt as Ethernet payload
ip.data = udp         # set UDP segment to be IP pkt payload
ip.len = len(ip)      # set IP length
udp.ulen = len(udp)  # set UDP length

repr(ip)              # implicitly computes IP checksum
# Output:
# "IP(len=28, src='127.0.0.1', dst='127.6.6.6', data=UDP(sport=1234, dport=5678))"

repr(eth)             # show Ethernet packet
# Output:
#
# "Ethernet(dst='ff:ff:ff:ff:ff:ff', src='08:00:27:bf:17:b4',
#           data=IP(len=28, src='127.0.0.1', dst='127.6.6.6',
#           data=UDP(sport=1234, dport=5678)))"
```

HON

Raw | Receiving Packets

Goal: Receive all packets on interface.

- Create socket with `socket.SOCK_RAW`:

```
ETH_P_ALL = socket.ntohs(0x0003) # all Ethernet protos
socket.socket(socket.AF_PACKET, # domain
              socket.SOCK_RAW,  # type
              ETH_P_ALL)        # protocol (non-empty / non-default)
```

- Put NIC to **promiscuous mode** and even get packets not intended for you.
- Receive packet:
  - `s.recv(4096)`
  - Return Ethernet frame as captured on interface.
  - Note: captures packets even if they are filtered by firewall.

Raw | Receiving Packets (Convenient)

Parse packets using third-party library `dpkt`.

```
import dpkt

pkt = s.recv(4096)            # receive via raw socket

dpkt.ethernet.Ethernet(pkt)   # print dpkt parsing result

# Output
# Ethernet(dst="\x08\x00'\xbf\x17\xb4", src='RT\x00\x125\x02',
#          data=IP(len=40, id=17571, p=6, sum=7709, src='\n\x00\x02\x02', dst='\n\x00\x02\x0f', opts='',
#          data=TCP(sport=54945, dport=22, seq=126587142, ack=4007547311, flags=16, sum=59372, opts='')))

eth = dpkt.ethernet.Ethernet(pkt)   # get Ethernet frame object
ip = eth.data                       # get IP packet object
tcp = ip.data                       # get TCP packet object
tcp.dport # access member variables
# Output:
# 22
```

Raw | Power

"With Great Power Comes Great Responsibility" -

Network Link Sniffing

- See every packet on the wire.
  And many of those in the air.
- Passive attacks such as local eavesdropping become easy.

Arbitrary Packet Injection

- You can inject completely arbitrary packets.
- "Spoof" any arbitrary MAC or IP address(es).
- Low resource requirements (in contrast to full TCP/IP stack).

Wrap-Up

Questions?

Take-Home Messages

- **Sockets** are **the API** for network programming and an important concept.
- **UDP socket** offer atomic send/receive operations.
- **TCP sockets** require more attention (connection handling, partial receive/send).
- **Raw sockets** give your complete control, but also put burden on you.

Further Reading

- Python 3.6 - Low-level networking interface
- Python 3.6 - Struct module
- Stevens, Fenner, Rudoff: UNIX Network Programming, Volume 1
  - Book examples in C, we use Python.