changelog

12 Nov 2024: add “detecting connection close”

port numbers

we run multiple programs on a machine

- IP addresses identifying machine — not enough

port numbers

we run multiple programs on a machine

- IP addresses identifying machine — not enough

so, add 16-bit *port numbers*

- think: multiple PO boxes at address

port numbers

we run multiple programs on a machine

- IP addresses identifying machine — not enough

so, add 16-bit *port numbers*

- think: multiple PO boxes at address

0–49151: typically assigned for particular services

- 80 = http, 443 = https, 22 = ssh, ...
- usually <1024 reserved for sysadmin-only-use

49152–65535: allocated on demand

- default "return address" for client connecting to server

sockets

port number helps identify *sockets*

sockets = file applications use to access networks

port number independence

typically "5-tuple" identifies "socket":
(protocol=TCP or UDP, source IP, dest IP, source port, dest port)

means can have:

- two sockets with same source IP+source port, but different destinations
- two sockets with same dest IP+des port, but different sources
- two sockets with different protocols, but everything else same

special cases:

- dest IP/port can be 'wildcard' (all zeroes usually)
- used for server that can be connected to
- used for unconnected UDP sockets

UDP format

(IPv4/IPv6 header)

source port	destination port
length	checksum

(application data)

BSD sockets

BSD = Berkely Standard Distribution (of Unix)

interface designed for stream connections

current C version standardized by POSIX

pretty much default interface programming with TCP connections

...across all platforms

sockets as generic interface

sockets designed to be protocol-agnostic

support all sorts of protocols with addresses

support streams and datagrams

unconnected UDP sockets (Python)

```
import socket
# ipv4
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM,
                     socket.IPPROTO_UDP)
# ipv6
sock = socket.socket(socket.AF_INET6, socket.SOCK_DGRAM)
sock = socket.socket(socket.AF_INET6, socket.SOCK_DGRAM,
                     socket.IPPROTO_UDP)

sock.bind((local_host, local_port))
# local_host == "0.0.0.0" (v4) or "::" (v6) for "all possible

data1, (host1, port1) = sock.recvfrom(MAX_SIZE)
sock.sendto(data, (host2, port2))
```

unconnected UDP sockets (Python, generic)

```
import socket
possible_local_addrs = socket.getaddrinfo(
        local_host, local_port,
        family=socket.AF_UNSPEC,
        type=socket.SOCK_DGRAM,
        flags=socket.AI_PASSIVE)

for af, socktype, proto, canonname, sockaddr in possible_local_addrs:
    sock = socket.socket(af, socktype, proto)
    sock.bind(sockaddr)
    # then use sock for recvfrom/sendto somehow
```

unconnected UDP sockets (Python, generic)

```
import socket
possible_remote_addrs = socket.getaddrinfo(
        remote_host, remote_port,
        family=socket.AF_UNSPEC,
        type=socket.SOCK_DGRAM,
        flags=0)

for af, socktype, proto, canonname, sockaddr in possible_remote_addrs:
    sock = socket.socket(af, socktype, proto)
    try:
        # hope local_host is acceptable for 'af'?
        # maybe choose local_host based on 'af' value?
        sock.bind((local_host, local_addr))
        sock.sendto(data, ...)
        ...
    except ...:
        ...
        # hope another address works?
        continue
```

aside: getaddrinfo

lookup host+port by name

can handle both IPv4 and IPv6

AI_PASSIVE = for bind()

returns list of addresses

- sometimes: just choose one address (send message to server)
- sometimes: want to make socket for each address (run server)

unconnected UDP sockets (C, IPv4)

```
int fd = socket(AF_INET, SOCK_DGRAM, 0 /* or IPPROTO_UDP */);
/* for 10.1.2.3 port 12345
   there are utility functions to help with this, so you
   shouldn't actually construct local_addr manually like this */
struct sockaddr_in local_addr;
memset(&local_addr, 0, sizeof(local_addr);
local_addr.sin_family = AF_INET;
local_addr.sin_port = htons(12345);
local_addr.sin_addr.s_addr = htonl((10<<24)|(1<<16)|(2<<8)|3),
bind(fd, &local_addr, sizeof(local_addr));
struct sockaddr_in remote1, remote2;
recvfrom(fd, buffer1, buffer1_size, 0, &remote1, sizeof(remote1));
recvfrom(fd, buffer2, buffer2_size, 0, &remote2, sizeof(remote2));
struct sockaddr_in remote3 = ...;
sendto(fd, buffer3, buffer3_size, 0, &remote3, sizeof(remote3));
```

connected UDP sockets (Python)

```
import socket
# ipv4
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
# ipv6
sock = socket.socket(socket.AF_INET6, socket.SOCK_DGRAM)

sock.bind((local_host, local_port))
sock.connect((remote_host, remote_port))
sock.send(b"one datagram")
sock.send(b"another datagram")
remote_data1 = sock.recv(MAX_SIZE)
remote_data2 = sock.recv(MAX_SIZE)
```

choosing v4/v6?

nicer to write applications that handle both

...without explicitly doing this differently

later socket API for that is getaddrinfo()

- also exists in C, but lots of extra stuff to handle types, etc.

netstat

```
$ netstat --udp -na
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
udp        0      0 128.143.71.27:60001     0.0.0.0:*
udp        0      0 128.143.71.27:60002     0.0.0.0:*
udp        0      0 128.143.71.27:60003     0.0.0.0:*
...
udp        0      0 128.143.71.27:68        128.143.67.64:67        ESTABLISHED
...
udp        0      0 127.0.0.1:123           0.0.0.0:*
udp        0      0 0.0.0.0:38269           0.0.0.0:*
udp        0      0 0.0.0.0:55407           0.0.0.0:*
udp        0      0 0.0.0.0:55786           0.0.0.0:*
...
udp6       0      0 :::45631                :::*
udp6       0      0 :::111                  :::*
udp6       0      0 fe80::1a66:daff:fe2:123 :::*
...
```

connection setup: client, TCP+IPv4

```
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# to select local host/port: socket.bind(('128.143.71.27', 54444))
# if no local host/port selected, OS assigns
sock.connect(('128.143.67.8', 443))
...
sock.send(...)
some_bytes = sock.recv(max_size)
```

connection setup: client, TCP+IPv6

```
sock = socket.socket(socket.AF_INET6, socket.SOCK_STREAM)
# to select local host/port: socket.bind(('3fff:1234::1', 54444))
# if no local host/port selected, OS assigns
sock.connect(('2607:f8b0:4004:c06::67', 443))
...
sock.send(...)
some_bytes = sock.recv(max_size)
```

connection setup: generic TCP, addrinfo

```
import socket
possible_addrs = socket.getaddrinfo(
        'www.cs.virginia.edu', '443',
        socket.AF_UNSPEC, socket.SOCK_STREAM)

for af, socktype, proto, canonname, sockaddr in possible_addrs:
    try:
        sock = socket.socket(af, socktype, proto)
    except OSError:
        sock = None
        continue
    try:
        sock.connect(sa)
    except OSError:
        sock.close()
        sock = None
        continue

if sock == None:
    # not successful
```

size mismatch? (1)

one end

other end

(get fds/socks:)

```
sock.send(b"12345")
```

```
b"123" == sock.recv(3)
b"45" == sock.recv(2)
```

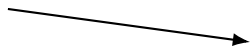
size mismatch? (2)

one end

other end

(get fds/socks:)

```
sock.send(b"12")
sock.send(b"345")
```



```
b"12345" == sock.recv(5)
```

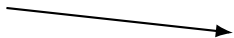
partial reads?

one end

other end

(get fds/socks:)

```
sock.send(b"X" * 6000)
```



sometimes `b"X" * 6000 == sock.recv(6000)`
or sometimes

```
b"X" * 1490 == sock.recv(6000)
b"X" * 2980 == sock.recv(4510)
b"X" * 1530 == sock.recv(1530)
```

partial writes

one end

other end

(get fds/socks:)

```
sock.setblocking(False)
sock.send(b"X" * 6000) == 1480
```

```
b"X" * 1480 == sock.recv(6000)
(+ no more data)
```

partial reads/writes

read/recv can have read less than requested

- read of zero bytes = end of file

send/write can write less than requested...
but only on error or if non-blocking mode requested

incomplete writes

`send` might write less than requested

- error after writing some data
- if blocking disabled

`recv` might read less than requested

- error after reading some data
- not enough data got there in time

reading and writing at once

threads: one reading thread, one writing thread *OR*

event-loop: use *non-blocking I/O* and select()/poll()/etc. functions

- select/poll/etc. tell you when one of many operations possible
- ...including across many sockets

detecting connection close

other end stops sending

- called close or (less common) shutdown or went down

(clean close) recv()/recvfrom() returns empty buffer

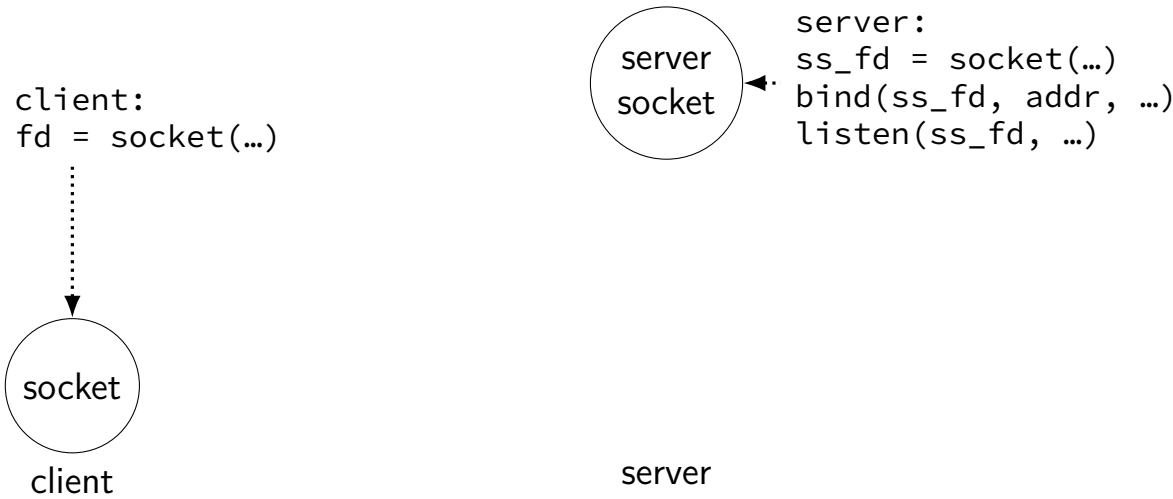
(unclean close) recv()/recvfrom() throw exception

- example: 'Connection reset by peer'
- (C: return -1 + set errno)

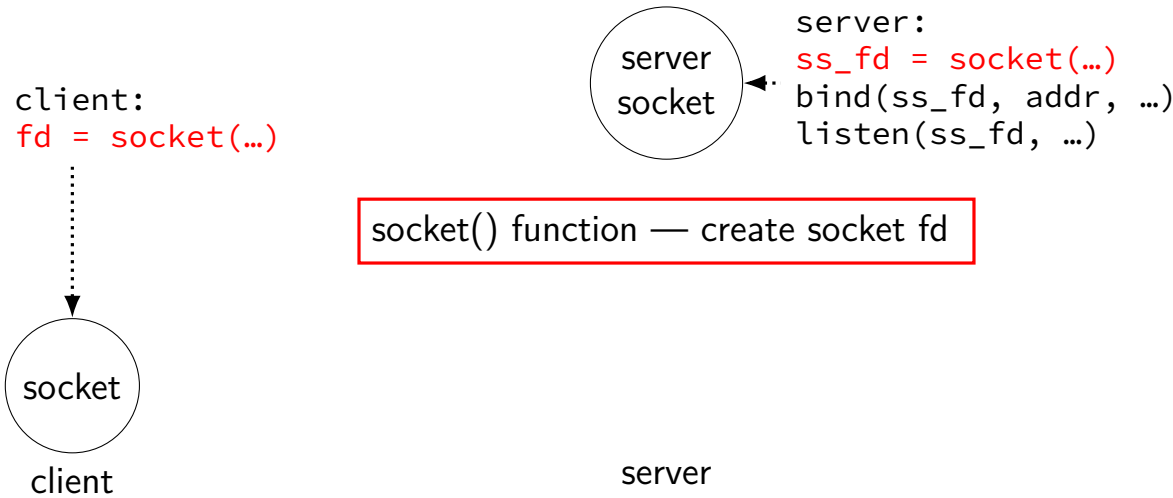
send()/sendall()

- usually BrokenPipeError, possibly other error
- (C: SIGPIPE, unless ignored/MSG_NOSIGNAL; then EPIPE)

sockets and server sockets (C)

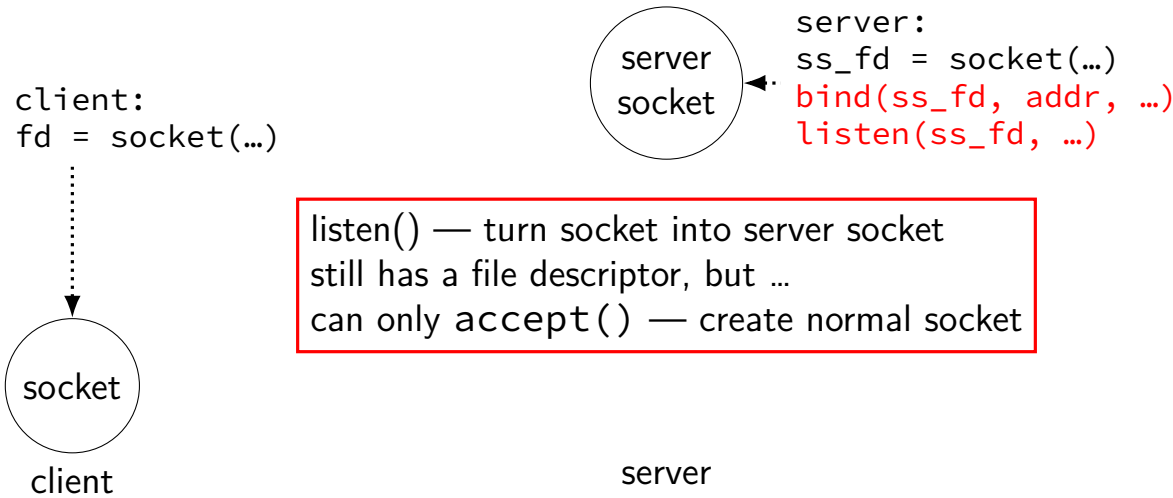


sockets and server sockets (C)

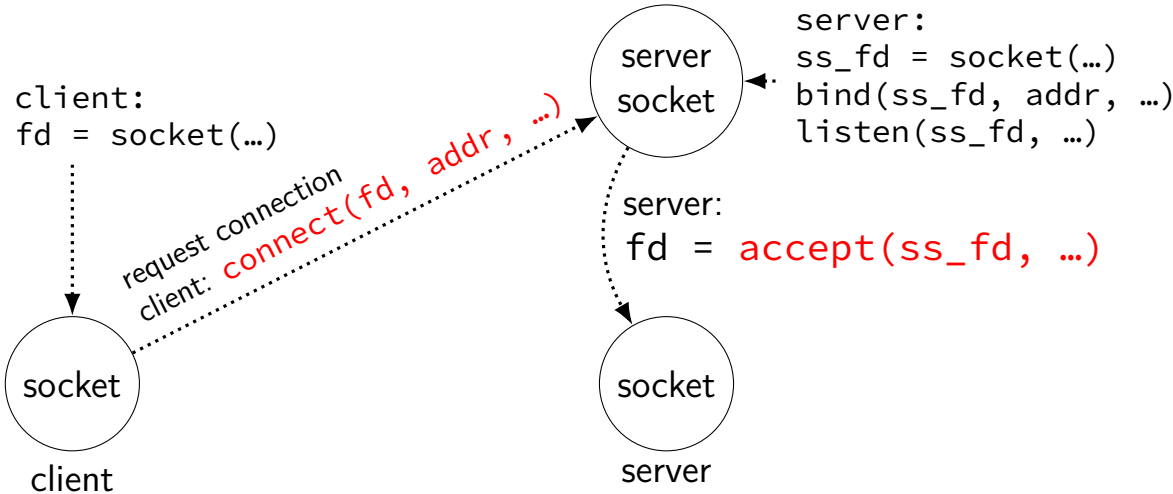


socket() function — create socket fd

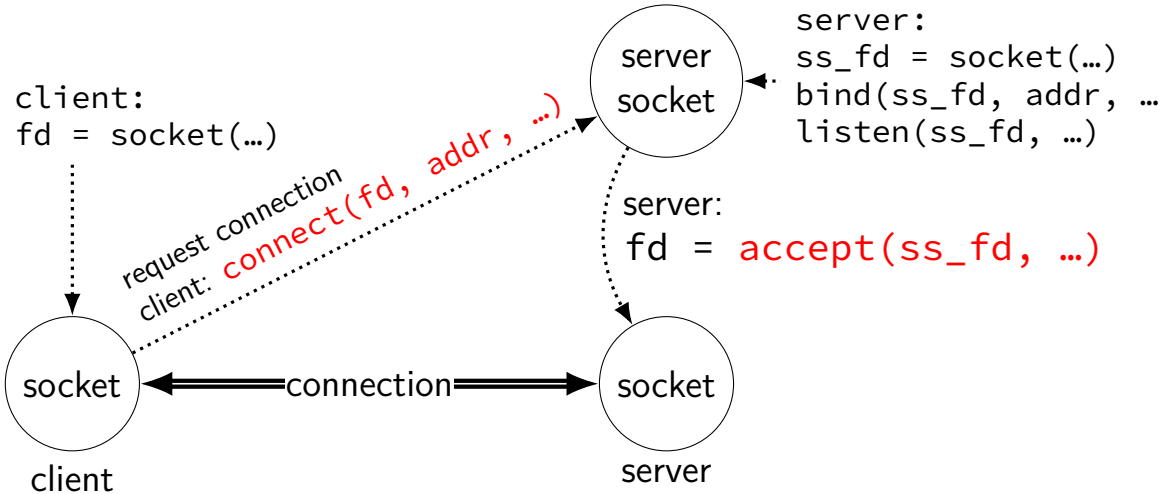
sockets and server sockets (C)



sockets and server sockets (C)

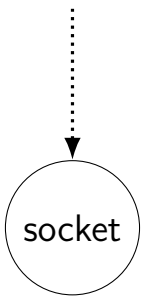


sockets and server sockets (C)



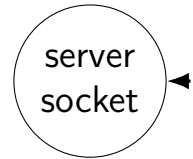
sockets and server sockets (Python)

```
client:
sock = socket.socket(…)
```



client

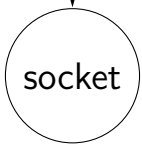
```
server:
ssock = \
 socket.socket(…)
ssock.bind(…)
ssock.listen()
```



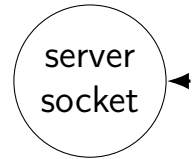
server

sockets and server sockets (Python)

```
client:
sock = socket.socket(…)
```



client

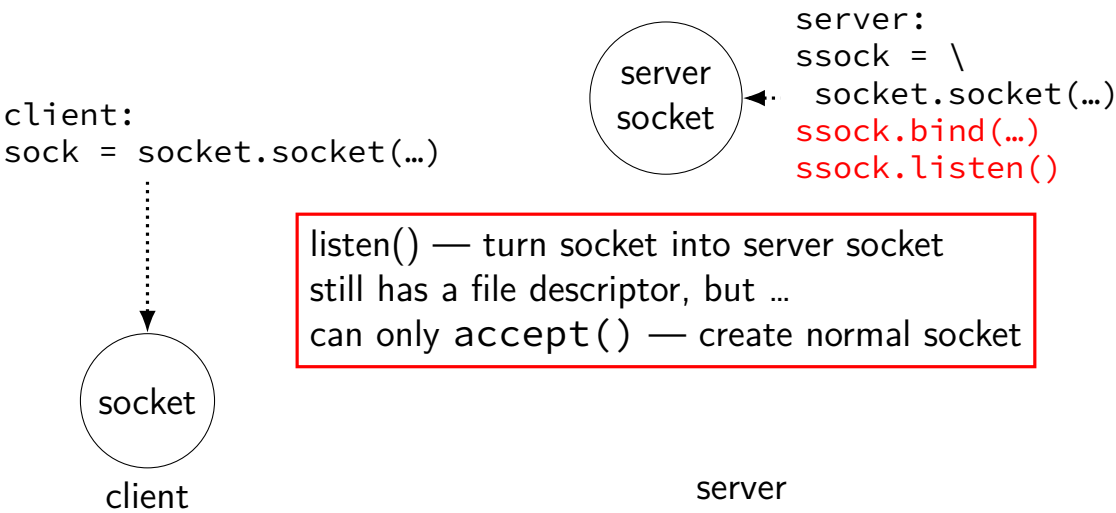



```
server:
ssock = \
 socket.socket(…)
ssock.bind(…)
ssock.listen()
```

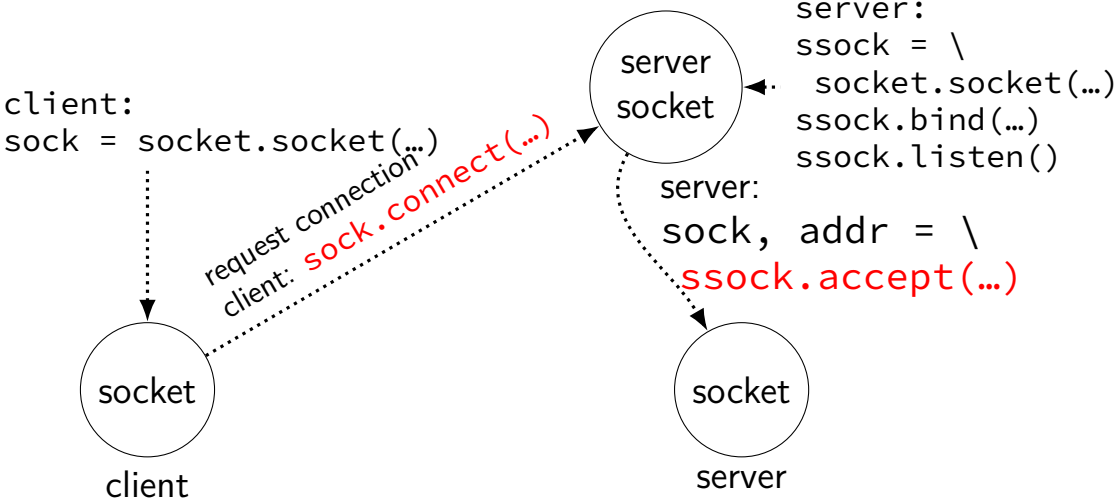
socket() function — create socket fd

server

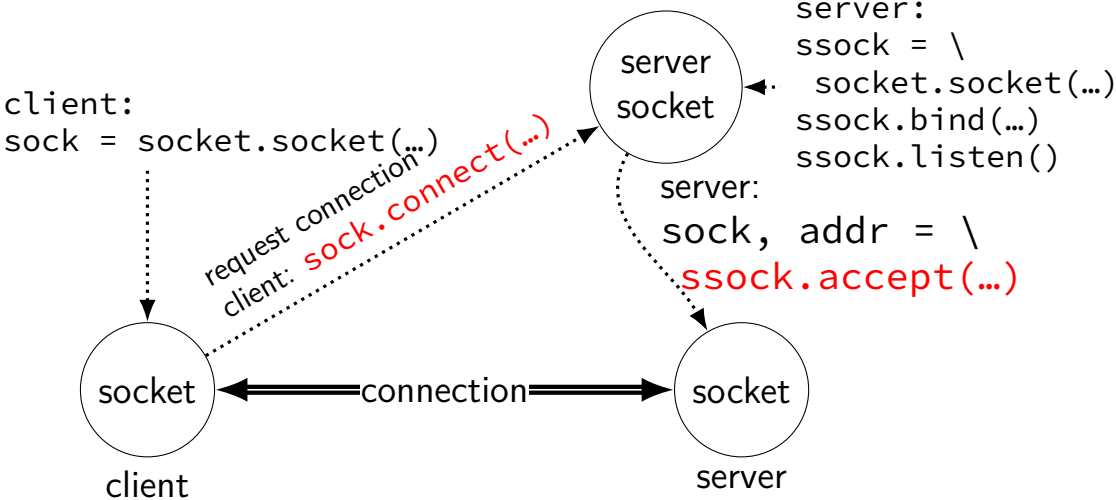
sockets and server sockets (Python)



sockets and server sockets (Python)



sockets and server sockets (Python)



connection setup: server, IPv4

```
import socket
ssock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# 0.0.0.0 == "any IP address"
ssock.bind(('0.0.0.0', 12345))
ssock.listen()
sock, remote_addr = ssock.accept()
Use(sock)
```

connection setup: server, IPv4

```
import socket
ssock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# 0.0.0.0 == "any IP address"
ssock.bind(('0.0.0.0', 12345))
ssock.listen()
sock, remote_addr = ssock.accept()
Use(sock)
```

connection setup: server, generic

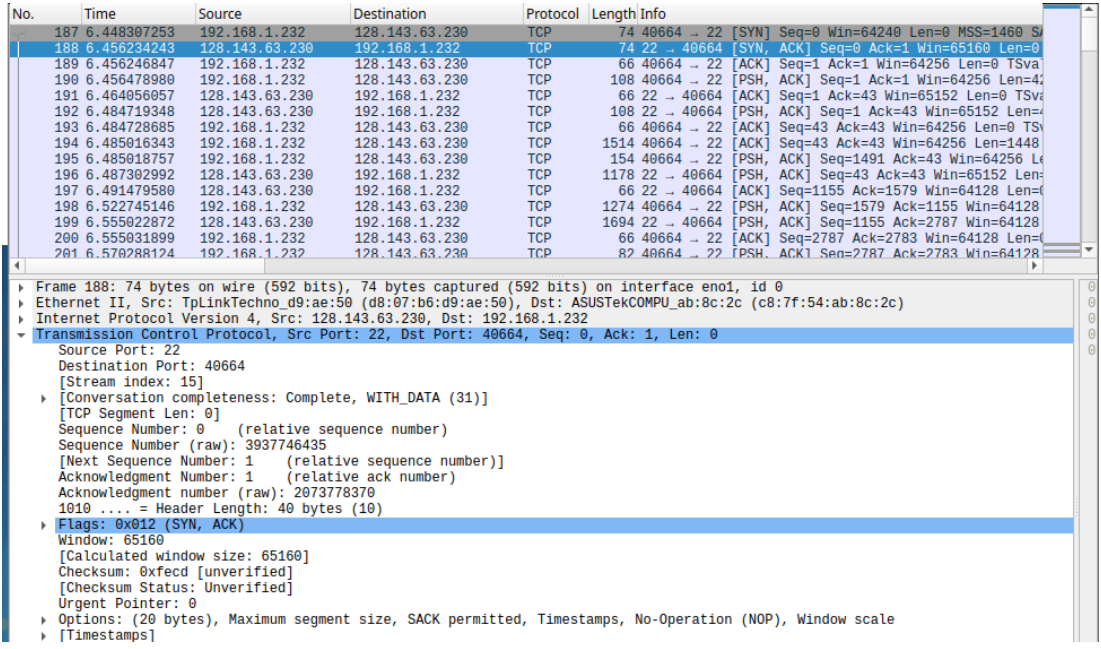
```
import select
import socket
all_addrs = socket.getaddrinfo(
        None, '12345',
        socket.AF_UNSPEC, socket.SOCK_STREAM)
ssocks = []
for af, socktype, proto, canonname, sockaddr in possible_addrs:
    ssock = socket.socket(af, socktype, proto)
    try:
        ssock.bind(sockaddr)
        ssock.listen()
        ssocks.append(ssock)
    except OSError:
        pass

which_ready, _, _ = select.select(ssocks, [], [])
for item in which_ready:
    sock, remote_addr = which_ready.accept()
    Use(sock)
```

three-way TCP handshake

from →to	flags	seq	ack	data
client →server	SYN	X	—	—
server→client	SYN+ACK	Y	$X+1$	—
client →server	ACK	$X+1$	$Y+1$	—
client →server	ACK	$X+1$	$Y+1$	initial data

X, Y selected at random

random selection to prevent attacker from 'spoofing' connection by sending packets with forged source addresses + guessing what to ACK


No. Time Source Destination Protocol Length Info
187 6.448307253 192.168.1.232 128.143.63.230 TCP 74 40664 → 22 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 S
188 6.456234243 128.143.63.230 192.168.1.232 TCP 74 22 → 40664 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0
189 6.456246847 192.168.1.232 128.143.63.230 TCP 66 40664 → 22 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSva
190 6.456478980 192.168.1.232 128.143.63.230 TCP 108 40664 → 22 [PSH, ACK] Seq=1 Ack=1 Win=64256 Len=4
191 6.464056057 128.143.63.230 192.168.1.232 TCP 66 22 → 40664 [ACK] Seq=1 Ack=43 Win=65152 Len=0 TSv
192 6.484719348 128.143.63.230 192.168.1.232 TCP 108 22 → 40664 [PSH, ACK] Seq=1 Ack=43 Win=65152 Len=
193 6.484728685 192.168.1.232 128.143.63.230 TCP 66 40664 → 22 [ACK] Seq=43 Ack=43 Win=64256 Len=0 TS
194 6.485016343 192.168.1.232 128.143.63.230 TCP 1514 40664 → 22 [ACK] Seq=43 Ack=43 Win=64256 Len=1448
195 6.485018757 192.168.1.232 128.143.63.230 TCP 154 40664 → 22 [PSH, ACK] Seq=1491 Ack=43 Win=64256 Le
196 6.487302992 128.143.63.230 192.168.1.232 TCP 1178 22 → 40664 [PSH, ACK] Seq=43 Ack=43 Win=65152 Len
197 6.491479580 128.143.63.230 192.168.1.232 TCP 66 22 → 40664 [ACK] Seq=1155 Ack=1579 Win=64128 Len=
198 6.522745146 192.168.1.232 128.143.63.230 TCP 1274 40664 → 22 [PSH, ACK] Seq=1579 Ack=1155 Win=64128
199 6.555022872 128.143.63.230 192.168.1.232 TCP 1694 22 → 40664 [PSH, ACK] Seq=1155 Ack=2787 Win=64128
200 6.555031899 192.168.1.232 128.143.63.230 TCP 66 40664 → 22 [ACK] Seq=2787 Ack=2783 Win=64128 Len=
201 6.570288124 192.168.1.232 128.143.63.230 TCP 82 40664 → 22 [PSH, ACK] Seq=2787 Ack=2783 Win=64128
Frame 188: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface eno1, id 0
Ethernet II, Src: TpLinkTechno_d9:ae:50 (d8:07:b6:d9:ae:50), Dst: ASUSTekCOMPU_ab:8c:2c (c8:7f:54:ab:8c:2c)
Internet Protocol Version 4, Src: 128.143.63.230, Dst: 192.168.1.232
Transmission Control Protocol, Src Port: 22, Dst Port: 40664, Seq: 0, Ack: 1, Len: 0
Source Port: 22
Destination Port: 40664
[Stream index: 15]
[Conversation completeness: Complete, WITH_DATA (31)]
[TCP Segment Len: 0]
Sequence Number: 0 (relative sequence number)
Sequence Number (raw): 3937746435
[Next Sequence Number: 1 (relative sequence number)]
Acknowledgment Number: 1 (relative ack number)
Acknowledgment number (raw): 2073778370
1010 .... = Header Length: 40 bytes (10)
Flags: 0x012 (SYN, ACK)
Window: 65160
[Calculated window size: 65160]
Checksum: 0xfecd [unverified]
[Checksum Status: Unverified]
Urgent Pointer: 0
Options: (20 bytes), Maximum segment size, SACK permitted, Timestamps, No-Operation (NOP), Window scale
[Timestamps]

so many messages?

TCP waits full round-trip before sending data; can do better

example: QUIC (figure from RFC 9000, TCP+TLS replacement)

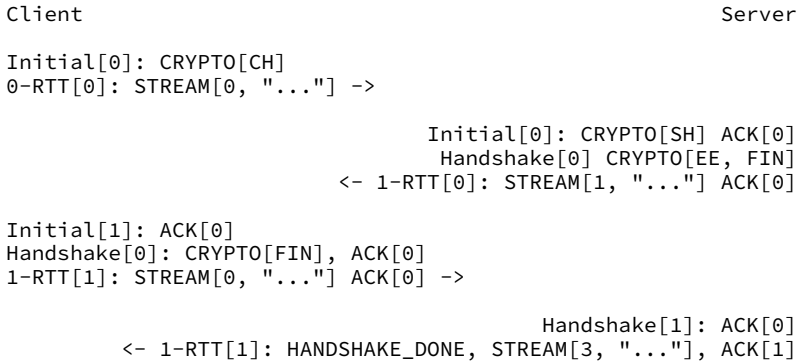
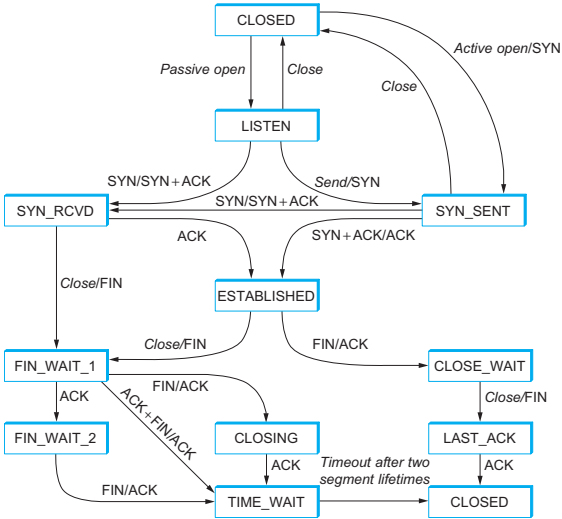


Figure 6: Example 0-RTT Handshake

TCP state machine



notation: event/action

not shown explicitly: timeout/resend

connection close

FIN (I want to close) + ACK (confirm close)

ACK side needs to wait just in case FIN resent

- keep port number from being reused
- TIME_WAIT state

RST

RST (reset) flag often used to respond to unexpected packets

example: data packet for never-established connection

connections on my desktop

```
cr4bd@reiss-t3620
: /zf14/cr4bd ; netstat --inet --inet6 --numeric
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 128.143.67.91:49202     128.143.63.34:22        ESTABLISHE
tcp        0      0 128.143.67.91:803       128.143.67.236:2049     ESTABLISHE
tcp        0      0 128.143.67.91:50292     128.143.67.226:22       TIME_WAIT
tcp        0      0 128.143.67.91:54722     128.143.67.236:2049     TIME_WAIT
tcp        0      0 128.143.67.91:52002     128.143.67.236:111      TIME_WAIT
tcp        0      0 128.143.67.91:732       128.143.67.236:63439    TIME_WAIT
tcp        0      0 128.143.67.91:40664     128.143.67.236:2049     TIME_WAIT
tcp        0      0 128.143.67.91:54098     128.143.67.236:111      TIME_WAIT
tcp        0      0 128.143.67.91:49302     128.143.67.236:63439    TIME_WAIT
tcp        0      0 128.143.67.91:50236     128.143.67.236:111      TIME_WAIT
tcp        0      0 128.143.67.91:22        172.27.98.20:49566      ESTABLISHE
tcp        0      0 128.143.67.91:51000     128.143.67.236:111      TIME_WAIT
tcp        0      0 127.0.0.1:50438         127.0.0.1:631           ESTABLISHE
tcp        0      0 127.0.0.1:631           127.0.0.1:50438         ESTABLISHE
```

socket options

'socket options' for extra settings

Python: `sock.setsockopt(level, key, value)`

e.g. `sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)`

C: `setsockopt(sock, level, key, value)`

e.g. `setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, 1)`

common level values

- `SOL_SOCKET` — OS-level
- `IPPROTO_IP`
- `IPPROTO_IPV6`
- `IPPROTO_TCP`

common SOL_SOCKET options

`SO_REUSEADDR` — allow bind()ing even if another socket for this port active

- common problem: server can't bind() because old session in TIME_WAIT

`SO_RCVBUF` — buffer size used by OS for receiving bytes

`SO_SNDBUF` — buffer size used by OS for sending bytes

`SO_LINGER` — make close() wait until messages acknowleged or timeout

(full list on Linux: `man 7 socket`)

efficiency problem

```
sock = ...
sock.send(b'About to send file with 16Kbytes')
sock.send(b'File is HTML; updated 16 November')
sock.send(file_data)
```

Naive approach:

- one packet for ‘about to send file with...’
- one packet for ‘file is HTML; ...’
- some number of packets for `file_data`

sending small TCP packets, so lots of overhead

Nagle's algorithm

(RFC 896)

delay sending data when smaller than full TCP segment

hope: by waiting we'll get a full TCP segment

typical wait: 200 ms?

TCP_NODELAY socket option can disable (at least on Linux)

broadcast/multicast

UDP IPv4: can broadcast to every machine on local network

- set the SO_BROADCAST socket option to 1
- send to special address 255.255.255.255
- received at that port number on all machines

UDP multicast: (send-to-many) groups

- IP_ADD_MEMBERSHIP (v4), IPV6_ADD_MEMBERSHIP (v6)
  - register to receive via setsockopt
- each 'multicast group' has IP address
- 224.0.0.0/24 and ff02::/16 = local network only
- local network: can implement by broadcasting + filtering
- (non-local-network multicast is more complex...)

service discovery

example: find printers on local network automatically

typical protocol mDNS (‘multicast DNS’)

uses IP addressses 224.0.0.251 / ff02::fb + port 5353

all machines on local network receive from all other machines on local network

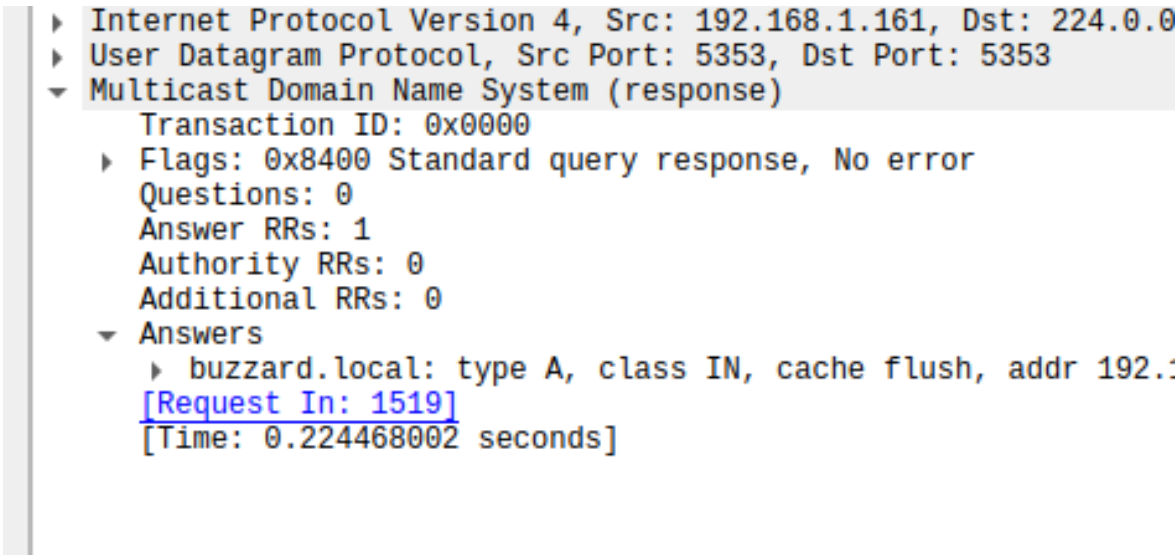
multicast DNS request

```
▸ Internet Protocol Version 4, Src: 192.168.1.161, Dst: 224.0.0.251
▸ User Datagram Protocol, Src Port: 5353, Dst Port: 5353
▾ Multicast Domain Name System (query)
    Transaction ID: 0x0000
  ▸ Flags: 0x0000 Standard query
    Questions: 6
    Answer RRs: 6
    Authority RRs: 0
    Additional RRs: 0
  ▾ Queries
    ▸ _services._dns-sd._udp.local: type PTR, class IN, "QM" question
    ▸ _http._tcp.local: type PTR, class IN, "QM" question
    ▸ _pdl-datastream._tcp.local: type PTR, class IN, "QM" question
    ▸ _printer._tcp.local: type PTR, class IN, "QM" question
    ▸ _scanner._tcp.local: type PTR, class IN, "QM" question
    ▸ _privet._tcp.local: type PTR, class IN, "QM" question
  ▾ Answers
    ▸ _services._dns-sd._udp.local: type PTR, class IN, _http._tcp.local
    ▸ _services._dns-sd._udp.local: type PTR, class IN, _ipp._tcp.local
    ▸ _services._dns-sd._udp.local: type PTR, class IN, _pdl-datastream._tcp.local
    ▸ _services._dns-sd._udp.local: type PTR, class IN, _printer._tcp.local
    ▸ _services._dns-sd._udp.local: type PTR, class IN, _scanner._tcp.local
    ▸ _services._dns-sd._udp.local: type PTR, class IN, _privet._tcp.local
```

multicast DNS response

```
▸ Internet Protocol Version 4, Src: 192.168.1.195, Dst: 224.0.0.251
▸ User Datagram Protocol, Src Port: 5353, Dst Port: 5353
▾ Multicast Domain Name System (response)
    Transaction ID: 0x0000
  ▸ Flags: 0x8400 Standard query response, No error
    Questions: 0
    Answer RRs: 5
    Authority RRs: 0
    Additional RRs: 5
  ▾ Answers
    ▸ _http._tcp.local: type PTR, class IN, Samsung M267x 287x Series (SEC8425192F4995)._http._tcp.local
    ▸ _pdl-datastream._tcp.local: type PTR, class IN, Samsung M267x 287x Series (SEC8425192F4995)._pdl-datastream._tcp.local
    ▸ _printer._tcp.local: type PTR, class IN, Samsung M267x 287x Series (SEC8425192F4995)._printer._tcp.local
    ▸ _scanner._tcp.local: type PTR, class IN, Samsung M267x 287x Series (SEC8425192F4995)._scanner._tcp.local
    ▸ _privet._tcp.local: type PTR, class IN, Samsung M267x 287x Series (SEC8425192F4995)._privet._tcp.local
  ▾ Additional records
    ▸ Samsung M267x 287x Series (SEC8425192F4995)._http._tcp.local: type TXT, class IN, cache flush
    ▸ Samsung M267x 287x Series (SEC8425192F4995)._pdl-datastream._tcp.local: type TXT, class IN, cache flush
    ▸ Samsung M267x 287x Series (SEC8425192F4995)._http._tcp.local: type NSEC, class IN, cache flush, next domain name Samsung M267x 287x Series (SEC8425192F4995)._http._tcp.local
    ▸ Samsung M267x 287x Series (SEC8425192F4995)._pdl-datastream._tcp.local: type NSEC, class IN, cache flush, next domain name Samsung M267x 287x Series (SEC8425192F4995)._pdl-datastream._tcp.local
    ▸ Samsung M267x 287x Series (SEC8425192F4995)._printer._tcp.local: type NSEC, class IN, cache flush, next domain name Samsung M267x 287x Series (SEC8425192F4995)._printer._tcp.local
    [Unsolicited: True]
```

.local domain



other address families

AF_UNIX (= AF_LOCAL in C, but not Python)

- sockets can be represented in files on disk
- support sending file descriptors usually

(not sure how portable?) AF_RAW

- usually how wireshark captures packets on Linux
- also supports writing packets

(Linux only) AF_NETLINK

- communicate with kernel networking code

AF_AX25, AF_APPLETALK, AF_DECNET, ...

- other interesting protocols

backup slides