LAYERING AND TCP SOCKETS

George Porter
Jan 14 and 16, 2019

ATTRIBUTION

- These slides are released under an Attribution-NonCommercial-ShareAlike 3.0 Unported (CC BY-NC-SA 3.0) Creative Commons license
- These slides incorporate material from:
  - Alex C. Snoeren, UC San Diego
  - Michael Freedman and Kyle Jamieson, Princeton University
  - Internet Society
ANNOUNCEMENTS

Practice problem set posted

Project 1 posted (Due Feb 4, 5pm)

Start early! The projects ARE NOT designed to be done in a single session
Outline

1. Layering
2. Client sockets
3. Server sockets
4. Socket options
5. Socket internals

WHAT ARE PROTOCOLS?

- Explicit and implicit conventions for how to communicate
  - Not for what is communicated
WHERE DO PROTOCOLS COME FROM?

- Standards bodies
  - IETF: Internet Engineering Task Force
  - ISO: International Standards Organization
- Community efforts
  - “Request for comments”
  - Bitcoin
- Corporations/industry
  - RealAudio™, Call of Duty multiplayer, Skype

HOW ARE PROTOCOLS SPECIFIED?

- Prose/BNF
  
- State transition diagrams
- Message Sequence Diagram
- Packet formats

By Stefan Birkner, cc-by-sa-2.5.2.0,1.0
TRANSMISSION CONTROL PROTOCOL (TCP)

- Remember 1 GB ~ 715,000 packets?
  - Don’t want to keep track of each packet, whether it got there, did it get lost? Did some get reordered??
- TCP offers *infinite bytestream* abstraction
  - If you put $N$ bytes into TCP connection as sender, those $N$ bytes will arrive to the destination in order, without loss, and without corruption
  - Compelling abstraction for higher-level applications such as web, video, gaming, ...

ROLE OF LAYERING IN PROTOCOLS

- Each layer offers useful semantics to layer above
  - IP gets packets to a destination host/server on the Internet (but is unreliable)
  - TCP uses IP to offer *reliable, in-order bytestream* abstraction
  - TCP useful for file transfer, as well as HTTP/web
TCP/IP PROTOCOL STACK

TRANSPORT PROTOCOLS

- Add services on top of IP
- User Datagram Protocol (UDP)
  - Data checksum
  - Best-effort
- Transmission Control Protocol (TCP)
  - Data checksum
  - Reliable byte-stream delivery
  - Flow control
    - Prevents receiver from being overloaded
  - Congestion control
    - Prevents the network from being overloaded
WHAT IS A SOCKET?

• What is a socket?
  • The point where a local application process attaches to the network
  • An interface between an application and the network
  • An application creates the socket
• The interface defines operations for
  • Creating a socket
  • Attaching a socket to the network
  • Sending and receiving messages through the socket
  • Closing the socket
PORTS

- IP addresses identify hosts
- Host has many applications
- Ports (16-bit identifier) identify a process

CLIENT AND SERVER OPERATIONS
SOCKET FAMILIES AND TYPES

• Socket Family
  • PF_INET denotes the Internet family
  • PF_UNIX denotes the Unix pipe facility
  • PF_PACKET denotes direct access to the network interface (i.e., it bypasses the TCP/IP protocol stack)

• Socket Type
  • SOCK_STREAM is used to denote a byte stream
  • SOCK_DGRAM is an alternative that denotes a message oriented service, such as that provided by UDP

CREATING A SOCKET

• int sockfd = socket(address_family, type, protocol);

• The socket number returned is the socket descriptor for the newly created socket

• int sockfd = socket (PF_INET, SOCK_STREAM, 0);
• int sockfd = socket (PF_INET, SOCK_DGRAM, 0);

• The combo of PF_INET and SOCK_STREAM implies TCP
struct sockaddr
{
    unsigned short sa_family;  /* Address family (e.g., AF_INET) */
    char sa_data[14];         /* Protocol-specific address information */
};

struct sockaddr_in
{
    unsigned short sin_family;  /* Internet protocol (AF_INET) */
    unsigned short sin_port;   /* Port (16-bits) */
    struct in_addr sin_addr;   /* Internet address (32-bits) */
    char sin_zero[8];          /* Not used */
};

struct in_addr
{
    unsigned long s_addr;  /* Internet address (32-bits) */
};

### HANDY HELPER FUNCTIONS FOR ADDRESSES

- Converting IP addr to/from strings
- Printable string to binary:
  - int inetpton();
- Binary to printable string:
  - const char * inet_ntop()
- Making the byte order consistent
  - uint16_t htons(uint16_t hostshort);
  - uint16_t ntohs(uint16_t netshort);
CONNECTING SOCKETS

- `int connect(int socket, const struct sockaddr *address, socklen_t address_len);`
- `socket` is the descriptor
- `address` describes the destination address
- `len` is the size of address
- Blocking call: waits until a connection is established
- Q: What kinds of errors might occur here?

SENDING DATA

- `ssize_t send(socket, buf, len, flags)`
  - We’re using blocking semantics of send
  - Always check that the right number of bytes were sent
  - Returns the number of bytes that were copied to the operating system kernel for transmission
RECEIVING RESPONSES

- ssize_t recv(int sockfd, void *buf, size_t len, int flags);
- Note:
  - recv() receives at least one bytes from the socket
  - It does not receive the same number of bytes that were sent via ‘send’
  - Returns 0 when the other side has closed the socket
    - Does not return 0 simply when the other side has nothing to send
- What does this mean?
  - You have to keep reading from the socket until you’ve received all the bytes you need

Outline

1. Layering
2. Client sockets
3. Server sockets
4. Socket options
5. Socket internals
SERVER OVERVIEW

Steps

1. Create network socket
2. Bind socket to an interface
3. Tell the socket to listen for incoming connections
4. Accept an incoming connection:
5. Read/write to the socket
6. Close the socket

Socket API used

1. socket()
2. bind()
3. listen()
4. accept()
5. send/recv()
6. close()

CLIENT AND SERVER OPERATIONS

Client

Server

open_clientfd

open_listeendif

connect

Connection request

accept

write

read

close

write

read

close

EOF
SERVER BIND

- Only server need to bind
  - \[ int \text{ bind}(int \text{ sockfd}, \text{ const struct sockaddr *my\_addr, socklen\_t addrlen}); \]

- sockfd
  - file descriptor socket() returned

- my\_addr
  - struct sockaddr\_in for IPv4
  - cast (struct sockaddr\_in*) to (struct sockaddr*)

```c
struct sockaddr\_in {  
  short sin\_family; // e.g. AF\_INET  
  unsigned short sin\_port; // e.g. htons(3490)  
  struct in\_addr sin\_addr; // see struct in\_addr, below  
  char sin\_zero[8]; // zero this if you want to  
};  

struct in\_addr {  
  unsigned long s\_addr; // load with inet\_atoni()  
};
```

SERVER LISTEN

- Now we can listen
  - \[ int \text{ listen}(int \text{ sockfd, int backlog}); \]

- sockfd
  - again, file descriptor socket() returned

- backlog
  - number of pending connections to queue

- For example,
  - listen(sockfd, 5);
SERVER ACCEPT

- Server must explicitly accept incoming connections
  - `int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen)`
- `sockfd`
  - again... file descriptor socket() returned
- `addr`
  - pointer to store client address, (struct sockaddr_in *) cast to (struct sockaddr *)
- `addrlen`
  - pointer to store the returned size of `addr`, should be `sizeof(*addr)`
- For example
  - `int isock=accept(sockfd, (struct sockaddr_in *) &caddr, &clen);`
SOCKET OPTIONS: MOTIVATION

• Basic “out of the box” socket functionality fine for most purposes
  • But what if you need to tweak the behavior?

• Can set/get ‘options’ on sockets

• These options apply to different layers of the network stack:
  • IP
  • TCP
  • Socket

TCP BUFFER SIZE OPTIONS

• Send and receive buffer sizes
  • What is the default?

    [gmporter@seed-f60-100 ~]$ cat /proc/sys/net/ipv4/tcp_rmem
    4096 87380 6291456

    [gmporter@seed-f60-100 ~]$ cat /proc/sys/net/ipv4/tcp_wmem
    4096 16384  4194304

  • Minimum  Default  Maximum

• Can we change that value?
  • Yes!
### SETTING/GETTING SOCKET OPTIONS

**GETSOCKOPT(2)**  
**Linux Programmer's Manual**  
**GETSOCKOPT(2)**

#### NAME

getsockopt, setsockopt - get and set options on sockets

#### SYNOPSIS

```c
#include <sys/types.h>  /* See NOTES */
#include <sys/socket.h>

int getsockopt(int sockfd, int level, int optname,  
    void *optval, socklen_t *optlen);
int setsockopt(int sockfd, int level, int optname,  
    const void *optval, socklen_t optlen);
```

<table>
<thead>
<tr>
<th>Level</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOL_SOCKET</td>
<td>SO_SNDBUF</td>
<td>Send buffer size</td>
</tr>
<tr>
<td></td>
<td>SO_REUSEADDR</td>
<td>Allow TCP port to be reused immediately</td>
</tr>
<tr>
<td></td>
<td>SO_SNDTIMEO</td>
<td>Set a send() timeout</td>
</tr>
<tr>
<td></td>
<td>SO_RCVTIMEO</td>
<td>Set a recv() timeout</td>
</tr>
</tbody>
</table>

### REUSING SOCKETS

```c
int optval = 1;
/* enable sockets to be immediately reused */
if (setsockopt(serv_sock, SOL_SOCKET,  
    SO_REUSEADDR, &optval, sizeof(optval)) != 0)  
{
    die_system("setsockopt() failed");
}  
```
PER-SOCKET TIMEOUTS

```c
struct timeval timeout;
timeout.tv_sec = 10;
timeout.tv_usec = 0;

if (setsockopt (sockfd, SOL_SOCKET, SO_RCVTIMEO, (char *)&timeout,
    sizeof(timeout)) < 0)
    error("setsockopt failed\n");

if (setsockopt (sockfd, SOL_SOCKET, SO_SNDTIMEO, (char *)&timeout,
    sizeof(timeout)) < 0)
    error("setsockopt failed\n");
```

Outline

1. Layering
2. Client sockets
3. Server sockets
4. Socket options
5. Socket internals
DIGGING INTO SEND() A BIT MORE

rv = connect(s, ...);
    ...
rv = send(s, buffer0, 1000, 0);
    ...
rv = send(s, buffer1, 2000, 0);
    ...
rv = send(s, buffer2, 5000, 0);
    ...
close(s);

AFTER 3 SEND() CALLS

[Diagram showing the process of sending and receiving data over a socket, with explained steps for each send call.]
AFTER FIRST RECV()

Sending sockets layer

SendQ

500 bytes

Receiving sockets layer

RecvQ

6000 bytes

Receiving program

Delivered

1 2 1

1 First send call (1000 bytes)
2 Second send call (2000 bytes)
3 Third send call (5000 bytes)

AFTER ANOTHER RECV()

Sending sockets layer

SendQ

500 bytes

Receiving sockets layer

RecvQ

2000 bytes

Receiving program

Delivered

3 2 1

1 First send call (1000 bytes)
2 Second send call (2000 bytes)
3 Third send call (5000 bytes)
WHEN DOES BLOCKING OCCUR?

- SendQ size: **SQS**
- RecvQ size: **RQS**
- `send(s, buffer, n, 0);`
  - \(n > SQS\): blocks until \((n - SQS)\) bytes xferred to RecvQ
  - If \(n > (SQS + RQS)\), blocks until receiver calls `recv()` enough to read in \(n - (SQS + RQS)\) bytes
- How does this lead to deadlock?
  - Trivial cause: both sides call `recv()` w/o sending data

MORE SUBTLE REASON FOR DEADLOCK

- SendQ size = 500; RecvQ size = 500