EECS 122: Introduction to Computer Networks Sockets Programming

Artur Rivilis
Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720-1776

Outline

- Socket API motivation, background
- Names, addresses, presentation
- API functions
- I/O multiplexing
- Tips/Troubleshooting
Motivation

- Applications need Application Programming Interface (API) to use the network

- API: set of function types, data structures and constants
  - Allows programmer to learn once, write anywhere
  - Greatly simplifies job of application programmer

Sockets (1)

- What exactly are sockets?
  - An endpoint of a connection
  - A socket is associated with each end-point (end-host) of a connection
  - Identified by address family and type.
- Can be further identified by IP address and port number
- Berkeley sockets (released with BSD 4.2 in 1983) is the de facto network API
  - Runs on Linux, *BSD, OS X, Windows
  - Led/fed off popularity of TCP/IP
Sockets (2)

- Similar to UNIX file I/O API (socket()) system call provides you with a file descriptor, which you can use like any other fd).
- Based on C, single threaded model
  - Does not require multiple threads
- Can build higher-level interfaces on top of sockets
  - e.g., Remote Procedure Call (RPC)

Types of Sockets (1)

- Different types of sockets implement different service models
  - We study two:
    - Stream: SOCK_STREAM
    - Datagram: SOCK_DGRAM
- Stream socket (TCP)
  - Connection-oriented (includes establishment + termination) – will study more in-depth soon.
  - No packet loss and in-order delivery guaranteed.
  - At-most-once delivery, no duplicates
  - E.g., SSH, HTTP, TELNET
- Datagram socket (UDP)
  - Connectionless (just data-transfer)
  - “Best-effort” delivery, possibly lower variance in delay
  - E.g., IP Telephony, streaming video & audio
Types of Sockets (2)

- How does application programming differ between stream and datagram sockets?
- Stream sockets
  - No need to packetize data
  - Data arrives in the form of a byte-stream
  - Receiver needs to separate messages in stream

TCP sends messages joined together, i.e. “Hi there! Hope you are well”

User application sends messages “Hi there!” and “Hope you are well” separately

Types of Sockets (3)

- Stream socket data separation:
  - Use records (data structures) to partition data stream
  - How do we implement variable length records?

  ┌───────┬───────┐
  │ A     │ B     │ C     │
  └───────┴───────┘

  size of record

  fixed length record
  fixed length record
  variable length record

  ┌───────┐
  │ 4     │
  └───────┘

- What if field containing record size gets corrupted?
  - Not possible! Why?
Types of Sockets (4)

- Datagram sockets
  - User packetizes data before sending
  - Maximum size of 64Kbytes
  - Further packetization at sender end and depacketization at receiver end handled by transport layer
  - Using previous example, “Hi there!” and “Hope you are well” will definitely be sent in separate packets at network layer

Naming and Addressing

- IP version 4 address
  - Identifies a single host
  - 32 bits
  - Written as dotted octets
    - e.g., 0x0a000001 is 10.0.0.1

- Host name
  - Identifies a single host
  - Variable length string
  - Maps to one or more IP address
    - e.g., www.berkeley.edu
    - gethostbyname() translates name to IP address

- Port number
  - Identifies an application on a host
  - 16 bit unsigned number

IPv6
- Developed at Xerox PARC
- 128 bit address (3.4 x 10^38 vs. 4.294 x 10^9)
Numerical Presentation

- We talk about two numeric presentations:
  - Big Endian
    - Architectures: Sun SPARC, Motorola 68000, PowerPC 970, IBM System/360
    - The most significant byte is stored in memory at the lowest address.
    - Example: \(4A3B2C1D\) hexadecimal will be stored as: 4A 3B 2C 1D
    - This is network-byte order.
  - Little Endian
    - Architectures: Intel x86, AMD64, DEC VAX
    - The least significant byte is stored in memory at the lowest address.
    - Example: \(4A3B2C1D\) hexadecimal will be stored as: 1D 2C 3B 4A

- What problems can arise because of this?
- What can we do to solve them?

Byte Ordering Solution

- The networking API provides us the following functions:

  ```c
  uint16_t htons(uint16_t host16bitvalue);
  uint32_t htonl(uint32_t host32bitvalue);
  uint16_t ntohs(uint16_t net16bitvalue);
  uint32_t ntohl(uint32_t net32bitvalue);
  ```

- Use for all numbers (int, short) to be sent across network
  - Including port numbers, but not IP addresses
Quiz!

- What is wrong with the following code?

```c
int factorial (int a) {
    if (a == 0 || a == 1)
        return 1;
    if (a < 0)
        return -1;
    return a * factorial (a-1);
}
```

The connection process

- Implements Transmission Control Protocol (TCP)
- Does NOT set up virtual-circuit!
- Sequence of actions:

1. `socket()`
2. `bind(80)`
3. `listen()`
4. `accept()`
5. `send()`
6. `recv()`
7. `close()`
8. `[connection established]`
Initialize (Client + Server)

```c
int sock;
if ((sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
    perror("socket");
    printf("Failed to create socket\n");
    abort();
}
```

- Handling errors that occur rarely usually consumes most of systems code
  - Exceptions (e.g., in Java – Why?) helps this somewhat.

---

Initialize (Server reuse addr)

```c
int optval = 1;
if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    perror("opening TCP socket");
    abort();
}
if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval)) <0) {
    perror("reuse address");
    abort();
}
```
**Initialize (Server bind addr)**

- **Want port at server end to use a particular number**

```c
struct sockaddr_in sin;
memset (&sin, 0, sizeof (sin));
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR;
sin.sin_port = htons (server_port);
if (bind(sock, (struct sockaddr *) &sin, sizeof (sin)) < 0) {
    perror("bind");
    printf("Cannot bind socket to address\n");
    abort();
}
```

**Why do we do this?**

---

**Initialize (Server listen)**

- **Wait for incoming connection**

  - **Parameter BACKLOG specifies max number of established connections waiting to be accepted (using accept())** – What might this help you avoid?

```c
if (listen (sock, BACKLOG) < 0) {
    perror ("error listening");
    abort ();
}
```
Establish (Client)

struct sockaddr_in sin;

struct hostent *host = gethostbyname (argv[1]);
unsigned int server_addr = *(unsigned long *) host-
    >h_addr_list[0];
unsigned short server_port = atoi (argv[2]);
memset (&sin, 0, sizeof (sin));

sin.sin_family = AF_INET;
sin.sin_addr.s_addr = server_addr;
sin.sin_port = htons (server_port);

if (connect(sock, (struct sockaddr *) &sin, sizeof (sin)) <
    0) {
    perror("connect");
    printf("Cannot connect to server\n");
    abort();
}

Establish (Server)

- Accept incoming connection

int addr_len = sizeof (addr);
int sock;

sock = accept (tcp_sock, (struct sockaddr *)
    &addr, &addr_len);

if (sock < 0) {
    perror ("error accepting connection");
    abort ();
}
Sending Data Stream

- Now that the connection is established, we want to send data:

```c
int send_packets (char *buffer, int buffer_len)
{
    sent_bytes = send (sock, buffer, buffer_len, 0);  // 1
    if (send_bytes < 0)
        perror ("send");
    return 0;
}
```

Receiving Data Stream

```c
int receive_packets(char *buffer, int buffer_len, int *bytes_read)
{
    int left = buffer_len - *bytes_read;
    received = recv(sock, buffer + *bytes_read, left, 0);  // 2
    if (received < 0) {
        perror ("Read in read_client");
        printf("recv in %s\n", __FUNCTION__);  // 3
    }
    if (received == 0) {
        return close_connection();
    }
    *bytes_read += received;
    while (*bytes_read > RECORD_LEN) {
        process_packet(buffer, RECORD_LEN);
        *bytes_read -= RECORD_LEN;
        memmove(buffer, buffer + RECORD_LEN, *bytes_read);  // 4
    }
    return 0;
}
```

Occurs when other side closes connection.
Comic fun

Kernel panic in the linked library with the named pipe!

Another exciting game of Clustenix

Any questions so far?

Quiz!

- We have the following packet header format (the numbers denote field size in bytes):

<table>
<thead>
<tr>
<th>length</th>
<th>type</th>
<th>source addr</th>
<th>dest addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

- What is wrong with the code below?

```c
typedef struct _pkt hdr_ {
    unsigned short length;
    char type;
    unsigned int src_addr;
    unsigned int dest_addr;
} pkt_hdr, *pkt_hdrptr;

char buffer[256]; pkt_hdr header;
/* assume fields filled in properly here */
memcpy(buffer, &header, sizeof (header));
send (sock, buffer, sizeof (header), 0);
```

Words aligned in memory: begins at address that is a multiple of bytes in word.

<table>
<thead>
<tr>
<th>length</th>
<th>type</th>
<th>source addr</th>
<th>dest addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>
Datagram Sockets

- Similar to stream sockets, except:
  - Sockets created using SOCK_DGRAM instead of SOCK_STREAM
  - No need for connection establishment and termination
  - Uses `recvfrom()` and `sendto()` in place of `recv()` and `send()` respectively
  - Data sent in packets, not byte-stream oriented

---

How to handle multiple connections?

- Where do we get incoming data?
  - Stdin (typically keyboard input)
  - All stream, datagram sockets
  - Asynchronous arrival, program doesn’t know when data will arrive

- Solution: I/O multiplexing using `select()`
  - Efficient for our purposes (preferred method).

- Solution: I/O multiplexing using polling
  - Very inefficient, never do this.

- Solution: Multithreading (POSIX et al)
  - More complex, requires mutex, semaphores, etc.
  - Commonly used for massive scalable networked applications.
  - Not covered, but feel free to try.
I/O Multiplexing: Polling

```c
int opts = fcntl (sock, F_GETFL);
if (opts < 0) {
    perror ("fcnt1(F_GETFL)"");
    abort ();
}
opts = (opts | O_NONBLOCK);
if (fcntl (sock, F_SETFL, opts) < 0) {
    perror ("fcntl(F_SETFL)"");
    abort ();
}
while (1) {
    if (receive_packets(buffer, buffer_len, &bytes_read) != 0) {
        break;
    }
    if (read_user(user_buffer, user_buffer_len, &user_bytes_read) != 0) {
        break;
    }
}
```

I/O Multiplexing: Select (1)

- **Select()**
  - Wait on multiple file descriptors/sockets and timeout
  - Application does not consume CPU cycles while waiting
  - Return when file descriptors/sockets are ready to be read or written or they have an error, or timeout exceeded

- **Advantages**
  - Simple
  - More efficient than polling(why?)

- **Disadvantages**
  - Does not scale to large number of file descriptors/sockets
  - More awkward to use than it needs to be
I/O Multiplexing: Select (2)

```c
fd_set read_set;
struct timeval time_out;
while (1) {
    FD_ZERO (read_set);
    set up parameters for select()
    FD_SET (stdin, read_set); /* stdin is typically 0 */
    FD_SET (sock, read_set);
    time_out.tv_usec = 100000; time_out.tv_sec = 0;
    run select()
    select_retval = select(MAX(stdin, sock) + 1, &read_set, NULL,
    interpret result
    NULL, &time_out);
    if (select_retval < 0) {
        perror ("select");
        abort ();
    }
    if (select_retval > 0) {
        if (FD_ISSET(sock, read_set)) {
            if (receive_packets(buffer, buffer_len, &bytes_read) != 0) {
                break;
            }
        }
        if (FD_ISSET(stdin, read_set)) {
            if (read_user(user_buffer, user_buffer_len,
                &user_bytes_read) != 0) {
                break;
            }
        }
    }
}
```

I/O Multiplexing: Select (3)

- Explanation:
  - FD_ZERO(fd_set *set) — clears a file descriptor set
  - FD_SET(int fd, fd_set *set) — adds fd to the set
  - FD_CLR(int fd, fd_set *set) — removes fd from the set
  - FD_ISSET(int fd, fd_set *set) -- tests to see if fd is in the set

- What is the fd_set? Just an array of file descriptors:
  ![Array of File Descriptors](image)

- When does the call return?
  - An error occurs on an fd.
  - There is data becomes available on an fd.
  - Space available in kernel's internal buffer for more to be written
to fd (not something you have to worry about).

- What do I check?
  - You use FD_ISSET to see if a particular fd is set, and if it is then
  you need to handle it in some way.
  - All none active fds are cleared (so you would need to reset the
    fd_set if you want to listen to an certain fd again).
  - More than one fd set may be set after select returns.
Common Mistakes + Hints

- **Common mistakes:**
  - C programming
    - Use GDB (Quick Guide: http://frmb.org/quickgdb.html)
    - Use printf for debugging, remember to do `fflush(stdout)` or `\n` at end of line.
  - Byte-ordering!
  - Use of `select()`
  - Separating records in TCP stream
  - Not knowing what exactly gets transmitted on the wire
    - Use tcpdump / Ethereal
- **Hints:**
  - Use man pages (inst or web.)
  - Check out WWW (ex. Beej's Guide), programming books
  - Look at sample code.
  - `setsockopt(int s, int level, int optname, const void *optval, socklen_t optlen);`

When things go wrong

- A few things may go wrong that you need to look out for:
  - A general error. These should always be checked and cleaned up for every sys call that is made.
  - Broken pipe: can happen any time pipe is broken, will get a SIGPIPE crash. Ignore SIGPIPE to solve.
  - Connection cleanly closed: typically you recv 0 when this happens. If you send twice you will also detect a closed socket by checking the error.
Socket API Reference

- `struct hostent *gethostbyname(const char *name);`
  
  ```c
  struct hostent {
    char *h_name; // Official name of the host.
    char **h_aliases; // A terminated array of alternate names for the host.
    int h_addrtype; // The type of address being returned, usually AF_INET.
    int h_length; // The length of the address in bytes.
    char **h_addr_list; // A zero-terminated array of network addresses for
                        // the host. Host addresses are in Network Byte Order.
  };
  ```
  
  `int gethostname(char *hostname, size_t size);`
  Returns the name of the computer that your program is running on.

- `int socket(int domain, int type, int protocol);`
  
- `int bind(int sockfd, struct sockaddr *my_addr, int addrlen);`
  
- `int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);`
  
- `int accept(int sockfd, void *addr, int *addrlen);`
  
- `int send(int sockfd, const void *msg, int len, int flags);`
  
- `int recv(int sockfd, void *buf, int len, unsigned int flags);`
  
- `setssockopt(int s, int level, int optname, const void *optval, socklen_t optlen);`

C Programming Reference

- `void * memcpy ( void * dest, const void * src, size_t num );`
  Copies num bytes from src buffer to memory location pointed by dest.

- `void * memset ( void * buffer, int c, size_t num );`
  Sets the first num bytes pointed by buffer to the value specified by c parameter.

- `void * memmove(void *s1, const void *s2, size_t n);`
  Copies n bytes from memory areas s2 to s1. It returns s1. If s1 and s2 overlap, all bytes are copied in a preserving manner.

- `void perror(const char *s );`
  Prints a system error message.

- `int select(int nfds, fd_set *readfds, fd_set *writfds, fd_set *exceptfds, struct timeval *timeout);`