Socket Tutorial

Prof. Chris GauthierDickey

COMP 3621: Computer Networking

University of Denver

Usage:

- You are free to use these slides, just give credit where credit is due (=

Prof. Chris GauthierDickey, University of Denver, 2010

What are sockets?

- When we broadly talk about sockets, we usually are referring to networking sockets
  - In particular, they refer to Berkeley sockets which were created some time ago (and operate slightly differently than POSIX sockets!)
  - Typically, we now use the POSIX sockets API
- Sockets are also just that: an API for network communication

Prof. Chris GauthierDickey, University of Denver, 2010

Sockets introduced...

- Conceptually, sockets are like a door [K&R ’09]
  - But more like a magic portal (think Stargate™):
    - We create one side and tell the OS where the other side opens--on another system!
    - The OSes handle the rest: communicating via TCP or UDP to the other system and establishing a connection (in the case of TCP)

Prof. Chris GauthierDickey, University of Denver, 2010

Sockets introduced...

- Like any door, we have to open and close it to send things through it
- We also have to know where this side opens (i.e., what network address so that the door can be two-way)
- We have to know where the other side opens (i.e., what the other address is)
- We also have to know how stable we want transport through door to be (i.e., TCP, UDP, or something else)

Prof. Chris GauthierDickey, University of Denver, 2010

Main Socket Functions [client side]

- As usual, these are in C:
  - getaddrinfo(), freeaddrinfo(): gets and frees address structures
  - socket(): creates a socket for communication
  - connect(): for connection-oriented sockets (TCP)
  - recv() & send() for TCP
  - recvfrom() & sendto() for UDP

Prof. Chris GauthierDickey, University of Denver, 2010

Client-side communication

- First we have to determine which type of transport protocol we want
  - UDP provides best effort delivery, with no guarantees
  - TCP provides in-order, reliable, congestion-controlled transport. TCP is ‘stream’ based.

Prof. Chris GauthierDickey, University of Denver, 2010

Creating a socket

- All network communication requires the creation of a socket
- The table lists the required parameters for both TCP and UDP sockets

```
#include <sys/socket.h>

// we have to specify the family, the type
// and the protocol when creating a socket
int socket(int family, int type, int protocol);
```

Parameters for creating a socket

	UDP	TCP
family	AF_INET, AF_INET6	AF_INET, AF_INET6
type	SOCK_DGRAM	SOCK_STREAM
protocol	IPPROTO_UDP	IPPROTO_TCP

Prof. Chris GauthierDickey, University of Denver, 2010

What does socket(...) return?

- A call to socket(...) returns what we call a ‘socket file descriptor’, or -1 if there was an error (check errno)
  - Think of it as a handle to some internal data structure the OS is using to provide networking with
  - In Unix, socket file descriptors and regular file descriptors behave similarly
  - The socket file descriptor will be used for every networking call so that the OS knows which network socket to use

Prof. Chris GauthierDickey, University of Denver, 2010

Resolving host names

- Note that to create a socket, we have to resolve the host name
- Older networking code uses gethostbyname() or gethostbyaddr(), but these are only good for IPv4
- We will instead use getaddrinfo() and freeaddrinfo()

getaddrinfo()

- The struct addrinfo is used for two purposes:
  - to pass in hints so that the right type of address will be created
  - to get the results of that call

```
// definition of getaddrinfo()
#include <netdb.h>
int getaddrinfo(const char *hostname,
        const char *service,
        const struct addrinfo *hints,
        struct addrinfo **result);
```

- hostname is either an actual host's name (e.g., www.cs.du.edu) or an ip address (130.253.8.41) as a string (i.e., with quotes around it)
- service is either the string service name (from /etc/services) or a decimal port name (again, in string format)
- hints should be filled with:
  - ai_flags: 0 or AI_CANONNAME
  - ai_family: AF_INET for IPv4 or AF_INET6 for IPv6
  - ai_socktype: SOCK_DGRAM for UDP or SOCK_STREAM for TCP
  - ai_protocol: IPPROTO_UDP or IPPROTO_TCP for UDP and TCP respectively
- result is a struct addrinfo pointer to a pointer that gets filled by the call of the function

Prof. Chris GauthierDickey, University of Denver, 2010

freeaddrinfo() and gai_strerror()

- The result you get from getaddrinfo *must* be freed using freeaddrinfo(), or you'll have a memory leak
  - Note that getaddrinfo returns a LIST of addresses (sometimes you have more than one result), which you can iterate over via its ai_next element. You just call freeaddrinfo() on the FIRST result from getaddrinfo()
- gai_strerror returns the string representation of the error returned by getaddrinfo

Prof. Chris GauthierDickey, University of Denver, 2010

getaddrinfo() example

```
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <strings.h>
struct addrinfo *getUDPAddressByName(const char *name, const char *port) {
    struct addrinfo hints;
    bzero(&hints, sizeof(hints));
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_DGRAM;
    hints.ai_protocol = IPPROTO_UDP;

    struct addrinfo *result;

    int result = getaddrinfo(name, port, &hints, &result);

    if (result != 0) {
        fprintf(STDERR, “Error resolving host %s and port %s: %s\n”, name, port, gai_strerror(result));
        return NULL;
    }

    return result;
}
```

Prof. Chris GauthierDickey, University of Denver, 2010

UDP client communication

- Assume for now that the server exists and may be waiting for packets
  - UDP is connectionless--we do not have to establish a connection to the server prior to sending data to it
  - UDP is bufferless: we may have a buffer we write to, and which is passed to the OS, but the OS only (usually) temporarily buffers the data to write it to the socket and then throws the buffer away

Prof. Chris GauthierDickey, University of Denver, 2010

UDP: sendto

- sendto(...) is used to send a UDP packet to the specified destination (the ‘to’ field).
  - **buff** is your buffer of data, nbytes is its length,
  - **flags** are any sendto flags you need (see the man pages),
  - **to** (the destination address) and **addrlen** are taken from the struct addrinfo returned by getaddrinfo()

```
#include <sys/socket.h>

// returns the number of bytes sent or
// -1 on error (must check errno)
ssize_t sendto(int sockfd,
    const void *buff,
    size_t nbytes,
    int flags,
    struct sockaddr *to,
    socklen_t *addrlen);
```

Prof. Chris GauthierDickey, University of Denver, 2010

UDP: recvfrom

- We call recvfrom when we want to retrieve a message from the UDP socket
  - **sockfd** is the valid socket from socket()
  - **buff** is a buffer to retreive the message into
  - **nbytes** is the size of buff in bytes
  - **flags** are flags for the socket for reading (typically pass in 0)
  - **from** is *filled in* by the call to recvfrom with the sender's address
  - **addrlen** is *filled in* by the call to recvfrom with the size of the sender's address
- You can legally pass NULL into **from** and **addrlen**, and it won't return the address info to you (obviously)

```
#include <sys/socket.h>

// returns the number of bytes sent or
// -1 on error (must check errno)
ssize_t recvfrom(int sockfd,
    void *buff,
    size_t nbytes,
    int flags,
    struct sockaddr *from,
    socklen_t *addrlen);
```

Prof. Chris GauthierDickey, University of Denver, 2010

UDP Server: bind()

- To create a UDP server, you have to bind the socket to an address and port number. Clients then send messages (via sendto) to that address.
  - **socket**: a valid socket created via socket()
  - **address**: the address we're binding to, which includes the port number--via getaddrinfo()
  - **address_len**: the length of the address struct, also via getaddrinfo()

```
#include <sys/socket.h>

// returns 0 if successful, -1 of there was
// an error (must check errno)
int bind(int socket,
        const struct sockaddr *address,
        socklen_t address_len);
```

Prof. Chris GauthierDickey, University of Denver, 2010

Creating a UDP client

- Create a socket, using an address from getaddrinfo() and a call to socket()
- Call sendto() with the address and port of the server
- Call recvfrom() to wait for a response from the server
  - Validate that the address was the server's address
- Call close() on the socket

Prof. Chris GauthierDickey, University of Denver, 2010

Creating a UDP Server

- Create a socket with an address and port using getaddrinfo() for the address, and socket() for the port
  - Set the AI_PASSIVE flag on the ai_flags field of the hints parameter to getaddrinfo. You can also set the hostname parameter to NULL so that you bind to any of the server's addresses.
- bind() the socket and address
- Wait for messages using recvfrom (keep the from address around for later)
- Respond to messages with sendto (and use the address in the from field when you received the message)
- close() when you're done!

Prof. Chris GauthierDickey, University of Denver, 2010

Preparing data to send

- From a C perspective, the buffers you use for sendto() and recvfrom() are simply chunks of memory without meaning.
- The buffers are copied byte-by-byte to the OS buffer to be sent on
- This is fine most of the time, except when you're trying to send integer data.

Prof. Chris GauthierDickey, University of Denver, 2010

Integer conversions

- In order to make networking compatible with different architectures, a set of functions was created in the socket API to convert from what we call *network byte-order* to *host byte-order.*
  - htonl() and htons() convert integers and shorts (16-bit values) from host byte-ordering to network byte-ordering
  - ntohl() and ntohs() convert integers and shorts (16-bit values) from network byte-ordering to host-byte ordering.
    - Network byte-ordering is big-endian, while some architectures are little-endian
    - You MUST convert back and forth for each integer you want to read over the internet.
      - Sometimes you'll forget and you'll get lucky some of the time, but eventually you'll cause a bug that enables an internet worm or trojan horse to propagate!

Prof. Chris GauthierDickey, University of Denver, 2010

Non-blocking I/O

- As you write networking code, the first thing you'll notice is that sendto() and recvfrom() block!
  - The OS will not return from your function until the operation is complete--which is a problem for recvfrom() when you're not sure you're going to get data
  - Imagine having to write code that periodically sends a packet, but in the meantime you're waiting for data:
    - In this case, if you call recvfrom, your call will not return to let you send the periodic data until it receives data
    - Instead we will use select

Prof. Chris GauthierDickey, University of Denver, 2010

Why select()?

- Well it's a common way to wait for sockets, and it's fairly standard. It's not the fastest method. Neither is poll(). You'll get somewhat OS specific to use the faster methods however!
- To use select(), we basically create a data structure called a file-descriptor set that we give to the OS. The OS then marks which descriptor is ready for reading, writing or has an exception. We check the set and then attempt to read or write to the socket.

Prof. Chris GauthierDickey, University of Denver, 2010

select() continued...

- Select also allows you to specify a timeout value down to the microsecond. This in theory also allows you to create a fairly high resolution timer (think about how you could do that).
- The main problem with select() is that even **after** the OS tells you a socket is ready, it could still block! So you further have to ensure that your socket is marked as non-blocking
  - The main side-effect of creating your socket as a non-blocking socket is that calls to read or write can return -1 with errno set to EAGAIN or EWOULDBLOCK (depending on the OS). This just means that the socket isn't actually ready.

Prof. Chris GauthierDickey, University of Denver, 2010

Creating a non-blocking socket

- First, we create the socket as we normally would.
- Then, we use fcntl() to set the O_NONBLOCK option on the socket.
- You can use fcntl() to check the options on the socket also.
- It's worth reading the fcntl() man pages to see what interesting things you can do with the socket!

```
#include <unistd.h>
#include <fcntl.h>

int res = fcntl(sockfd, F_SETFL,
O_NONBLOCK);
```

Prof. Chris GauthierDickey, University of Denver, 2010

Creating the FD_SET

- The fd_set is a data structure that you manipulate with macros:
  - FD_SET(int sockfd, fd_set *set)
  - FD_CLR(int sockfd, fd_set *set)
  - FD_ISSET(int sockfd, fd_set *set)
  - FD_ZERO(fd_set *set)

```
#include <sys/select.h>

// assume we have a socket, called sockfd

// declare the fd_sets
fd_set masterset;
fd_set readset;
fd_set writeset;
fd_set exceptset;

// zero them out
FD_ZERO(&masterset)
FD_ZERO(&readset);
FD_ZERO(&writeset);
FD_ZERO(&exceptset);

// now, for each socket you want to
// use select on, you'll FD_SET it
FD_SET(sockfd, &masterset)
FD_SET(sockfd, &readset)
FD_SET(sockfd, &writeset)
FD_SET(sockfd, &exceptset)
```

Prof. Chris GauthierDickey, University of Denver, 2010

Using select()

- select() requires a timeout value, which if we use NULL, will wait until at least one socket is ready for action
- Then we loop through all our sockets and check to see if they were set by select()
- Finally, we reset the fd_sets by copying the master set over them using a C struct copy

```
#include <sys/select.h>

struct timeval timeout;
timeout.tv_sec = 2;
timeout.tv_usec = 0;

// note that maxfd must be the highest integer
// value of the sockets you have created (socket
// numbers typically increase sequentially)
int res = select(maxfd + 1, &readset, &writeset,
                 &exceptset, &timeout);

// In this case, select will block for 2 seconds
// before returning. Setting timeout to 0 will
// cause it to return immediately

// you will check each socket using FD_ISSET
// before reading and writing to the socket
if (FD_ISSET(sockfd, &readset)) {
  // the socket has data and can be read from
}

if (FD_ISSET(sockfd, &writeset)) {
  // the socket is ready to be written to
}

// copy the masterset over the readset,
// writeset, and exceptset before you call
// select() again (because they're modified
// by select)
readset = masterset;
```

Prof. Chris GauthierDickey, University of Denver, 2010

References

- K&R ’09: James Kurose and Keith Ross, *Computer Networking: A Top-Down Approach,* 5th edition, Addison Wesley, 2009, ISBN 0-13-607967-9