CSE 124
January 7, 2016

Winter 2016, UCSD

Prof. George Porter

Outline

1. UNIX network sockets overview
   – `socket(), connect(), send(), recv(), close(), bind(), listen(), accept()`
   – Data structures needed for network sockets
   – Helper functions
2. We're going to build a simple 'echo' client
3. Then the server application
4. Then let's modify it!

Demo 1: Echo client/server

Part 1: The client

Client overview

Steps

1. Handle command line arguments
2. Create network socket
3. Connect to the server
4. Send the string
5. Receive the response
6. Close the socket

Socket API used

1. `n/a`
2. `socket()`
3. `connect()`
   - `struct sockaddr_in`
4. `send()`
5. `recv()`
6. `close()`

Step 1: Command line

- The client needs 3 parameters:
  - Address to connect to
  - String to send to the echo server
  - TCP port number of the echo server
- Let’s build argument handling first
- Note this part of the code doesn’t involve any network sockets APIs, except ‘in_port_t’:
  - typedef __uint16_t in_port_t;

Step 2: Creating the socket

```
#include <sys/socket.h>
int socket(int family, int type, int protocol);
```

	Family	Type	Protocol
TCP	PF_INET	SOCK_STREAM	IPPROTO_TCP
UDP		SOCK_DGRAM	IPPROTO_UDP

Creates a new socket;
PF_INET,SOCK_STREAM,IPPROTO_TCP defines a TCP connection

Step 3: Connecting to the server

1. Convert the destination address and destination port into a
   – struct sockaddr_in;

2. Connect to the server

Generic

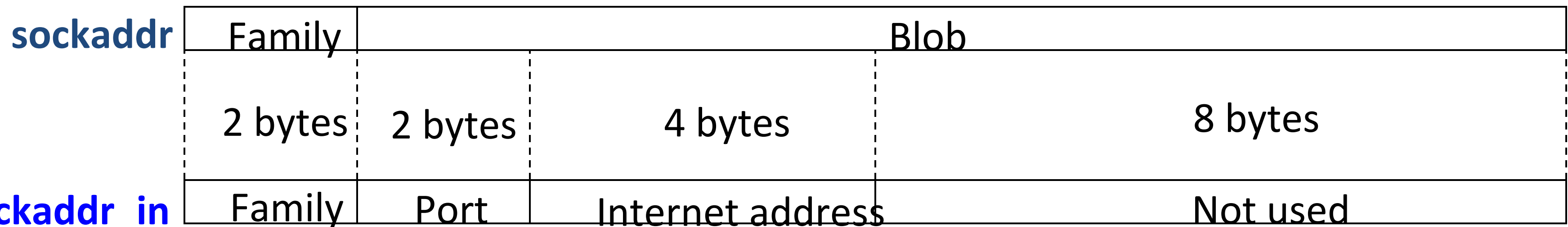
- struct sockaddr

```
struct sockaddr
{
  unsigned short sa_family;   /* Address family (e.g., AF_INET) */
  char sa_data[14];           /* Protocol-specific address information */
};
```

IP Specific

- struct sockaddr_in

```
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) */
};
```



Some details about addresses…

- Converting IP addrs to/from strings
  - Printable string to binary:
    - `int inet_pton();`
  - 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?

Step 4: sending the string

- `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 sent and acknowledged by the receiver

Step 5: Receive the response

- **`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' (we'll see why in Chapter 7)
  - Returns 0 when the client has closed the socket
- What does this mean?
  - You have to keep reading from the socket until you've received all the bytes you need

Step 6: close the socket

- **`int close(int`** `socket`**`);`**
- Closes the socket

Client summary

Steps

1. Handle command line arguments
2. Create network socket
3. Connect to the server
4. Send the string
5. Receive the response
6. Close the socket

Socket API used

1. `n/a`
2. `socket()`
3. `connect()`
   - `struct sockaddr_in`
4. `send()`
5. `recv()`
6. `close()`

Server overview

Steps

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

Socket API used

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

Step 1: Command line

- The server needs 1 parameter:
  - Port to listen on
- Similar to the client
- No real network code in this part

Step 2: Creating the socket

```
#include <sys/socket.h>
int socket(int family, int type, int protocol);
```

	Family	Type	Protocol
TCP	PF_INET	SOCK_STREAM	IPPROTO_TCP
UDP		SOCK_DGRAM	IPPROTO_UDP

Same as in the client

Step 3: bind

- We need to tell the socket what IP address and port to listen for incoming connections
- **int bind(int** sockfd**, const struct sockaddr ***addr**, socklen_t** len**);**
- We don't care which IP address to bind to
  - servAddr.sin_addr.s_addr = htonl(INADDR_ANY);
- We do care about the listening port
  - servAddr.sin_port = htons(servPort);

Step 4: listen

- Tells the OS that this socket is going to be a server socket
  - E.g., that it should listen for incoming commands
- **`int listen(int sockfd, int backlog);`**
  - 'backlog' specifies how many pending connections can exist before additional ones are refused

Step 5: Accepting new connections

- A “listening” socket isn’t actually used to send/ receive data…
  - Otherwise e.g., a web server could only handle one client at a time
  - Because port 80 (www) would be “used up” by a single client

- Instead, listening socket used to accept a new connection, which gets its very own *per-connection* ‘client’ socket

Step 5: Accepting new connections

- **int accept(int** sockfd**, struct sockaddr ***addr**, socklen_t ***addrlen**);**
- Blocking call, waits for incoming connections
- Returns new, per-connection client socket ID
- addr is filled in with information about the incoming client
  - E.g., their IP address and port

Step 6: send/recv

- Same as with the client
- Reminder:
  - recv() returns at least one byte
  - Can return a different amount of bytes than the corresponding send call
- Example:
  - send(1000 bytes)
  - recv() returns 400 bytes, next recv() returns 100 bytes, and a third recv() call returns 500 bytes

Step 7: closing the socket

- Same as with the client
- But remember that you need to close the ‘client’ socket as well as the ‘server’ socket

IPv6

- Chapter 2.9 covers support for IPv6
- Mostly using slight variations of the data structures
- Our testbed doesn't support v6, so we won't be really using it this term