Computer Networks
ICS 651

Unix Sockets API

Unix Sockets API – principles

- a socket is an endpoint of communication -- we need (at least) two sockets to communicate, just as we need at least two telephones
- a socket pair is uniquely identified by protocol, two IP address, and two port numbers
- when we first create a socket, it has only the first of these attributes, the protocol number

Unix Sockets API – principles

- a socket pair is uniquely identified by:

1) protocol: TCP or UDP (8-bit numbers, e.g. 6 for TCP, 17/0x11 for UDP)

2) two IP addresses (32-bit numbers, for example 128.171.10.123), one for each socket

3) for IPv6, this becomes two 128-bit numbers

4) two port numbers (16-bit numbers, for example 1234), one for each socket

Unix Sockets API
Managing Sockets

Unix Sockets API -- Managing Sockets

```
int socket(int domain,
           int type,
           int protocol);
```

creates a socket (the return value is a file descriptor)

```
int close(int sockfd);
```

closes a socket (or any other file descriptor)

Unix Sockets API
Managing Sockets

`close` completely eliminates a socket.

Sometimes it's useful to tell the system we are done writing on a socket:

```
int shutdown(int sockfd, int how);
```

shuts down reading from a socket (`how == SHUT_RD`), writing to a socket (`how == SHUT_WR`), or both

Most programs use close rather than shutdown, but there are times when you need to indicate you will never again write to a socket (`how == SHUT_WR`).

Unix Sockets API connecting as a client

```
int connect
  (int sockfd,
   struct sockaddr *serv_addr,
   socklen_t addrlen);
```

requests a connection. The address is found in the first addrlen bytes of memory pointed to by serv_addr.

See ip(7) and ipv6(7) for details.

Unix Sockets API
accepting connections as a server

```
int bind(int sockfd,
         struct sockaddr *my_addr,
         socklen_t addrlen);
```

binds the given socket to the given address, found in the first addrlen bytes of memory pointed to by my_addr. See ip(7) and ipv6(7) for details.

```
int listen(int sockfd, int backlog);
```

specifies willingness to accept connections, and how many incoming connections can be queued.

Unix Sockets API
accepting connections as a server

```
int accept
     (int sockfd,
      struct sockaddr *addr,
      socklen_t *addrlen);
```

waits for an actual connection, returning a new socket to be used for communication, and the address of the peer.

Unix Sockets API
Domain Names (older, easier way)

```
int gethostname(char *name,
                         int len);
```

- if len is greater than the length of the domain name of the local host, fills in name.

```
struct hostent *
   gethostbyname(const char *n);
```

given a null-terminated name (domain name or dotted IP address), returns a host entry, if possible. See gethostbyname(3) for details of the hostent structure.

Unix Sockets AP, Domain Names (newer, more powerful way)

```
int getaddrinfo
   (const char *node,
    const char *service,
    const struct addrinfo *hints,
    struct addrinfo **res);
```

- can return multiple addresses
- service is a port number or service name such as http (port 80)
- hints may request a specific address type
- the result is dynamically allocated, and must be `free`'d

Unix Sockets API
Sending Data

```
int send
   (int s,
    const void *buf, int len,
    int flags);
int sendto
   (int s,
    const void *msg, int len,
    unsigned int flags,
    const struct sockaddr *to,
    socklen_t tolen);
int write
   (int fd,
    const void *buf, int count);
```

- these return the length sent

Unix Sockets API
Sending Data without Connections

- send and write are equivalent (for sockets), and are be used when the sockets are connected. All TCP sockets, and those UDP sockets on which connect has been used, may use `send` or `write`
- `sendto` is used for those UDP sockets that are not connected, and allows send-time decision of where to send. In other words, we can send to many different destinations on a single UDP socket.

Unix Sockets API
Receiving Data

```
int recv(int s,
           void *buf, int len, int flags);
int recvfrom
  (int s, void *buf, int len, int flags,
   struct sockaddr *from,
   socklen_t *fromlen);
int read(int fd, void *buf, int count);
```

correspond to the sending operations, returning:

- the length received,
- 0 if (the OS knows that) the socket has been closed, or
- -1 if there was an error.

Unix Sockets API
Receiving TCP Data

When calling recv on a TCP socket, we may get in return

- the same number of bytes that were sent in one send operation, or
- fewer bytes than were sent in one send operation, or
- more bytes than were sent in one send operation

It's up to your code to deal with this!

TCP Receive Loop

```
char buffer [BIG_ENOUGH];
int rcvd = 0;
while (rcvd < sizeof (buffer)) {
  int new_bytes = recv (s,
     buffer + rcvd,
     sizeof (buffer) - rcvd, 0);
  if (new_bytes <= 0)
      /* exit or return or break  */
  /* new_bytes > 0 */
  rcvd += new_bytes;
  if (received_valid (buffer, rcvd)
      break;
}
process_data (buffer, rcvd);
```