Unix C API for UDP

kv5002

Dr Alun Moon

Computer Science

March 19, 2019

UDP

User Datagram Protocol

- UDP is a simple message-oriented transport layer protocol that is documented in RFC 768.
- it provides no guarantees to the upper layer protocol for message delivery and the UDP layer retains no state of UDP messages once sent.

> *I'd tell you the one about UDP. . . but you might not get it!*

Applications

A number of UDP's attributes make it especially suited for certain applications.

- It is transaction-oriented, suitable for simple query-response protocols such as the Domain Name System or the Network Time Protocol.
- It provides datagrams, suitable for modeling other protocols such as IP tunneling or remote procedure call and the Network File System.
- It is simple, suitable for bootstrapping or other purposes without a full protocol stack, such as the DHCP and Trivial File Transfer Protocol.
- It is stateless, suitable for very large numbers of clients, such as in streaming media applications such as IPTV.
- The lack of retransmission delays makes it suitable for real-time applications such as Voice over IP, online games, and many protocols using Real Time Streaming Protocol.
- Because it supports multicast, it is suitable for broadcast information such as in many kinds of service discovery and shared information such as Precision Time Protocol and Routing Information Protocol.

Receive message on socket

```
ssize_t recvfrom(int sockfd,
                 void *buf, size_t len, int flags,
                 struct sockaddr *src_addr,
                 socklen_t *addrlen);
```

`sockfd` file descriptor of socket to receive message from.

`buf` buffer to write the message into.

`len` size of the buffer

`src_addr` pointer to address structure to be filled in with the source address of the message.

`addrlen` a value-result argument. Before the call, it should be initialized to the size of the buffer associated with `src_addr`. Upon return, `addrlen` is updated to contain the actual size of the source address.

Send message on socket

```
ssize_t sendto(int sockfd,
               const void *buf, size_t len, int flags,
               const struct sockaddr *dest_addr,
               socklen_t addrlen);
```

sockfd file descriptor of socket to send message on.

buf buffer containing message to send.

len size of the message.

src_addr pointer to destination address structure.

addrlen size of the address structure.

Building a framework and API

The API for UDP (and TCP)

- all return values need checking for errors
- parameters for pointers to buffers and structures
- parameters for size of buffers and structures

We (you) might want to write *wrapper* functions to simplify and organise the API, and help write logical program structures.

Get an address

notes

Wrapper for getaddrinfo

- Handle error reporting
- Simplify parameters and return
- Parameters:

  node The address to find, NULL means make a server.

  service port number or service to look-up.

  address pointer to an struct addrinfo structure to fill in.

- Return value,
  - true if succeeded
  - false if failed, errors reported to stderr

Get an address

```
int getaddr(const char *node, const char *service,
                struct addrinfo **address )
{
    struct addrinfo hints = { .ai_flags = 0,
        .ai_family = AF_INET, .ai_socktype = SOCK_DGRAM,
    };
    if( node ) hints.ai_flags = AI_ADDRCONFIG;
    else       hints.ai_flags = AI_PASSIVE;
    int err = getaddrinfo( node, service, &hints, address);
    if(err) {
        fprintf(stderr, "Error getting address: %s\n",
                    gai_strerror( err ) );
        return false;
    }
    return true;
}
```

Create a socket

notes

Wrapper for socket

- Handle error reporting
- Create a socket for IPv4 UDP messages
- Parameters:
  - none
- Return value,
  - socket file descriptor if succeeded
  - 0 (false) if failed, errors reported to stderr

Create a socket

```
int mksocket(void )
{
    int sfd = socket(AF_INET, SOCK_DGRAM, 0);
    if ( sfd == -1 ) {
        fprintf(stderr, "error making socket: %s\n",
                        strerror(errno) );
        return 0;
    }
    return sfd;
}
```

Bind socket to an address

notes

Wrapper to bind

- Handle error reporting
- binds a created socket to an address (server)
- Parameters:

  sfd socket file descriptor

  addr pointer to struct sockaddr with address to bind to

  addlen size of address structure.
- Return value,
  - true if succeeded
  - false if failed, errors reported to stderr

Bind socket to an address

```
int bindsocket(int sfd,
                   const struct sockaddr *addr,
                   socklen_t addrlen )
{
    int err = bind( sfd, addr, addrlen );
    if ( err == -1 ) {
        fprintf(stderr, "error binding socket: %s\n",
                    strerror(errno) );
        return false;
    }
    return true;
}
```

Convert to text

notes

- Convert data in `sockaddr_in` IPv4 socket address into URI *host*:*port* notation.
- parameters:

  `addr` pointer to socket address structure
- return value
  - pointer to buffer (string) with result URI

Convert to text

```
char uri[80];
char *addrtouri(struct sockaddr_in *addr)
{
sprintf(uri, "%s:%d",
                 inet_ntoa(addr->sin_addr),
                 htohs(addr->sin_port) );
return uri;
}
```

Server

notes

- blocks waiting for messages
- creates reply from message
- sends reply (if any)
- parameters:

  `srvrsock` bound socket to wait for messages

  `handlemsg` function to handle message and create reply
- return value
  - Does not return (`while(true)` loop)

Server

```
int server(int srvrsock, handle_t handlemsg)
{
    const size_t buffsize = 4096; /* 4k */
    char  message[buffsize], replybuffer[buffsize];
    size_t msgsize, reply;
    struct sockaddr  clientaddr;
    socklen_t addrlen=sizeof(clientaddr);

    while(true) {
        msgsize = recvfrom(srvrsock, message, buffsize, 0,
                    &clientaddr, &addrlen );
        reply = handlemsg( message, msgsize,
                           replybuffer, buffsize,
                           (struct sockaddr_in*)&clientaddr);
        if(reply) sendto(srvrsock, replybuffer, reply, 0,
                     &clientaddr, addrlen );
    }
```

Server

typedef notes

- Types for function pointers are tricky, especially as parameters
- `typedef` helps simplify this
- type `handle_t` is a pointer to a function that
- parameters

  `char *` pointer to incoming message

  `size_t` size of incoming message

  `char *` pointer to buffer for reply

  `size_t` size of reply buffer

  `struct sockaddr_in *` pointer to an IPv4 socket address structure of the message's origin.

- returns

  `size_t` value containing the reply message length.

Server

```
typedef size_t (*handle_t)(
                           char*, size_t,
                           char*, size_t,
                           struct sockaddr_in *);
```

Exit mechanism

- server is:
  - waiting for message
  - in unending loop
- how to exit cleanly?
  - register signal handler to respond to interrupt (ctrl + C)
  - signal handler calls (exit)
  - register exit function with `atexit`
  - exit function closes server socket
  - socket variable must be global

Exit mechanism

```
int sock;

void finished(int sig)
{
    exit(0);
}

void cleanup(void)
{
close(sock);
}
```

main

```
int main ( int argc , char *argv[] )
{
    struct addrinfo *serveraddr;

    atexit(cleanup);
    signal(SIGINT, finished);

    if( !getaddr(NULL, argv[1], &serveraddr) ) exit(1);;

    if( !(sock = mksocket()) ) exit(1);;

    if( !bindsocket(sock, serveraddr->ai_addr, serveraddr->ai_

    server(sock, udpecho);
}
```

function pointers

In this example several function pointers are used

- to register the signal handler
- to register the clean-up function for use on exiting
- to supply the message handler to the server
  - this makes the server code generic
  - the protocol is implemented by the supplied function to respond to a single message