Unix C API for TCP

kv5002

Dr Alun Moon

Computer Science

March 14, 2019

A note on man pages

How they are referred to

- Entries in the man pages are given with the section number in parenthesis.

quoted as	man command	
ip(7)	`man 7 ip`	`man ip.7`
udp(7)	`man 7 udp`	`man udp.7`
tcp(7)	`man 7 tcp`	`man tcp.7`
service(5)	`man 5 service`	`man service.5`

The Unix API for the TCP/IP stack is well documented in the manual pages.

Manual sections

1. Executable programs or shell commands
2. System calls (functions provided by the kernel)
3. Library calls (functions within program libraries)
4. Special files (usually found in `/dev`)
5. File formats and conventions eg `/etc/passwd`
6. Games
7. Miscellaneous (including macro packages and conventions), e.g. man(7), groff(7)
8. System administration commands (usually only for root)
9. Kernel routines [Non standard]

Network byte order

byteorder(3), htons(3), nstohs(3)

- addresses are 32bit (in ip4)
- ports are 16bit
- *byte order matters*

Utility functions in `<arpa/inet.h>`, standard types in `<stdint.h>`

- convert to and from network and host byte ordering
- work on 16bit (short) and 32bit (long) values

Address lookup

getaddrinfo(3)

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

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

void freeaddrinfo(struct addrinfo *res);

const char * gai)strerror(int errcode);
```

getaddrinfo

- Given node and service, which identify an Internet host and a service, getaddrinfo() returns one or more addrinfo structures, each of which contains an Internet address that can be specified in a call to bind(2) or connect(2).
- getaddrinfo() returns 0 if it succeeds, or a nonzero error code: The gai_strerror() function translates these error codes to a human readable string, suitable for error reporting.
- The getaddrinfo() function allocates and initializes a linked list of addrinfo structures, one for each network address that matches node and service, subject to any restrictions imposed by hints, and returns a pointer to the start of the list in res. The items in the linked list are linked by the ai_next field. The sorting function used within getaddrinfo() is defined in RFC 3484;

addrinfo structure

```
struct addrinfo {
   int ai_flags;
   int ai_family;
   int ai_socktype;
   int ai_protocol;
   socklen_t ai_addrlen;
   struct sockaddr *ai_addr;
   char *ai_canonname;
   struct addrinfo *ai_next;
};
```

Field values

to set in `hints`

`ai_family` set to

- `AF_INET` for ip v4 (see ip(7))
- `AF_INET6` for ip v6 (see ipv6(7))
- `AF_UNSPEC` for either

`ai_socktype` to

- `SOCK_STREAM` for TCP sockets (see tcp(7))
- `SOCK_DGRAM` for UDP sockets (see udp(7))

`ai_flags` combine flags by OR

- `AI_PASSIVE` and `node` is `NULL`, then the returned socket addresses will be suitable for bind(2)ing a socket that will accept(2) connections. *Used in creating servers.*
- `AI_CANONNAME` the `ai_canonname` field of the first of the `addrinfo` structures in the returned list is set to point to the official name of the host.

Example

set-up and query

```
struct addrinfo  hints;
struct addrinfo *results;
int err;

hints.ai_family = AF_UNSPEC;
hints.ai_socktype = 0;
hints.ai_protocol = 0;
hints.ai_flags = AI_CANONNAME;

err = getaddrinfo( "hesabu.net", "http", &hints, &results);

if( err) {
    fprintf(stderr, "Error trying to open %s:%s\n    %s\n",
                 argv[1], argv[2], gai_strerror(err));
    exit(EXIT_FAILURE);
}
```

Example

handle results

```
while( results ) {
  struct sockaddr_in *ipaddr;
  ipaddr = (struct sockaddr_in *)results->ai_addr;
  printf("    canonical name: %s\n", results->ai_canonname);
  printf(" address : %s\n", inet_ntoa( ipaddr->sin_addr ) );
  printf("    port          : %d\n", ntohs(ipaddr->sin_port) )

  results = results->ai_next;
}
```

typically just use

```
struct sockaddr_in *ipaddr =
                   (struct sockaddr_in *)results->ai_addr;
```

Sockets

socket(7), socket(2)

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

sockfd = socket(int socket_family, int socket_type,
                int protocol);
```

`socket()` creates an endpoint for communication and returns a file descriptor that refers to that endpoint. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

typical use

```
int sfd = socket(AF_INET, SOCK_STREAM, 0);
if( sfd == -1 ) { /* error handling */ }
```

Connecting to a server

connect(2)

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sockfd,
            const struct sockaddr *addr, socklen_t addrlen);
```

The `connect()` system call connects the socket referred to by the file descriptor `sockfd` to the address specified by `addr`. The `addrlen` argument specifies the size of `addr`. The format of the address in `addr` is determined by the address space of the socket `sockfd`; see socket(2) for further details.

Using results from getaddrinfo

The getaddrinfo() function has been used to look a server and port. The results are pointed to by results.

```
err = getaddrinfo( "hesabu.net", "http", &hints, &results);
if( err == -1 ) { /* error handling */ }

err = connect( sfd, results->ai_addr, results->ai_addrlen);
if( err == -1 ) { /* error handling */ }
```

Creating a client using TCP

- Lookup server with name/ip address and port/service `getaddrinfo(3)`
- create a socket `socket(2)`
- connect to the server `connect(2)`
- use `send(2)` and `recv(2)` to send and receive messages.

Example

```
int s = socket(AF_INET, SOCK_STREAM, 0);
if( s == -1){
    perror("Error creating socket");
    exit(EXIT_FAILURE);
}

int c = connect(s, results->ai_addr, results->ai_addrlen);
if( s == -1){
    perror("Error connecting to server");
    exit(EXIT_FAILURE);
}
```

Sending Messages

- declare buffer as `char buffer[BUFSIZE]`
- also add `char *message = buffer;`
- use formatted string writing
  `sprintf(buffer, "formats..%d\n", data);`
- with string(3) functions, particularly
  message = strcat(message, ”some string”);
- send with
  `sent = send(c, buffer, strlen(buffer), 0);`
- check for errors `sent == -1`

Example

```
const size_t msgsize = 4096;
char msgbuff[msgsize];
char *msg;

strcpy(msgbuf, "Messsage Header\r\n");
strcat(msgbuf, "v1.1\r\n");
msg = strchr(msgbuf, '\0');
msg += sprintf(msg, "Key:%d\r\n", value);

sent = send(con, msgbuff, strlen(msgbuff), 0);
```

Receiving Messages

- declare buffer as `char buffer[BUFSIZE]`
- byte count `size_t bytes`
- receive with
  `bytes = recv(c, buffer, BUFSIZE-1, 0);`
- check for errors `bytes == -1`
- add terminating zero for string manipulation
  `buffer[bytes]='\0';`

Example

```
const size_t bufsize=4096;
char msgbuf[bufsize];
size_t bytes;

bytes = recv(con, msgbuf, bufsize-1, 0);
if( bytes == -1 ) {/*handle error*/}
msgbuf[bytes]='\0';
```

Unpacking messges

- Typically messages are line oriented with carriage-return line-feed characters (CRLF) as line terminators
- Some protocols are permissive on what counts as a line ending
- lines are formatted, often key:value pairs.

strtok(3)

The `strtok` pair of functions are ideal for splitting the message into lines and parts.

Splitting a message into lines

Reentrant version of strtok

```
/* assume char *message is full message */
char *line;
char *rest;
for(
    line = strtok_r(message, "\r\n", &rest);
    line!=NULL;
    line = strtok_r(NULL, "\r\n", &rest);
   ) {
    /* handle line in here
}
```

Need reentrant version to remember where we are in `rest`

Beware

As used here, `strtok_r` will skip blank lines. Not helpful if you want to identify the end of an HTTP header.

Split a message into key:value pairs

plain strtok

```
char *key;
char *value;
key   = strtok(line, ":");
value = strtok(NULL, ":");
```

strtok remembers it's place internally, which means it can't be nested.

Creating a server using TCP

- Create server address with 'listening port' `getaddrinfo(3)`
- create a socket `socket(2)`
- `bind(2)` the socket to the address
- tell the socket to `listen(2)`
- use `accept(2)` to listen for connections
- use `send(2)` and `recv(2)` to send and receive messages.

Example

```
int s = socket(AF_INET, SOCK_STREAM, 0);

hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;

err = getaddrinfo( NULL, "65421", &hints, &results);
err = bind(s, results->ai_addr, results->ai_socklen);
err = listen(s, 1);

int cfd;
struct sockaddr client;
socklen_t size = sizeof(client);
cfd = accept(s, &client,  &size);
```

send and receive on file-descriptor cfd

Notes & hints

- *Always* check return values for errors
  - see errno(3)
  - see perror(2)
- Assume text (for now)
- *Add* zero terminating byte to string in read buffer

how do I know if I have the whole message? The PROTOCOL defines the beginning and end of a message.

my read buffer isn't big enough! You'll get the message in parts

- bigger buffer
- mechanism to assemble message from multiple calls to `recv`