Network IPC: Sockets

Advanced Programming in the UNIX Programming Environment

Chun-Ying Huang <chuang@cs.nctu.edu.tw>

Introduction

The classical IPC's allow processes running on the same computer to communicate with one another

To allow processes running on different computers with one another: Use Network IPC

Network IPC can be used for both inter-machine communication and intra-machine communication

In this Chapter, we focus majorly on TCP/IP sockes

Socket Descriptors

A socket is an abstraction of a communication endpoint

Socket descriptors are implemented as file descriptors in the UNIX System

Many of the functions that deal with file descriptors, such as *read* and *write*, will work with a socket descriptor

Socket Descriptors (Cont'd)

Creating a socket

- `int socket(int domain, int type, int protocol);`
- Returns: file (socket) descriptor if success, or -1 on error

domain: AF_INET, AF_INET6, AF_UNIX, AF_UNSPEC

type: SOCK_DGRAM, SOCK_RAW, SOCK_STREAM

protocol: 0 (default), IPPROTO_TCP, IPPROTO_UDP, …

Note: the above constants (defines) may locate in different header files

- Most of domain/type resides in sys/socket.h
- IPPROTO_* resides in netinet/in.h

Socket Descriptors and File I/O Functions

Function	Behavior with socket
close	deallocates the socket
dup, dup2	duplicates the file descriptor as normal
fchdir	fails with errno set to ENOTDIR
fchmod	unspecified
fchown	implementation defined
fcntl	some commands supported, including F_DUPFD, F_GETFD, F_GETFL, F_GETOWN, F_SETFD, F_SETFL, and F_SETOWN
fdatasync, fsync	implementation defined
fstat	some stat structure members supported, but how left up to the implementation
ftruncate	unspecified
ioctl	some commands work, depending on underlying device driver
lseek	implementation defined (usually fails with errno set to ESPIPE)
read	equivalent to recv without any flags
write	equivalent to send without any flags

Release a Socket Descriptor

Communication on a socket is bidirectional

We can disable I/O on a socket with the shutdown function

Synopsis

- `int shutdown(int sockfd, int how);`
- Returns: zero if success, or -1 on error

how: SHUT_RD, SHUT_WR, and SHUT_RDWR

Why do we need shutdown?

- shutdown closes the socket descriptor immediately (independent of the number of references to the descriptor)
- shutdown is able to half-close a socket descriptor

Addressing

How to identify a socket?

We need addressing schemes

- AF_UNIX
  – a pathname
- AF_INET + SOCK_STREAM + IP_PROTO_TCP
  – IPv4 address and TCP port number
- AF_INET + SOCKET_DGRAM + IP_PROTO_UDP
  – IPv4 address and UDP port number
- AF_INET6 + SOCK_STREAM + IP_PROTO_TCP
  – IPv6 address and TCP port number
- AF_INET6 + SOCKET_DGRAM + IP_PROTO_UDP
  – IPv6 address and UDP port number

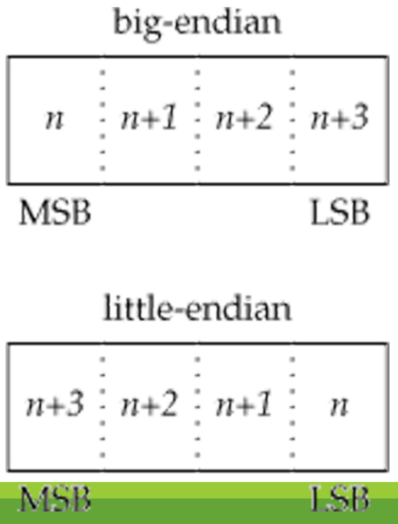
Address and Byte Ordering

When communicating with processes running on the same computer, we generally don't have to worry about byte ordering

It is a problem when addresses are represented in numeric forms

The byte order is a characteristic of the processor architecture

- It only affects larger data types, e.g., an integer



Address and Byte Ordering (Cont’d)

The byte ordering becomes visible to applications when they exchange formatted data

The TCP/IP protocol suite uses big-endian byte order

- The network byte order

Four common functions are provided to convert byte orders

- `uint32_t htonl(uint32_t hostint32);`
- `uint16_t htons(uint16_t hostint16);`
- `uint32_t ntohl(uint32_t netint32);`
- `uint16_t ntohs(uint16_t netint16);`

Address Formats: Generic

The generic sockaddr structure on Linux

```
struct sockaddr {
    sa_family_t sa_family;      /* address family */
    char sa_data[14];           /* variable-length address */
};
```

However, it may be a little bit different on some other systems

```
struct sockaddr {
    unsigned char sa_len        /* total length */
    sa_family_t sa_family;      /* address family */
    char sa_data[14];           /* variable-length address */
};
```

Address Formats: IPv4 on Linux

```
typedef uint16_t in_port_t;
typedef uint32_t in_addr_t;

struct in_addr {
     in_addr_t s_addr;              /* IPv4 address */
};

struct sockaddr_in {
     sa_family_t sin_family;        /* address family */
     in_port_t sin_port;            /* port number */
     struct in_addr sin_addr;       /* IPv4 address */
     unsigned char sin_zero[8];
};
```

Address Formats: IPv6 on Linux

```
typedef uint16_t in_port_t;

struct in6_addr {
     union {                                /* IPv4 address */
          uint8_t  __u6_addr8 [16];
          uint16_t __u6_addr16[8];
          uint32_t __u6_addr32[4];
     } __in6_u;
};

struct sockaddr_in6 {
     sa_family_t sin_family;       /* address family */
     in_port_t sin6_port;          /* port number */
     uint32_t sin6_flowinfo;       /* IPv6 flow info */
     struct in6_addr sin6_addr;    /* IPv6 address */
     uint32_t sin6_scope_id;       /* IPv6 scope id */
};
```

Conversion of Address

To print an address in a readable format, we have to convert a numeric address to a text

-
  ```
  const char *inet_ntop(int af,
                        const void *src, char *dst, socklen_t cnt);
  ```

- Returns: pointer to address string on success, or NULL on error

To convert a text formatted address to a numeric address

-
  ```
  int inet_pton(int af, const char *src, void *dst);
  ```

- Returns: 1 on success, 0 if the format is invalid, or -1 on error and errno=EAFNOSUPPORT

Example: Address Conversion

See netipc/addr.c

```
if(inet_pton(AF_INET, argv[1], &addr4) == 1) {
    printf("IPv4: 0x%08x\n", htonl(addr4.s_addr));
}

if(inet_pton(AF_INET6, argv[1], &addr6) == 1) {
    printf("IPv6: 0x%08x%08x%08x%08x\n",
        htonl(addr6.__in6_u.__u6_addr32[0]),
        htonl(addr6.__in6_u.__u6_addr32[1]),
        htonl(addr6.__in6_u.__u6_addr32[2]),
        htonl(addr6.__in6_u.__u6_addr32[3]));
}
```

Address Lookup – Known Hosts

How do we get all known hosts in the system?

Check the /etc/hosts file

In a UNIX program, we can get all known hosts by the function:

- `struct hostent *gethostent(void);`
- Returns: valid pointer if success, or NULL on error
- NOTE: gethostent() is not thread-safe

```
struct hostent {
        char   *h_name;                 /* official name of host */
        char  **h_aliases;              /* alias list */
        int     h_addrtype;             /* host address type */
        int     h_length;               /* length of address */
        char  **h_addr_list;            /* list of addresses */
}
```

Address Lookup

Known hosts

Known protocols (IP, TCP, UDP, …)

Known services (TELNET, FTP, WWW, …)

Address Lookup – Known Hosts (Cont'd)

Two functions related to gethostent()

sethostent

- `void sethostent(int stayopen);`
- Open host database if it is not already open
- Rewind it if it is already open

endhostent

- `void endhostent(void);`
- Close the host database

NOTE: The two functions have different meanings when lookup via DNS

Address Lookup, an Example – Get All Known Hosts

```
int main() {
        int i;
        char buf[64];
        struct hostent *h;
        while((h = gethostent()) != NULL) {
                if(h->h_addrtype != AF_INET)
                        continue;
                printf(“name=%s, addr={ “, h->h_name);
                for(i = 0; h->h_addr_list[i] != NULL; i++) {
                        printf("%s ", inet_ntop(AF_INET,
                               h->h_addr_list[i], buf, sizeof(buf)));
                }
                printf("}\n");
        }
        return(0);
}
```

```
name=cshome.cs.nctu.edu.tw, addr={ 140.113.235.101 }
name=csduty.cs.nctu.edu.tw, addr={ 140.113.235.102 }
```

Address Lookup – Known Protocols

How do we get all known protocols in the system?

Check the /etc/protocols file

In a UNIX program, we can get all known protocols by the function:

- `struct protoent *getprotoent(void);`
- Returns: valid pointer if success, or NULL on error
- NOTE: getprotoent() is not thread-safe

```
struct protoent {
        char  *p_name;                 /* official protocol name */
        char **p_aliases;              /* alias list */
        int    p_proto;                /* protocol number */
}
```

Address Lookup
– Known Protocols (Cont'd)

Two functions related to getprotoent()

setprotoent

- `void setprotoent(int  stayopen);`
- Open protocol database if it is not already open
- Rewind it if it is already opened

endprotoent

- `void endprotoent(void);`
- Close the protocol database

Address Lookup, an Example – Get All Known Protocols

```
int main() {
      int i;
      struct protoent *p;
      while((p = getprotoent()) != NULL) {
            printf("name=%s (%d), ", p->p_name, p->p_proto);
            printf("alias={ ");
            for(i = 0; p->p_aliases[i] != NULL; i++)
                  printf("%s ", p->p_aliases[i]);
            printf("}\n");
      }
      return(0);
}
```

```
name=ip (0), alias={ IP }
name=icmp (1), alias={ ICMP }
name=igmp (2), alias={ IGMP }
...
name=tcp (6), alias={ TCP }
...
name=udp (17), alias={ UDP }
...
```

Address Lookup – Get a Specific Protocol

In addition to iteratively list all protocols, we can retrieve information about a given protocol

- `struct protoent *getprotobyname(const char *name);`
- `struct protoent *getprotobynumber(int proto);`
- Returns: valid pointer if success, or NULL on error
- NOTE: Both the two functions are not thread-safe

Behavior of setprotoent

- By default, getprotocolby* functions close the protocol database after a query
- If we use setprotoent with stayopen = true (1), the getprotocolby* functions do not close the database

Address Lookup – Known Services

How do we get all known services in the system?

Check the /etc/services file

In a UNIX program, we can get all known services by the function:

- `struct servent *getservent(void);`
- Returns: valid pointer if success, or NULL on error
- NOTE: getservent() is not thread-safe

```
struct servent {
      char  *s_name;            /* official service name */
      char **s_aliases;         /* alias list */
      int    s_port;            /* port number */
      char  *s_proto;           /* protocol to use */
}
```

Address Lookup
– Known Services (Cont'd)

Two functions related to getservent()

setservent

- `void setservent(int stayopen);`
- Open service database if it is not already open
- Rewind it if it is already open

endservent

- `void endprotoent(void);`
- Close the service database

Address Lookup, an Example – Get All Known Services

```
int main() {
        int i;
        struct servent *s;
        while((s = getservent()) != NULL) {
                printf("name=%s (%s/%d), ",
                        s->s_name, s->s_proto, ntohs(s->s_port));
                printf("alias={ ");
                for(i = 0; s->s_aliases[i] != NULL; i++)
                        printf("%s ", s->s_aliases[i]);
                printf("}\n");
        }
        return(0);
}
```

```
...
name=telnet (tcp/23), alias={ }
name=ftp (tcp/21), alias={ }
name=pop3 (tcp/110), alias={ pop-3 }
name=www (tcp/80), alias={ http }
...
```

Address Lookup – Get a Specific Service

In addition to iteratively list all services, we can retrieve information about a given service

-
  ```
  struct servent *getservbyname(const char *name,
                                const char *proto);
  ```

-
  ```
  struct servent *getservbyport(int port, const char *proto);
  ```
  - NOTE: port should be in network byte order
- Returns: valid pointer if success, or NULL on error
- NOTE: Both the two functions are not thread-safe

Behaviors of setservent

- By default, each getservby* function close the service database after a query
- If we use setservent with stayopen = true (1), the getservby* function will not close the database

Address Lookup, Hostname via DNS

In addition to iteratively list all known hosts from system database, we can retrieve information about a given host via DNS

- `struct hostent *gethostbyname(const char *name);`

-
  ```
  struct hostent *gethostbyaddr(const void *addr,
                                socklen_t len, int type);
  ```
  - type can be either AF_INET or AF_INET6
- Returns: valid pointer if success, or NULL on error
- NOTE: Both the two functions are not thread-safe

Behaviors of sethostent

- By default, gethostby* functions query DNS using UDP protocol
- If we use sethostent with stayopen = true (1), the gethostby* functions use TCP to query the DNS (and keep the connection alive)

Thread-Safe Query of Address and Port (1/5)

The getaddrinfo function

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

Parameters

- node: the node to be queried (name or address)
- service: name of the service
- hints: query criteria
  - flags (see the next slide)
  - address family (AF_INET/AF_INET6)
  - socktype (SOCK_DGRAM/SOCK_STREAM, can be 0)
  - protocol (can be 0)
  - Other fields must be zero
- res: return the queried result

Returns: zero if success, or nonzero error code on error

Thread-Safe Query of Address and Port (2/5)

The addrinfo data structure

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

Thread-Safe Query of Address and Port (3/5)

The ai_flags

Flag	Description
AI_ADDRCONFIG	Query for whichever address type (IPv4 or IPv6) is configured
AI_ALL	Look for both IPv4 and IPv6 addresses (used only with AI_V4MAPPED)
AI_CANONNAME	Request a canonical name (as opposed to an alias)
AI_NUMERICHOST	Return the host address in numeric format
AI_NUMERICSERV	Return the service as a port number
AI_PASSIVE	Socket address is intended to be bound for listening
AI_V4MAPPED	If no IPv6 addresses are found, return IPv4 addresses mapped in IPv6 format

Thread-Safe Query of Address and Port (4/5)

Handle error returned from getaddrinfo

If getaddrinfo fails, we can not use perror or strerror to generate an error message

We need to call gai_strerror to convert the error code returned into an error message

- `const char *gai_strerror(int error);`

Thread-Safe Query of Address and Port (5/5)

```
$ ./getaddrinfo google.com www
74.125.45.100:80
209.85.171.100:80
74.125.67.100:80
```

```
int main(int argc, char *argv[]) {
    int s;
    struct addrinfo hints, *result, *rp;
    if (argc < 3) {
        fprintf(stderr, "usage: %s host port\n", argv[0]);
        exit(-1);
    }
    bzero(&hints, sizeof(struct addrinfo));
    hints.ai_family = AF_INET;          /* allow IPv4 or IPv6 */
    hints.ai_socktype = SOCK_STREAM; /* stream socket */
    hints.ai_flags = 0;
    hints.ai_protocol = 0;              /* any protocol */
    if((s=getaddrinfo(argv[1], argv[2], &hints, &result))!=0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
        exit(-1);
    }
    for(rp = result; rp != NULL; rp = rp->ai_next) {
        struct sockaddr_in *p = (struct sockaddr_in*) rp->ai_addr;
        printf( "%s:%d\n", inet_ntoa(p->sin_addr), ntohs(p->sin_port));
    }
    return(0);
}
```

Thread-Safe Query of Name and Service

The inverse function of getaddrinfo

Synopsis

-
  ```
  int getnameinfo(const struct sockaddr *sa, socklen_t salen,
                  char *host, size_t hostlen,
                  char *serv, size_t servlen, int flags);
  ```
- Returns: zero if success, or nonzero on error

Flag	Description
NI_DGRAM	The service is datagram (UDP) based rather than stream (TCP)
NI_NAMEREQD	An error is returned if the hostname cannot be determined
NI_NOFQDN	Return only the hostname part of the fully qualified domain name for local hosts
NI_NUMERICHOST	Return the numeric form of the host address instead of the name
NI_NUMERICSERV	Return the numeric form of the service address (i.e., the port number) instead of the name

Thread-Safe Query of Name and Service, an Example

```
int main(int argc, char *argv[]) {
        struct sockaddr_in sin;
        char host[64], serv[64];
        int s;
        if (argc < 3) {
                fprintf(stderr, "usage: %s ip port\n", argv[0]);
                exit(-1);
        }
        bzero(&sin, sizeof(sin));
        sin.sin_family = AF_INET;
        sin.sin_addr.s_addr = inet_addr(argv[1]);
        sin.sin_port = htons(atoi(argv[2]));
        if((s = getnameinfo((struct sockaddr*) &sin, sizeof(sin),
                host, sizeof(host), serv, sizeof(serv), 0)) != 0) {
                fprintf(stderr, "getnameinfo: %s\n", gai_strerror(s));
                exit(-1);
        }
        printf("%s:%s\n", host, serv);
        return(0);
}
```

```
$ ./getnameinfo 74.125.45.100 80
yx-in-f100.google.com:www
```

Associate Addresses with Sockets

Usually, a client does not need to bind an address with a socket

- The server automatically chooses the address for the socket

However, a server has to bind an address with a socket

-
  ```
  int bind(int sockfd, const struct sockaddr *addr,
                socklen_t addrlen);
  ```
- Returns: zero if success, or -1 on error
- The address we specify must be valid for the machine
  - It can be zero – bound to all interfaces
- The port number in the address cannot be less than 1,024 (only superuser can do that)
- Usually, only one socket endpoint can be bound to a given address

Discover the Address Bound to a Socket

Get the local address bound to a socket

-
  ```
  int getsockname(int  s, struct sockaddr *name,
                       socklen_t *namelen);
  ```

Get the remote address bound to a socket

- If the socket is connected to a peer

-
  ```
  int getpeername(int  s, struct sockaddr *name,
                       socklen_t *namelen);
  ```

Notes

- The *name* and the *namelen* must be declared first
- Before calling getsockname or getpeername, the *namelen* must be set to the length of the *name* data structure

Connection Establishment

If a client is dealing with a connection-oriented network service (SOCK_STREAM)

It has to create a connection before exchanging data

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

- Returns: zero if success, or -1 on error
- If *sockfd* is not bound to an address, connect will bind a default address for the caller

Listen for an Incoming Connection

A server can announce that it is willing to accept connect requests

The *listen* function

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

- Returns: zero if success, or -1 on error
- backlog
  - The number of outstanding connect requests in a queue
  - In Linux, the max allowable is 128 (defined by the SOMAXCONN constant)
  - Once the queue is full, the system will reject additional connect requests
  - The *backlog* must be chosen based on the expected load of the server

Accept Incoming Connections

Once a server has called listen, the socket used can receive connect requests

The *accept* function

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

- Returns: file (socket) descriptor if success, or -1 on error
- The returned descriptor is the socket connected to the client
- This new socket descriptor has the same socket type and address family as the original *sockfd*
- The *addr* holds the address and the port of the client
  - It can be NULL if we do not need these information
- If no connect requests are pending, accept will block until one arrives

Establish IPv4 TCP Connections – Summary

Server side

- `fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)`
- Provide a sockaddr_in data structure – sin
- `bind(fd, (struct sockaddr*) &sin, sizeof(sin))`
- `listen(fd, backlog)`
- `pfd = accept(fd, &psin, sizeof(psin))`

Client side

- `fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)`
- Provide a sockaddr_in data structure – sin
- `connect(fd, (struct sockaddr*) &sin, sizeof(sin))`

Data Transfer – Send Data

We can use the *write* function to send data via descriptors

However, there are more flexible functions can be used

- For *connection oriented* only

  ```
  ssize_t send(int sockfd, const void *buf, size_t nbytes,
               int flags);
  ```

  - It is equivalent to *write* if the *flags* is set to zero

- For both *Connection oriented* and *connectionless*

  ```
  ssize_t sendto(int sockfd, const void *buf, size_t nbytes,
                 int flags, const struct sockaddr *destaddr,
                 socklen_t destlen);
  ```

  - We have to specify a *destaddr* in connectionless mode

- Returns: number of bytes sent if success, or -1 on error

Data Transfer – Send Data (Cont’d)

Flags

Flag	Description
MSG_DONTROUTE	Don't route packet outside of local network
MSG_DONTWAIT	Enable non-blocking operation (equivalent to using O_NONBLOCK)
MSG_EOR	This is the end of record if supported by protocol
MSG_OOB	Send out-of-band data if supported by protocol

Data Transfer – Receive Data

We can use the *read* function to receive data via descriptors

There are also more flexible functions can be used

- For *connection oriented* only

  ```
  ssize_t recv(int sockfd, void *buf, size_t nbytes,   int flags);
  ```

  - It is equivalent to *read* if the *flags* is set to zero
- For both *Connection oriented* and *connectionless*

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

  - The *addr* contains the address/port of the data sender (for connectionless mode only)
- Returns: number of bytes received if success, 0 if no messages are available and peer has done an orderly shutdown, or -1 on error

Data Transfer – Receive Data (Cont'd)

Flags

Flag	Description
MSG_OOB	Receive out-of-band data if supported by protocol
MSG_PEEK	Return packet contents without consuming packet
MSG_TRUNC	Return that the real length of the packet, even if it was longer than the passed buffer (Only valid for packet sockets)
MSG_WAITALL	Wait until all data is available, i.e., the passed buffer is all filled (SOCK_STREAM only)

A Server Example – TCP ECHO Server (1/5)

```
int main(int argc, char *argv[]) {
      pid_t pid;
      int fd, pfd, val;
      struct sockaddr_in sin, psin;
      if(argc < 2) {
            fprintf(stderr, "usage: %s port\n", argv[0]);
            return(-1);
      }
      signal(SIGCHLD, SIG_IGN);
      if((fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
            perror("socket");
            return(-1);
      }
      val = 1;
      if(setsockopt(fd,
            SOL_SOCKET, SO_REUSEADDR, &val, sizeof(val)) < 0) {
            perror("setsockopt");
            return(-1);
      }
```

A Server Example – TCP ECHO Server (2/5)

```
bzero(&sin, sizeof(sin));
sin.sin_family = AF_INET;
sin.sin_port = htons(atoi(argv[1]));
if(bind(fd, (struct sockaddr*) &sin, sizeof(sin)) < 0) {
        perror("bind");
        return(-1);
}
if(listen(fd, SOMAXCONN) < 0) {
        perror("listen");
        return(-1);
}
```

A Server Example – TCP ECHO Server (3/5)

```
while(1) {
        val = sizeof(psin);
        bzero(&psin, sizeof(psin));
        if((pfd=accept(fd, (struct sockaddr*) &psin, &val))<0) {
                perror("accept");
                return(-1);
        }
        if((pid = fork()) < 0) {
                perror("fork");
                return(-1);
        } else if(pid == 0) {      /* child */
                close(fd);
                serv_client(pfd, &psin);
                exit(0);
        }
        /* parent */
        close(pfd);
}
```

A Server Example – TCP ECHO Server (4/5)

```
void serv_client(int fd, struct sockaddr_in *sin) {
      int len;
      char buf[2048];
      printf("connected from %s:%d\n",
            inet_ntoa(sin->sin_addr), ntohs(sin->sin_port));
      while((len = recv(fd, buf, sizeof(buf), 0)) > 0) {
            if(send(fd, buf, len, 0) < 0) {
                  perror("send");
                  exit(-1);
            }
      }
      printf("disconnected from %s:%d\n",
            inet_ntoa(sin->sin_addr), ntohs(sin->sin_port));
      return;
}
```

A Server Example – TCP ECHO Server (5/5)

Running the server

- $ ./echosrv 12345
- If you run *netstat -na* command, you should see:
  - `tcp    0    0 0.0.0.0:12345    0.0.0.0:*    LISTEN`

Running the client (using telnet)

- Type something and press Enter
- You should see the same string echoed back
- Press ^] and type quit to terminate the client

```
$ telnet localhost 12345
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
test 123
test 123
hello world
hello world
^]
telnet> quit
Connection closed.
```

Socket Options

The behavior of sockets can be controlled by options

Interfaces

- int setsockopt(int sockfd, int level, int option, const void *val, socklen_t len);
- int getsockopt(int sockfd, int level, int option, void *val, socklen_t *lenp);
- Returns: zero if success, or -1 on error
- The *level* argument
  - Identify the protocol (by a protocol number) to apply
    - For example, IPPROTO_IP, IPPROTO_TCP, …
  - If the option is a generic socket-level option, then level is set to SOL_SOCKET

Generic Socket Options

Option	Type of val	Description
SO_ACCEPTCONN	int	Return whether a socket is enabled for listening (getsockopt only)
SO_BROADCAST	int	Broadcast datagrams if *val is nonzero
SO_DEBUG	int	Debugging in network drivers enabled if *val is nonzero
SO_DONTROUTE	int	Bypass normal routing if *val is nonzero
SO_ERROR	int	Return and clear pending socket error (getsockopt only)
SO_KEEPALIVE	int	Periodic keep-alive messages enabled if *val is nonzero
SO_LINGER	struct linger	Delay time when unsent messages exist and socket is closed
SO_OOBINLINE	int	Out-of-band data placed inline with normal data if *val is nonzero

Generic Socket Options (Cont’d)

Option	Type of val	Description
SO_RCVBUF	int	The size in bytes of the receive buffer
SO_RCVLOWAT	int	The minimum amount of data in bytes to return on a receive call
SO_RCVTIMEO	struct timeval	The timeout value for a socket receive call
SO_REUSEADDR	int	Reuse addresses in bind if *val is nonzero
SO_SNDBUF	int	The size in bytes of the send buffer
SO_SNDLOWAT	int	The minimum amount of data in bytes to transmit in a send call
SO_SNDTIMEO	struct timeval	The timeout value for a socket send call
SO_TYPE	int	Identify the socket type (getsockopt only)

Q & A