Network Inter-Process Communication: Sockets

Socket Descriptors

- Socket Descriptors
  - abstraction of a communication endpoint
  - implemented as file descriptors in the UNIX system
  - many functions can deal with file descriptors as well as socket descriptors
- `socket(2)`: `int socket(int domain, int type, int protocol);`
  - return: file descriptor if OK, -1 error
  - `domain`: `AF_INET`, `AF_INET6`, `AF_UNIX`, `AF_UNSPEC`
  - `type`: `SOCK_DGRAM`, `SOCK_RAW`, `SOCK_STREAM`
  - `protocal`: 0 (default), `IPPROTO_TCP`, `IPPROTO_UDP`, ...
  - constants may locate in different header files, most of
    - `domain`, `type` resides in `sys/socket.h`
    - `protocal` resides in `netinet/in.h`
- Socket Descriptors and File I/O Functions

Function	Behavior with socket
`close(2)`	Deallocates the socket
`dup(2)`, `dup2(2)`	Duplicates the file descriptor as normal
`fchdir(2)`	Fails with `errno` set to `ENOTDIR`
`fchmod(2)`	Unspecified
`fchown(2)`	Implementation defined
`fcntl(2)`	Some commands supported, including `F_DUPFD`, `F_GETFD`, `F_GETFL`, `F_GETOWN`, `F_SETFD`, `F_SETFL`, and `F_SETOWN`
`fdatasync(2)`, `fsync(2)`	Implementation defined
`fstat(2)`	Some stat structure members supported, but how left up to the implementation
`ftruncate(2)`	unspecified
`ioctl(2)`	some commands work, depending on underlying device driver

Function	Behavior with socket
`lseek(2)`	implementation defined (usually fails with `errno` set to `ESPIPE`)
`read(2)`	equivalent to `recv(2)` without any flags
`write(2)`	equivalent to `send(2)` without any flags

- Release a Socket Descriptor
  - communication on a socket is bidirectional
  - `shutdown(2)`: `int shutdown(int sockfd, int how);`
    - return: 0 OK, -1 error
    - `how`: `SHUT_RD`, `SHUT_WR`, `SHUT_RDWR`
    - close the socket descriptor immediately
    - able to half-close a socket descriptor

Addressing

- Schemes
  - `AF_UNIX`: local communication
  - `AF_INET` + `SOCK_STRAEM` + `IP_PROTO_TCP`
  - `AF_INET` + `SOCK_DGRAM` + `IP_PROTO_UDP`
  - `AF_INET6` + `SOCK_STRAEM` + `IP_PROTO_TCP`
  - `AF_INET6` + `SOCK_DGRAM` + `IP_PROTO_UDP`
- Byte Ordering
  - `h`: host, `n`: network
  - `l`: 4 bytes, `s`: 2 bytes
  - `htonl(3)`: `uint32_t htonl(uint32_t hostlong);`
  - `htons(3)`: `uint16_t htons(uint16_t hostshort);`
  - `ntohl(3)`: `uint32_t ntohl(uint32_t netlong);`
  - `ntohs(3)`: `uint16_t ntohs(uint16_t netshort);`
- Address Format
  - Linux

```
struct socketaddr {
  sa_family_t sa_family;
  char sa_data[14];
};

typedef uint16_t in_port_t;
typedef uint32_t in_addr_t;

struct in_addr {
  in_addr_t s_addr;
};

struct sockaddr_in {
  sa_family_t sin_family;
  in_port_t sin_port;
  struct in_addr sin_addr;
```

```
  unsigned char sin_zero[8];
};

struct in6_addr {
  union {
    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;
  in_port_t sin6_port;
  uint32_t sin6_flowinfo;
  struct in6_addr sin6_addr;
  uint32_t sin6_scope_id;
};
```

- `struct sockaddr` may be different on some other systems

```
struct sockaddr {
  unsigned char sa_len;
  sa_family_t sa_family;
  char sa_data[14];
}
```

- conversion of address
  - `inet_ntop(3)`: `const char *inet_ntop(int af, const void *src, char *dst, socklen_t size);`
    - return: pointer to `dst` OK, `NULL` error
    - network address structure to readable format
    - `size`: `dst` size
    - `src` is datatype `struct in_addr` or `struct in6_addr`
  - `inet_pton(3)`: `int inet_pton(int af, const char *src, void *dst);`
    - return: 1 on success, 0 if `src` is invalid, -1 if `af` is not valid with `errno` set to `EAFNOSUPPORT`
    - readable format to network address structure
    - `dst` is datatype `struct in_addr` or `struct in6_addr`
  - example

```
if (inet_pton(AF_INET, argv[1], &addr4) == 1) {
  printf("IPv4: 0x%.8x\n", htonl(addr4.s_addr));
}
```

```
  if (inet_pton(AF_INET6, argv[1], &addr6) == 1) {
    printf("IPv6: 0x%.8x%.8x%.8x%.8x\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
  - check `/etc/hosts`
  - `gethostent(3)`: `struct hostent *gethostent(void);`
    - return: pointer if success, `NULL` error
    - get all known hosts
    - 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 */
};
```

  - `sethostent(3)`: `void sethostent(int stayopen);`
    - open host database if it is not already open
    - rewind it if it is already open
    - DNS: `stayopen`
      - 1: TCP socket is used for name server queries and the connection remains open
      - 0: UDP datagrams is used for name server queries
  - `endhostent(3)`: `void endhostent(void);`
    - close the host database
    - DNS: ends TCP connection for name server queries
  - example

```
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;
}
```

```
$ ./gethostent
name=localhost, addr={ 127.0.0.1 }
name=ee904-itri-pc2, addr={ 127.0.1.1 }
name=ip6-localhost, addr={ 127.0.0.1 }
```

- Known Protocols
  - check `/etc/protocols`
  - `getprotoent(3)`: `struct protoent *getprotoent(void);`
    - return: pointer if success, `NULL` error
    - get all known protocols
    - not thread safe

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

}
```

  - `setprotoent(3)`: `void setprotoent(int stayopen);`
    - open protocol database if it is not already open
    - rewind it if it is already opened
  - `endprotoent(3)`: `void endprotoent(void);`
    - close the protocol database
  - example

```
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;
}
```

```
$ ./getprotoent
name=ip (0), alias={ IP }
name=hopopt (0), alias={ HOPOPT }
name=icmp (1), alias={ ICMP }
name=igmp (2), alias={ IGMP }
...
name=hip (139), alias={ HIP }
name=shim6 (140), alias={ Shim6 }
name=wesp (141), alias={ WESP }
name=rohc (142), alias={ ROHC }
```

- `getprotobyname(3)`: `struct protoent *getprotobyname(const char *name);`
- `getprotobynumber(3)`: `struct protoent *getprotobynumber(int proto);`
  - return: pointer if success, `NULL` error
  - both of them are not thread safe
  - by default, they close protocol database
  - the database can remains open if `setprotoent(1)` is called

- Known Services
  - `struct servent *getservent(void);`
    - return: pointer if success, `NULL` error
    - get all known services
    - 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 */
}
```

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

- `void endservent(void)`
  - close the service database
- example

```
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;
}
```

```
$ ./getservent
name=tcpmux (tcp/1), alias={ }
name=echo (tcp/7), alias={ }
name=echo (udp/7), alias={ }
name=discard (tcp/9), alias={ sink null }
...
name=csync2 (tcp/30865), alias={ }
name=dircproxy (tcp/57000), alias={ }
name=tfido (tcp/60177), alias={ }
name=fido (tcp/60179), alias={ }
```

- `struct servent *getservbyname(const char *name, const char *proto);`
- `struct servent *getservbyport(int port, const char *proto);`
  - return: pointer if success, `NULL` error
  - `port` **should be in network byte order**
  - both of them are not thread safe
  - by default, they close service database
  - the database can remains open if `setservent(1)` is called

- Host by DNS
  - `gethostbyname(3)`: `struct hostent *gethostbyname(const char *name);`
  - `gethostbyaddr(3)`: `struct hostent *gethostbyaddr(const void *addr, socklen_t len, int type);`
    - return: pointer if success, `NULL` error

    - `type` can be either `AF_INET` or `AF_INET6`
    - both of them are not thread safe
    - by default, they do queries by UDP protocol
    - the queries use TCP and keeps alive if `sethostent(1)` is called

- Thread-Safe Query of Address and Port

  - `getaddrinfo(3)`: `int getaddrinfo(const char *node, const char *service, const struct addrinfo *hints, struct addrinfo **res);`

    - return: 0 OK, nonzero error
    - `node`: the node to be queried (name or address)
    - `service`: service name
    - `hints`: query criteria
      - same data type as `*res`
        - `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

        - `ai_family`: `AF_INET` or `AF_INET6`
        - `ai_socktype`: `SOCK_DGRAM` or `SOCK_STREAM`, can be zero
        - `ai_protocol`: can be zero
        - other fields must be zero
    - `res`: result

```
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;
  };
```

- `gai_strerror(3)`: `const char *gai_strerror(int errcode);`
  - **this function should be used instead of `perror(3)` or `strerror(3)`**
  - handle error code return `getaddrinfo(3)`
- example

```
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 */
  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;
}
```

```
$ ./getaddrinfo google.com www
216.58.200.238:80
```

- Thread-Safe Query of Name and Service
  - `getnameinfo(3)`: `int getnameinfo(const struct sockaddr *addr, socklen_t addrlen, char *host, socklen_t hostlen, char *serv, socklen_t servlen, int flags);`
    - return: 0 OK, nonzero error
    - `flags`

Flag	Description

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. port number) instead of the name

- 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]);  // convert to
network byte order
  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 216.58.200.238 80
tsa03s01-in-f14.1e100.net:http
```

Associate Addresses with Sockets

- Usually Client Does not Need to Bind
  - server automatically chooses the address for the socket
- Server has to Bind
  - `bind(2)`: `int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);`

        - return: 0 OK, -1 error
        - `addr` must be valid for machine
            - zero - bound to all interfaces
        - port number in `addr` cannot be less than 1024 (only superuser can do that)
        - usually, only one socket endpoint can be bound to a given address
- Discover the Address Bond to a Socket
    - `getsockname(2)`: `int getsockname(int sockfd, struct sockaddr *addr, socklen_t *addrlen);`
    - `getpeername(2)`: `int getpeername(int sockfd, struct sockaddr *addr, socklen_t *addrlen);`
        - return: 0 OK, -1 error
        - `getsockname(2)` get local address bound to a socket
        - `getpeername(2)` get remote address bound to a socket
        - `addrlen` must be set to length of `addr` before calling

Connection Establishment

- `connect(2)`: `int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);`
    - return: 0 OK, -1 error
    - client create connection before exchanging data (`SOCK_STREAM`)
    - if `sockfd` is not bound to an address, default address will be bound
- `listen(2)`: `int listen(int sockfd, int backlog);`
    - return: 0 OK, -1 error
    - server is willing to accept connect requests
    - `backlog`
        - number of outstanding connect requests in a queue
        - max 128 (`SOMAXCONN`) in Linux
        - system will reject additional requests once the queue is full
- `accept(2)`: `int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);`
    - return: file descriptor connected to the client, -1 error
    - the new file descriptor has same socket type and address family as `sockfd`
    - `addr` holds client address and port number
        - set `NULL` if we do not need
    - if no requests are pending, `accept(2)` will block
- Connections Summary

```
// server
fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
bind(fd, (struct sockaddr *)&sin, sizeof(sin));
listen(fd, backlog);
pfd = accept(fd, &psin, sizeof(psin));
```

```
// client
fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
connect(fd, (struct sockaddr *)&sin, sizeof(sin));
```

- Send Data
  - `send(2)`: `ssize_t send(int sockfd, const void *buf, size_t len, int flags);`
  - `sendto(2)`: `ssize_t sendto(int sockfd, const void *buf, size_t len, int flags, const struct sockaddr *dest_addr, socklen_t addrlen);`
    - return: number of bytes sent, -1 error
    - `send(2)` is for connection oriented only
      - equal to `write(2)` if `flags` is zero
    - `sendto(2)` is for both connection oriented and connectionless
      - `dest_addr` need to be specified in connectionless mode
    - `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

- Receive Data
  - `recv(2)`: `ssize_t recv(int sockfd, void *buf, size_t len, int flags);`
  - `recvfrom(2)`: `ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags, struct sockaddr *src_addr, socklen_t *addrlen);`
    - return: number of bytes received, 0 if no message available (`EOF`), -1 error
    - `recv(2)` is for connection oriented only
      - equal to `read` if `flags` is zero
    - `recvfrom(2)` is for both connection oriented and connectionless
      - `src_addr` need to be specified in connectionless mode
    - `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)

- Example: TCP Echo Server

```
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;
}

int main(int argc, char *argv[]) {
  pid_t pid;
  int fd, pfd;
  unsigned 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);
  }

  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);
  }
  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) {
      close(fd);
      serv_client(pfd, &psin);
      exit(0);
    }
    close(pfd);
  }
}
```

```
$ ./echosrv 1234 &
[1] 19160
$ netstat -na | grep tcp
tcp        0      0 0.0.0.0:1234            0.0.0.0:*
LISTEN
...
$ telnet localhost 1234
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
connected from 127.0.0.1:53930
abcde
abcde
12345
12345
hello
hello
^]
telnet> quit
Connection closed.
disconnected from 127.0.0.1:53930
```

Socket Options

- `setsockopt(2)`: `int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen);`
- `getsockopt(2)`: `int getsockopt(int sockfd, int level, int optname, void *optval, socklen_t *optlen);`
  - return: 0 OK, -1 error
  - `level`
    - identify the protocol to apply
      - e.g. `IPPROTO_IP`, `IPPROTO_TCP`
    - if the option is a generic socket-level option
      - then it 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(2)` 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(2)` 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
`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(2)` only)