Lecture 21. Network IPC: Sockets

The IPC methods of communication studied (pipes, FIFOs, message queues, semaphores, shared memory) allow processes running on the same computer to communicate with each other. This lecture covers methods that allow processes running on different computers to communicate with each other.

The **socket network IPC interface** allows processes to communicate with each other on the same machine or on different machines. The socket IPC network interface can be used to communicate using many different network protocols, but we will limit the discussion to the TCP/IP protocol suite.

Socket Descriptors

A socket is an abstraction of a communication endpoint. Applications use sockets to communicate, just like they use file descriptors. In fact, **socket descriptors** are implemented as file descriptors under UNIX. Many of the functions available to file descriptors (`read()`, `write()` etc.) are available to sockets.

A socket can be created with the `socket()` function:

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

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

Returns file (socket) descriptor if OK, -1 if error.

Arguments:
`domain`—determines the nature of the communication, including the address format. It is one of the domains shown in Table 1
`type`—the type of the socket, which determines the communication characteristics. The types available are shown in Table 2
`protocol`—usually 0, to select the default protocol for the given domain and type. When multiple protocols are supported for the same domain/type, the protocol argument selects a particular protocol.

A **datagram** is a connectionless service, which needs no logical connection between the peers that communicate. A datagram message is analogous to a letter,

in that it contains the address of the peer it is destined to, but the order and reliability of the service are not guaranteed. A datagram service is specified by:
SOCK_DGRAM—socket type (2nd argument to `socket()`);
SOCK_RAW—datagram interface directly to the underlying network layer (e.g. TCP or UDP transport layers are bypassed). Superuser privileges are required to create a raw socket.

A **connection-oriented protocol** requires a logical connection between the communicating peers. It is analogous to a phone call, in that once the connection is in place, a bidirectional conversation can take place, and the connection is a peer-to-peer communication channel. The messages do not, therefore, contain addressing information. A connection-oriented protocol is specified by one of the following socket types:
SOCK_STREAM—byte stream service. Reading data from a SOCK_STREAM socket does not guarantee the delivery of all the bytes written by the sender. The retrieval of all these bytes may require several system calls;
SOCK_SEQPACKET—message-based service. The amount of data read from such a socket is the same as the amount of data sent. The Stream Control Transmission Protocol (SCTP) provides a sequential packet service in the Internet domain.

Table 1. Communication Domains

Domain	Description
`AF_INET`	IPv4 Internet domain
`AF_INET6`	IPv6 Internet domain
`AF_UNIX`	UNIX domain
`AF_UNSPEC`	unspecified

Table 2. Socket Types

Type	Description
`SOCK_DGRAM`	fixed-length, connectionless, unreliable messages
`SOCK_RAW`	datagram interface to IP (optional in POSIX.1)
`SOCK_SEQPACKET`	fixed-length, sequenced, reliable, connection-oriented messages
`SOCK_STREAM`	sequenced, reliable, bidirectional, connection-oriented byte streams

As mentioned earlier, the system calls available to file descriptors are available to sockets. The following table summarizes all the functions available to sockets.

Function	Behavior with socket
`close` (Section 3.3)	deallocates the socket
`dup`, `dup2` (Section 3.12)	duplicates the file descriptor as normal
`fchdir` (Section 4.22)	fails with `errno` set to `ENOTDIR`
`fchmod` (Section 4.9)	unspecified
`fchown` (Section 4.11)	implementation defined
`fcntl` (Section 3.14)	some commands supported, including `F_DUPFD`, `F_GETFD`, `F_GETFL`, `F_GETOWN`, `F_SETFD`, `F_SETFL`, and `F_SETOWN`
`fdatasync`, `fsync` (Section 3.13)	implementation defined
`fstat` (Section 4.2)	some `stat` structure members supported, but how left up to the implementation
`ftruncate` (Section 4.13)	unspecified
`getmsg`, `getpmsg` (Section 14.4)	works if sockets are implemented with STREAMS (i.e., on Solaris)
`ioctl` (Section 3.15)	some commands work, depending on underlying device driver
`lseek` (Section 3.6)	implementation defined (usually fails with `errno` set to `ESPIPE`)
`mmap` (Section 14.9)	unspecified
`poll` (Section 14.5.2)	works as expected
`putmsg`, `putpmsg` (Section 14.4)	works if sockets are implemented with STREAMS (i.e., on Solaris)
`read` (Section 3.7) and `readv` (Section 14.7)	equivalent to `recv` (Section 16.5) without any flags
`select` (Section 14.5.1)	works as expected
`write` (Section 3.8) and `writev` (Section 14.7)	equivalent to `send` (Section 16.5) without any flags

Communication on a socket is bidirectional. I/O to a socket can be disabled with the `shutdown()` function:

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

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

Returns 0 if OK, -1 if error.

Arguments:
`sockfd`—file (socket) descriptor, returned by `socket()`
`how`—may have one of the following values:

- SHUT_RD—reading from the socket is disabled
- SHUT_WR—writing to the socket is disabled
- SHUT_RDWR—disables both

A socket is closed with the `close()` function. Why are both `shutdown()` and `close()` needed? `shutdown()` frees the file (socket) descriptor explicitly, disabling all remaining references to the file (socket) descriptor, whereas `close()` deallocates the network endpoint only when the last active reference is closed.

The following short program creates a socket:

```
#include <sys/socket.h> /* for 1st, 2nd arguments */
#include <netinet/in.h> /* for 3rd argument */
#include <stdio.h> /* for printf() */
#include <errno.h> /* for errno */
#include <string.h> /* for strerror() */

int main(void)
{
     int sockfd;

     sockfd=socket(AF_INET6, SOCK_RAW, IPPROTO_IPV6);
     printf("Created socket with sockfd=%d.\n", sockfd);
     printf("errno=%d: %s\n", errno, strerror(errno));
return 0;
}
```

The output shows the *`sockfd`* created, and that the operation is only permitted with superuser privileges:

```
$ sudo ./mysocket
Password:
Created socket with sockfd=3.
errno=0: Undefined error: 0
```

Addressing

So far, we studied how to create and destroy a socket. In this section, we study how to identify the process with which the communication through sockets takes place. The **network address** of the remote computer is used to identify the peer host. The **port number**, representing a service, is used to identify the peer process.

Byte Ordering

The two types of byte ordering are:
big endian—the highest byte address occurs in the least significant byte (highest is rightmost, number grows right to left, contrary to the normal way of representing binary numbers)
little endian— the lowest byte address occurs in the least significant byte (highest is leftmost, number grows left to right, the normal way of representing numbers)

For example, a 32-bit integer with the hexadecimal value `0x04030201` will be read differently on machines with different byte ordering types:
on the big endian system: ch[0]=4 (leftmost non-zero digit)
on the little endian system: ch[0]=1 (rightmost non-zero digit)

The byte orderings for the four most popular platforms are:

Operating system	Processor architecture	Byte order
FreeBSD 5.2.1	Intel Pentium	little-endian
Linux 2.4.22	Intel Pentium	little-endian
Mac OS X 10.3	PowerPC	big-endian
Solaris 9	Sun SPARC	big-endian

In order for different computers running different platforms to communicate, the specific byte orderings on each system must be translated into the byte ordering used by the protocol. For instance, TCP/IP uses big endian byte ordering. The following functions provide byte ordering translation:

```
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostint32);
                                    Returns 32-bit integer in network byte order.
uint16_t htons(uint16_t hostint16);
                                    Returns 16-bit integer in network byte order.
uint32_t ntohl(uint32_t netint32);
                                    Returns 32-bit integer in host byte order.
uint16_t ntohs(uint16_t netint16);
                                    Returns 16-bit integer in host byte order.
```

`htonl()`—host to network long
`htons()`—host to network short
`ntohl()`—network to host long
`ntohs()`—network to host short

The following example shows the little endian-to-big endian conversion:

$$n = 1234 = 1024 + 128 + 64 + 16 + 2 = 2^{10} + 2^{7} + 2^{6} + 2^{4} + 2^{1} =$$
$$= \underbrace{0000\,0100}_{\text{byte 1}}\underbrace{1101\,0010}_{\text{byte 0}} = 0x04D2$$

If n is in little endian byte order, meaning the least significant byte is the lowest address byte, to convert it into big endian, the byte ordering must be reversed:

$$m = \underbrace{1101\,0010}_{\text{byte 0}}\underbrace{0000\,0100}_{\text{byte 1}} = 2^{15} + 2^{14} + 2^{12} + 2^{9} + 2^{2} = 53{,}764$$

The following program illustrates the use of the host-to-network short integer conversion function `htons()`:

```
#include <arpa/inet.h>
#include <stdio.h>

int main(void)
{
     uint32_t n=1234;

     uint32_t m=htonl(n);
     printf("In network byte order: n=%d\n", m);
     uint16_t p=htons(n);
     printf("In network byte order: n=%d\n", p);
return 0;
}
```

The output shows that we obtain the same value as above only for a 16-bit integer:

```
$ ./mybyteorder
In network byte order: n=-771489792
In network byte order: n=53764
```

Address Formats

An **address** identifies the destination socket in a communication domain (one of the four). The address format is specific to the domain. For this reason, in order for addresses with different formats to be passed to socket functions, the differently formatted addresses are stored in a generic `sockaddr` structure, specifying the domain, as opposed to just a number. For BSD, this is:

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

In the IPv4 Internet domain (AF_INET) a socket address is represented by the `sockaddr_in` structure:

```
struct sockaddr_in
{
     sa_family_t    sin_family;    /* address family */
     in_port_t      sin_port;      /* port number */
     struct in_addr sin_addr;      /* IPv4 address */
};
```

where the `in_addr` structure is:

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

In the IPv6 Internet domain (AF_INET6) a socket address is represented by a `sockaddr_in6` structure:

```
struct sockaddr_in6
{
     sa_family_t         sin6_family;   /* address family */
     in_port_t           sin6_port;     /* port number */
     uint32_t            sin6_flowinfo; /* traffic class and
flow info */
     struct in6_addr     sin6_addr;     /* IPv6 address */
```

```
     uint32_t              sin6_scope_id; /* set of interfaces for
scope */
};
```

where the `in6_addr` structure is:

```
struct in6_addr
{
     uint8_t               s6_addr[16];   /* IPv6 address */
};
```

Although `sockaddr_in` and `sockaddr_in6` are different, they can be passed to the same socket functions as `sockaddr` argument types. Later, we will study the AF_UNIX UNIX domain sockets, with their own structures.

The following two functions convert IPv4/IPv6 between binary and string dot notation formats, in network byte order:

```
#include <arpa/inet.h>

const char *inet_ntop(int domain, const void *restrict addr,
                         char *restrict str, socklen_t size);
```

Returns pointer to address if OK, NULL if error.

```
int inet_pton(int domain, const char *restrict str,
                void *restrict addr);
```

Returns 1 if OK, 0 if format is invalid, -1 if error.

Arguments:
domain—one of the AF_INET and AF_INET6 values
addr—buffer to hold address, 32 bits for AF_INET, and 128 bits for AF_INET6
str—buffer to hold the text string representing an IPv4/IPv6 address, of sizes INET_ADDRSTRLEN (for AF_INET) and INET6_ADDRSTRLEN (for AF_INET6)

The 4-byte IP address in IPv4 of my MacBook Pro is **71.183.210.141**. Each digit group between the dots represents a byte. Converted into binary, this IP address can be broken into:

$71 = 64 + 4 + 2 + 1 = 2^6 + 2^2 + 2^1 + 2^0 = 0100\,0111$

$183 = 128 + 32 + 16 + 4 + 2 + 1 = 2^7 + 2^5 + 2^4 + 2^2 + 2^1 + 2^0 = 1011\,0111$

$210 = 128 + 64 + 16 + 2 = 2^7 + 2^6 + 2^4 + 2^1 = 1101\,0010$

$141 = 128 + 8 + 4 + 1 = 2^7 + 2^3 + 2^2 + 2^0 = 1000\,1101$

This is in network byte order, meaning the highest order byte is the leftmost byte (little endian). Converted into an integer, this address is, still in network byte order:

$$\underbrace{0100\,0111}_{\text{byte 0}}\underbrace{1011\,0111}_{\text{byte 1}}\underbrace{1101\,0010}_{\text{byte 2}}\underbrace{1000\,1101}_{\text{byte 3}} =$$

$$= \underbrace{2^{31} + 2^{26} + 2^{25} + 2^{24}}_{1\,191\,182\,336} + \underbrace{2^{23} + 2^{21} + 2^{20} + 2^{18} + 2^{17} + 2^{16}}_{11\,993\,088} +$$

$$+ \underbrace{2^{15} + 2^{14} + 2^{12} + 2^{9}}_{53\,760} + \underbrace{2^{7} + 2^{3} + 2^{2} + 2^{0}}_{141} = 1\,203\,229\,325$$

This number is what the function `inet_pton()` (peer-to-network), when called with my IP address, 71.183.210.141, returns into the `addr.sin_addr` member of the `sockaddr_in` structure. The function `ntohl()` (network-to-host long) will convert that into `uint32_t` number. The following program illustrates this fact:

```
#include<stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <string.h>
#include <errno.h>

int main(int argc, char *argv[])
{
     struct in_addr addr;
     char *byte_order=malloc(sizeof(argv[1]));

     if(argc<2)
     {
          fprintf(stderr, "usage: ./myIPaddress4 [ip_address]
\n");
          return -1;
     }

     inet_pton(AF_INET, argv[1], &addr);
```

```
     fprintf(stdout, "Network byte order: %d\n”,
ntohl(addr.s_addr));
     long back=htonl(ntohl(addr.s_addr));
     fprintf(stdout, "Host byte order: %ld\n", back);

     inet_ntop(AF_INET, &addr.s_addr, byte_order,
sizeof(argv[1])*2);
     fprintf(stdout, "Dot quad: %s\n", byte_order);
     free(byte_order);
return 0;
}
```

The output shows the conversion of the quad dot IP peer address into network byte ordered integer, and back:

```
$ ./myIPaddress4 71.183.210.141
Network byte order: 1203229325
Host byte order: 2379396935
Dot quad: 71.183.210.141
```

The network byte order is the same as previously calculated by hand. The conversion into host byte order, explained on an earlier example, is done by reversing the byte order of the 4 bytes from the network byte ordered integer.

Address Lookup

Having seen the address structures for IPv4 and IPv6, we would like to write applications that are unaware of the internals of each of these structures. Instead, we want do pass socket addresses as `sockaddr` structures. These will work with multiple protocols that provide the same type of service.

The network configuration information returned by the functions that interface with the network is either kept in static files (/etc/hosts, /etc/services etc.), or can be managed by a name service (DNS—Domain Name System, or NIS—Network Information Service).

The hosts known by a computer system are returned by `gethostent()`:

```
#include <netdb.h>

struct hostent *gethostent(void);
```

Returns pointer if OK, NULL if error.

```
void sethostent(int stayopen);

void endhostent(void);
```

`gethostent()` opens the **host database file**, and returns the next entry in the file. The `sethostent()` function opens the file or rewinds it if already open. `endhostent()` closes the file.
`gethostent()` returns a pointer to a structure `hostent`:

```
struct hostent
{
     char      *h_name;        /* name of host */
     char      **h_aliases;    /* pointer to alternate host name 
array */
     int       h_addrtype;     /* address type */
     int       h_length;       /* length in bytes of the address 
*/
     char      **h_addr_list; /* pointer to array of network 
addresses */
     …
};
```

The addresses returned are in network byte order.

The following program illustrates the contents of this structure for my MacBook Pro:

```
#include <netdb.h>   /* for gethostent() */
#include <stdio.h>   /* for printf() */
#include <stdlib.h> /* for malloc() */
#include <arpa/inet.h>

int main(void)
{
     struct hostent *host_ptr=malloc(sizeof(struct hostent));

     host_ptr=gethostent();
     printf("Name of host: %s.\n", host_ptr->h_name);

     char **aliases_ptr;
     for (aliases_ptr=host_ptr->h_aliases; *aliases_ptr!=NULL; 
aliases_ptr++)
     {
```

```
          printf("Alias: %s.\n", *aliases_ptr);
     }

     switch(host_ptr->h_addrtype)
     {
     case 0:
          printf("Address type: %d: %s.\n",
                    host_ptr->h_addrtype, "AF_UNSPEC");
          break;
     case 1:
          printf("Address type: %d: %s.\n",
                    host_ptr->h_addrtype, "AF_UNIX");
          break;
     case 2:
          printf("Address type: %d: %s.\n",
                    host_ptr->h_addrtype, "AF_INET");
          break;
     default:
          break;
     }

     printf("Address length: %d bytes.\n", host_ptr->h_length);

     char **address_list_ptr;
     for (address_list_ptr=host_ptr->h_addr_list;
               *address_list_ptr!=NULL; address_list_ptr++)
     {
          printf("Address list: %s.\n", inet_ntoa((*(struct
in_addr *)*address_list_ptr)));
     }
return 0;
}
```

The output is:

```
$ ./mygethostent
Name of host: localhost.
Address type: 2: AF_INET.
Address length: 4 bytes.
Address list: 127.0.0.1.
```

Network names and numbers can be obtained with the following functions:

```
#include <netdb.h>
```

```
struct netent *getnetbyaddr(uint32_t net, int type);

struct netent *getnetbyname(const char *name);

struct netent *getnetent(void);

void setnetent(int stayopen);

void endnetent(void);
```

All return pointer if OK, NULL if error.

…where the `netent` structure has the following fields:

```
struct netent
{
     char       *n_name;        /* network name */
     char       **n_aliases;    /* alternate network name array 
pointer */
     int        n_addrtype;     /* address type */
     uint32_t   n_net;          /* network number */
     …
};
```

The following program shows the use of `getnetent()` to return a `netent` struct and its members:

```
#include <netdb.h> /* for getnetbyaddr() */
#include <stdio.h> /* for printf() */
#include <stdlib.h> /* for malloc() */

struct netent *netentry;

int main(void)
{
     netentry=malloc(sizeof(struct netent));
     //netentry=getnetbyaddr(2379396935, AF_INET);
     netentry=getnetent();
     printf("netentry->n_net=%d\n", netentry->n_net);
     printf("netentry->n_addrtype=%d\n", netentry->n_addrtype);
     printf("netentry->n_name=%s\n", netentry->n_name);
     char **ptr;
     int i=0;
     for (ptr=&netentry->n_aliases[0]; *ptr!=NULL; ptr++)
     {
          printf("netentry->n_aliases[%d]=%s\n", i, *ptr);
```

```
            i++;
        }
return 0;
}
```

The output is:

```
$ ./mynetent
netentry->n_net=127
netentry->n_addrtype=2
netentry->n_name=loopback
netentry->n_aliases[0]=loopback-net
```

The **loopback** device is a virtual network interface used for a computer to communicate with itself, for purposes of diagnostics and troubleshooting, but especially to connect to servers running on the local host.

The following functions map between protocol names and numbers:

```
#include <netdb.h>

struct protoent *getprotobyname(const char *name);

struct protoent *getprotobynumber(int proto);

struct protoent *getprotoent(void);
```

All return pointer to struct if OK, NULL if error.

```
void setprotoent(int stayopen);

void endprotoent(void);
```

The `protoent` structure has the following entries:

```
struct protoent
{
     char *p_name;        /* protocol name */
     char **p_aliases;    /* pointer to alternate protocol name
array */
     int p_proto;         /* protocol number */
     …
};
```

The following program, gracefully borrowed from `stackoverflow`, shows the use of `getprotoent()` to retrieve members of the `protoent` structure:

```
#include <stdio.h>
#include <netdb.h>

int main (int argc, char *argv[])
{
     int i;
     struct protoent *proto=getprotobyname("ipv6");
     if (proto!=NULL)
     {
          printf("Official name: %s\n", proto->p_name);
          printf("Port#: %d\n", proto->p_proto);
          for (i = 0; proto->p_aliases[i] != 0; i++)
          {
               printf("Alias[%d]: %s\n", i+1, proto-
>p_aliases[i]);
          }
     }
     else
          perror("protocol not found");
return 0;
}
```

The output is:

```
$ ./myprotoent
Official name: ipv6
Port#: 41
Alias[1]: IPV6
```

Services are represented by the port number portion of the address. In struct `sockaddr_in`, the member `in_port_t sin_addr` represents the port. A service name can be mapped to a port number by calling `getservbyname()`, and vice-versa by calling `getservbyport()`. Alternately, the services database can be scanned sequentially with `getservent()`:

```
#include <netdb.h>

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

struct servent *getservbyport(int port, const char *proto);
```

```
struct servent *getservent(void);
void setservent(int stayopen);
void endservent(void);
```

All three return pointer to struct if OK, NULL if error.

The `servent` structure has the following members:

```
struct servent
{
     char *s_name;         /* service name */
     char **s_aliases;     /* pointer to alternate service name
array */
     int s_port;           /* port number */
     char *s_proto;        /* name of protocol */
     …
};
```

The following program uses `getservent()` to scan all services on this host and print out the members of `servent`:

```
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>

struct servent *serventry;

int main(void)
{
     serventry=malloc(sizeof(struct servent));
     for (;;)
     {
          printf("—————————————————————————————————————\n");
          serventry=getservent();
          printf("serventry->s_port=%d\n", serventry->s_port);
          printf("serventry->s_name=%s\n", serventry->s_name);
          printf("serventry->s_proto=%s\n", serventry->s_proto);
          for (int i=0; serventry->s_aliases[i]!=0; i++)
          {
               printf("serventry->s_aliases[%d]=%s\n", i,
serventry->s_aliases[i]);
          }
          printf("—————————————————————————————————————\n");
```

```
     }
return 0;
}
```

The first few lines of the output look like this:

```
-------------------------------------------
serventry->s_port=256
serventry->s_name=rtmp
serventry->s_proto=ddp
-------------------------------------------
-------------------------------------------
serventry->s_port=256
serventry->s_name=tcpmux
serventry->s_proto=udp
-------------------------------------------
-------------------------------------------
serventry->s_port=256
serventry->s_name=tcpmux
serventry->s_proto=tcp
-------------------------------------------
-------------------------------------------
serventry->s_port=512
serventry->s_name=nbp
serventry->s_proto=ddp
-------------------------------------------
```