

Network Programming

Application Programming Interface: API

Network programming at different levels:

- send Ethernet, ATM, ... packets
- exchange UDP/TCP (and raw IP) packets
- RPC, transactions, Xlib, Corba, ...

Goals:

- efficient (avoid copies)
- change protocols with minimal program changes – at link time, at run time

Socket API

- introduced in 1981 by BSD 4.1
- originally only Unix \rightsquigarrow WinSock (“almost” same)
- connects *sockets* on two hosts
- data transfer phase: like Unix file descriptors; but: no open, notion of permissions, seek, ...
- implemented as library (SVR4) or system calls (BSD)
- mainly, two services: datagram and stream communications
- also provides buffering
- alternatives: TLI (SVR4)



foKus

Review: Unix file I/O

file descriptor: small, positive integer

```
int fd;
```

```
char buf[256];
```

```
fd = open("a.txt", O_RDWR | O_CREAT, S_IRUSR);
```

```
write(fd, buf, sizeof(buf));
```

```
close(fd)
```

\rightsquigarrow roughly same for sockets, except for setup



foKus

Creating a socket

- like filedescriptor (type int)
- domain, type and protocol:

domain: PF_INET, PF_UNIX, ...

	datagram	SOCK_DGRAM	UDP
type:	reliable, sequenced stream	SOCK_STREAM	TCP
	raw	SOCK_RAW	IP

protocol: usually 0 = default for type

```
socket(family, type, protocol);
#include <sys/types.h>
#include <sys/socket.h>
if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket");
}
```



Endpoint addresses

Each socket association has five components:
(protocol, local address, local port, remote address, remote port)

protocol: socket

local address, port: bind

remote address, port: connect, sendto



Address conversion

internally: 32-bit integers

host name to numeric: `gethostbyname()`

dotted quad to numeric: `inet_addr()`

numeric to dotted quad: `inet_ntoa()`

```
struct sin_addr {
    u_long s_addr;           /* 32-bit host address */
}
struct sockaddr_in {
    short sin_family;       /* AF_INET */
    u_short sin_port;      /* network byte order */
    struct sin_addr sin_addr; /* network address */
}
```

Data representation

- integers:
 - little endian least significant byte first DEC, Intel
 - big endian most significant byte first Sun, SGI, HP, ...
- *network byte order* = big endian
- `m = ntohl(m)`: network-to-host byte order, 32 bit
- `m = htonl(m)`: host-to-network byte order
- `ntohs, htons`: short (16 bit)
- other problems:
 - characters (ASCII, 8859-x, multi-byte)
 - floating point, structured data, ...
 - \Rightarrow XDR, ASN.1 (canonical representation)

Binding the local address

```
int bind(int s, struct sockaddr *addr, int addresslen);
```

- only one socket per port and protocol (almost)
- optional for clients \Rightarrow system chooses random local port

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int s;
struct sockaddr_in sin;

if ((s = socket(PF_INET, SOCK_DGRAM, 0)) < 0) { /* error */ }
memset((char *)&sin, 0, sizeof(sin));
sin.sin_family      = AF_INET;
sin.sin_port        = htons(6000); /* 0: let the system choose */
sin.sin_addr.s_addr = INADDR_ANY; /* allow any interface */
if (bind(s, (struct sockaddr *)&sin, sizeof(sin)) < 0) { /* */ }
```



foKus

Sending and receiving data

call	device	buffering	opt	peer?
read(), write()	any	single	N	N
readv(), writev()	any	s/g	N	N
recv(), send()	socket	single	Y	N
recvfrom(), sendto()	socket	single	Y	Y
recvmsg(), sendmsg()	socket	s/g	Y	Y

- block size read (may) \neq block size written
- read from stream: may read fewer bytes than requested
- scatter/gather: read/write from/to several buffers at once



foKus

Datagram communications: sendto()

```
int sendto(int s, char *msg, int len, int flags,  
           struct sockaddr *to, int tolen);
```

- sends *len* bytes from buffer *buf*
- to location *to*, with options *flags* (usu. 0)
- successful return ↗ delivery



Datagram communication: recvfrom()

```
int recvfrom(int s, char *buf, int len, int flags,  
             struct sockaddr *from, int *fromlen);
```

- receives **up to** *len* bytes into *buf*
- sets *from* to source address of data
- sets *fromlen* to valid length of *from*
- returns number of bytes received, 0 on EOF



Scatter/gather I/O

Allows to write/read to/from several buffers in one call \Rightarrow write header and data, etc.

```
struct iovec {
    caddr_t iov_base; /* base address of buffer */
    int     iov_len;  /* length of buffer */
}
int writev(int s, struct iovec iov[], int iovcnt);
int readv(int s, struct iovec iov[], int iovcnt);
```

- writes *iovcnt* buffers from array *iov*
- returns total number of bytes read/written
- on reading, may not fill all buffers



foKus

Client/server model

- common, but not the only use of sockets ...
- server process (“daemon”) waits for connection requests
- file server, time server, X server, ...
- asymmetric, but data exchange in both directions
- single process can be both client and server
- two modes:
 - sequential: one connection at a time
 - parallel: each connection forks a new server
- location usually through well-known port number \Rightarrow one per host
- somehow need to find host that offers service



foKus

Connecting to a server: `connect()`

```
int connect(int s, struct sockaddr *name, int namelen);
```

- client issues `connect()` to
 - establish remote address, port (stream, datagram sockets)
 - establish connection (stream socket)
- fails: host not listening to port, timeout

Establish connection queue: `listen()`

```
int listen(int s, int backlog)
```

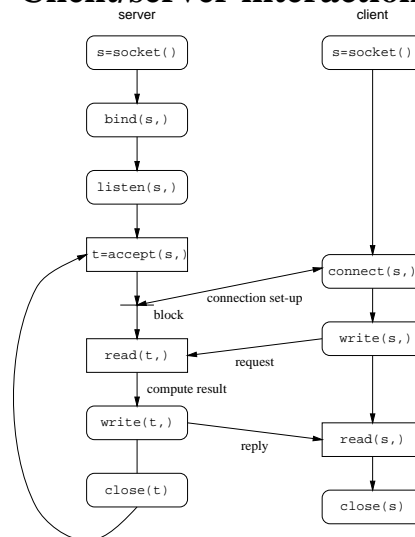
- only used for stream sockets
- socket being listened to can't be used for client
- does **not** wait for connections, but needed
- allows *backlog* pending connection requests while waiting for `accept()`

Accepting connections: `accept()`

```
int accept(int s, struct sockaddr *addr, int *addrlen);
```

- by connection-oriented server, after `listen()`
- can't "preview" connection \Rightarrow accept and close
- returns a **new** socket for data exchange with client
- optionally, returns address of client

Client/server interaction



Stream client

```

/*
 * client host port
 */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

extern struct in_addr host2ip(char *host);

int main(int argc, char *argv[])
{
    int s;
    struct sockaddr_in sin;
    char msg[80] = "Hello World!";
    int n;

    if (argc < 3) { printf("%s host port\n", argv[0]); return -1; }

```



foKus

```

    if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket"); return -1;
    }
    sin.sin_family = AF_INET;
    sin.sin_port = htons(atoi(argv[2]));
    sin.sin_addr = host2ip(argv[1]);
    if (connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
        perror("connect"); return -1;
    }
    if (write(s, msg, strlen(msg)+1) < 0) {
        perror("write"); return -1;
    }
    if ((n = read(s, msg, sizeof(msg))) < 0) {
        perror("read"); return -1;
    }
    printf("%d bytes: %s\n", n, msg);
    if (close(s) < 0) { perror("close"); return -1; }
    return 0;
}

```



foKus

Stream server: echo

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

int main(int argc, char *argv[])
{
    int s, t;
    struct sockaddr_in sin;
    char msg[80];
    int sinlen;

    if (argc < 2) { printf("%s port\n", argv[0]); return -1; }

    if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket"); return -1;
    }
}
```



```
sin.sin_family      = AF_INET;
sin.sin_port        = htons(atoi(argv[1]));
sin.sin_addr.s_addr = INADDR_ANY;
if (bind(s, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
    perror("bind"); return -1;
}
if (listen(s, 5) < 0) { perror("listen"); return -1; }
sinlen = sizeof(sin);
if ((t = accept(s, (struct sockaddr *)&sin, &sinlen)) < 0) {
    perror("accept"); return -1;
}
printf("From %s:%d.\n",
       inet_ntoa(sin.sin_addr), ntohs(sin.sin_port));
if (read(t, msg, sizeof(msg)) < 0) {perror("read"); return -1;}
if (write(t, msg, strlen(msg)) < 0) {perror("write"); return -1;}
if (close(t) < 0) { perror("close"); return -1; }
if (close(s) < 0) { perror("close"); return -1; }
return 0;
}
```



Event-based programs

- `read()` is blocking \Rightarrow server only works with single socket
- need I/O multiplexing \Rightarrow event-based programming
- also need to handle time-outs, connection requests
- all events (mouse clicks, windows, etc.) handled by event loop
- **do**
 wait for event(s)
 handle event (hopefully short)
forever
- harder to maintain state, recursion
- alternative 1: “interrupts” (signals) \Rightarrow called at any time
- alternative 2: threads (separate scheduling, same address space)

Multiplexing with `select()`

```
int select(int nfd, fd_set *readfds, fd_set *writefds,
          fd_set *exceptfds, struct timeval *timeout)
```

- block until ≥ 1 file descriptors have something to be read, written, or an exception, or timeout
- set bit mask for descriptors to watch using `FD_SET`
- returns with bits for ready descriptors set \Rightarrow check with `FD_ISSET`
- cannot specify amount of data ready

select() server example

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

int main(int argc, char *argv[])
{
    int s = -1, t = -1;
    int n;
    struct sockaddr_in sin;
    char msg[80];
    fd_set rfds;
    int sinlen;
    struct timeval tv;

    if (argc < 2) { printf("%s port\n", argv[0]); return -1; }
```



```
if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket"); return -1;
}
sin.sin_family      = AF_INET;
sin.sin_port        = htons(atoi(argv[1]));
sin.sin_addr.s_addr = INADDR_ANY;
if (bind(s, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
    perror("bind"); return -1;
}
if (listen(s, 5) < 0) { perror("listen"); return -1; }
tv.tv_sec = 10;
tv.tv_usec = 0;
while (1) {
    FD_ZERO(&rfds);
    if (s > 0) FD_SET(s, &rfds);
    if (t > 0) FD_SET(t, &rfds);
    n = select(64, &rfds, NULL, NULL, &tv);
    if (n == 0) { printf("Timeout!\n"); }
    else if (n > 0) {
        if (s > 0 && FD_ISSET(s, &rfds)) {
            sinlen = sizeof(sin);
            if ((t = accept(s, (struct sockaddr *)&sin, &sinlen)) < 0) {
                perror("accept"); return -1;
            }
        }
    }
}
```



```

    }
    printf("From %s:%d.\n",
          inet_ntoa(sin.sin_addr), ntohs(sin.sin_port));
}
if (t > 0 && FD_ISSET(t, &rfdset)) {
    if ((n=read(t,msg,sizeof(msg))) < 0) {perror("read"); return -1;}
    if (n == 0) { close(t); t = -1; }
    else {
        if (write(t,msg,n)<0) {perror("write");return -1;}
    }
}
}
}
if (close(t) < 0) { perror("close"); return -1; }
if (close(s) < 0) { perror("close"); return -1; }
return 0;
}

```

inetd

- server processes (“daemons”): routed, snmpd, in.rlogind, ftpd, mountd, telnetd, ...
- each service, one process → lots of idle processes, listening sockets
- inetd: Internet services daemon
- opens sockets, forks processes when request arrives
- stdin/stdout connected to network
- but: process creation overhead

Evaluation

- not *quite* the same as files \Rightarrow `open("foo.com:3456",...)?`
- fixed-length address structure for IPv6?
- asymmetric \Rightarrow client, server
- cannot easily pass QOS parameters (use `ioctl`)
- only single address, no point-to-multipoint
- different IPC (Unix sockets, pipes, ...) for local communication \Rightarrow no distribution transparency