Client-Server Communication Paradigm

Client Application

Host With TCP/IP

LAN

WAN

Server Application

Host With TCP/IP

LAN

router

router

router

router

router
Application Interface to Protocols
Network Programming

- Network applications use TCP/IP to communicate with each other.
- TCP/IP protocol software resides in the computer's operating system.
- An application program interacts with the operating system to request service.
TCP/IP Network Programming

APIs

- In practice, a few APIs exist.
- Berkeley UNIX Sockets
  - Initiated by ARPA (Advanced Research Project Agency) in early 1980s
  - By the University of California, Berkeley
  - Included in release 4.1 of the Berkeley Software Distribution (bsd)
  - Has been adopted by many systems, including Linux (a de facto standard)
  - Known as socket API (or socket interface, sockets.)
- Microsoft Windows Sockets
  - A variant of socket API
- AT&T TLI (Transport Layer Interface)
  - for System V UNIX
The Socket Interface
System Calls

- **System calls** are mechanisms that OS uses to transfer control between application and the operating system procedures.
- To a programmer, system calls look and act like function calls.
- When received a system call, OS directs the call to an internal procedure that performs the requested operation.
The basic I/O operations in Linux

- An application program calls `open` to initiate input or output.
- The system returns an integer called a file descriptor that the application uses in further I/O operations
- Three arguments
  - the name of a file or device to open
  - A set of bit flags that controls special cases (e.g., create one if not exists)
  - An access mode (e.g., read, write, etc.)

Example
```c
int desc;
desc = open("filename", O_RDWR, 0);

read (desc, buffer, 128); //bytes

close (desc);
```
Socket API

- Socket API is a set of functions and the parameters that each function requires and the semantics of the operation it performs.
- Follow conventional I/O primitives notation and semantics
  - Use basic I/O functions whenever possible
  - Add additional functions for those operations that cannot be expressed conveniently
- Allow multiple families of protocols
  - E.g., TCP/IP protocol family – PF_INET
The basic I/O operations in Linux

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>open</strong></td>
<td>Prepare a device or a file for input and output operations</td>
</tr>
<tr>
<td><strong>close</strong></td>
<td>Terminate use of a previously opened device or file</td>
</tr>
<tr>
<td><strong>read</strong></td>
<td>Obtain data from an input device or file, and place it in the application program’s memory</td>
</tr>
<tr>
<td><strong>write</strong></td>
<td>Transmit data from the application program’s memory to an output device or file</td>
</tr>
<tr>
<td><strong>lseek</strong></td>
<td>Move to a specific position in a file or device (e.g., disk)</td>
</tr>
<tr>
<td><strong>ioctl</strong></td>
<td>Control a device or the software used to access it (e.g., specify buffer size or change character set mapping)</td>
</tr>
</tbody>
</table>
OS implements file descriptors as an array of pointers to internal data structures.

OS maintains a separate file descriptor table for each process.

Socket descriptor
- When a process opens a socket, OS places a pointer to the internal data structure for that socket.
- It is in the same process’ descriptor table as file descriptors.
- OS returns the table index (i.e. the socket descriptor) to the calling program.
## Summary of Socket Calls (1/2)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>socket</strong></td>
<td>Create a descriptor for use in network communication</td>
</tr>
<tr>
<td><strong>connect</strong></td>
<td>Connect to a remote peer (client)</td>
</tr>
<tr>
<td><strong>send (write)</strong></td>
<td>Send outgoing data across a TCP connection or a UDP datagram</td>
</tr>
<tr>
<td><strong>recv (read)</strong></td>
<td>Acquire incoming data from a TCP connection or the next incoming UDP datagram</td>
</tr>
<tr>
<td><strong>close</strong></td>
<td>Terminate comm. and de-allocate a descriptor</td>
</tr>
<tr>
<td><strong>bind</strong></td>
<td>Bind a local IP addr. and protocol port to a socket</td>
</tr>
<tr>
<td><strong>listen</strong></td>
<td>Place the socket in passive mode and set the # of incoming TCP conn. the system will enqueue (server)</td>
</tr>
<tr>
<td><strong>accept</strong></td>
<td>Accept the next incoming conn. (server)</td>
</tr>
</tbody>
</table>
Summary of Socket Calls (2/2)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>recvmsg</td>
<td>Receive next incoming UDP datagram (variation of recv).</td>
</tr>
<tr>
<td>recvfrom</td>
<td>Receive next incoming UDP datagram and record its source endpoint address.</td>
</tr>
<tr>
<td>sendmsg</td>
<td>Send an outgoing UDP datagram.</td>
</tr>
<tr>
<td>sendto</td>
<td>Send an outgoing UDP datagram, usually to a prerecorded endpoint address.</td>
</tr>
<tr>
<td>shutdown</td>
<td>Terminate a TCP conn. in one or both directions.</td>
</tr>
<tr>
<td>getpeername</td>
<td>After a conn. arrives, obtain the remote machine’s endpoint address from a socket.</td>
</tr>
<tr>
<td>getsockopt</td>
<td>Obtain the current options for a socket.</td>
</tr>
<tr>
<td>setsockopt</td>
<td>Change the options for a socket.</td>
</tr>
</tbody>
</table>
Using Socket Calls in a client-server communication paradigm

**CLIENT SIDE**
- socket
- connect
- send
- recv
- close

**SERVER SIDE**
- socket
- bind
- listen
- accept
- recv
- send
- close

Sequence of calls made by a client and a server using TCP.
System data structures for socket()

- `socket()`: create a new socket
- A new data structure is created by OS to hold the info for communication
- A new descriptor table entry is created to contain a pointer to the data structure.

Operating System

```
data structure for a socket

family: PF_INET
service: SOCK_STREAM
local IP:
remote IP:
local port:
remote port:
```

- family: PF_INET
- service: SOCK_STREAM
- local IP:
- remote IP:
- local port:
- remote port:
Making a socket active or passive

- Server
  - Configure a socket to *wait* for an incoming connection
  - The socket is said to be *passive*

- Client
  - Configure a socket to *initiate* a connection to server
  - The socket is said to be *active*
Predefined symbolic constants and data structures for socket calls

- SOCK_DGRAM, SOCK_STREAM
- #include <sys/types.h>
- #include <sys/socket.h>
Specifying an endpoint address

- Address family
  - e.g., AF_INET (PF_INET)
- Each protocol family defines its own representation of its endpoint address
TCP/IP endpoint address

```c
struct sockaddr_in {
    u_char    sin_len;  /* total length */
    u_short   sin_family; /* type of address: 2-byte */
    u_short   sin_port;  /* protocol port number */
    struct in_addr sin_addr; /* IP address (declared to be */
                            /* u_long on some systems) */
    char      sin_zero[8]; /* unused (set to zero) */
};
```
Algorithms and Issues in Client Software Design
Introduction

Client program

- How to initiate communication?
- Select TCP/IP protocol and address families
- How to use TCP or UDP as the transport layer protocol?
- How to contact a server?
- How to use socket calls to interact with the protocols?
Algorithm: A TCP Client to form a connection to a server for communication

1. Find the IP address and protocol port number of the server with which communication is desired
2. Allocate a socket
3. Specify that the connection needs an arbitrary, unused protocol port on the local machine, and allow TCP to choose one
4. Connect the socket to the server
5. Communication with the server using the application-level protocol (this usually involves sending requests and awaiting replies)
6. Close the connection
Identify the location of a server

- Have server’s domain name or IP address as a constant when the program is compiled
  - inflexible
- Find the server when the program is invoked.
  - As an input argument
- Find the server location info from stable storage (e.g., a file), or
- Use a separate protocol to find a server (e.g., multicast, broadcast, etc)

- Make the client program more general
- Make it possible to change server locations
Look up a domain name

- `socketaddr_in` requires a 32-bit IP address in binary
- Socket APIs that convert a dotted decimal address (1.2.3.4) into a 32-bit IP address in binary
  - `inet_addr`
    - Take an ASCII string address and return the equivalent IP address in binary
  - `gethostbyname`
    - Take an ASCII string address and return the address of a hostent structure
gethostbyname()

```c
struct hostent {
    char    *h_name;    /* official host name   */
    char    **h_aliases; /* other aliases   */
    int     h_addrtype; /* address type */
    int     h_length;  /* address length */
    char    **h_addr_list; /* list of address */
};
#define h_addr h_addr_list[0]  /* for backward compatibility */
```

- *Lists* of host names and aliases (a host may have more than one interface)
struct hostent *hptr;
char    *examplename = “merlin.cs.purdue.edu”;

If ( hptr = gethostbyname(examplename) ) {
    /* IP address is now in hptr->h_addr */
} else {
    /* error in name – handle it */
}

struct hostent {
    char    *h_name;    /* official host name */
    char    **h_aliases; /* other aliases */
    int h_addrtype; /* address type */
    int h_length; /* address length */
    char    **h_addr_list; /* list of address */
}; // #define h_addr h_addr_list[0] /* for backward compatibility
Look up a well-known port by name

- Look up the protocol port for a service
- `getservbyname(string service, string protocol)`

```c
struct servent {
    char    *s_name;       /* official service name */
    char**  s_aliases;    /* other aliases */
    int     s_port;       /* port for this service */
    char    *s_proto;     /* protocol to use */
};
```
Look up a well-known port by name: sample code

```c
struct servent *sptr;
char    *examplename = "merlin.cs.purdue.edu";

If ( sptr = getservbyname("smtp", "tcp") ) {
    /* port number is now in sptr->s_port */
} else {
    /* error in name – handle it */
}
```
Port number and network byte order

- `getservbyname()` returns the protocol port in network byte order
  - It is in the form for use in `sockaddr_in`

- Network byte order vs. byte order in local machine(!)
Look up a protocol by name

- A protocol name is mapped to an integer constant
- `getprotobyname()`

```
struct protoent {
    char    *p_name;    /* official protocol name */
    char   **p_aliases; /* list of aliases allowed */
    int    *p_proto;    /* official protocol number */
};
```
Look up a protocol by name:
sample code

```c
struct protoent *pptr;

If ( pptr = getprotobynamed(“udp”) ) {
    /* official protocol number is now in pptr->p_proto */
} else {
    /* error in name – handle it */
}
```
Algorithm: A TCP Client to form a connection to a server for communication

1. Find the IP address and protocol port number of the server with which communication is desired
2. Allocate a socket
3. Choose an arbitrary, unused protocol port on the local machine
4. Connect the socket to the server
5. Communication with the server using the application-level protocol (this usually involves sending requests and awaiting replies)
6. Close the connection
2. Allocate a Socket

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

int s;      /* socket descriptor*/
s = socket( PF_INET, SOCK_STREAM, 0);
```
3. Choose a local protocol port

- The socket call allows an application to leave the local IP address unfilled.
- TCP/IP software will choose a local one automatically at the time the client connects to a server.
4. Connect the socket to the server

- connect()
  - Allow a client to initiate a connection
  - Return value: 0: success; 1: failure

```c
retcode = connect(sd, remaddr, remaddrlen);
```

- `sd`: socket descriptor
- `remaddr`: remote endpoint of the connection of type `sockaddr_in`
- `remaddrlen`: in bytes
connect()

Performs four tasks

- Test to ensure the specified socket is valid and has not been connected.
- Fills in the remote endpoint address in the socket
- Choose a local endpoint address for the connection if not having one
- Initiate a connection and return a value whether succeeded or not
5. Communicating with server using **TCP**: sample code

```c
#define BLEN 120  /* buffer length to use */
char   *req = "request for some port";
char   buf[BLEN];   /* buffer for answer */
char   *bptr;   /* pointer to buffer */
int    n;   /* number of bytes read */
int    buflen;   /* space left in buffer */

bptr = buf;
buflen = BLEN;

send( sd, req, strlen(req), 0 );   /* Send request */

/* read response (may come in many pieces) */
while ( ( n = recv(sd, bptr, buflen, 0) ) > 0 ) {
    bptr += n;
    buflen -= n;
}
```
Receiving a response from a TCP connection

- TCP is stream-oriented
  - deliver the sequence of bytes that the sender transmits;
  - Do not guarantee to deliver to receiver in the same grouping as they were sent.
- TCP may choose to accumulate many bytes in its output buffer before sending a segment.
6. Closing a TCP connection

- close()
  - Terminate the connection gracefully and deallocate the socket.

- TCP is a two-way communication
  - Terminate a connection requires coordination among the client and the server

- Partial close - shut down a TCP connection in one direction
  - `errcode = shutdown(sd, direction)`
  - direction: an integer (0: no further input is allowed; 1: no further output is allowd; 2: shutdown in both directions)
  - Client finishes sending may use `shutdown()`
  - The server receives an end-of-file signal
  - After sending the last response, it can close the connection.
Algorithm: A UDP Client to form a connection to a server for communication

1. Find the IP address and protocol port number of the server with which communication is desired
2. Allocate a socket
3. Choose an arbitrary, unused protocol port on the local machine, or allow client to choose one
4. Specify the server to which message must be sent
5. Communication with the server using the application-level protocol (this usually involves sending requests and awaiting replies)
6. Close the connection
A client application can use a UDP socket in one of two basic modes:

- **Connected and unconnected.**

**Connected mode**
- The client calls `connect()` to specify a remote endpoint address
- Client can send and receive messages without specifying the remote address repeatedly.
- Suitable for client app that interacts with only one server at a time

**Unconnected mode**
- Does not connect the socket to a specific remote endpoint
- It specifies the remote destination each time it sends a message
- Suitable for interacting with multiple servers
Communicating with a server with UDP

- Connect() with SOCK_DGRAM
  - Do not test validity or reachability of the remote endpoint address
  - It records the remote endpoint info in the socket data structure

- UDP
  - message transfer
  - send(), recv()

- Close()
  - Do not inform the remote endpoint

- Shutdown()
  - For a connected UDP
  - Stop further transmission in a given direction
  - Again, no control message is sent to the other side.
Discussion

- Client applications using UDP must handle reliability functions themselves if needed.
- Reliability techniques, e.g.,
  - Packet sequencing
  - Acknowledgements
  - Timeouts
  - Retransmission
/* Allocate a socket */
    
s = socket(PF_INET, SOCK_DGRAM, 0);
    
    if ( s < 0 )
        
        errexit(“can’t create socket: %s\n”, strerror(errno));

/* Connect the socket */
    
    if (connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
        
        errexit( “can’t connect to %s.%s: %s\n”, host, service , strerror(errno) );

    return s;
}
Example Client Software
A Procedure to Form Connections

/* connectsock.c - connectsock */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <string.h>
#include <stdlib.h>
#ifndef INADDR_NONE
#define INADDR_NONE 0xffffffff#endif /* INADDR_NONE */
extern int errno;
int errexit (const char *format, …);

extern int errno;
int errexit (const char *format, …);
`connectsock` – allocate & connect a socket using TCP or UDP

```c
int connectsock( const char *host, const char *service, const char *transport )
```

- `host` - name of host to which connection is desired
- `service` - service associated with the desired port
- `transport` - name of transport protocol to use ("tcp" or "udp")

```c
struct hostent *phe; /* pointer to host information entry */
struct servent *pse; /* pointer to service information entry */
struct protoent *ppe; /* pointer to protocol information entry */
struct sockaddr_in sin; /* an Internet endpoint address */
int s, type; /* socket descriptor and socket type */
memset(&sin, 0, sizeof(sin)); /* clean up */
sin.sin_family = AF_INET; /* address family: Internet */
```
/* Map service name to port number */
    if ( pse = getservbyname(service, transport) )
        sin.sin_port = pse -> s_port;
    else if ((sin.sin_port = htons((unsigned short)atoi(service))) == 0)
        errexit("can't get \"%s\" service entry\n", service);
/* Map host name to IP address, allowing for dotted decimal */
    if ( phe = gethostbyname(host) )
        memcpy(&sin.sin_addr, phe->h_addr, phe->h_length);
    else if ((sin.sin_addr.s_addr=inet_addr(host)) == INADDR_NONE)
        errexit("can't get \"%s\" host entry\n", host);
/* Map transport protocol name to protocol number */
    if ( (ppe = getprotobynumber(transport)) == 0)
        errexit("can't get \"%s\" protocol entry\n", transport);
/* Use protocol to choose a socket type */
    if(strcmp(transport, "udp") == 0)
        type = SOCK_DGRAM;
    else
        type = SOCK_STREAM;
/* Allocate a socket */
    s = socket(PF_INET, type, ppr->p_proto);
    if ( s < 0 )
        errexit("can't create socket: %s\n", strerror(errno));

/* Connect the socket */
    if (connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
        errexit("can't connect to %s.%s: %s\n", host, service ,
                 strerror(errno) );

    return s;
}
Algorithms for TCP server software design
Server architecture

**SERVER SIDE**

- Create a socket
- Bind the socket to a well-known port to receive requests
- Enter an infinite loop to accept the next request from clients
- Process the request
- Formulate a reply
- Send reply back
bind()

- sockaddr_in – IP address and port number
- getportbyname
  - Used by a server to map a service name into the corresponding well-known port number
- INADDR_ANY (a socket interface constant)
  - To allow a multi-homed hosts and routers to have a single server accept incoming communication addressed to any of the hosts’s IP addresses
listen()

- An input argument – specifying the length of an internal request queue for the socket
  - Each incoming TCP connection request
- Place the socket in passive mode

SERVER SIDE
- socket
- bind
- listen
- accept
- recv
- send
- close
accept()

- Obtain the next incoming connection request (i.e., extract the request from the request queue)
- It returns a socket descriptor to be used for the new connection.
- Once accepted the connection, use read() to obtain application protocol requests from the client
- Use write() to send replies back.
- Use close() to release the socket
Server architecture: concurrent vs. iterative

- **Iterative server**
  - Process one request at a time

- **Concurrent server**
  - Handle multiple requests at one time, i.e., permit multiple requests to proceed concurrently

- **Multiple threads of execution**
  - Each thread handles one request
Four types of servers

<table>
<thead>
<tr>
<th>Low Request Processing time</th>
<th>Iterative connectionless</th>
<th>Iterative connection-oriented</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concurrent connectionless</td>
<td>Concurrent connectionless</td>
<td>Concurrent connection-oriented</td>
</tr>
</tbody>
</table>
Iterative connectionless

Client
- `connect()` – to specify a server’s address
- `write()` – to send data (internal data structure contains both two endpoints address)

Server
- `recvfrom()`
  - Server uses to receive the sender’s address
- `retcode = recvfrom(s, buf, len, flags, from, fromlen);`
- Uses an unconnected socket
- `sendto()` to specify both a datagram to be sent and an address to which it goes.
- `retcode = sendto(s, message, len, flags, toaddr, toaddrlen);`
  - S: unconnected socket
  - Generates reply addresses explicitly use
Concurrent, Connection-Oriented Servers (TCP)
Concurrent Sever: Goals

- Provide faster response times to multiple clients
- Suitable for applications that
  - Form a response required significant I/O
  - Diverse processing times
  - Server executed on a multi-processor computer
Concurrent sever: Using separate process

- A master server process begins execution initially
- Master opens a socket at the well-known port
- Wait for the next request
- Create a slave server process to handle each request
- A slave process exits when complete the communication with the client
- A concurrent server creates a new process for each connection.
- `fork()` – a system call
- Both master and slave processes execute the same code.
- `execve()` – have the slave process execute an independently written code after the call to fork.
The Process Structure

- **Master**
- **slave\_1**, **slave\_2**, **slave\_n**
- **client**
- **socket used for conn. requests**
- **socket used for an individual conn.**
- **Internet**
- **Server Application Process**
- **Operating System**
Concurrent Connection-Oriented Server Algorithm:
Using separate process

Master 1: Create a socket and bind to the well-known address for the service being offered
Master 2: Place the socket in passive mode, making it ready for use by a server
Master 3: Repeatedly call `accept` to receive the next connection request from the client, and create a new slave process to handle the response.
Slave 1: receive a connection request upon creation
Slave 2: interact with the client using the connection: read requests and send back replies
Slave 3: close the connection and exit.
Concurrent Connection-Oriented Server Algorithm: Using one single process

- In many OSs, process creation and context switching are expensive
- Use `select()` (a system call) for asynchronous I/O.

1. Create a socket and bind to the well-known address for the service. Add socket to the I/O list.
2. Use `select` to wait for I/O on existing sockets
3. If original socket is ready, use `accept` to obtain the next connection, and add the new socket to the I/O list
4. If some socket other than the original is ready, use `read` to obtain the next request, and use `write` to send the response.
5. return to step 2
Concurrent
Connection-Oriented Server
Algorithm: Using separate process
/* passiveTCP.c - passiveTCP - create a passive socket for use in a TCP server */

int passivesock(const char *service, const char *transport, int qlen);

int passiveTCP(const char *service, int qlen)
{
    return passivesock(service, "tcp", qlen);
}
Passive TCP: sample code (2/7)

/* passivesock.c - passivesock */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <string.h>
#include <netdb.h>

extern int errno;
int errexit(const char *format, ...);
u_short portbase = 0; // port base, for non-root servers
Passive TCP: sample code (3/7)

```c
/*----------------------------------------------*/
    passivesock –
    allocate & bind a server socket using TCP or UDP
/*----------------------------------------------*/
int passivesock(const char *service,
                const char *transport, int qlen)
/*
 * Arguments:
 *     service  - service associated with the desired port
 *     transport - transport protocol to use ("tcp" or "udp")
 *     qlen      - maximum server request queue length
*/
```
Passive TCP: sample code (4/7)

{
    struct servent *pse;
    /* pointer to service information entry */
    struct protoent *ppe;
    /* pointer to protocol information entry */
    struct sockaddr_in sin;
    /* an Internet endpoint address */
    int s, type;
    /* socket descriptor and socket type */

    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = INADDR_ANY;
}
/* Map service name to port number */
if ( pse = getservbyname(service, transport) )
    sin.sin_port = htons(ntohs((u_short)pse->s_port)
        + portbase);
else if ( (sin.sin_port = htons((u_short)atoi(service)))
    == 0 )
    errexit("can't get \"%s\" service entry\n", service);

/* Map protocol name to protocol number */
if ( (ppe = getprotobynameltransport)) == 0)
    errexit("can't get \"%s\" protocol entry\n", transport);
/* Use protocol to choose a socket type */
    if (strcmp(transport, "udp") == 0)
        type = SOCK_DGRAM;
    else type = SOCK_STREAM;

/* Allocate a socket */
    s = socket(PF_INET, type, ppe->p_proto);
    if (s < 0)
        errexit("can't create socket: %s\n", strerror(errno));
/* Bind the socket */

if (bind(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
    errexit("can't bind to %s port: %s
        strerror(errno));

if (type == SOCK_STREAM && listen(s, qlen) < 0)
    errexit("can't listen on %s port: %s
        strerror(errno));

return s;
Socket API: Functionality

- Allocate local resources for communication
- Specify local and remote communication endpoints
- Initiate a **connect** (client side)
- Send a datagram (client side)
- **Wait for an incoming connection** (server side)
- Send or receive data
- Determine when data arrives
- Generate urgent data
- Handle incoming urgent data
- Terminate a connection gracefully
- Handle connection termination from the remote site
- Abort communication
- Handle error conditions or a connection abort
- **Release local resource** when communication finishes.
Appendix
Generic address structure (1/3)

Generic address structure

- The goal is to allow a software to manipulate protocol addresses without knowing the details of how every protocol family defines its address representation.

- e.g.,
  - a procedure that accepts an arbitrary protocol endpoint spec as an argument
  - Choose one of several possible actions depending on the address type

- (address family, endpoint address in the family)
  - A constant denotes a predefined address types
  - The representation for the specified address type.
To keep programs portable and maintainable, TCP/IP code should not use the `sockaddr` structure in declarations.

- It can be used only as an overlay.
- The code should reference only the `sa_family` field.
Generic address structure (3/3)

/* used for declaring variables to store endpoint address

struct sockaddr { /* struct to hold an address */
    u_char     sa_len; /* total length */
    u_short    sa_family; /* type of address */
    char       sa_data[14]; /* value of address */
};