Tutorial 2: COMP4621

Socket Programming in C

Fall 2024

TA: Xinyu YANG
E-mail: xinyu.yang@connect.ust.hk

Outline

- Data structures
- Socket API in C
- UDP server-client example
- TCP server-client example
- *getaddrinfo*() and *struct addrinfo*

Data Structures: *sockaddr*

- The struct *sockaddr* holds socket address information for many types of sockets

```
struct sockaddr {
    unsigned short    sa_family;    // address family, AF_xxx
    char              sa_data[14];  // 14 bytes of protocol address
};
```

- *sa_family* can be a variety of things: for Internet it'll be *AF_INET (IPv4)* or *AF_INET6 (IPv6)*
- *sa_data* contains a destination address and port number for the socket. But you don't want to tediously pack the address in the *sa_data* by hand.

Data Structures: *sockaddr*

- To deal with struct *sockaddr*, programmers created another structure: *struct sockaddr_in* ("in" for "Internet") to be used with IPv4.

```
// (IPv4 only--see struct sockaddr_in6 for IPv6)

struct sockaddr_in {
    short int          sin_family;  // Address family, AF_INET
    unsigned short int sin_port;    // Port number
    struct in_addr     sin_addr;    // Internet address
    unsigned char      sin_zero[8]; // Same size as struct sockaddr
};
```

  - Note that *sin_zero* (is included to pad the structure to the length of a struct sockaddr) should be set to all zeros with the function *memset().*
  - Also, notice that *sin_family* corresponds to *sa_family* in a struct *sockaddr* and should be set to "*AF_INET*".
  - Finally, the *sin_port* must be in *Network Byte Order*

Data Structures: *sin_addr*

- The *sin_addr* field is a struct *in_addr*.

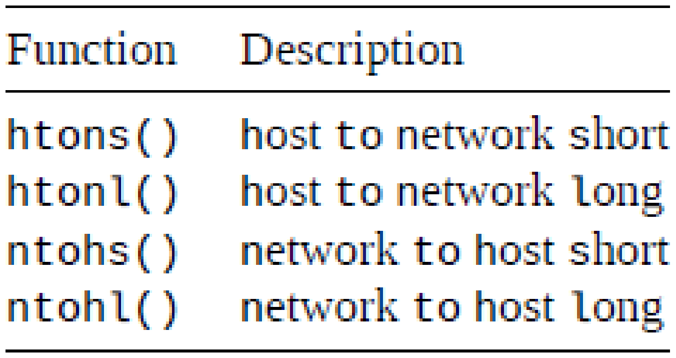
```
// (IPv4 only--see struct in6_addr for IPv6)

// Internet address (a structure for historical reasons)
struct in_addr {
    uint32_t s_addr; // that's a 32-bit int (4 bytes)
};
```

  - If you have declared *ina* to be of type struct *sockaddr_in*, then *ina.sin_addr.s_addr* references the 4-byte (32bit) IP address (in Network Byte Order)

Socket API Functions

- Function·0



Function	Description
htons()	host to network short
htonl()	host to network long
ntohs()	network to host short
ntohl()	network to host long

- Byte Order Conversion
  - Different communicating hosts may use different byte ordering.
  - There are two ways to store two bytes in memory: with the lower-order byte at the starting address (*little-endian* byte order) or with the high-order byte at the starting address (*big-endian* byte order). We call them collectively host byte order.
  - To avoid confusion, Networking protocols such as TCP are based on a specific *network byte order*. The Internet protocols use big-endian byte ordering.
  - There are two types of numbers that you can convert: short (two bytes) and long (four bytes).
  - Before sending/upon receiving, port numbers and IP addresses must be converted to/from network byte order

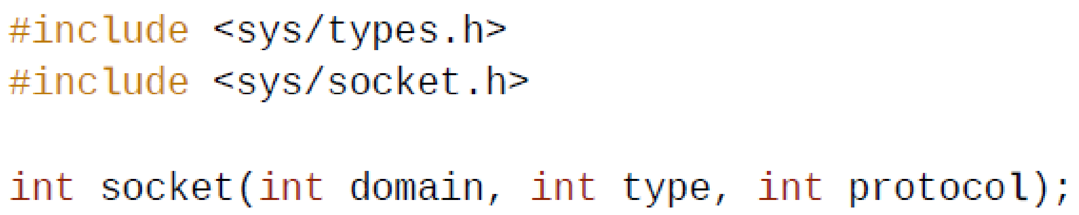
An Example

```c
struct sockaddr_in my_addr;

my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(3490); // short, network byte order
my_addr.sin_addr.s_addr = inet_addr("10.12.110.57");
memset(my_addr.sin_zero, '\0', sizeof my_addr.sin_zero);
```

Socket API Functions

- Function·1



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

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

- Return Value

    Returns a new socket descriptor that you can use to do what sockets do:

- Arguments

    - *domain:* describes the kind of the socket. It’s going to be PF_INET for IPv4 and PF_INET6 for IPv6
    - *type:* It can be a number of things. The common setting is SOCK_STREAM for reliable TCP sockets, SOCK_DGRAM for unreliable fast UDP sockets, and SOCK_RAM for constructing packets by hand.
    - *protocol:* The protocol parameter tells which protocol to use with a certain socket type. When using SOCK_STREAM and SOCK_DGRAM, you can just set the protocol to 0 and it’ll use the proper protocol automatically. Otherwise, you can use getprotobyname() to look up the proper protocol number.

Socket API Functions

- Function·2

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

int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

- Arguments
  - *sockfd* is the socket file descriptor returned by *socket()*.
  - *my_addr* is a pointer to a struct *sockaddr* that contains information about your address, namely, port and IP address.
  - *addrlen* is the length in bytes of that address.

An Example

```c
int sockfd;
struct sockaddr_in my_addr;

sockfd = socket(PF_INET, SOCK_STREAM, 0);

my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(3490); // short, network byte order
my_addr.sin_addr.s_addr = inet_addr("10.12.110.57");
memset(my_addr.sin_zero, '\0', sizeof my_addr.sin_zero);

bind(sockfd, (struct sockaddr *)&my_addr, sizeof my_addr);
```

Socket API Functions

- Function·3

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

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

- Return Value

  Returns zero on success, or -1 on error (and errno will be set accordingly).

- Arguments
  - *sockfd* is the socket descriptor
  - The *backlog* parameter is how many pending connections you can have before the kernel starts rejecting new ones. So as the new connections come in, you should be quick to accept() them so that the backlog doesn’t fill. Try setting it to 10 or so, and if your clients start getting “Connection refused” under heavy load, set it higher.

Socket API Functions

- Function·4

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

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

- Return Value

  Returns zero on success, or -1 on error (and errno will be set accordingly).

- Arguments
  - *sockfd:* the socket descriptor to be connected to the remote server
  - *serv_addr:* the address of the server you are interested in connecting to
  - *addrlen:* this is filled with the sizeof() the structure returned in the *serv_addr* parameter.

Socket API Functions

- Function·5

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

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

- Return Value

accept() returns the newly connected socket descriptor, or -1 on error, with errno set appropriately.

- Arguments
  - *sockfd:* the listening socket descriptor
  - *addr:* this is filled in with the address of the site that's connecting to you
  - *addrlen:* this is filled with the sizeof() the structure returned in the addr parameter. You can safely ignore it if you're getting a struct sockaddr_in back because that's the type you passed in for addr.

Socket API Functions

- Function·6

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

ssize_t send(int sockfd, const void *buf, size_t len, int flags);
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags, const struct sockaddr *to,
socklen_t tolen);
```

- Description
  - *send()* is used for TCP SOCK_STREAM connected sockets, and *sendto()* is used for UDP SOCK_DGRAM unconnected datagram sockets. With the unconnected sockets, you must specify the destination of a packet each time you send one, and that's why the last parameters of *sendto()* define where the packet is going.
- Return Value

Returns the number of bytes sent, or -1 on error (and errno will be set accordingly)

- Arguments
  - *sockfd:* the socket
  - *buf:* a pointer to the data you want to send
  - *len:* the number of bytes you want to send
  - *flags:* specify more information about how the data is to be sent

Socket API Functions

- Function·7

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

ssize_t recv(int sockfd, const void *buf, size_t len, int flags);
ssize_t recvfrom(int sockfd, const void *buf, size_t len, int flags, struct sockaddr *from,
socklen_t fromlen);
```

- Description

  Once you have a socket up and connected, you can read incoming data from the remote side using the recv() (for TCP SOCK_STREAM sockets) and recvfrom() (for UDP SOCK_DGRAM sockets)

- Return Value

  Returns the number of bytes actually received (which might be less than you requested in the len parameter), or -1 on error (and errno will be set accordingly).

- Arguments
  - *sockfd:* the socket
  - *buf:* a pointer to the data you receive
  - *len:* the number of bytes you receive
  - *flags:* specify more information about how the data is to be sent

Socket API Functions

- Function·8

```
int close(int sockfd);
```

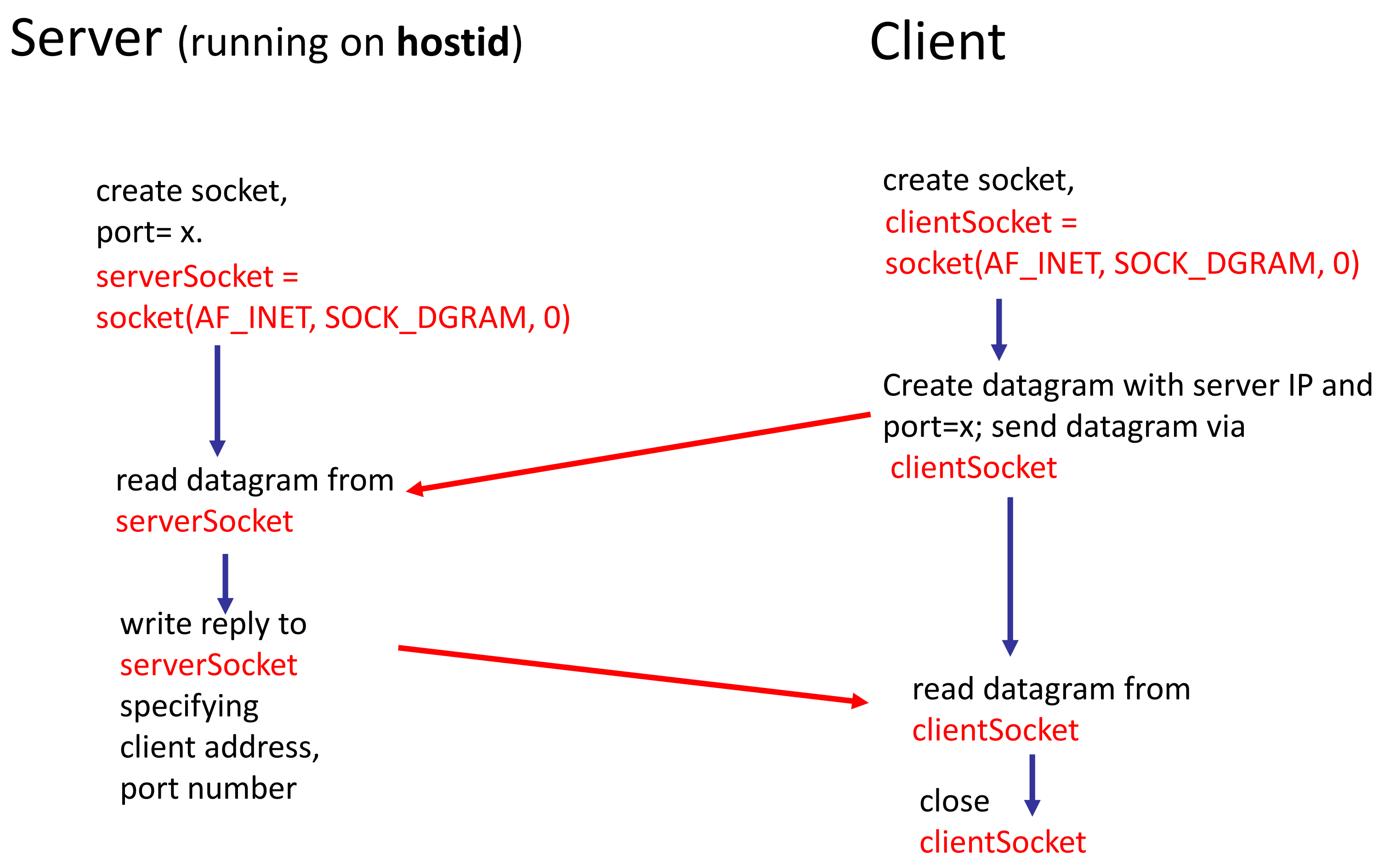
- Return Value

  Returns zero on success, or -1 on error (and errno will be set accordingly).

How to use all of this: Our First Example

- Client:
  - User types a line of text
  - Client program sends the line to server
- Server:
  - Server receives line of text
  - Capitalizes all the letters
  - Sends modified line to client
- Client:
  - Receives line of text
  - Displays it on screen

Client/server socket interaction: UDP



Server (running on **hostid**)

create socket,
port= x.
serverSocket =
socket(AF_INET, SOCK_DGRAM, 0)

read datagram from
serverSocket

write reply to
serverSocket
specifying
client address,
port number

Client

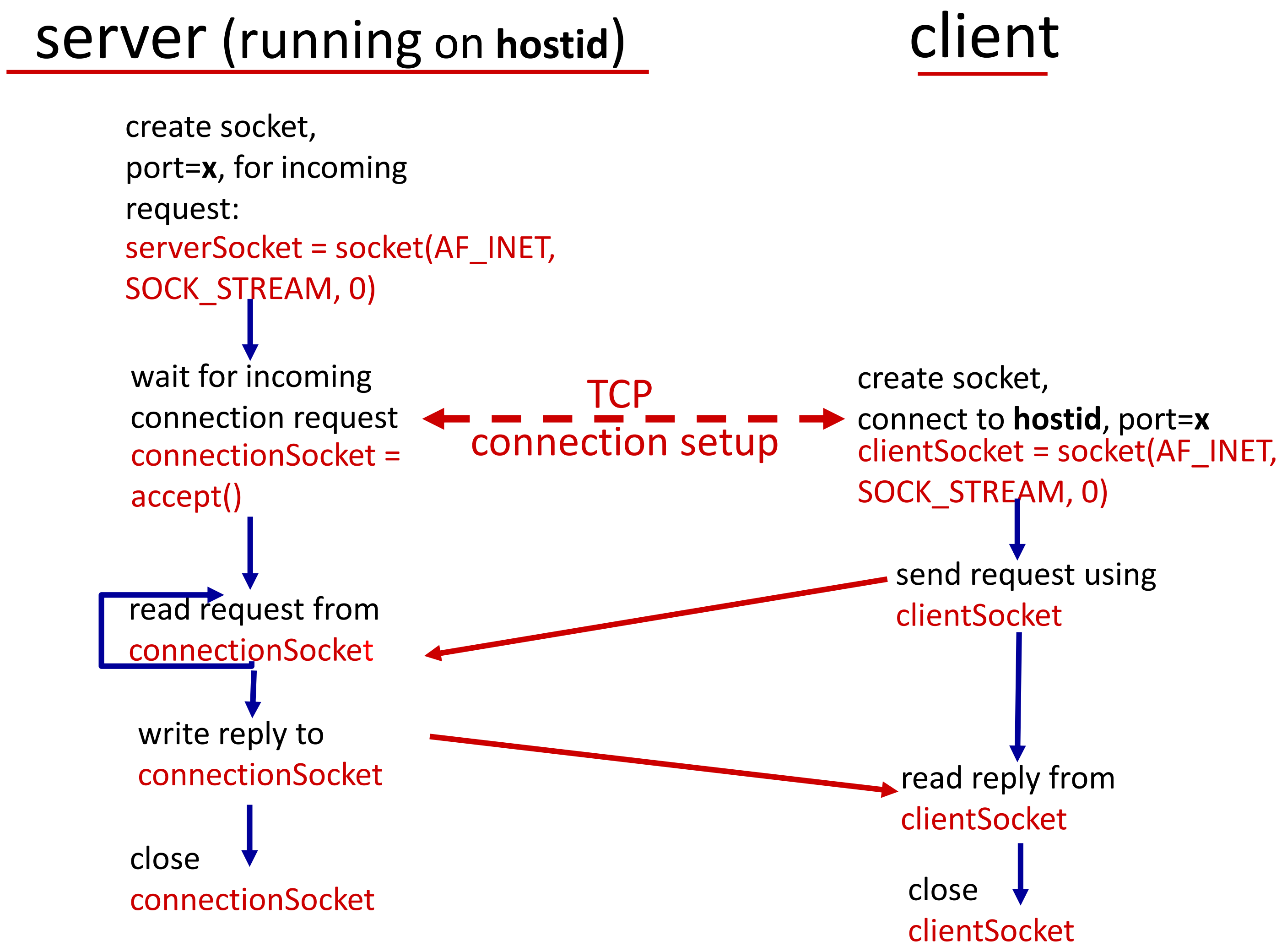
create socket,
clientSocket =
socket(AF_INET, SOCK_DGRAM, 0)

Create datagram with server IP and
port=x; send datagram via
clientSocket

read datagram from
clientSocket

close
clientSocket

Client/server socket interaction: TCP



server (running on **hostid**)

create socket,
port=**x**, for incoming
request:
serverSocket = socket(AF_INET,
SOCK_STREAM, 0)

wait for incoming
connection request
connectionSocket =
accept()

read request from
connectionSocket

write reply to
connectionSocket

close
connectionSocket

TCP
connection setup

client

create socket,
connect to **hostid**, port=**x**
clientSocket = socket(AF_INET,
SOCK_STREAM, 0)

send request using
clientSocket

read reply from
clientSocket

close
clientSocket

UDP Socket programming in C: Summary

UDP Server :

1. using socket(), create a UDP socket.
2. using bind(), bind the socket to the server address.
3. using recvfrom(), wait until the datagram packet arrives from the client.
4. process the datagram packet and use sendto() to send a reply to the client.
5. go back to Step 3.

UDP Client :

1. using socket(), create a UDP socket.
2. using sendto(), send a message to the server.
3. using recvfrom(), wait until response from the server is received.
4. process reply and go back to step 2, if necessary.
5. using close(), close socket descriptor and exit.

TCP Socket programming in C: Summary

TCP Server
1. using socket(), Create TCP socket.
2. using bind(), Bind the socket to server address.
3. using listen(), put the server socket in a passive mode, where it waits for the client to approach the server to make a connection
4. using accept(), at this point, connection is established between client and server, on a new socket and they are ready to transfer data.
5. Go back to Step 3.

TCP Client
1. using socket(), create TCP socket.
2. using connect(), connect newly created client socket to server.

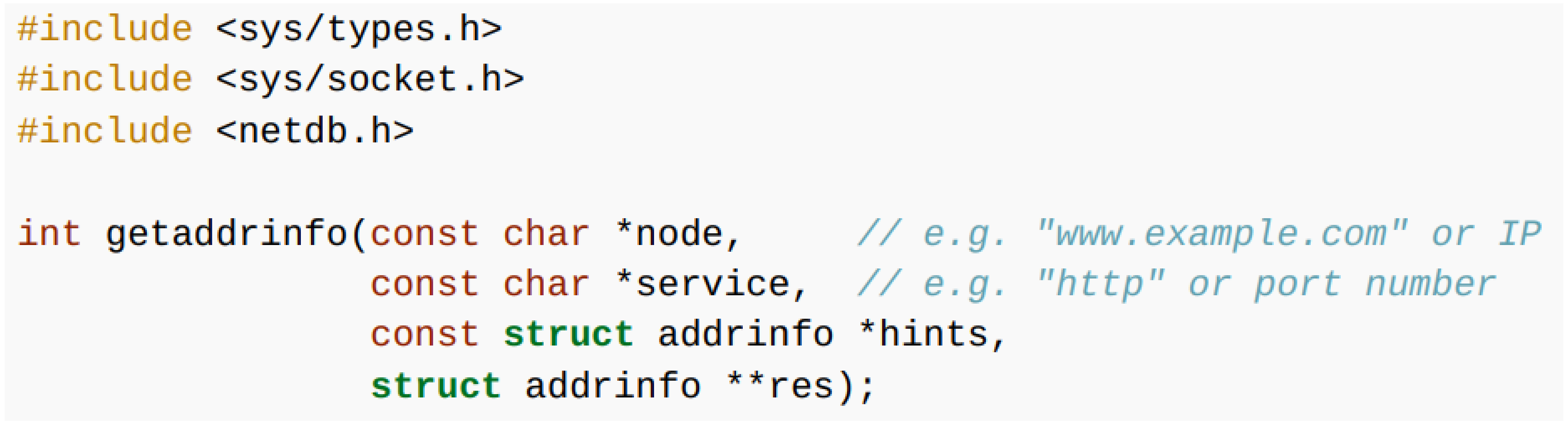
addrinfo

```
struct addrinfo {
    int              ai_flags;     // AI_PASSIVE, AI_CANONNAME, etc.
    int              ai_family;    // AF_INET, AF_INET6, AF_UNSPEC
    int              ai_socktype;  // SOCK_STREAM, SOCK_DGRAM
    int              ai_protocol;  // use 0 for "any"
    size_t           ai_addrlen;   // size of ai_addr in bytes
    struct sockaddr *ai_addr;      // struct sockaddr_in or _in6
    char            *ai_canonname; // full canonical hostname

    struct addrinfo *ai_next;      // linked list, next node
};
```

- Defined in netdb.h, so you need: **#include <netdb.h>**
- Stores information about the type of sockets supported by the caller.
- Oftentimes, a call to *getaddrinfo()* fills out *addrinfo* for you

getaddrinfo()



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

int getaddrinfo(const char *node,     // e.g. "www.example.com" or IP
                const char *service,  // e.g. "http" or port number
                const struct addrinfo *hints,
                struct addrinfo **res);
```

- Node: domain name or IP address.
- Service: port number or the protocol (e.g., http, ftp, https).
- Hints: point to an addrinfo struct you privde.
- Res: A list of results.

An Example

```c
// !!! THIS IS THE OLD WAY !!!

int sockfd;
struct sockaddr_in my_addr;

sockfd = socket(PF_INET, SOCK_STREAM, 0);

my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(MYPORT);     // short, network byte order
my_addr.sin_addr.s_addr = inet_addr("10.12.110.57");
memset(my_addr.sin_zero, '\0', sizeof my_addr.sin_zero);

bind(sockfd, (struct sockaddr *)&my_addr, sizeof my_addr);
```

The old fashion: Call htons, inet_addr, memset functions manually.

```c
struct addrinfo hints, *res;
int sockfd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;     // fill in my IP for me

getaddrinfo(NULL, "3490", &hints, &res);

// make a socket:

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

// bind it to the port we passed in to getaddrinfo():

bind(sockfd, res->ai_addr, res->ai_addrlen);
```

You can get all the address information by calling getaddrinfo()!

Summary

- We have learned the basic data structures and functions for socket programming in C
- We also learned to write a simple server-client application

Try it yourself*: Use the demo code available on Canvas to implement the clients and servers therein by using getaddrinfo() instead.*

Next

- We will look at advanced techniques on how to build high performance client-server applications

Task

Answer the following questions:

1. Describe the components of a socket address structure. Why do we use the data structure sockaddr_in instead of socketaddr?
2. What is the purpose of the bind() function in socket programming? Does the client require it? Why?
3. Explain the purpose of the listen() function in socket programming. How does it work with the accept() function to establish connections between clients and servers? What does the argument “backlog” in the listen() function mean?
4. For the recv() and recvfrom() functions, what does the argument “size_t len” mean and what’s the difference between it and the return value?

Submit a pdf document on Canvas before 11:59pm tonight!