

# UDP Sockets Programming

- Creating UDP sockets.
  - Client
  - Server
- Sending data.
- Receiving data.
- Connected Mode.

# Creating a UDP socket

```
int socket(int family,int type,int proto);

int sock;

sock = socket( PF_INET,
               SOCK_DGRAM,
               0);

if (sock<0) { /* ERROR */ }
```

# Binding to well known address (typically done by server only)

```
int mysock;  
struct sockaddr_in myaddr;  
  
mysock = socket(PF_INET, SOCK_DGRAM, 0);  
myaddr.sin_family = AF_INET;  
myaddr.sin_port = htons( 1234 );  
myaddr.sin_addr = htonl( INADDR_ANY );  
  
bind(mysock, &myaddr, sizeof(myaddr));
```

# Sending UDP Datagrams

```
ssize_t sendto( int sockfd,  
                void *buff,  
                size_t nbytes,  
                int flags,  
                const struct sockaddr* to,  
                socklen_t addrlen);
```

`sockfd` is a UDP socket

`buff` is the address of the data (`nbytes` long)

`to` is the address of a `sockaddr` containing the destination address.

Return value is the number of bytes sent, or -1 on error.

# sendto ()

- You can send 0 bytes of data!
- Some possible errors :
  - EBADF**, **ENOTSOCK**: bad socket descriptor
  - EFAULT**: bad buffer address
  - EMSGSIZE**: message too large
  - ENOBUFS**: system buffers are full

# More `sendto()`

- The return value of `sendto()` indicates how much data was accepted by the O.S. for sending as a datagram - not how much data made it to the destination.
- There is no error condition that indicates that the destination did not get the data!!!

# Receiving UDP Datagrams

```
ssize_t recvfrom( int sockfd,  
                 void *buff,  
                 size_t nbytes,  
                 int flags,  
                 struct sockaddr* from,  
                 socklen_t *fromaddrlen);
```

`sockfd` is a UDP socket

`buff` is the address of a buffer (`nbytes` long)

`from` is the address of a `sockaddr`.

Return value is the number of bytes received and put into `buff`, or `-1` on error.

# recvfrom()

- If `buff` is not large enough, any extra data is lost forever...
- You can receive 0 bytes of data!
- The `sockaddr` at `from` is filled in with the address of the sender.
- You should set `fromaddrlen` before calling.
- If `from` and `fromaddrlen` are NULL we don't find out who sent the data.



# More `recvfrom()`

- Same errors as `sendto`, but also:
  - `EINTR`: System call interrupted by signal.
- Unless you do something special - `recvfrom` doesn't return until there is a datagram available.

# Typical UDP client code

- Create UDP socket.
- Create `sockaddr` with address of server.
- Call `sendto()`, sending request to the server. **No call to `bind()` is necessary!**
- Possibly call `recvfrom()` (if we need a reply).

# Typical UDP Server code

- Create UDP socket and bind to well known address.
- Call `recvfrom()` to get a request, noting the address of the client.
- Process request and send reply back with `sendto()`.

# UDP Echo Server

```
int mysock;
struct sockaddr_in myaddr, cliaddr;
char msgbuf[MAXLEN];
socklen_t clien;
int msglen;

mysock = socket(PF_INET, SOCK_DGRAM, 0);
myaddr.sin_family = AF_INET;
myaddr.sin_port = htons( S_PORT );
myaddr.sin_addr = htonl( INADDR_ANY );
bind(mysock, &myaddr, sizeof(myaddr));
while (1) {
    len=sizeof(cliaddr);
    msglen=recvfrom(mysock, msgbuf, MAXLEN, 0, cliaddr, &clien);
    sendto(mysock, msgbuf, msglen, 0, cliaddr, clien);
}
```

**NEED TO CHECK FOR ERRORS!!!**

# Debugging

- Debugging UDP can be difficult.
- Write routines to print out sockaddr.
- Use trace, strace, ptrace, truss, etc.
- Include code that can handle unexpected situations.

# Timeout when calling `recvfrom()`

- It might be nice to have each call to `recvfrom()` return after a specified period of time even if there is no incoming datagram.
- We can do this by using `SIGALRM` and wrapping each call to `recvfrom()` with a call to `alarm()`

# recvfrom() and alarm()

```
signal(SIGALRM, sig_alarm);
alarm(max_time_to_wait);
if (recvfrom(...)<0)
    if (errno==EINTR)
        /* timed out */
    else
        /* some other error */
else
    /* no error or time out
    - turn off alarm */
alarm(0);
```

There are some other (better) ways to do this ...

# Connected mode

- A UDP socket can be used in a call to `connect ()` .
- This simply tells the O.S. the address of the peer.
- No handshake is made to establish that the peer exists.
- No data of any kind is sent on the network as a result of calling `connect ()` on a UDP socket.



# Connected UDP

- Once a UDP socket is *connected*:
  - can use `sendto ()` with a null dest. address
  - can use `write ()` and `send ()`
  - can use `read ()` and `recv ()`
    - only datagrams from the peer will be returned.
  - Asynchronous errors will be returned to the process.

**OS Specific, some won't do this!**



# Asynchronous Errors

- What happens if a client sends data to a server that is not running?
  - ICMP “port unreachable” error is generated by receiving host and sent to sending host.
  - The ICMP error may reach the sending host after `sendto()` has already returned!
  - The next call dealing with the socket could return the error.

# Back to UDP connect()

- Connect() is typically used with UDP when communication is with a single peer only.
- Many UDP clients use connect().
- Some servers (TFTP).
- It is possible to disconnect and connect the same socket to a new peer.

# TCP/IP Sockets in C: Practical Guide for Programmers

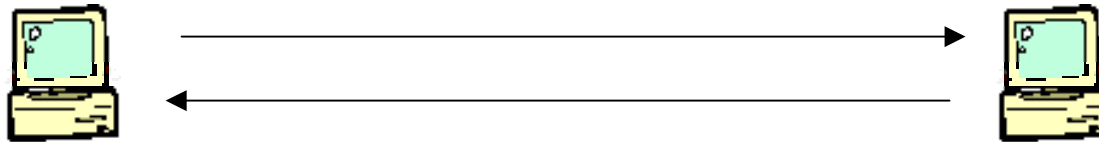


---

Michael J. Donahoo  
Kenneth L. Calvert

# Computer Chat

- How do we make computers talk?



- How are they interconnected?

Internet Protocol (IP)



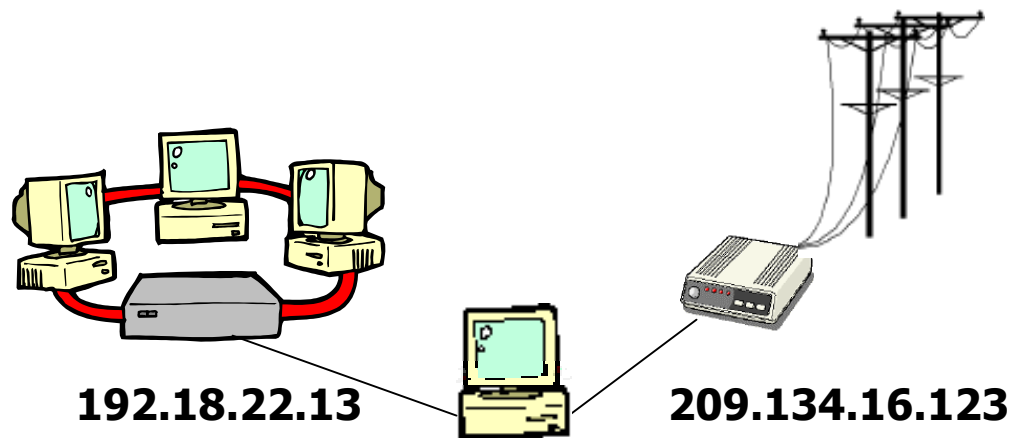
# Internet Protocol (IP)

---

- Datagram (packet) protocol
- Best-effort service
  - Loss
  - Reordering
  - Duplication
  - Delay
- Host-to-host delivery  
(not application-to-application)

# IP Address

- 32-bit identifier
- Dotted-quad: 192.118.56.25
- www.mkp.com -> 167.208.101.28
- Identifies a host interface (not a host)





# Transport Protocols

---

**Best-effort not sufficient!**

- Add services on top of IP
- User Datagram Protocol (UDP)
  - Data checksum
  - Best-effort
- Transmission Control Protocol (TCP)
  - Data checksum
  - Reliable byte-stream delivery
  - Flow and congestion control



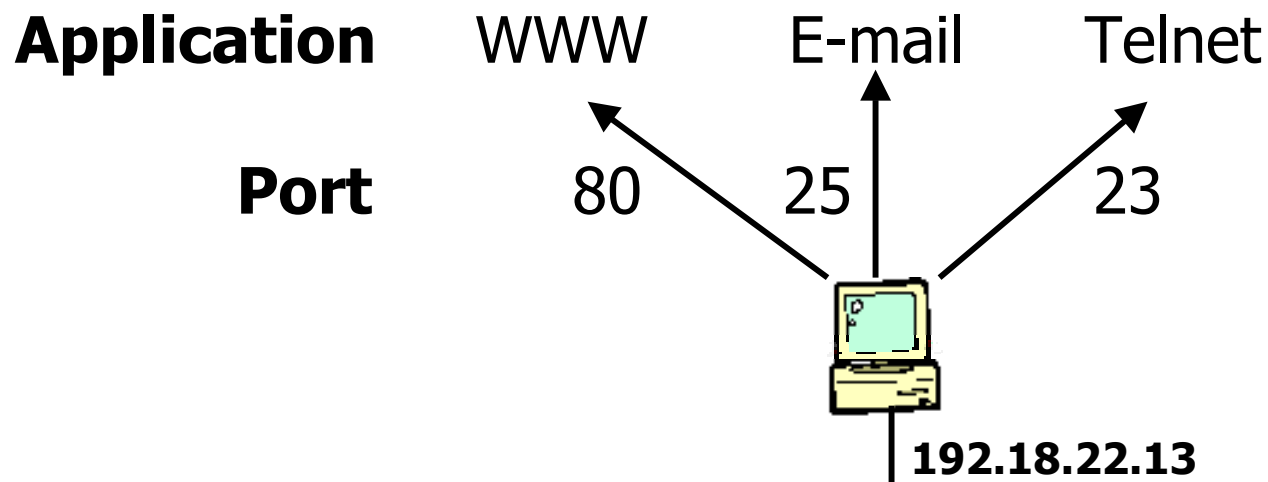


# Ports

---

## Identifying the ultimate destination

- IP addresses identify hosts
- Host has many applications
- Ports (16-bit identifier)





# Socket

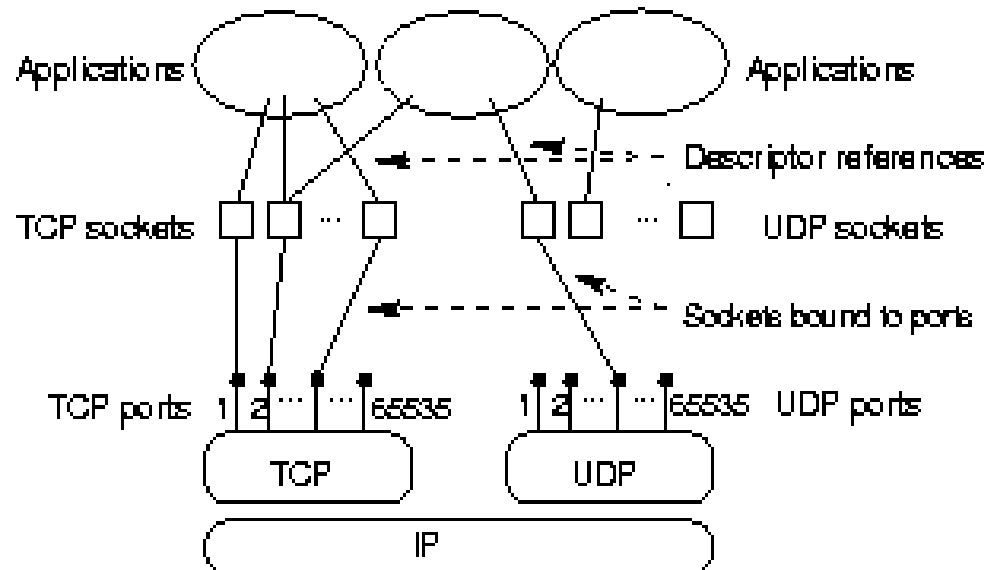
---

## How does one speak TCP/IP?

- Sockets provides interface to TCP/IP
- Generic interface for many protocols

# Sockets

- Identified by protocol and local/remote address/port
- Applications may refer to many sockets
- Sockets accessed by many applications





# TCP/IP Sockets

---

- `mySock = socket(family, type, protocol);`
- TCP/IP-specific sockets

	Family	Type	Protocol
TCP	PF_INET	SOCK_STREAM	IPPROTO_TCP
UDP		SOCK_DGRAM	IPPROTO_UDP

- Socket reference
  - File (socket) descriptor in UNIX
  - Socket handle in WinSock

Generic

```

■ struct sockaddr
{
    unsigned short sa_family; /* Address family (e.g., AF_INET) */
    char sa_data[14];        /* Protocol-specific address information */
};

```

IP Specific

```

■ struct sockaddr_in
{
    unsigned short sin_family; /* Internet protocol (AF_INET) */
    unsigned short sin_port;   /* Port (16-bits) */
    struct in_addr sin_addr;   /* Internet address (32-bits) */
    char sin_zero[8];         /* Not used */
};
struct in_addr
{
    unsigned long s_addr;     /* Internet address (32-bits) */
};

```

sockaddr

Family	Blob		
2 bytes	2 bytes	4 bytes	8 bytes
Family	Port	Internet address	Not used

sockaddr\_in



# Clients and Servers

---

- Client: Initiates the connection

Client: Bob

Server: Jane

"Hi. I'm Bob." →

← "Hi, Bob. I'm Jane"

"Nice to meet you, Jane." →

- Server: Passively waits to respond



# TCP Client/Server Interaction

---

Server starts by getting ready to receive client connections...

## Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

## Server

1. Create a TCP socket
2. Assign a port to socket
3. Set socket to listen
4. Repeatedly:
  - a. Accept new connection
  - b. Communicate
  - c. Close the connection



# TCP Client/Server Interaction

---

```
/* Create socket for incoming connections */  
if ((servSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)  
    DieWithError("socket() failed");
```

## Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

## Server

1. **Create a TCP socket**
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
  - a. Accept new connection
  - b. Communicate
  - c. Close the connection





# TCP Client/Server Interaction

---

```
echoServAddr.sin_family = AF_INET;           /* Internet address family */
echoServAddr.sin_addr.s_addr = htonl(INADDR_ANY); /* Any incoming interface */
echoServAddr.sin_port = htons(echoServPort); /* Local port */
```

```
if (bind(servSock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0)
    DieWithError("bind() failed");
```

## Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

## Server

1. Create a TCP socket
2. **Bind socket to a port**
3. Set socket to listen
4. Repeatedly:
  - a. Accept new connection
  - b. Communicate
  - c. Close the connection



# TCP Client/Server Interaction

---

```
/* Mark the socket so it will listen for incoming connections */  
if (listen(servSock, MAXPENDING) < 0)  
    DieWithError("listen() failed");
```

## Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

## Server

1. Create a TCP socket
2. Bind socket to a port
3. **Set socket to listen**
4. Repeatedly:
  - a. Accept new connection
  - b. Communicate
  - c. Close the connection



# TCP Client/Server Interaction

---

```
for (;;) /* Run forever */  
{  
    clntLen = sizeof(echoClntAddr);  
  
    if ((clntSock=accept(servSock,(struct sockaddr *)&echoClntAddr,&clntLen)) < 0)  
        DieWithError("accept() failed");
```

## Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

## Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
  - a. **Accept new connection**
  - b. Communicate
  - c. Close the connection



# TCP Client/Server Interaction

---

Server is now blocked waiting for connection from a client

Later, a client decides to talk to the server...

## Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

## Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
  - a. **Accept new connection**
  - b. Communicate
  - c. Close the connection



# TCP Client/Server Interaction

---

```
/* Create a reliable, stream socket using TCP */  
if ((sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)  
    DieWithError("socket() failed");
```

## Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

## Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
  - a. Accept new connection
  - b. Communicate
  - c. Close the connection



# TCP Client/Server Interaction

---

```
echoServAddr.sin_family    = AF_INET;           /* Internet address family */
echoServAddr.sin_addr.s_addr = inet_addr(servIP); /* Server IP address */
echoServAddr.sin_port      = htons(echoServPort); /* Server port */
```

```
if (connect(sock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0)
    DieWithError("connect() failed");
```

## Client

1. Create a TCP socket
2. **Establish connection**
3. Communicate
4. Close the connection

## Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
  - a. **Accept new connection**
  - b. Communicate
  - c. Close the connection



# TCP Client/Server Interaction

---

```
if ((clntSock=accept(servSock,(struct sockaddr *)&echoClntAddr,&clntLen)) < 0)  
    DieWithError("accept() failed");
```

## Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

## Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
  - a. Accept new connection
  - b. Communicate
  - c. Close the connection



# TCP Client/Server Interaction

---

```
echoStringLen = strlen(echoString);      /* Determine input length */  
  
/* Send the string to the server */  
if (send(sock, echoString, echoStringLen, 0) != echoStringLen)  
    DieWithError("send() sent a different number of bytes than expected");
```

## Client

1. Create a TCP socket
2. Establish connection
3. **Communicate**
4. Close the connection

## Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
  - a. **Accept new connection**
  - b. **Communicate**
  - c. **Close the connection**





# TCP Client/Server Interaction

---

```
/* Receive message from client */  
if ((recvMsgSize = recv(clntSocket, echoBuffer, RCVBUFSIZE, 0)) < 0)  
    DieWithError("recv() failed");
```

## Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

## Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
  - a. Accept new connection
  - b. Communicate
  - c. Close the connection



# TCP Client/Server Interaction

---


`close(sock);`

## Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. **Close the connection**

`close(clntSocket)`

## Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly: 
  - a. Accept new connection
  - b. Communicate
  - c. **Close the connection**



# TCP Tidbits

---

- Client must know the server's address and port
- Server only needs to know its own port
- No correlation between `send()` and `recv()`

**Client**      **Server**

`send("Hello Bob")`

`recv() -> "Hello "`

`recv() -> "Bob"`

`send("Hi ")`

`send("Jane")`

`recv() -> "Hi Jane"`



# Closing a Connection

---

- `close()` used to delimit communication
- Analogous to EOF

## Echo Client

`send(string)`

while (not received entire string)

`recv(buffer)`

`print(buffer)`

`close(socket)`

## Echo Server

`recv(buffer)`

while(client has not closed connection)

`send(buffer)`

`recv(buffer)`

`close(client socket)`



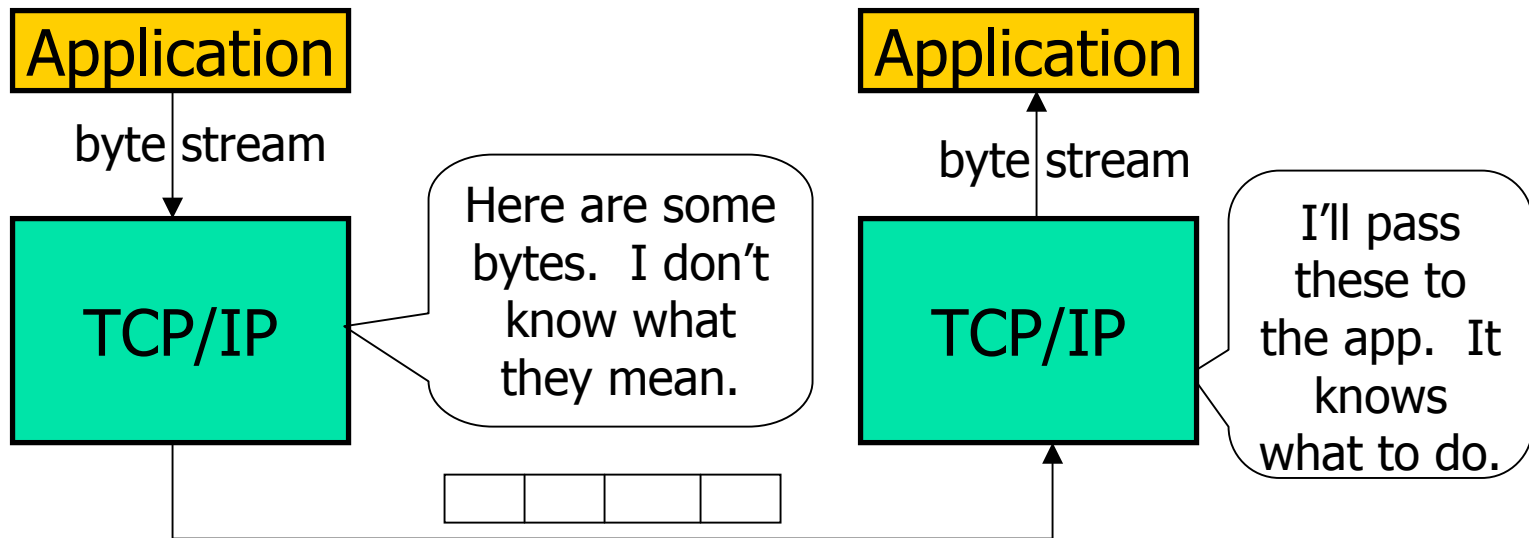
# Constructing Messages

---

...beyond simple strings

# TCP/IP Byte Transport

- TCP/IP protocols transports **bytes**



- Application protocol provides semantics



# Application Protocol

---

- Encode information in bytes
- Sender and receiver must agree on semantics
- Data encoding
  - Primitive types: strings, integers, and etc.
  - Composed types: message with fields



# Primitive Types

---

- String

- Character encoding: ASCII, Unicode, UTF
- Delimit: length vs. termination character

	0	77	0	111	0	109	0	10
	M		o		m		\n	
3	77		111		109			





# Primitive Types

---

- Integer

- Strings of character encoded decimal digits

49	55	57	57	56	55	48	10
'1'	'7'	'9'	'9'	'8'	'7'	'0'	\n

- Advantage:
  1. Human readable
  2. Arbitrary size
- Disadvantage:
  1. Inefficient
  2. Arithmetic manipulation



# Primitive Types

---

- Integer

- Native representation

Little-Endian	0	0	92	246	4-byte two's-complement integer
	23,798				
Big-Endian	246	92	0	0	

- Network byte order (Big-Endian)

- Use for multi-byte, binary data exchange
    - htonl(), htons(), ntohs(), ntohl()



# Message Composition

---

- Message composed of fields
  - Fixed-length fields

integer	short	short
---------	-------	-------

- Variable-length fields

M	i	k	e		1	2	\n
---	---	---	---	--	---	---	----

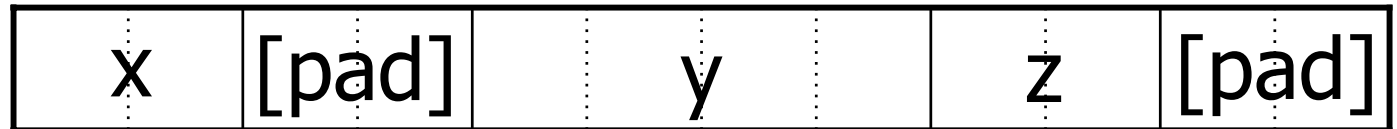


# “Beware the bytes of padding” -- Julius Caesar, Shakespeare

---

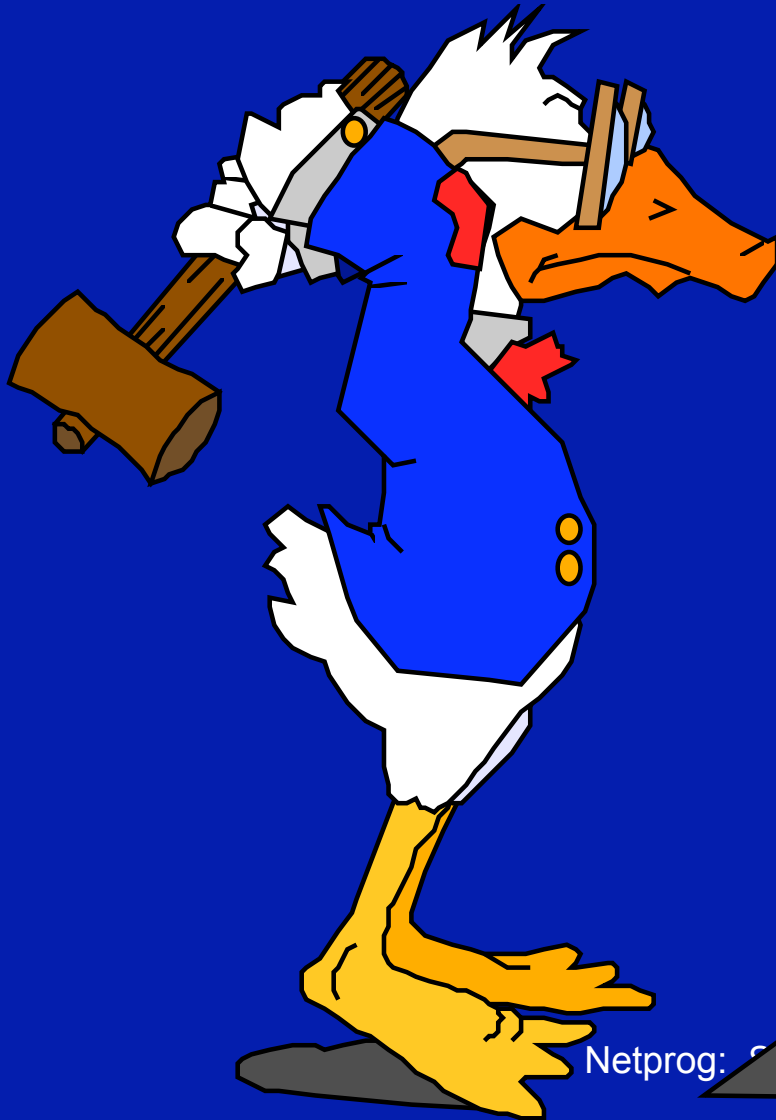
- Architecture alignment restrictions
- Compiler pads structs to accommodate

```
struct tst {  
    short x;  
    int y;  
    short z;  
};
```

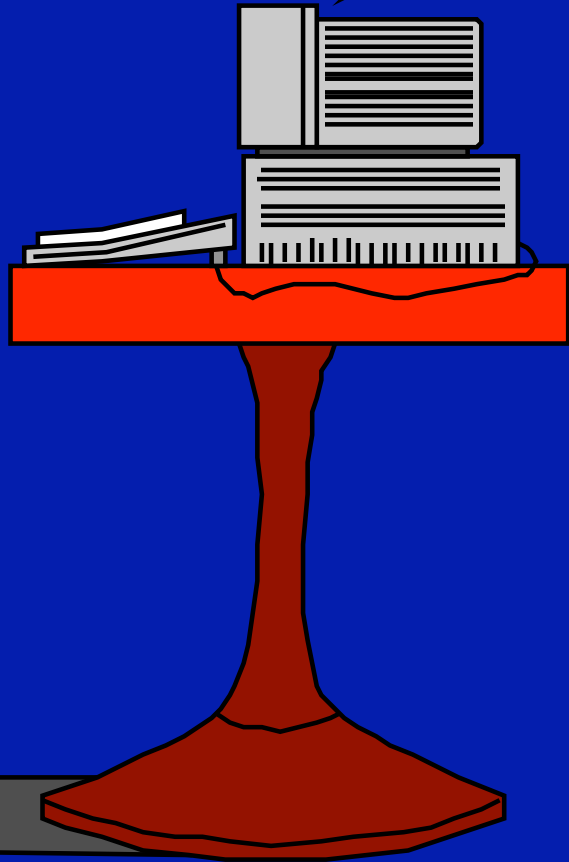


- Problem: Alignment restrictions vary
- Solution: 1) Rearrange struct members  
2) Serialize struct by-member

# Sockets Programming



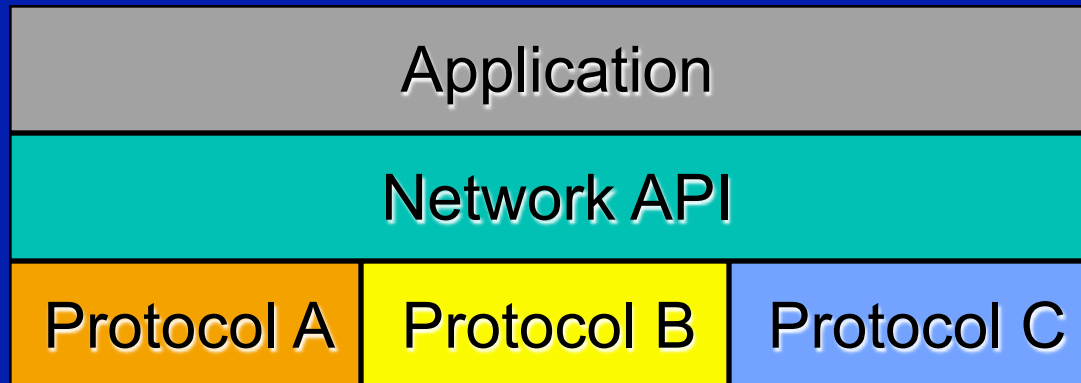
Socket to me!



Netprog: 5

# Network Application Programming Interface (API)

- The services provided (often by the operating system) that provide the interface between application and protocol software.



# Network API wish list

- Generic Programming Interface.
- Support for message oriented and connection oriented communication.
- Work with existing I/O services (when this makes sense).
- Operating System independence.
- Presentation layer services

# Generic Programming Interface

- Support multiple communication protocol suites (families).
- Address (endpoint) representation independence.
- Provide special services for Client and Server?



# TCP/IP

- TCP/IP does not include an API definition.
- There are a variety of APIs for use with TCP/IP:
  - Sockets
  - TLI, XTI
  - Winsock
  - MacTCP

# Functions needed:

- Specify local and remote communication endpoints
- Initiate a connection
- Wait for incoming connection
- Send and receive data
- Terminate a connection gracefully
- Error handling

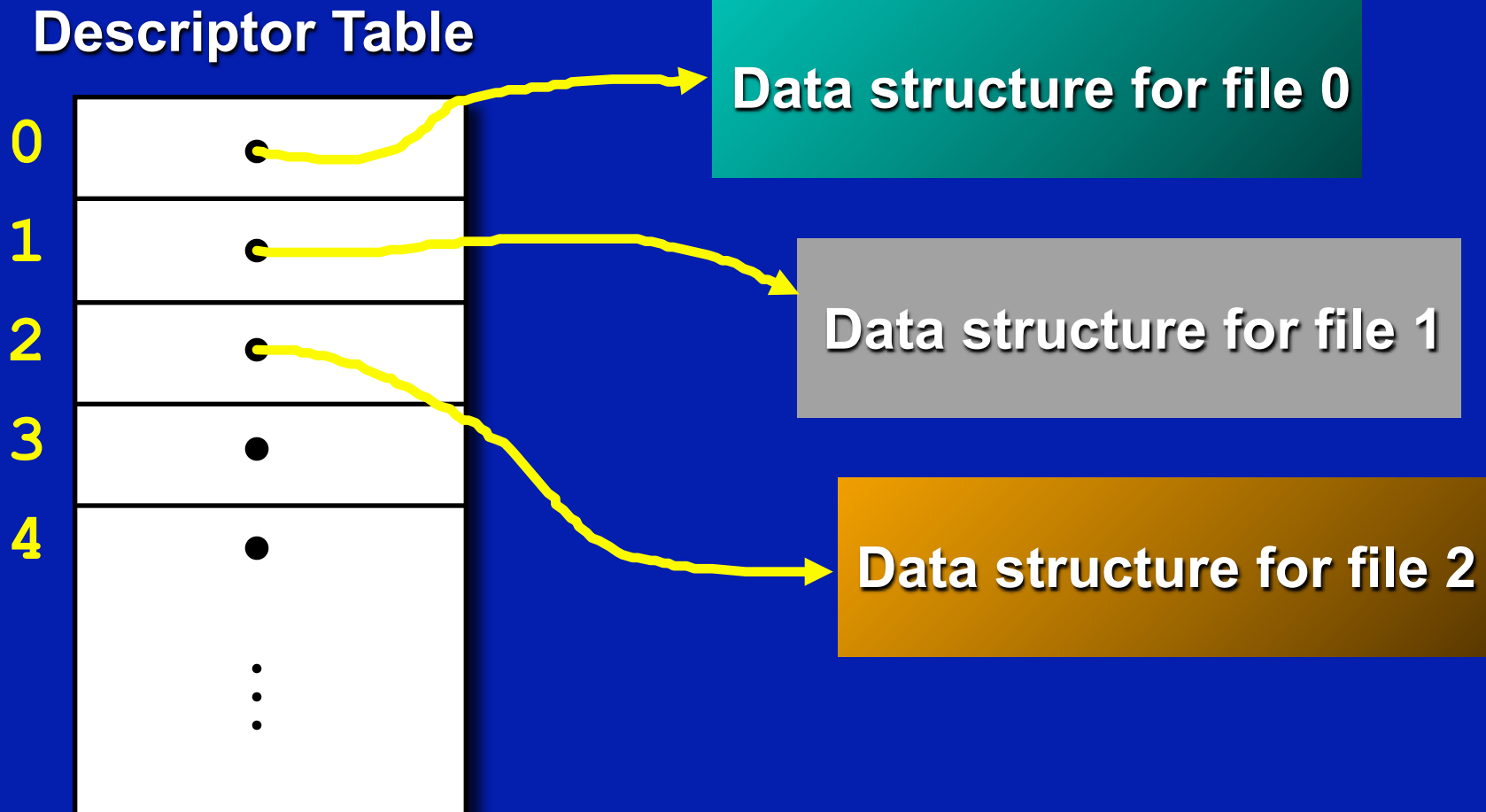
# Berkeley Sockets

- Generic:
  - support for multiple protocol families.
  - address representation independence
- Uses existing I/O programming interface as much as possible.

# Socket

- A socket is an abstract representation of a communication endpoint.
- Sockets work with Unix I/O services just like files, pipes & FIFOs.
- Sockets (obviously) have special needs:
  - establishing a connection
  - specifying communication endpoint addresses

# Unix Descriptor Table



# Socket Descriptor Data Structure

Descriptor Table

0	•
1	•
2	•
3	•
4	•
	•
	•
	•

Family: PF\_INET  
Service: SOCK\_STREAM  
Local IP: 111.22.3.4  
Remote IP: 123.45.6.78  
Local Port: 2249  
Remote Port: 3726

# Creating a Socket

```
int socket(int family, int type, int proto);
```

- `family` specifies the protocol family (**PF\_INET** for TCP/IP).
- `type` specifies the type of service (**SOCK\_STREAM**, **SOCK\_DGRAM**).
- `protocol` specifies the specific protocol (usually 0, which means *the default*).

# socket ()

- The `socket ()` system call returns a socket descriptor (small integer) or `-1` on error.
- `socket ()` allocates resources needed for a communication endpoint - but it does not deal with endpoint addressing.



# Specifying an Endpoint Address

- Remember that the sockets API is generic.
- There must be a generic way to specify endpoint addresses.
- TCP/IP requires an IP address and a port number for each endpoint address.
- Other protocol suites (families) may use other schemes.

# Necessary Background Information: POSIX data types

<code>int8_t</code>	signed 8bit int
<code>uint8_t</code>	unsigned 8 bit int
<code>int16_t</code>	signed 16 bit int
<code>uint16_t</code>	unsigned 16 bit int
<code>int32_t</code>	signed 32 bit int
<code>uint32_t</code>	unsigned 32 bit int

`u_char, u_short, u_int, u_long`

# More POSIX data types

`sa_family_t`

address family

`socklen_t`

length of struct

`in_addr_t`

IPv4 address

`in_port_t`

IP port number

# Generic socket addresses

```
struct sockaddr {  
    uint8_t      sa_len;   
    sa_family_t  sa_family;  
    char         sa_data[14];  
};
```

Used by kernel



- `sa_family` specifies the address type.
- `sa_data` specifies the address value.

# sockaddr

- An address that will allow me to use sockets to communicate with my kids.
- address type **AF\_DAVESKIDS**
- address values:

<b>Andrea</b>	<b>1</b>	<b>Mom</b>	<b>5</b>
<b>Jeff</b>	<b>2</b>	<b>Dad</b>	<b>6</b>
<b>Robert</b>	<b>3</b>	<b>Dog</b>	<b>7</b>
<b>Emily</b>	<b>4</b>		

# AF\_DAVESKIDS

- Initializing a sockaddr structure to point to Robert:

```
struct sockaddr robster;
```

```
robster.sa_family = AF_DAVESKIDS;
```

```
robster.sa_data[0] = 3;
```

Really old picture! →

Netprog: Sockets API



# AF\_INET

- For AF\_DAVESKIDS we only needed 1 byte to specify the address.
- For AF\_INET we need:
  - 16 bit port number
  - 32 bit IP address

**IPv4 only!**



# struct sockaddr\_in (IPv4)

```
struct sockaddr_in {
    uint8_t          sin_len;
    sa_family_t      sin_family;
    in_port_t        sin_port;
    struct in_addr    sin_addr;
    char             sin_zero[8];
};
```

*A special kind of sockaddr structure*



# struct in\_addr

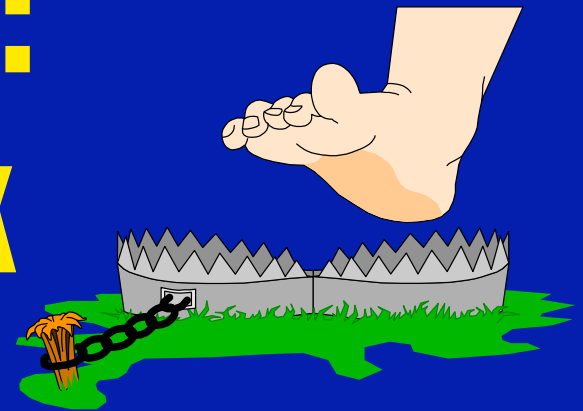
```
struct in_addr {  
    in_addr_t    s_addr;  
};
```

`in_addr` just provides a name for the 'C' type associated with IP addresses.

# Network Byte Order

- All values stored in a `sockaddr_in` must be in network byte order.
  - `sin_port` a TCP/IP port number.
  - `sin_addr` an IP address.

**Common Mistake:  
Ignoring Network  
Byte Order**



# Network Byte Order Functions

'h' : host byte order

'n' : network byte order

's' : short (16bit)

'l' : long (32bit)

```
uint16_t htons (uint16_t) ;  
uint16_t ntohs (uint16_t) ;
```

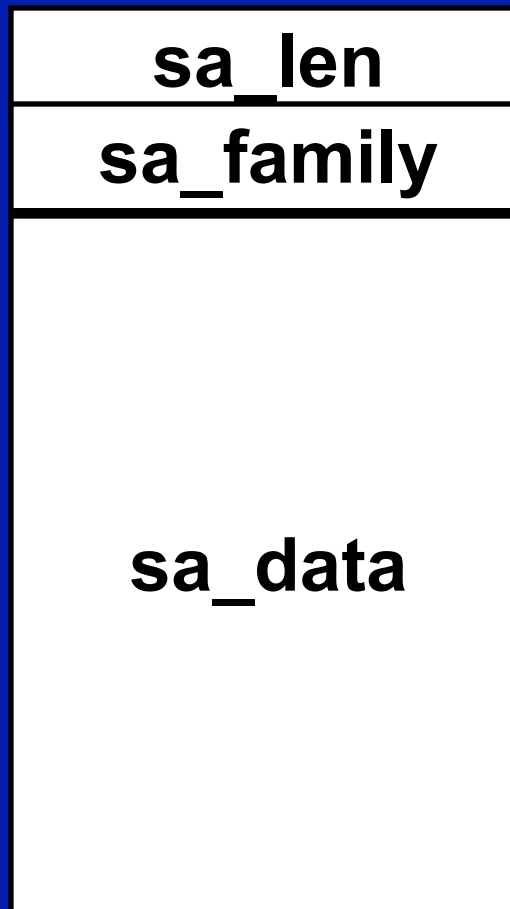
```
uint32_t htonl (uint32_t) ;  
uint32_t ntohl (uint32_t) ;
```

# TCP/IP Addresses

- We don't need to deal with `sockaddr` structures since we will only deal with a real protocol family.
- We can use `sockaddr_in` structures.

**BUT:** The C functions that make up the sockets API expect structures of type `sockaddr`.

# sockaddr



# sockaddr\_in



# Assigning an address to a socket

- The `bind()` system call is used to assign an address to an existing socket.

```
int bind( int sockfd,  
          const struct sockaddr *myaddr,  
          int addrlen);  
const! →
```

- `bind` returns 0 if successful or -1 on error.

# bind()

- calling `bind()` assigns the address specified by the `sockaddr` structure to the socket descriptor.
- You can give `bind()` a `sockaddr_in` structure:

```
bind( mysock,  
      (struct sockaddr*) &myaddr,  
      sizeof(myaddr) );
```

# bind() Example

```
int mysock, err;
struct sockaddr_in myaddr;

mysock = socket(PF_INET, SOCK_STREAM, 0);
myaddr.sin_family = AF_INET;
myaddr.sin_port = htons( portnum );
myaddr.sin_addr = htonl( ipaddress);

err=bind(mysock, (sockaddr *) &myaddr,
        sizeof(myaddr));
```



# Uses for `bind()`

- There are a number of uses for `bind()`:
  - Server would like to bind to a well known address (port number).
  - Client can bind to a specific port.
  - Client can ask the O.S. to assign *any available* port number.

# Port schmort - who cares ?

- Clients typically don't care what port they are assigned.
- When you call bind you can tell it to assign you any available port:

```
myaddr.port = htons(0);
```

# What is my IP address ?

- How can you find out what your IP address is so you can tell `bind()` ?
- There is no realistic way for you to know the right IP address to give `bind()` - what if the computer has multiple network interfaces?
- specify the IP address as: `INADDR_ANY`, this tells the OS to take care of things.

# IPv4 Address Conversion

```
int inet_aton( char *, struct in_addr *);
```

Convert ASCII dotted-decimal IP address to network byte order 32 bit value. Returns 1 on success, 0 on failure.

```
char *inet_ntoa(struct in_addr);
```

Convert network byte ordered value to ASCII dotted-decimal (a string).

# Other socket system calls

- General Use

- `read()`
- `write()`
- `close()`

- Connection-oriented (TCP)

- `connect()`
- `listen()`
- `accept()`

- Connectionless (UDP)

- `send()`
- `recv()`

# Socket Hijacking

**Author:**

**Neelay S. Shah**

Senior Software Security Consultant  
Foundstone Professional Services

**Rudolph Araujo**

Technical Director  
Foundstone Professional Services

### Abstract

Sockets are one of the most widely used inter-process communication primitives for clientserver applications due to a combination of the following factors. Sockets:

- Allow for bi-directional communication
- Allow processes to communicate across the network
- Are supported by most operating systems

What application developers need to be aware of is that attackers can target these same client-server applications by "hijacking" the server socket. Insecurely bound server sockets allow an attacker to bind his / her own socket on the same port, gaining control of the client connections and ultimately allowing the attacker to successfully steal sensitive application user information as well as launch denial of service attacks against the application server.

In this white paper we discuss the socket hijacking vulnerability on Windows, the impact of the vulnerability and what it takes to successfully exploit the vulnerability. We also review existing mitigating factors, the cause of the vulnerability as well as its remediation.

This white paper is intended towards all software developers, architects, testers and system administrators.

Foundstone has released a free tool "Foundstone Socket Security Auditor" which identifies the insecurely bound sockets on the local system. The free tool can be found at <http://www.foundstone.com/us/resources-free-tools.asp>.

### Discussion

Sockets are identified by an IP address and port number. Port number can be in the range of 0 to 65535 whereas the IP address can be any of the underlying IP addresses associated with the system including the loopback address. The socket library also supports a wildcard IP address (`INADDR_ANY`) that binds the socket to the specified port on all underlying IP addresses associated with the system. This feature is extremely attractive (and hence widely used) from an application development point of view for the following reasons:

- The application developer does not need to write code to programmatically enumerate the underlying IP addresses (associated with the system) and then use one or more of them to bind the listening server socket.
- In scenarios where the server has multiple network routable IP addresses, there is no additional overhead needed for exchanging the server's listening IP address with the client. The client could use any one of the server's network routable address and connect successfully to the server.

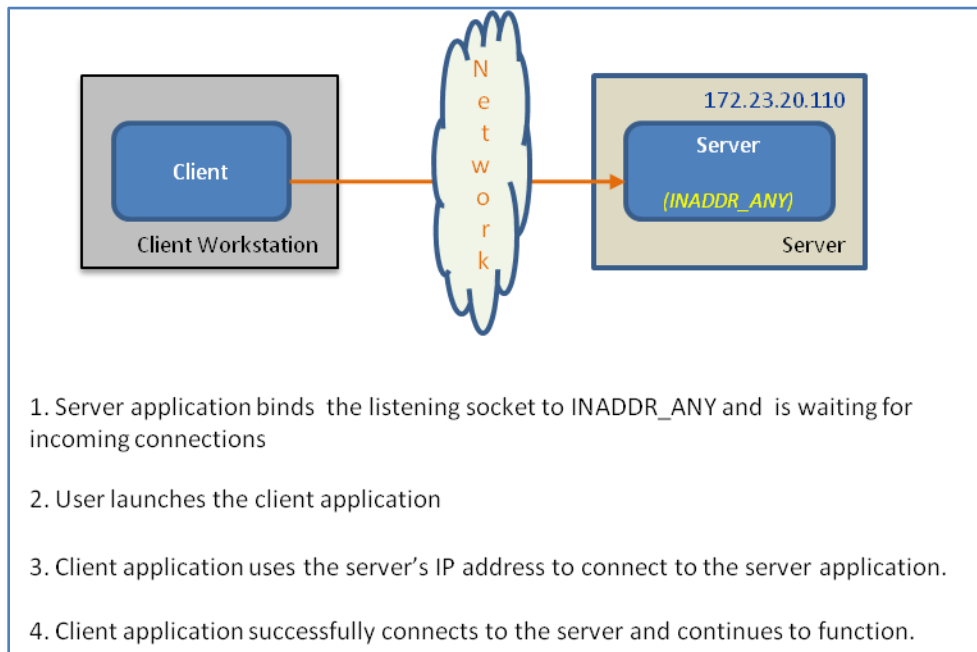


Figure 1: Setup of a typical client-server application communicating using sockets

However, it is possible to bind more than one socket to the same port. For instance, there could be an application server with a listening socket bound to `INADDR_ANY:9000` and another malicious application server with its listening socket bound to `172.23.20.1101:9000`. Note that both the applications are running on

<sup>1</sup> Assuming 172.23.20.110 is the IP addresses associated with the system.



## Socket Hijacking

the same system, the only difference (as far as their listener sockets are concerned) is the binding of the listener socket. The legitimate application server has bound its listening socket to the wildcard IP address (`INADDR_ANY`) whereas the malicious application server has bound its listening socket to a specific IP address (172.23.20.110).

When the client initiates a connection to the server, the client needs to use the routable address (172.23.20.110) and the port (9000) to connect to the server. When the connection request reaches the server, it is the responsibility of the network stack on the server to forward the connection to the listener. Now there are two sockets listening on the same port (9000), and the network stack can forward the connection to only one of the listening sockets. Thus, the network stack needs to resolve this conflict and choose one of the two sockets to forward the connection to.

For this, the network stack inspects the incoming client request which is targeted for 172.23.20.110:9000. Based on this information, the network stack resolves in favor of the malicious application since it had bound its listening socket specifically on 172.23.20.110. Thus the malicious application gets the client connection and can communicate further with the client. This is referred to as "Socket Hijacking" i.e. the malicious application has successfully hijacked the legitimate application's listener socket.

The following figure illustrates the client-server communication setup in the event of socket hijacking:

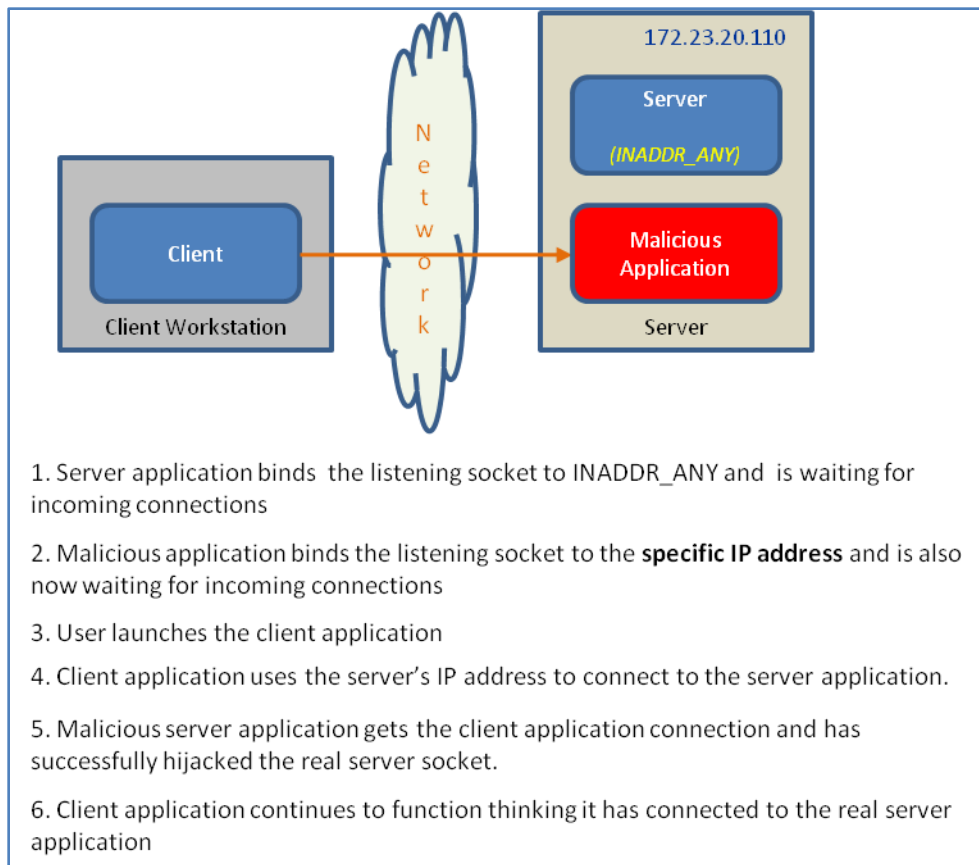


Figure 2: Client-Server communication setup in a "socket hijacking" scenario

### Impact of the vulnerability

Now that we understand and have discussed "socket hijacking" in detail, let's turn our focus towards the impact of the socket hijacking vulnerability; or in other words what damage an attacker can perform by exploiting the socket hijacking vulnerability.

Hijacking the listener socket of the legitimate server essentially allows the attacker to setup a "**spoof server**" and hijack client connections without having to poison the client application in any way i.e. the client application still connects to the same IP address and the port as before however the attacker gets hold of the client connection. Having received the client connection, the attacker will then be in a position to potentially carry out much more damaging things such as:

- Information Disclosure – Depending on the transport security primitives and the actions the client and the server carry out based on the messages on the socket, the attacker could gain knowledge of sensitive data such as user credentials and even launch man-in-the-middle attacks.
- Denial of Service – The real server has no notification of the client connection and as such the attacker would be successful in causing denial of service to legitimate client(s).

### Exploiting the vulnerability

So the next question is: "What does the attacker need in order to successfully exploit this vulnerability?"

Following are the key considerations and the mitigating factors with respect to successful exploitation of this vulnerability.

- The attacker needs to have sufficient access to the system with the vulnerable application. The attacker does not need to have privileged access but needs to be able to execute his malicious application on the system.
- On Windows Server 2003 and later a default ACL is applied to all sockets and as such a limited rights user cannot hijack a socket opened by a different user unless the application explicitly used an insecure ACL while creating the socket.
- Ports 0-1023 are privileged ports on Windows XP SP2 & later. On these operating systems, the attacker would need administrator/super-user privileges to hijack sockets which are bound to ports in the range 0-1023.

### Identifying the vulnerability

The vulnerability is introduced due to binding the socket insecurely. Let us look at the signature of insecure invocation of the "bind" API which is used to bind the socket to the underlying IP address and port. Since the socket is bound to wildcard IP Address (`INADDR_ANY`), this code snippet is susceptible to "socket hijacking" on Windows

```
SOCKET sListener = ::socket(AF_INET, SOCK_STREAM, 0);
//Check for error return code

sockaddr_in service;
service.sin_family = AF_INET;
service.sin_addr.S_un.S_addr = ::htonl(INADDR_ANY);
service.sin_port = htons(9000);

int iRet = ::bind(sListener, (sockaddr*)&service, sizeof(service));
//Check for error return code
```

### Remediating the vulnerability

Listener sockets must be bound securely by turning on the exclusive address use option

(SO\_EXCLUSIVEADDRUSE) on the socket so that an attacker cannot hijack the server socket. The following code snippet shows the secure binding of a listener socket:

```
SOCKET sListener = ::socket(AF_INET, SOCK_STREAM, 0);
//Check for error return code

sockaddr_in service;
service.sin_family = AF_INET;
service.sin_addr.S_un.S_addr = ::htonl(INADDR_ANY);
service.sin_port = htons(9000);

int iValLen = sizeof(BOOL);
BOOL bExclusiveUseAddr = TRUE;
int iFail = ::setsockopt(sTCPSTerver, SOL_SOCKET, SO_EXCLUSIVEADDRUSE, (char
*)&bExclusiveUseAddr, iValLen);
//Check for error return code

int iRet = ::bind(sListener, (sockaddr*) &service, sizeof(service));
//Check for error return code
```

### Acknowledgements

Rudolph Araujo provided significant support with reviewing the white paper.

### About Foundstone Professional Services

Foundstone® Professional Services, a division of McAfee. Inc., offers expert services and education to help organizations continuously and measurably protect their most important assets from the most critical threats. Through a strategic approach to security, Foundstone identifies and implements the right balance of technology, people, and process to manage digital risk and leverage security investments more effectively. The company's professional services team consists of recognized security experts and authors with broad security experience with multinational corporations, the public sector, and the US military.

### References

1. Socket Hijacking - Chapter 15, Writing Secure Code Vol. 2, Michael Howard et. al. ISBN: 0-7356-1722-8
2. Using SO\_REUSEADDR and SO\_EXCLUSIVEADDRUSE - [http://msdn2.microsoft.com/en-us/library/ms740621\(VS.85\).aspx](http://msdn2.microsoft.com/en-us/library/ms740621(VS.85).aspx)

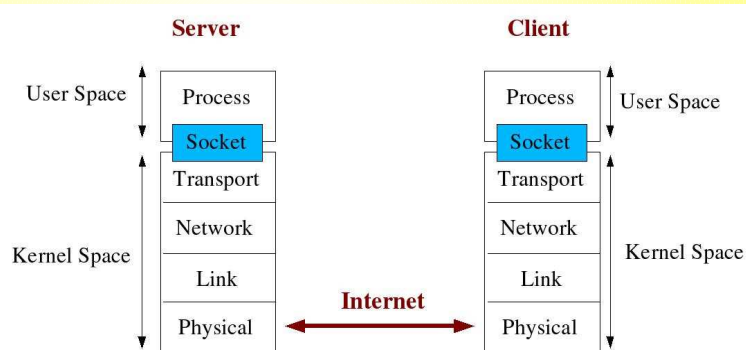
# Socket Programming

Kameswari Chebrolu  
Dept. of Electrical Engineering, IIT Kanpur

## What is a socket?

- Socket: An interface between an application process and transport layer
  - The application process can send/receive messages to/from another application process (local or remote) via a socket
- In Unix jargon, a socket is a file descriptor – an integer associated with an open file
- Types of Sockets: **Internet Sockets**, unix sockets, X.25 sockets etc
  - Internet sockets characterized by IP Address (4 bytes) and port number (2 bytes)

## Socket Description



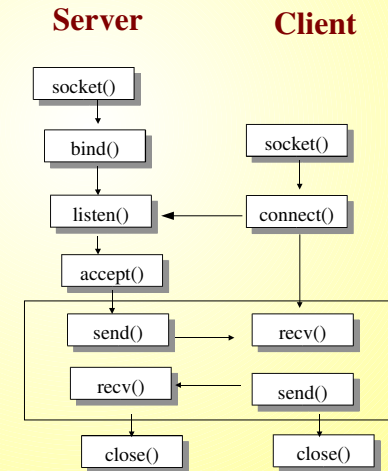
## Types of Internet Sockets

- Stream Sockets (SOCK\_STREAM)
  - Connection oriented
  - Rely on TCP to provide reliable two-way connected communication
- Datagram Sockets (SOCK\_DGRAM)
  - Rely on UDP
  - Connection is unreliable

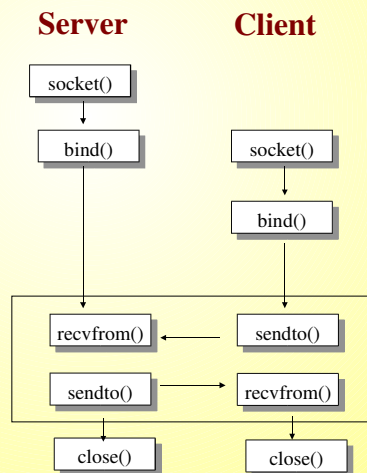
## Background

- Two types of “Byte ordering”
  - Network Byte Order: High-order byte of the number is stored in memory at the lowest address
  - Host Byte Order: Low-order byte of the number is stored in memory at the lowest address
  - Network stack (TCP/IP) expects Network Byte Order
- Conversions:
  - htons() - Host to Network Short
  - htonl() - Host to Network Long
  - ntohs() - Network to Host Short
  - ntohl() - Network to Host Long

## Connection Oriented Protocol



## Connectionless Protocol



## socket() -- Get the file descriptor

- `int socket(int domain, int type, int protocol);`
  - domain should be set to `AF_INET`
  - type can be `SOCK_STREAM` or `SOCK_DGRAM`
  - set protocol to 0 to have socket choose the correct protocol based on type
  - `socket()` returns a socket descriptor for use in later system calls or -1 on error

## socket structures

- struct sockaddr: Holds socket address information for many types of sockets

```
struct sockaddr {
    unsigned short sa_family; //address family AF_XXX
    unsigned short sa_data[14]; //14 bytes of protocol addr
}
```

- struct sockaddr\_in: A parallel structure that makes it easy to reference elements of the socket address

```
struct sockaddr_in {
    short int     sin_family; // set to AF_INET
    unsigned short int sin_port; // Port number
    struct in_addr sin_addr; // Internet address
    unsigned char  sin_zero[8]; //set to all zeros
}
```

## Dealing with IP Addresses

- int inet\_aton(const char \*cp, struct in\_addr \*inp);

- Example usage:

```
struct sockaddr_in my_addr;
my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(MYPORT);
inet_aton("10.0.0.5",&(my_addr.sin_addr));
memset(&(my_addr.sin_zero),'\0',8);
```

- inet\_aton() gives non-zero on success and zero on failure

- To convert binary IP to string: inet\_ntoa()

```
printf("%s",inet_ntoa(my_addr.sin_addr));
```

## bind() - what port am I on?

- Used to associate a socket with a port on the local machine
  - The port number is used by the kernel to match an incoming packet to a process
- int bind(int sockfd, struct sockaddr \*my\_addr, int addrlen)
  - sockfd is the socket descriptor returned by socket()
  - my\_addr is pointer to struct sockaddr that contains information about your IP address and port
  - addrlen is set to sizeof(struct sockaddr)
  - returns -1 on error
- my\_addr.sin\_port = 0; //choose an unused port at random
- my\_addr.sin\_addr.s\_addr = INADDR\_ANY; //use my IP addr

## connect() - Hello!

- Connects to a remote host
- int connect(int sockfd, struct sockaddr \*serv\_addr, int addrlen)
  - sockfd is the socket descriptor returned by socket()
  - serv\_addr is pointer to struct sockaddr that contains information on destination IP address and port
  - addrlen is set to sizeof(struct sockaddr)
  - returns -1 on error
- At times, you don't have to bind() when you are using connect()



## listen() - Call me please!

- Waits for incoming connections
- `int listen(int sockfd, int backlog);`
  - `sockfd` is the socket file descriptor returned by `socket()`
  - `backlog` is the number of connections allowed on the incoming queue
  - `listen()` returns -1 on error
  - Need to call `bind()` before you can `listen()`

## accept() - Thank you for calling !

- `accept()` gets the pending connection on the port you are `listen()`ing on
- `int accept(int sockfd, void *addr, int *addrlen);`
  - `sockfd` is the listening socket descriptor
  - information about incoming connection is stored in `addr` which is a pointer to a local `struct sockaddr_in`
  - `addrlen` is set to `sizeof(struct sockaddr_in)`
  - `accept` returns *a new socket file descriptor* to use for this accepted connection and -1 on error

## send() and recv() - Let's talk!

- The two functions are for communicating over stream sockets or connected datagram sockets.
- `int send(int sockfd, const void *msg, int len, int flags);`
  - `sockfd` is the socket descriptor you want to send data to (returned by `socket()` or got with `accept()`)
  - `msg` is a pointer to the data you want to send
  - `len` is the length of that data in bytes
  - set flags to 0 for now
  - `send()` returns the number of bytes actually sent (may be less than the number you told it to send) or -1 on error

## send() and recv() - Let's talk!

- `int recv(int sockfd, void *buf, int len, int flags);`
  - `sockfd` is the socket descriptor to read from
  - `buf` is the buffer to read the information into
  - `len` is the maximum length of the buffer
  - set flags to 0 for now
  - `recv()` returns the number of bytes actually read into the buffer or -1 on error
  - If `recv()` returns 0, the remote side has closed connection on you

## sendto() and recvfrom() - DGRAM style

- `int sendto(int sockfd, const void *msg, int len, int flags, const struct sockaddr *to, int tolen);`
  - *to* is a pointer to a struct `sockaddr` which contains the destination IP and port
  - *tolen* is `sizeof(struct sockaddr)`
- `int recvfrom(int sockfd, void *buf, int len, int flags, struct sockaddr *from, int *fromlen);`
  - *from* is a pointer to a local struct `sockaddr` that will be filled with IP address and port of the originating machine
  - *fromlen* will contain length of address stored in *from*

## close() - Bye Bye!

- `int close(int sockfd);`
  - Closes connection corresponding to the socket descriptor and frees the socket descriptor
  - Will prevent any more sends and recvs

## Miscellaneous Routines

- `int getpeername(int sockfd, struct sockaddr *addr, int *addrlen);`
  - Will tell who is at the other end of a connected stream socket and store that info in *addr*
- `int gethostname(char *hostname, size_t size);`
  - Will get the name of the computer your program is running on and store that info in *hostname*

## Miscellaneous Routines

- `struct hostent *gethostbyname(const char *name);`

```
struct hostent {
    char    *h_name;      //official name of host
    char    **h_aliases; //alternate names for the host
    int     h_addrtype;  //usually AF_INET
    int     h_length;    //length of the address in bytes
    char    **h_addr_list; //array of network addresses for the host
}
#define h_addr h_addr_list[0]
```

- Example Usage:

```
struct hostent *h;
h = gethostbyname("www.iitk.ac.in");
printf("Host name : %s\n", h->h_name);
printf("IP Address: %s\n", inet_ntoa(*(struct in_addr *)h->h_addr));
```

## Summary

- Sockets help application process to communicate with each other using standard Unix file descriptors
- Two types of Internet sockets: `SOCK_STREAM` and `SOCK_DGRAM`
- Many routines exist to help ease the process of communication

## References

- Books:
  - Unix Network Programming, volumes 1-2 by W. Richard Stevens.
  - TCP/IP Illustrated, volumes 1-3 by W. Richard Stevens and Gary R. Wright
- Web Resources:
  - Beej's Guide to Network Programming
    - [www.ecst.csuchico.edu/~beej/guide/net/](http://www.ecst.csuchico.edu/~beej/guide/net/)

# IPv6 for Developers and socket address structure

Kapil

# Presentation Agenda

- Socket API extensions
- Name Service API changes
- Tools and Recommendations
- Miscellaneous Topics

# Programming with IPv6

- This presentation geared towards C programming
- Java programmers can go to sleep
  - Java API is already IP version agnostic

# Programming with IPv6

- Most applications will require minimal changes to support IPv6
  - Change the socket, name-service, and UI
- Network-intensive applications will require a bit more
  - IDS, firewall, network/security analysis tools
  - Security tools that use addresses in protocol

# A Few Notes about Java and IPv6

- Class `InetAddress` will handle IPv4 & IPv6 addr
  - Methods that support IPv6 features
    - `isLinkLocalAddress()`
  - Methods that are version-agnostic
    - `toString()`, `getByAddress()`, `getAllByName()`, etc
  - `Inet4Address` and `Inet6Address` are subclasses
- Socket calls all use `InetAddress`
- Unless you are doing something specific to IPv4, not porting is necessary for Java code
  - Underlying OS must support IPv6



# Socket API Extensions

- Basic **socket()** system call is unchanged
  - Just a new protocol family for IPv6
    - `s = socket(AF_INET, SOCK_DGRAM, 0); [IPv4]`
    - `s = socket(AF_INET6, SOCK_DGRAM, 0); [IPv6]`
  - IPv4-only sockets continue to work as they always have
  - System calls that bind or receive address to/from IPv6 sockets must use IPv6 socket addresses
    - `bind()`, `connect()`, `sendmsg()`, `sendto()`
    - `accept()`, `recvfrom()`, `recvmsg()`, `getpeername()`, `getsockname()`

# IPv6 Address Structure

- **struct in6\_addr** versus **struct in\_addr**
  - Usually defined in /usr/include/netinet/in.h
  - Often see *int* or *uint* to carry IPv4 addresses
    - Makes IP address variables harder to find in code

Officially:

```
struct in6_addr {  
    u_int8_t s6_addr[16];  
}
```

Often implemented as:

```
struct in6_addr {  
    union {  
        u_int8_t  u6_addr8[16];  
        u_int16_t u6_addr16[8];  
        u_int32_t u6_addr32[4];  
    } u6_addr;  
}  
  
#define s6_addr    u6_addr.u6_addr8  
#define s6_addr16 u6_addr.u6_addr16  
#define s6_addr32 u6_addr.u6_addr32
```

# struct in6\_addr

- A few useful constants and macros (<netinet/in.h>)
  - const struct in6\_addr **in6addr\_any**; /\* :: \*/
    - INADDR\_ANY is v4 equivalent
  - const struct in6\_addr **in6addr\_loopback**; /\* ::1 \*/
    - INADDR\_LOOPBACK is v4 equivalent
  - #define **INET6\_ADDRSTRLEN** 46
    - Longest string representation of IPv6 address
  - **IN6\_IS\_ADDR\_UNSPECIFIED(a)**
  - **IN6\_IS\_ADDR\_LOOPBACK(a)**
  - **IN6\_IS\_ADDR\_MULTICAST(a)**
  - **IN6\_IS\_ADDR\_LINKLOCAL(a)**
  - **IN6\_IS\_ADDR\_SITELOCAL(a)**
  - **IN6\_IS\_ADDR\_V4\_MAPPED(a)**
  - **IN6\_IS\_ADDR\_V4\_COMPAT(a)**
  - **IN6\_ARE\_ADDR\_EQUAL(a,b)**
  - Multicast scope macros

# IPv6 Socket Addresses

- New socket address structure defined for IPv6
  - Usually defined in `/usr/include/netinet/in.h`

BSD 4.3-based:

```
struct sockaddr_in6 {
    u_int16_t          sin6_family;
    u_int16_t          sin6_port;
    u_int32_t          sin6_flowinfo;
    struct sockaddr_in6 sin6_addr;
    u_int32_t          sin6_scope_id;
};
```

BSD 4.4-based:

```
#define SIN6_LEN
struct sockaddr_in6 {
    u_int8_t          sin6_len;
    u_int8_t          sin6_family;
    u_int16_t         sin6_port;
    u_int32_t         sin6_flowinfo;
    struct sockaddr_in6 sin6_addr;
    u_int32_t         sin6_scope_id;
};
```

# IPv6 Socket Addresses

- Most system calls that pass in or receive socket addresses use a generic (struct sockaddr \*)
  - Cast your specific type of sockaddr\_\* to a sockaddr before passing in/out
  - Generic struct sockaddr is not large enough to hold IPv6 socket address
  - Define new generic sockaddr\_storage which has enough space to hold largest sockaddr system supports
    - Has ss\_family member that overlaps sin\_family & sin6\_family
    - Usually defined in /usr/include/sys/socket.h

# IPv6 Socket Addresses

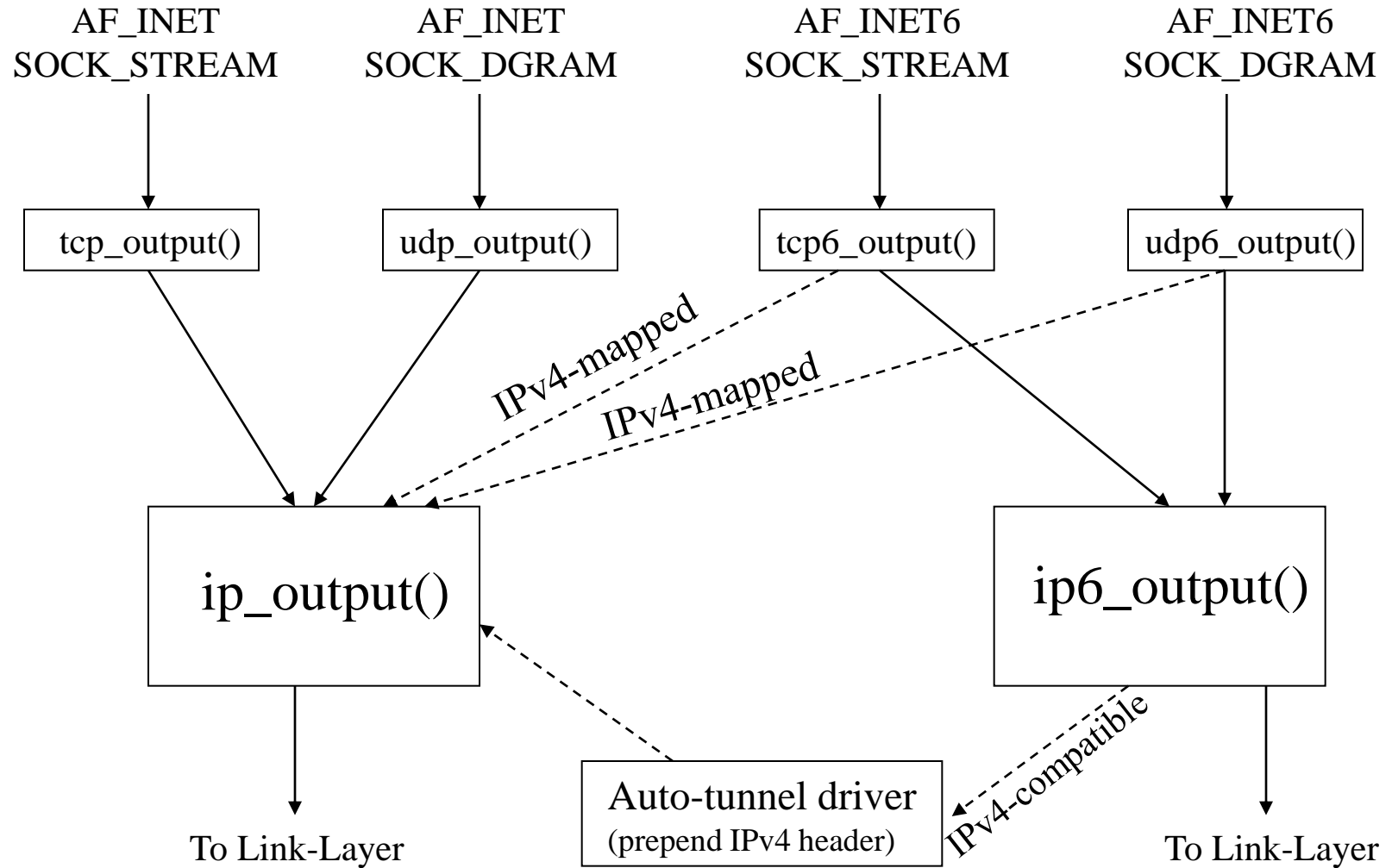
## sockaddr\_storage

```
struct sockaddr_storage ss;  
int ss_len;  
  
get_sock_addr((struct sockaddr *)&ss);  
  
switch (ss.ss_family) {  
    case AF_INET:  
        sin = (struct sockaddr_in *)&ss;  
        ss_len = sizeof(struct sockaddr_in);  
        break;  
    case AF_INET6:  
        sin6 = (struct sockaddr_in6 *)&ss;  
        ss_len = sizeof(struct sockaddr_in6);  
        break;  
    [...]  
}  
  
ret = bind(s, (struct sockaddr *)&ss, ss_len);
```

# IPv6 and IPv4 Interoperability

- An IPv6 socket can talk to and accept IPv4 connections
  - Assuming dual-stacks active
  - To connect to an IPv4 address via an IPv6 socket
    - Use IPv4-Mapped address (e.g. ::FFFF:192.168.0.1)
    - Use IPv4-Compatible address (e.g. ::192.168.0.1)
  - Accepting connections on an IPv6 socket
    - IPv4 connections will return IPv6 address as IPv4-mapped/compatible
- Can use `IN6_IS_ADDR_V4_MAPPED` to test
- Use `AF_INET6` sockets for applications that will support both IPv4 and IPv6

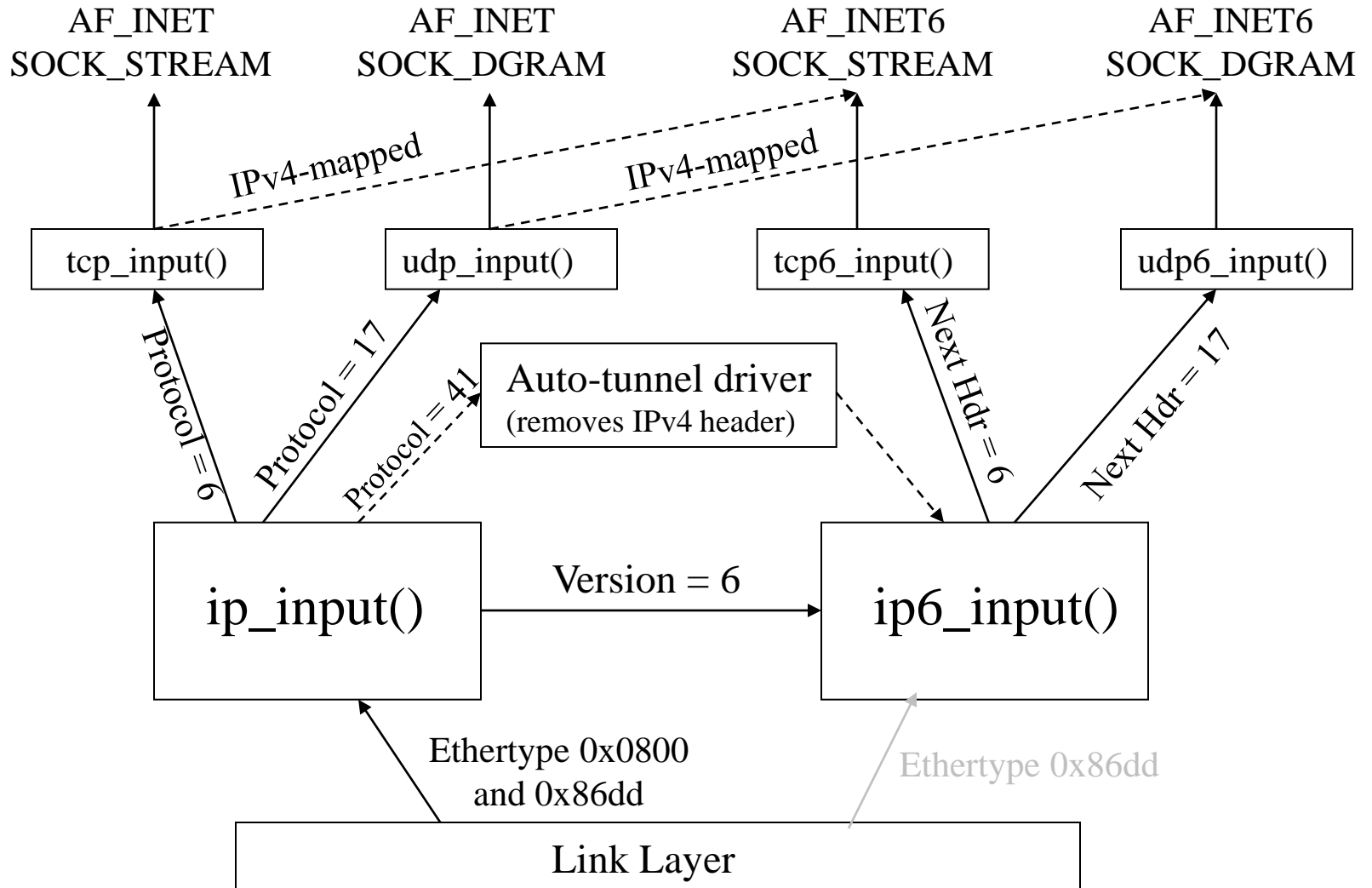
# Dual-Stacked Nodes: Sending IPv4 and IPv6 Packets





# Dual-Stacked Nodes

## Receiving IPv4 and IPv6 Packets



# IPv6 Socket Options

- Changing socket type
  - If an IPv6 application inherits a socket from a v4-only application and wants to make it a v6 socket

```
int addrform = PF_INET6;

setsockopt(s, IPPROTO_IPV6, IPV6_ADDRFORM,
           (char *)&addrform, sizeof(addrform));
```

- Changing Hop Limit

```
int hoplimit = 10;

setsockopt(s, IPPROTO_IPV6, IPV6_UNICAST_HOPS,
           (char *)&hoplimit, sizeof(hoplimit));
```

# IPv6 Socket Options

- Multicast Options
  - IPV6\_MULTICAST\_IF -- set interface (int)
  - IPV6\_MULTICAST\_HOPS -- set hop limit (int)
  - IPV6\_MULTICAST\_LOOP -- toggle loopback (int)
  - IPV6\_ADD\_MEMBERSHIP -- (struct ipv6\_mreq)
  - IPV6\_DROP\_MEMBERSHIP -- (struct ipv6\_mreq)

# Displaying and Interpreting IPv6 Addresses

- Replace `inet_ntoa()` and `inet_addr()` functions with protocol-agnostic versions
  - `inet_ntop()` -- network format to presentation format
    - `inet_ntop(int af, void *src, char *dst, int cnt)`
      - `src` is `in_addr`, `in6_addr`, etc
      - `dst` is char array of `cnt` bytes (`INET[6]_ADDRSTRLEN`)
  - `inet_pton()` -- presentation format to network format
    - `inet_pton(int af, const char *src, void *dst)`
      - `src` is string representation
      - `dst` is pointer to `in_addr`, `in6_addr`, etc.

# Hostname and Address Lookups

- DON'T USE `gethostbyname()/gethostbyaddr()`
  - Interface to address results is cumbersome with respect to address type
  - Not thread-safe
- `gethostbyname()` behavior can be changed with resolver flag
- `gethostbyname2()` allows you to specify an address family
- `gethostbyaddr()` already has address family

# Hostname and Address Lookups

- Preferred interface is protocol independent
  - getaddrinfo() and getnameinfo()
  - Allows multiple addresses with independent types
    - Addresses are returned as linked-list of type struct addrinfo

```
struct addrinfo {
    int    ai_flags;        /* AI_PASSIVE, AI_CANONNAME */
    int    ai_family;      /* PF_xxx */
    int    ai_socktype;    /* SOCK_xxx */
    int    ai_protocol;    /* 0 or IPPROTO_x for IPv4 & IPv6 */
    size_t ai_addrlen;     /* length of ai_addr */
    char  *ai_canonname;   /* canonical name for hostname */
    struct sockaddr *ai_addr; /* binary address */
    struct addrinfo *ai_next; /* next structure in linked list */
};
```

# Hostname and Address Lookups

- `getaddrinfo()`

- `getaddrinfo(const char *node, const char *service, const struct addrinfo *hints, struct addrinfo **results);`

```
struct addrinfo hints, *res, *res0;
int error;

memset(&hints, 0, sizeof(hints));
hints.ai_family = PF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
error = getaddrinfo("www.kame.net", "http", &hints, &res0);
[...]

for (res = res0; res; res = res->ai_next) {
    s = socket(res->ai_family,
              res->ai_socktype, res->ai_protocol);
    [...]
    error = connect(s, res->ai_addr, res->ai_addrlen);
    [...]
}
```

# Hostname and Address Lookups

- `getnameinfo()` -- inverse lookup
  - `getnameinfo(const struct sockaddr *sa, size_t sa_len, char *host, size_t hostlen, char *serv, size_t servlen, int flags);`



# Advanced Socket API for IPv6

## RFC 2292

- Details on header structures
  - IPv6 header
  - Extension Headers,
  - ICMPv6 headers
  - Neighbor Discovery message formats
- RAW sockets and ICMPv6 filters
- Ancillary data
  - How to get IPv6 Extension data from socket
- Specifying and receiving Packet Information
  - Src/Dst addr, in/out interface, in/out hop limit, next hop addr
- API for hop-by-hop, destination, routing options
- Future API for flow, PMTU, Neighbor reachability

# IPv6 Programming: UI Considerations

- Reading addresses -- use `inet_pton()` as needed
  - Configuration/Data files and user input need to change
    - Larger address strings -- larger buffers to read and parse
  - Address lookups will return multiple addresses per host
  - Do you need to handle [`<ipv6-addr>`]:`<port>` format?
  - Check for overflow
- Writing addresses -- use `inet_ntop()` as needed
  - Text or GUI output will require larger screen area
  - Building log messages will require larger buffers
  - Data file formats may need change (addr type and size)
    - Integration with databases or other processes

# General IPv6 Programming: Tools

- There are some good software tools publicly available that can automatically determine if an IPv4 program contains IP-specific calls, and suggest needed changes. A few are:
  - <http://msdn.microsoft.com/library/>
    - IPv6 Guide for Windows Sockets Applications
    - Checkv4.exe utility program
  - <http://www.sun.com/software/solaris/ipv6/>
    - IPv6 Socket Scrubber
  - [http://www.sun.com/software/solaris/ipv6/porting\\_guide\\_ipv6.pdf](http://www.sun.com/software/solaris/ipv6/porting_guide_ipv6.pdf)
    - Porting Networking Applications to the IPv6 APIs
  - Linux tools also available

# General IPv6 Programming: Recommendations

- Build application-specific address structure in the code.
  - This would typically be a structure that includes the address type, address data, and optionally address size. This allows a single structure for dealing with multiple address types.
- Build small set of functions that deal with these address structures
  - Functions may include: setting, comparing, printing, etc., address structures.
- Hostname lookups
  - Expect multiple addresses to be returned. This should be obvious for hosts with multiple IPv4 addresses, but account for several IP addresses (at least 2) per interface. Also, consider link-local, multicast, and anycast addresses.
- When replacing IPv4 addresses in code
  - Rename variables or structure members so that the compiler can help you find all instances of the address variable that need to be adjusted
- Use of "struct sockaddr\_storage" and cast to the appropriate sockaddr\_\*

# General IPv6 Programming: Recommendations

- When processing packets, look for:
  - IPv6 extension headers -- may need to skip for transport layer access
  - Tunneling of IPvX-in-IPvX (how many layers of encapsulation are sufficient to handle?)
  - BPF issues
- Write protocol-independent code
  - Will you be retired when IPv8 is deployed??

# IPv6 Miscellaneous Issues

- Libpcap and Berkeley Packet Filters (BPF)
  - Used by tcpdump, ethereal, etc.
  - Have supported IPv6 for a few years
    - But must be enabled in the build
  - tcpdump ip6
  - tcpdump net 2001:480:31:10::/64
  - Filter on port X -- verify that v4 & v6 will be processed
    - tcpdump -d port 22
    - Look for ldh[12] and compare to 0x86dd

# Checking BPF code for IPv6

```
root# tcpdump -d port 22
(000) ldh    [12]
(001) jeq    #0x86dd    jt 2   jf 10
(002) ldb    [20]
(003) jeq    #0x84      jt 6   jf 4
(004) jeq    #0x6       jt 6   jf 5
(005) jeq    #0x11     jt 6   jf 23
(006) ldh    [54]
(007) jeq    #0x16     jt 22  jf 8
(008) ldh    [56]
(009) jeq    #0x16     jt 22  jf 23
(010) jeq    #0x800    jt 11  jf 23
(011) ldb    [23]
(012) jeq    #0x84     jt 15  jf 13
(013) jeq    #0x6      jt 15  jf 14
(014) jeq    #0x11     jt 15  jf 23
(015) ldh    [20]
(016) jset   #0x1fff   jt 23  jf 17
(017) ldxb   4*([14]&0xf)
(018) ldh    [x + 14]
(019) jeq    #0x16     jt 22  jf 20
(020) ldh    [x + 16]
(021) jeq    #0x16     jt 22  jf 23
(022) ret    #96
(023) ret    #0
```