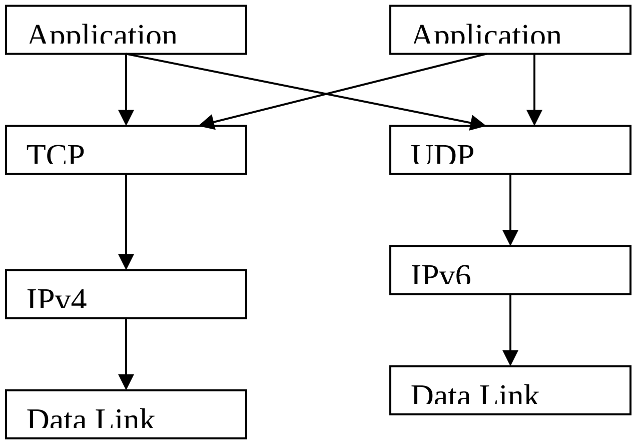
IPv4 and ipV6 INTEROPERABILITY

Till the time, IPv6 is established all over the world, there is a need for one to host dual stacks – that is both IPv4 and IPv6 are running concurrently as shown below:

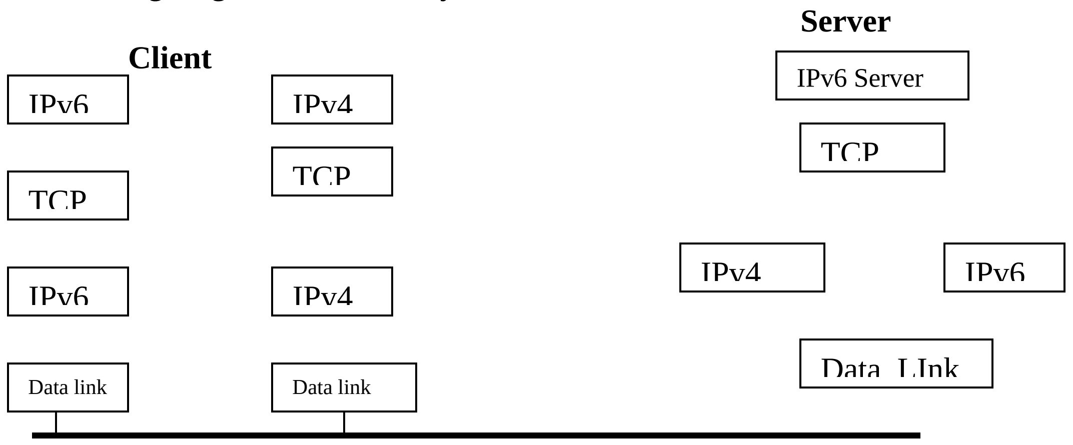
IPv4 , AF_INET **IPv6, AF_INET6**
sockaddr_in () **sockaddr_in6()**



While being so, the client can be on IPv4 or IPv6 format and the server must be able to accept both application.. Here the peculiarities that are involved when the client is on IPv4 and the server is on IPv6 format and when the client is IPv6 and the server is in IPv4 format are discussed.

IPv4 client and IPv6 server:

Following diagram shows the system when the client is either IPv4 or IPv6



Enet hdr	IPv4 Hdr	TCP Hdr	TCP Data

Enet Hdr	IPv6 hdr	TCP Hdr	TCP Datqa

The system considered is on Ethernet for the sake of simplicity. The server is running on dual host and the client can be of either IPv4 or IPv6. The server has created an IPv6 listening TCP socket. It is bound to the IPv6 wild card address and TCP port 8888.

To make connection, the client sends the SYN segment to the server. In this case it appears as Ethernet header (contains a type field 0x0800which identifies the frame as IPv4 frame), followed by IPv4 header (destination IP address in IPv4 format) and the TCP header that contains destination port.

IN case of IPv6 client connection, the clients sends SYN segment to the server. . In this case it appears as Ethernet header (contains a type field 0x86dd which identifies the frame as IPv6 frame), followed by IPv6 header (destination IP address in IPv6 format) and the TCP header that contains destination port.

The receiving datalink looks at the Ethernet type field and passes each frame to the appropriate IP module. The IPv4 module in conjunction with TCP module detects that the destination socket is an IPv6 socket and the source IPv4 address in the IPv4 header is converted into equivalent IPv4 mapped IPv6 address. That mapped address is returned to the IPv6 socket as the client's IPv6 address when accept returns to the server with IPv4 client connection. All remaining datagram for this connections are IPv4 datagram. When accept returns to the server with the IPv6 client connection, the client's IPv6 address does not change from whatever source address appears in the IPv6 header. All remaining datagram for this connections are IPv6 datagram

Steps that allow an IPv4 TCP client to communicate with an IPv6 server:

- The IPv6 server starts, creates an IPv6 listening socket, and we assume it binds the wildcards address to the socket.
- The IPv4 client calls **gethostbyname** and finds an A record for the server. The server host will have both A record and AAAA record, since it supports both protocols but the IPv4 client asks for only an A record.
- The client calls connect and the client host sends an IPv4 SYN to the server.
- The server host receives the IPv4 SYN directed to the IPv6 to the listening socket, sets a flag indicating that this connection is using IPv4 mapped IPv6 addresses and responds with an IPv4 SYN/ACK. When the connection is established, the address returned to the server by accept is the IPv4 mapped IPv6 address.
- All communication between this client and the server tak place using IPv4 datagram
- Both client and server explicitly do not know that they are communicating with different address schemes.

The scenario is similar for an IPv6 UDP server, but the address format can change for each datagram. If anIPv6 server receives a datagram from an IPv4 client, the address returned by **recvfrom** will be the client's IPv4 mapped IPv6 address. The server responds to this clients requests by calling **sendto** with the IPv4 mapped IPv6 address as the destination. This address format tells the kernel to send IPv4 datagram to the client. But the next datagram received to the server could be an IPv6 datagram, and **recvfrom** will return the IPv6 address. IF the server responds, the kernel will generate an IPv6 datagram.

Most dual stack hosts should use the following rules in dealing with listening sockets:

- A listening IPv4 socket can accept incoming connection from only IPv4 client.
- If server has a listening IPv6 socket that has bound the wildcard address, that socket can accept incoming connections from either IPv4 client or IPv6 client. For connection from IPv4 client the server's local address for the connection will be the corresponding IPv4 mapped IPv6 address.
- If a server has a listening IPv6 socket that has bound an IPv6 address other than an IPv4 mapped IPv6 address, then the socket can accept incoming connections from IPv6 clients only.

IPv6 client and IPV4 Server:

This scene is the swapping of the client and server protocols used in the previous case. Consider the IPv6 TCP client running on dual stack host.

- An IPv4 server starts on an IPv4 only host and creates an IPv4 listening socket.
- The IPv6 client starts, calls **gethostbyname()** asking for only IPv6 addresses. (It enables the RES_USE_INET6 option).Since the IPv4 only server host has only A record, an IPv4 mapped IPv6 address is returned to the client.
- The IPv6 client calls **connect** with IPv4 mapped IPv6 address in the IPv6 socket address structure. The kernel detects the mapped address and automatically sends an IPv4 SYN to the server

- The server responds with IPv4 SYN/ACK and the connection is established using IPv4 datagrams.

The scenario can be summarized as given below:

- If an IPv4 TCP client calls connect specifying an IPv4 address, or If an IPv4 UDP client calls sendto specifying an IPV4 address, nothing special is done. Shown as IPv4 arrows in the figure.
- IF an IPv6 TCP client calls connect specifying an IPv6 address, or if an IPv6 UDP client calls sendto specifying an IPv6 address, nothing special is done. These are two arrows labeled IPv6 in the figure.
- IF an IPv6 TCP client specifies an IPv4 mapped IPv6 address to connect or if an IPv6 UDP client specifies an IPv4 mapped.IPv6 address to **sendto**, the kernel detects the mapped address and causes an IPv4 client cannot specified causes an IPv4 datagram to sent, instead of IPv6 datagram. These are the two dashed arrows in the figure.
- An IPv4 client cannot specify an IPv6 address to either connect or sendto because a 16 byte IPv6 address does not fit in the 4 byte **in_addr** structure within the IPv4 client to the IPv6 protocol box in the figure.

IN the previous section, (an IPv4 datagram arriving for an IPv6 server socket), the conversion of the received address to the IPv4 mapped IPv6 address is done by the mernel and returned transparently to the application by accept or recvfrom. IN this secion, (an IPv4 datagram needing to be sent on an IPv6 socket) the conversion of the IPv4 address to the IPv4 mapped IPv6 address is done by the resolver according to the rules. And the mapped address is then passed transparently by the application to the connect or sendto.

T H R E A D S

When a process needs something to be performed by another entity, it forks a child process and lets the child perform processing. (similar to concurrent server program.) The problem associated with this are:

a. All descriptors are copied from parent to child thereby occupying more memory.
b. Inter process communication requires to pass information between the parent and child after each fork. Returning the information form the child to parent takes more work.

Threads being a light weight process help to overcome these drawbacks. However, as they share the same variables – known as global variables - they need to be synchronized to avoid errors. IN addition to these variables, threads also share

- Process instructions
- Data,
- Open files
- Signal handlers and signal dispositions
- Current working directory
- User Groups Ids.

Each thread has its own thread ID, set of registers, program counter and stack pointer, stack, errno, signal mask and priority.

Basic thread functions:

pthread_create function:

int ptheread_create (pthread_t) *tid, const pthread_attr_t *attr, void * (*func) (*void), void *arg);

tid : is the thread ID whose data type is **pthread_t -** unsigned integer. ON successful creation of thread, its ID is returned through the pointer tid.

pthread_atttr_t : Each thread has a number of attributes – priority, initi8al stack size, whether is demon thread or not. If this variable is specified, it overrides the default. To accept the default, attr argument is set to null pointer.

***func :** When the thread is created, a function is specified for it to execute. The thread starts by calling this function and then terminates either explicitly (by calling pthread_exit) or implicitly by letting this function to return. The address of the function is specified as the **func** argument. And this function is called with a single pointer argument, **arg.** If multiple arguments are to be passed, the address of the structure can be passed

The function takes one argument – a generic pointer (void *) and returns a generic pointer (void *). This lets us to pass one pointer to the thread and return one pointer.

The return is normally 0 if OK or nonzero on an error

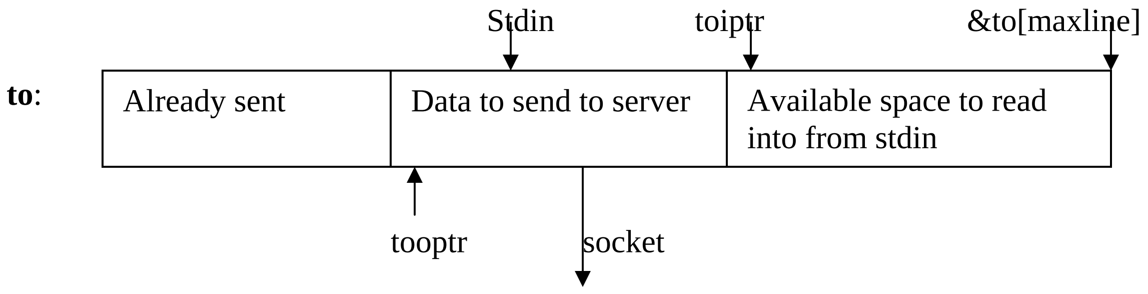
pthread_join function :
int pthread_join (pthread_t tid, void ** status)
We can wait for a given thread to terminate by calling pthread_join . We must specify the tid of the thread that we wish to wait for. If the status pointer is non null, the return value from the thread is stored in the location pointed to by status.

pthread_self function : Each thread has an ID that identifies it within a given process. The thread ID is returned by pthread_create. This function fetches this value for itself by using this function:
pthread_t pthread_self(void);

pthread_detach function :
A thread is joinable (the default) or detached. When a joinable thread terminates, its thread ID and exit status are retained until thread calls **pthread_join().** But a detached thread for example daemon thread- when it terminates all its resources are released and we cannot wait for it terminate.
When one thread needs to know when another thread terminates, it is best to leave the thread joinable.
Int pthread_detach (pthread_t tid);

pthread_exit function:
One way for the thread to terminate is to call pthread_exit().
void pthread_exit (void *status);

The str_cli function uses fputs and the fgets to write to and read from the server. While doing so, fgets my be putting the data from **stdin** into the buffer wherein already there is data waiting to be **writen.** This will block the other processes as the function is waiting to write the pending data. If there is data to be read form the server by the function **readen** at the client, it will be blocked. Similarly, if a line of input is available from the socket we can block in the subsequent call to fputs, if the standard output is slower thant the network. IN such situation non blocking methods are used. This requires creating elaborate arrangement of buffer management: where pointer are used to find the data that is already sent, data that are yet to be sent and available space in the read buffer. As shown below:



The program comes to be about 100 lines . This can be reduced by using threads. In this, call to pthread is given in the main function where in the fgets function are invoked. When the thread returns, fputs is invoked.

RAW SOCKETS

INTRODUCTION:

Raw sockets, are those that bypass the TCP and IP layers and pass the ICMPv4, (Internet Control Message Protocol), IGMPv4 ()Internet Group Management Protocol – used with multicasting) and ICMPv6 packets directly to the link layers.

This allows the application to build ICMP and IGMP entirely as user processes instead of putting more code into the kernel. Examples are route discovery daemon which processes router advertisement and router solicitation are built this way.

With raw sockets a process can read and write IPv4 datagram with IPv4 protocol filed (an 8 bit filed in IPv4 packet) that is not processed by the kernel. Most kernels process datagrams containing values of 1 (ICMP), 2 (IGMP), 6 (TCP), and 17 (UDP). But values like 89 (OSPF) routing protocol does not use TCP or UDP but uses IP directly by setting the protocol field to 89.

With raw sockets, a process can build its own IPv4 header using the IP_HDRINCL socket option
We learn the raw socket creation, input and the output and developing few programs that work with IPv4 and IPv6.

RAW SOCKET CREATION

1. To create raw sockets, the second argument in socket function SOCK_RAW. And the third argument is nonzero (normally) as shown below:

```
Int sockfd;
Sockfd = socket (AF_INET, SOCK_RAW, protocol);
```

In this the protocol is th one of the constants defined by IPPROTO_XXX which is done by including <netinet/in.h> header. For example IPPROO_ICMP. Only super user can create raw socket.

2. The IP-HDRINCL socket option can be set to:

```
const int ON =1;
if (setsocketopt(sockfd, IPPROTO_IP, IP_HDRINCL, &ON, soze0f(ON)) <0)
        error
```

3. Bind may not be called on raw sockets. If called, it sets the local IP address and not the port number as there is no concept of port number with raw sockets. With regard to output, calling bind sets the IP address that will be used for datagrams sent on the raw socket (only if IP_HDRINCL socket option is not set). If bind is not called, the kernel sets the source IP address of the outgoing interface.

4. connect can be call on the raw socket but this is also rare. This function sets only the foreign address and again there is no concept of port number. With regard to output, calling connect lets us call write or send instead of sendto, since the destination IP address is already specified.

<u>Raw Socket Output</u>:

The output of raw socket is governed by the following rules:

- Normal output is performed by calling **sendto** or **sendmsg** and specifying the destination IP address. IN case the socket has been connected, **write** and **send** functions can be used.
- If the **IP_HDRINCL** option is **not set,** the IP header will be built by the kernal and it will be prepend it to the data.
- If **IP_HDRINCL** is **set,** the header format will remain the same and the process builds the entire IP header except the IPv4 identification field which is set to 0 by the kernel
- The kernel fragments the raw packets that exceed the outgoing interface.

IPv6 Differences:

- All fields in the protocol headers sent or received on a raw IPv6 sockets are in network byte order.
- There ae no option fields in IPv6 format. Almost all fields in an IPv6 header and all extension headers (Optional header that follow have their own length field. There is a separate fragmentation header.) are available to the application through socket options.
- Checksum are handled differently.

<u>IPv6_CHECKSUM Socket option</u>

- In case of ICMPv4, the checksum is calculated by the application. Whereas in the application it is done by the kernel.

<u>Raw Socket Input</u>:

The question to be answered in this is which received IP datagrams does the kernel pass to raw sockets.

- Received TCP and UDP packets are never passed to a raw socket.
- Most ICMP packets are passed to a raw socket after the kernel has finished processing the ICMP message. BSD derived implementations pass all received ICMP raw sockets other than echo requests, timestamp request and address mask request. These three ICMP messages are processed entirely by the kernel.
- All IGMP packets are passed to a raw sockets, after the kernel has finished processing the IGMP message.
- All IP datagram with a protocol field that kernel does not understand are passed to a raw socket. The only kernel processing done on these packets is the minimal verification of some IP header field: IP version, IPv4 Header checksum, header length and the destination IP address.
- If the datagram arrives in fragments, nothing is passed to a raw sockets until all fragments have arrived and have been reassembled.

When kernel has to pass IP datagram, it should satisfy all the three tests given below:

- If a nonzero protocol is specified when the raw socket is created (third argument to socket), then the received datagram's protocol field must match this value or the datagram is not delivered.
- IF a local IP address is bound to the raw socket by bind, then the destination IP address of the received datagram must match this bound address or the datagram is not delivered.
- IF foreign IP address was specified for the raw socket by connect, then the source IP address of the received datagram must match this connected address or datagram is not delivered.

If a raw socket is created with protocol of 0, and neither bind or connect is called, then that socket receives a copy of every raw datagram that kernel passes to raw sockets.

When a received datagram is passed to a raw IPv4 socket, the entire datagram, including the IP header, is passed to the process.

ICMPv6 Type Filtering:

A raw ICMPv6 is a superset of ICMPv4, ARP and IGMP and hence the socket can receive many more packets compared to ICMPv4 socket. To reduce the number of packets passed form kernet ot the application , an application specific filter is provided A filter is declared with a data type of **struct icmp_filter** which is defined by including <netinet/icmp6.h> header. The current filter for a raw socket is set and fetched using **setsockopt** and **getsockopt** with a level of **IPPROTO_ICMPv6** and optname **ICMP_FILTER**.

Ping Program:

In this ICMP echo request is sent to some IP address and that the node responds with an ICMP echo reply. These two ICMP messages are supported under IPv4 and IPv6. Following figure shows the format of the ICMP messages.

Type	Code	Checksum
Identifier		Sequence number
Optional Data		

Type	Code	Description	Handled by error no
0	0	Echo reply	User process (ping)
3		Destination unreachable	
	0	Network unreachable	EHOSTUNREACH
	1	Host unreachable	EHOSTUNREACH
`	2	Protocol un reachable	ECONNREFUSED.
	4	port unreachable	ECONNREFUSED

5		REDIRECT	
	0	Redirect for network	Kernel updates routing table
	1		

Checksum is the standard Internet Checksum,
Identifier is set to the process ID of the ping process and the sequence number is incremented by one for each packet that we send. An 8 buit timestamp is stored when a packet is sent as optional data. The rules of ICMP requires that the identifier, sequence number and any optional data be returned in the echo reply Storing the timestamp in the packet lets us calculate the RTT when the reply is received.