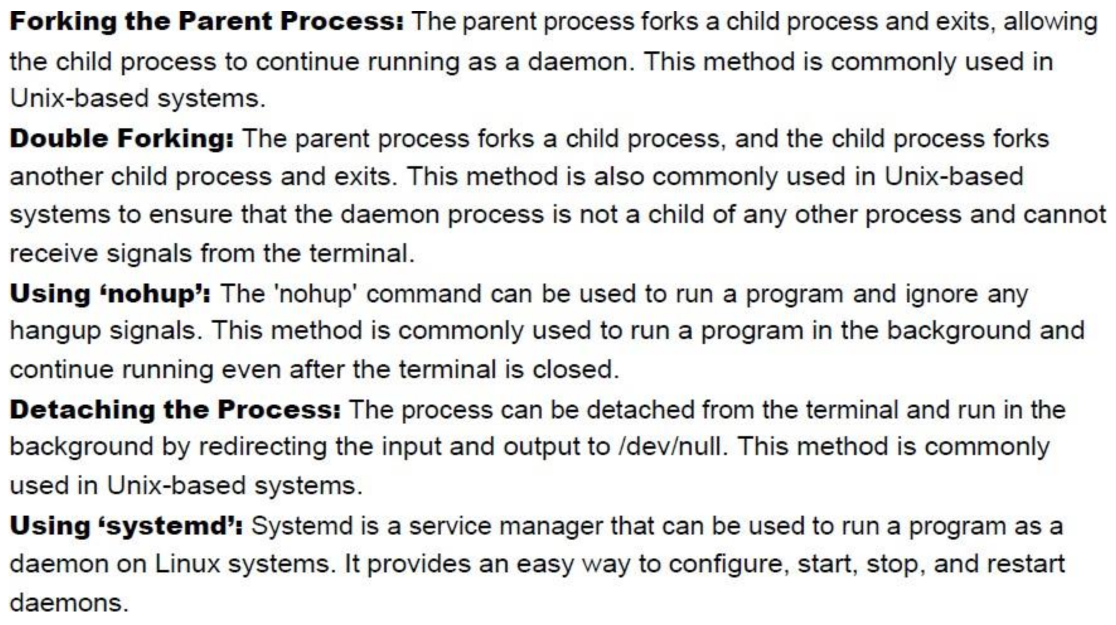
• What is daemon process. List out the numerous ways to start a daemon. Explain the significance of daemon_init function

A computer program that runs as a background process, rather than being under the direct control of an interactive user.



Forking the Parent Process: The parent process forks a child process and exits, allowing the child process to continue running as a daemon. This method is commonly used in Unix-based systems.

Double Forking: The parent process forks a child process, and the child process forks another child process and exits. This method is also commonly used in Unix-based systems to ensure that the daemon process is not a child of any other process and cannot receive signals from the terminal.

Using 'nohup': The 'nohup' command can be used to run a program and ignore any hangup signals. This method is commonly used to run a program in the background and continue running even after the terminal is closed.

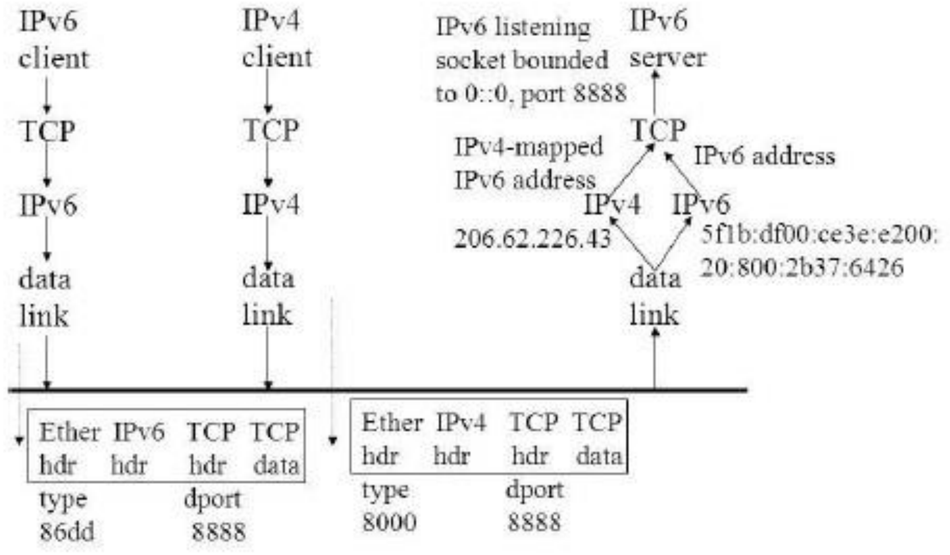
Detaching the Process: The process can be detached from the terminal and run in the background by redirecting the input and output to /dev/null. This method is commonly used in Unix-based systems.

Using 'systemd': Systemd is a service manager that can be used to run a program as a daemon on Linux systems. It provides an easy way to configure, start, stop, and restart daemons.

The daemon_init function is significant because it provides a portable and reliable way to daemonize a process in Unix-like operating systems. It is used by many different software packages, including web servers, database servers, and other system services. It is used by many different software packages to run in the background and provide continuous services.

• **Summarize the steps that allow an IPV4 TCP client to communicate with IPV6 server using dual stack**

Ans:

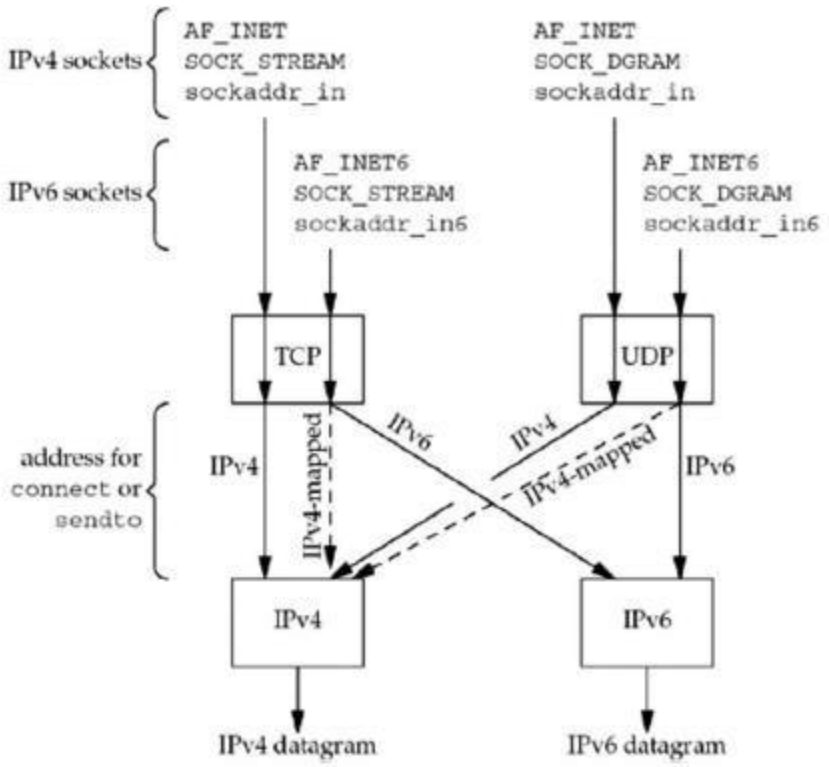


Steps that allow an IPv4 TCP client to communicate with an IPv6 server as follows:

- The IPv6 server starts creating an IPv6 listening socket, and we assume it binds the wildcard address to the socket.
- The IPv4 client calls `gethostbyname` and finds an A record for the server. The server host will have both an A record and an AAAA record since it supports both protocols, but the IPv4 client asks for only an A record.
- The client calls connect and the client's host sends an IPv4 SYN to the server.
- The server host receives the IPv4 SYN directed to the IPv6 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.
- When the server host sends to the IPv4-mapped IPv6 address, its IP stack generates IPv4 datagrams to the IPv4 address. Therefore, all communication between this client and server takes place using IPv4 datagrams.
- Unless the server explicitly checks whether this IPv6 address is an IPv4-mapped IPv6 address, the server never knows that it is communicating with an IPv4

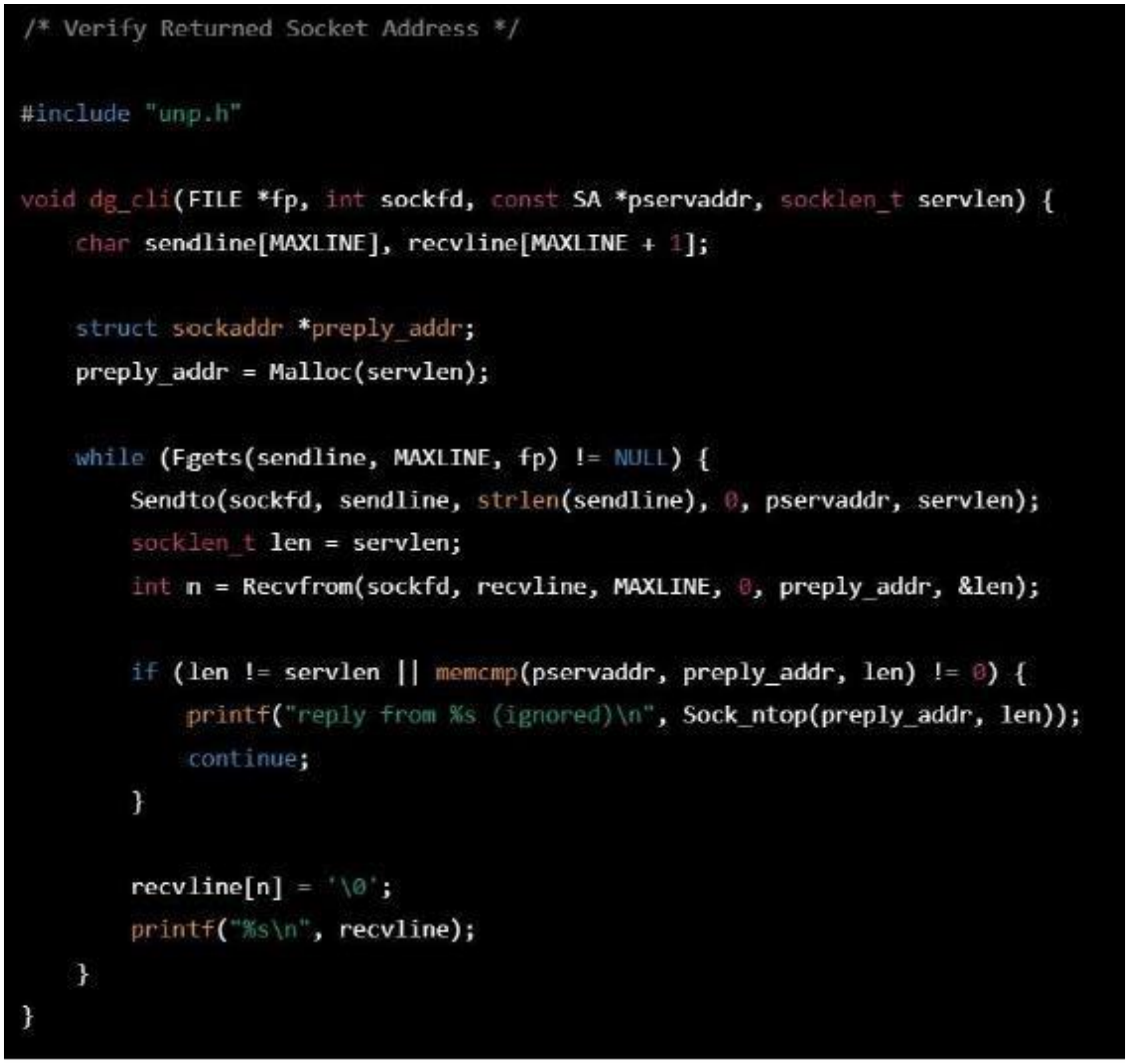
Explain with a neat block diagram Processing of client requests, depending on address type and socket type.

\ns:



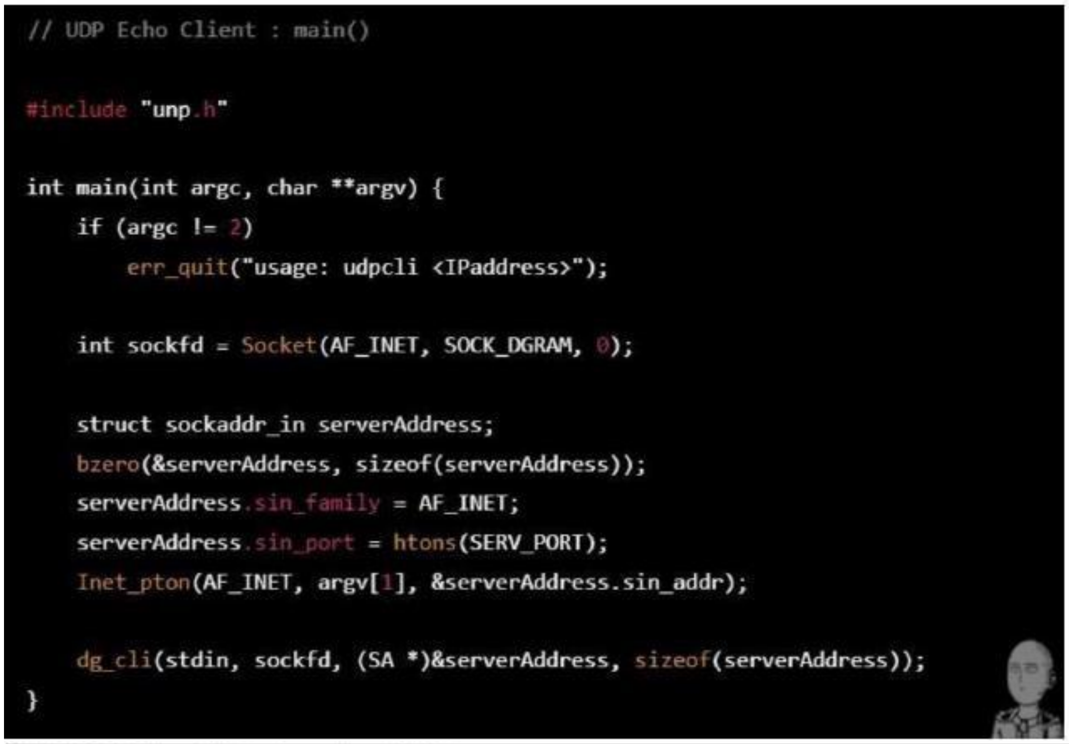
- An IPv4 server starts on an IPv4-only host and creates an IPv4 listening socket. The IPv6 client calls `connect` with the IPv4-mapped IPv6 address in the IPv6 socket address structure.
- The kernel detects the mapped address and automatically sends an IPv4 SYN to the server.
- The server responds with an IPv4 SYN/ACK, and the connection is established using IPv4 datagrams.
- 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.
- 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.

- **Develop the “C” program for dg_cli function that verifies returned socket address**

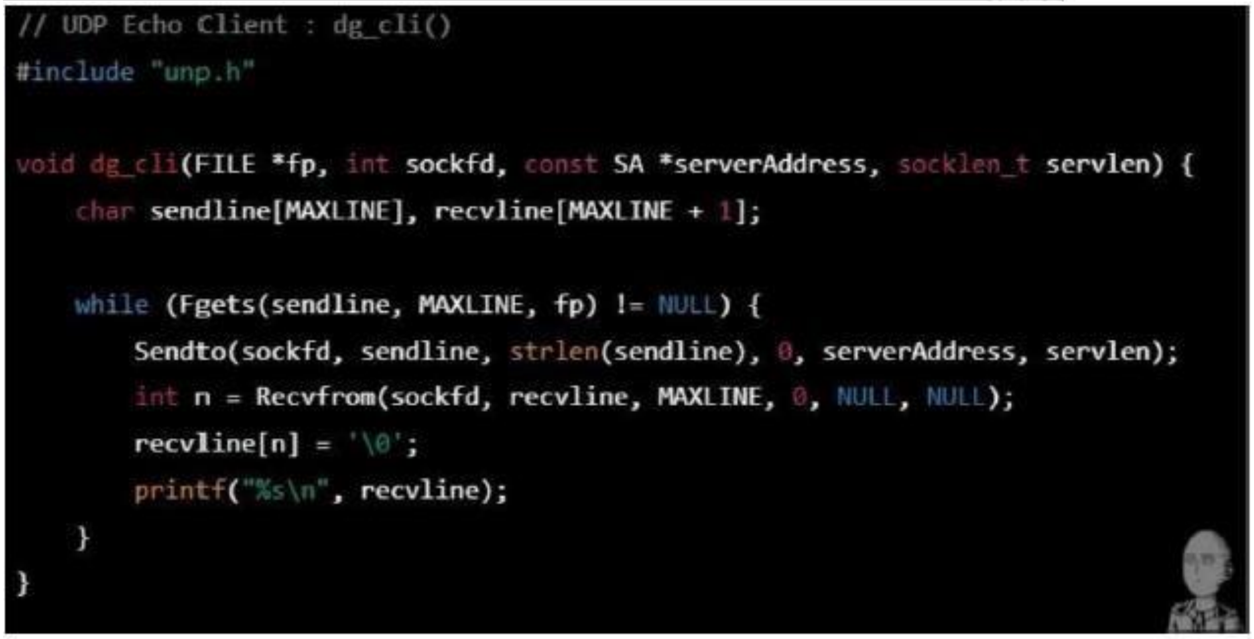


```c
/* Verify Returned Socket Address */

#include "unp.h"

void dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen) {
    char sendline[MAXLINE], recvline[MAXLINE + 1];

    struct sockaddr *preply_addr;
    preply_addr = Malloc(servlen);

    while (Fgets(sendline, MAXLINE, fp) != NULL) {
        Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
        socklen_t len = servlen;
        int n = Recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);

        if (len != servlen || memcmp(pservaddr, preply_addr, len) != 0) {
            printf("reply from %s (ignored)\n", Sock_ntop(preply_addr, len));
            continue;
        }

        recvline[n] = '\0';
        printf("%s\n", recvline);
    }
}
```

- **Develop the "C" program to demonstrate the the UDPechoclient: main function and dg_cli function**

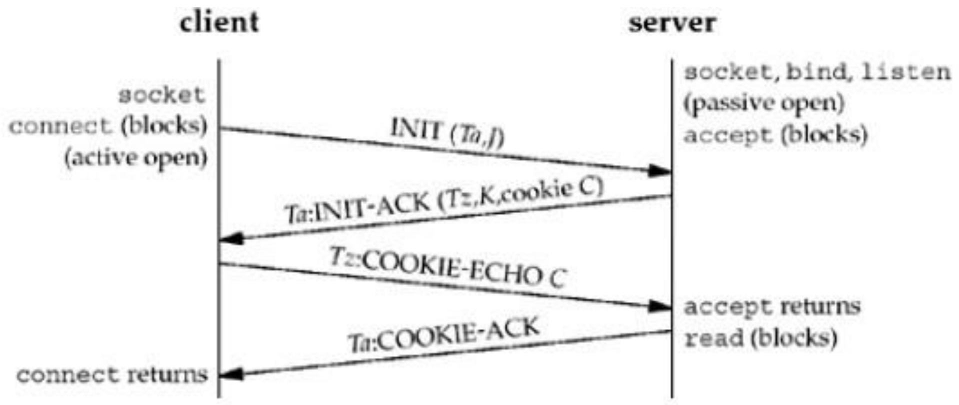


```c
// UDP Echo Client : main()

#include "unp.h"

int main(int argc, char **argv) {
    if (argc != 2)
        err_quit("usage: udpcli <IPaddress>");

    int sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

    struct sockaddr_in serverAddress;
    bzero(&serverAddress, sizeof(serverAddress));
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_port = htons(SERV_PORT);
    Inet_pton(AF_INET, argv[1], &serverAddress.sin_addr);

    dg_cli(stdin, sockfd, (SA *)&serverAddress, sizeof(serverAddress));
}
```



```c
// UDP Echo Client : dg_cli()
#include "unp.h"

void dg_cli(FILE *fp, int sockfd, const SA *serverAddress, socklen_t servlen) {
    char sendline[MAXLINE], recvline[MAXLINE + 1];

    while (Fgets(sendline, MAXLINE, fp) != NULL) {
        Sendto(sockfd, sendline, strlen(sendline), 0, serverAddress, servlen);
        int n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
        recvline[n] = '\0';
        printf("%s\n", recvline);
    }
}
```

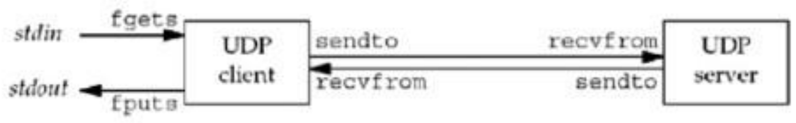
• **Construct a suitable sequence diagram to indicate the functionality of 4-way handshake for the establishment of an association in SCTP protocol**



Optional Theory, just for understanding concept

- The server must be prepared to accept an incoming association. This preparation is normally done by calling socket, bind, and listen.
- The client issues an active open by calling connect or by sending a message. This causes the client SCTP to send an INIT message to tell the server the client's list of IP addresses, the number of outbound streams the client is requesting, and the number of inbound streams the client can support.
- The server acknowledges the client's INIT message with an INIT-ACK message, which contains the server's list of IP addresses, number of outbound streams the server is requesting, number of inbound streams the server can support, and a state cookie.
- The client echos the server's state cookie with a COOKIE-ECHO message.
- The server acknowledges that the cookie was correct and that the association was established with a COOKIE-ACK message.
- The minimum number of packets required for this exchange is four; hence, this process is called SCTP's four-way handshake.

- **With a neat flowchart explain how you would implement Echo Client and Server programs using UDP and demonstrate their working**



1. Server: Create a socket
   - The server creates a socket using the socket() function.
   - The function returns a socket descriptor, which is used to identify the socket in all future operations.
2. Server: Bind the socket to an address and port
   - The server binds the socket to a specific address and port using the bind() function.
   - This step is used to associate the socket with a specific IP address and port, so that it can receive incoming data.
3. Server: Receive data
   - The server uses the recvfrom() function to receive data from the socket.
   - The function stores the received data in the buffer and the address of the sender in the address struct provided.
4. Server: Send data back
   - The server uses the sendto() function to send the data back to the sender.
   - The function sends the data in the buffer to the specified address using the socket descriptor.
5. Server: Close the socket
   - The server closes the socket using the close() function, which takes the socket descriptor as its argument.
6. Client: Create a socket
   - The client creates a socket using the socket() function.
   - The function returns a socket descriptor, which is used to identify the socket in all future operations.
7. Client: Send data
   - The client uses the sendto() function to send data to the server.
   - The function sends the data in the buffer to the specified address using the socket descriptor.
8. Client: Receive data
   - The client uses the recvfrom() function to receive data from the socket.
   - The function stores the received data in the buffer and the address of the sender in the address struct provided.
9. Client: Close the socket
   - The client closes the socket using the close() function, which takes the socket descriptor as its argument.

EchoClient.c

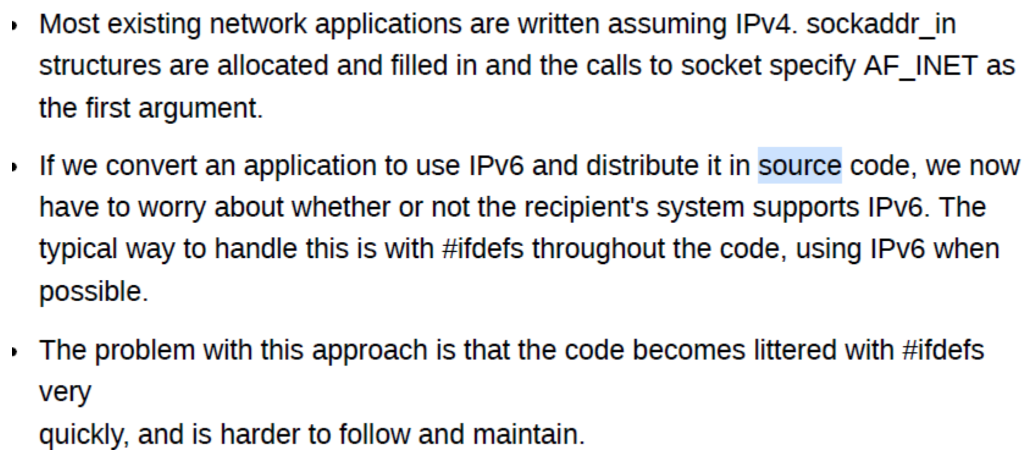
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 8080
#define SERVER_ADDRESS "127.0.0.1"
#define BUFFER_SIZE 1024

int main() {
    int sockfd;
    char buffer[BUFFER_SIZE];
    struct sockaddr_in server_addr;

    // Creating socket file descriptor
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        perror("Socket creation failed");
        exit(EXIT_FAILURE);
    }

    memset(&server_addr, 0, sizeof(server_addr));

    // Filling server information
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    if (inet_pton(AF_INET, SERVER_ADDRESS, &server_addr.sin_addr) <= 0) {
        perror("Invalid address/ Address not supported");
        exit(EXIT_FAILURE);
    }

    while (1) {
        printf("Enter message to send: ");
        fgets(buffer, BUFFER_SIZE, stdin);

        sendto(sockfd, (const char *)buffer, strlen(buffer), MSG_CONFIRM, (const
struct sockaddr *)&server_addr, sizeof(server_addr));

        int n = recvfrom(sockfd, (char *)buffer, BUFFER_SIZE, MSG_WAITALL, NULL,
NULL);
        buffer[n] = '\0';
        printf("Echoed message from server: %s\n", buffer);
    }

    close(sockfd);
    return 0;
}
```

EchoServer.c

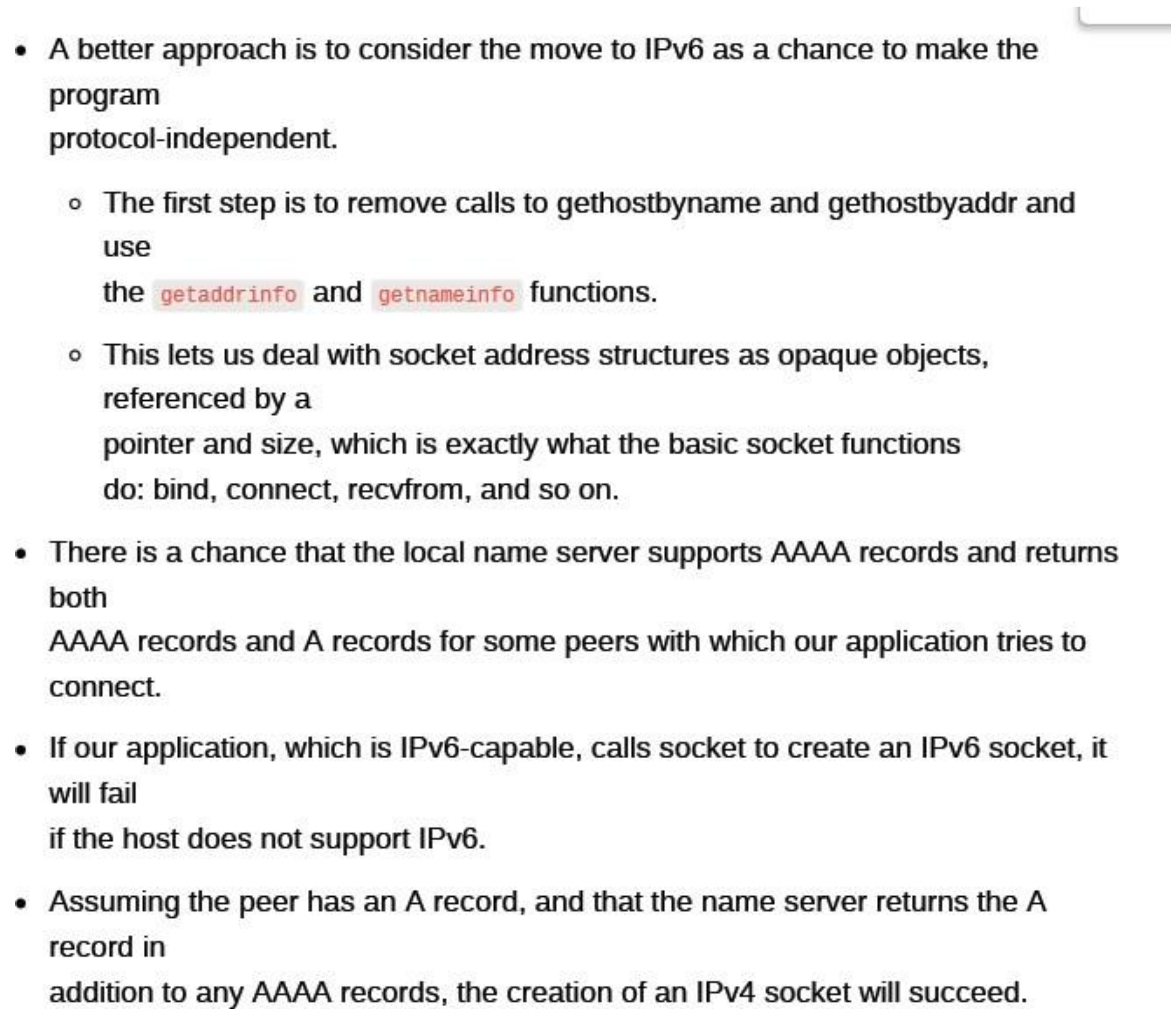
```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 8080
#define BUFFER_SIZE 1024

int main() {
    int sockfd;
    char buffer[BUFFER_SIZE];
    struct sockaddr_in server_addr, client_addr;
    socklen_t len;

    // Creating socket file descriptor
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        perror("Socket creation failed");
        exit(EXIT_FAILURE);
    }

    memset(&server_addr, 0, sizeof(server_addr));
    memset(&client_addr, 0, sizeof(client_addr));

    // Filling server information
    server_addr.sin_family = AF_INET; // IPv4
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);

    // Bind the socket with the server address
    if (bind(sockfd, (const struct sockaddr *)&server_addr, sizeof(server_addr))
== -1) {
        perror("Bind failed");
        exit(EXIT_FAILURE);
    }

    printf("UDP Echo Server started on port %d\n", PORT);

    while (1) {
        len = sizeof(client_addr);

        int n = recvfrom(sockfd, (char *)buffer, BUFFER_SIZE, MSG_WAITALL, (struct
sockaddr *)&client_addr, &len);
        buffer[n] = '\0';
        printf("Received message from %s:%d: %s\n",
inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port), buffer);

        sendto(sockfd, (const char *)buffer, strlen(buffer), MSG_CONFIRM, (const
struct sockaddr *)&client_addr, len);
        printf("Echoed message sent.\n");
    }
    close(sockfd);
    return 0;
}
```

Discuss how source code portability from IPV4 applications could be converted to useIPV6

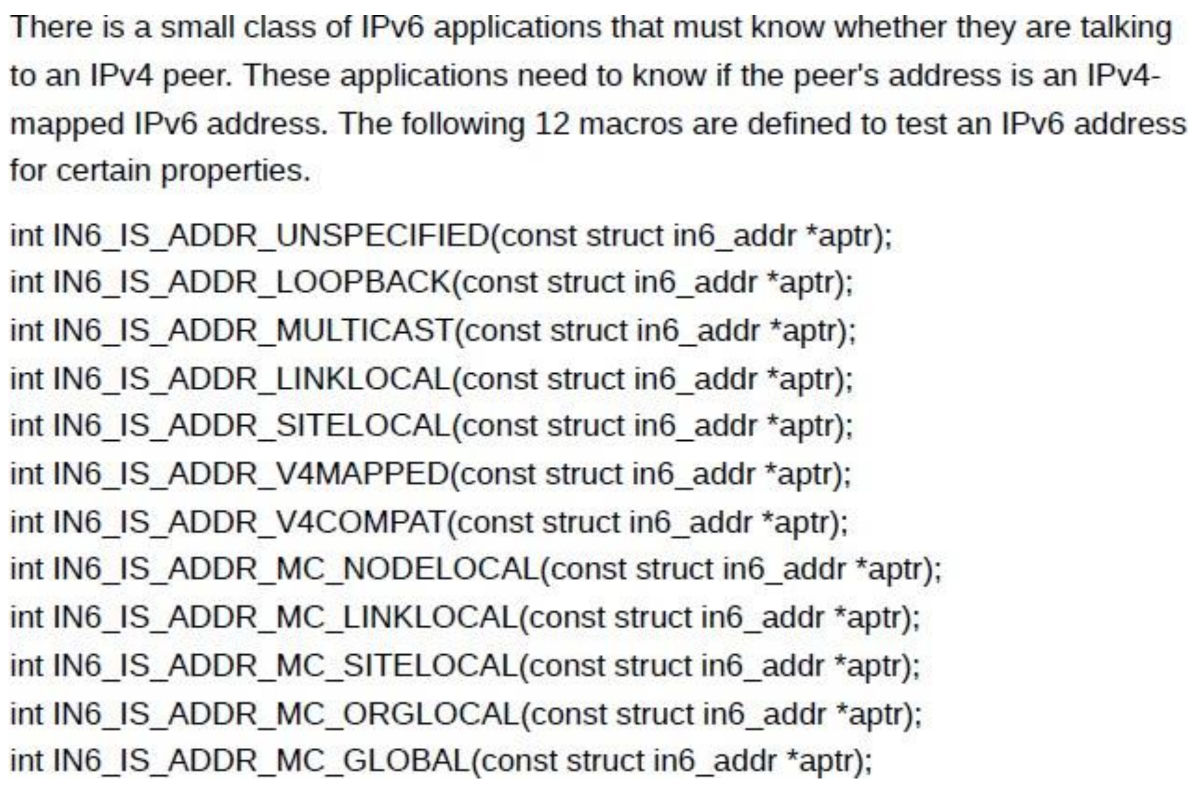


- Most existing network applications are written assuming IPv4. sockaddr_in structures are allocated and filled in and the calls to socket specify AF_INET as the first argument.
- If we convert an application to use IPv6 and distribute it in source code, we now have to worry about whether or not the recipient's system supports IPv6. The typical way to handle this is with #ifdefs throughout the code, using IPv6 when possible.
- The problem with this approach is that the code becomes littered with #ifdefs very
  quickly, and is harder to follow and maintain.



- A better approach is to consider the move to IPv6 as a chance to make the program
  protocol-independent.
  - The first step is to remove calls to gethostbyname and gethostbyaddr and use
    the `getaddrinfo` and `getnameinfo` functions.
  - This lets us deal with socket address structures as opaque objects, referenced by a
    pointer and size, which is exactly what the basic socket functions
    do: bind, connect, recvfrom, and so on.
- There is a chance that the local name server supports AAAA records and returns both
  AAAA records and A records for some peers with which our application tries to connect.
- If our application, which is IPv6-capable, calls socket to create an IPv6 socket, it will fail
  if the host does not support IPv6.
- Assuming the peer has an A record, and that the name server returns the A record in
  addition to any AAAA records, the creation of an IPv4 socket will succeed.

- **Explain IPV6 addressing testing Macros**



There is a small class of IPv6 applications that must know whether they are talking to an IPv4 peer. These applications need to know if the peer's address is an IPv4-mapped IPv6 address. The following 12 macros are defined to test an IPv6 address for certain properties.

```
int IN6_IS_ADDR_UNSPECIFIED(const struct in6_addr *aptr);
int IN6_IS_ADDR_LOOPBACK(const struct in6_addr *aptr);
int IN6_IS_ADDR_MULTICAST(const struct in6_addr *aptr);
int IN6_IS_ADDR_LINKLOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_SITELOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_V4MAPPED(const struct in6_addr *aptr);
int IN6_IS_ADDR_V4COMPAT(const struct in6_addr *aptr);
int IN6_IS_ADDR_MC_NODELOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_MC_LINKLOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_MC_SITELOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_MC_ORGLOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_MC_GLOBAL(const struct in6_addr *aptr);
```