UDP Socket:

What is UDP?

- UDP is a connectionless, unreliable, datagram protocol
- UDP provides a connectionless service, as there need not be any longterm relationship between a UDP client and server. For example, a UDP client can create a socket and send a datagram to a given server and then immediately send another datagram on the same socket to a differ- ent server. Similarly, a UDP server can receive several datagrams on a single UDP socket, each from a different client.
- Each UDP datagram has a length. The length of a datagram is passed to the receiving application along with the data.
- UDP can use either IPV4 or IPV6.
- In general, most TCP servers are concurrent and most UDP servers are iterative.

Why UDP is used if it is unreliable ?

No retransmission delays – UDP is suitable for time-sensitive applications that can't afford retransmission delays for dropped packets. Examples include Voice over IP (VoIP), online games, and media streaming.

Speed – UDP's speed makes it useful for query-response protocols such as DNS, in which data packets are small and transactional.

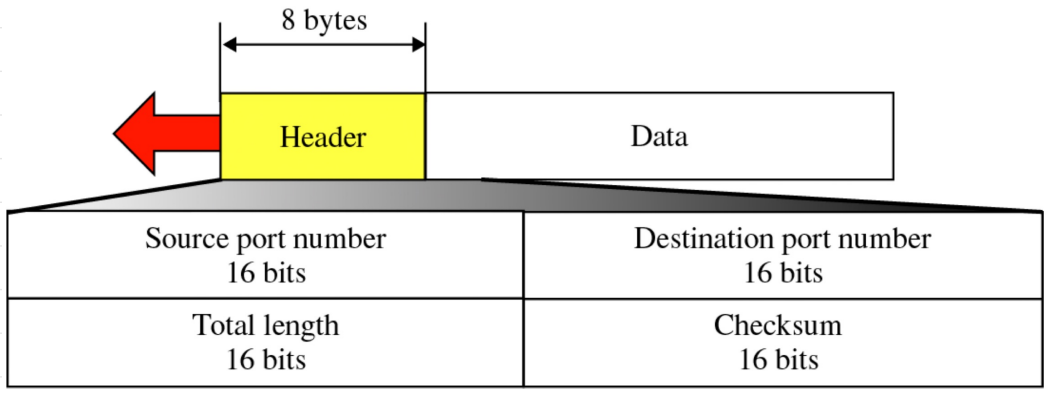
Suitable for broadcasts – UDP's lack of end-to-end communication makes it suitable for broadcasts, in which transmitted data packets are addressed as receivable by all devices on the internet. UDP broadcasts can be received by large numbers of clients without server-side overhead.

What are the disadvantages of UDP?

UDP's lack of connection requirements and data verification can create a number of issues when transmitting packets. These include:

- No guaranteed ordering of packets.
- No verification of the readiness of the computer receiving the message.
- No protection against duplicate packets.
- No guarantee the destination will receive all transmitted bytes. UDP, however, does provide a checksum to verify individual packet integrity.

User Datagram



UDP packets called user datagram.

UDP wraps datagrams with a UDP header, which contains four fields totaling eight bytes.

Source port – The port of the device sending the data. This field can be set to zero if the destination computer doesn't need to reply to the sender.

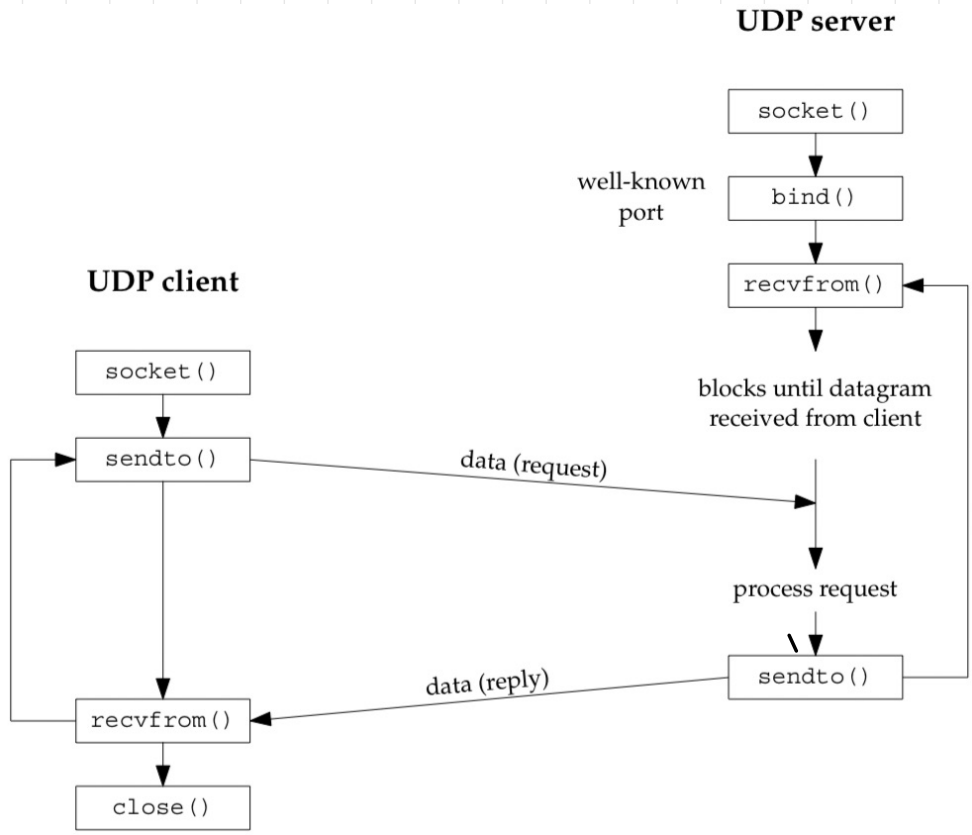
Destination port – The port of the device receiving the data. UDP port numbers can be between 0 and 65,535.

Length – Specifies the number of bytes comprising the UDP header and the UDP payload data. The limit for the UDP length field is determined by the underlying IP protocol used to transmit the data

Checksum – The checksum allows the receiving device to verify the integrity of the packet header and payload. It is optional in IPv4 but was made mandatory in IPv6.

The minimum length is 8 bytes, the length of the header. The field size sets a theoretical limit of 65,535 bytes (8 byte header + 65,527 bytes of data) for a UDP datagram. However the actual limit for the data length, which is imposed by the underlying IPv4 protocol, is 65,508 bytes (2**16 = 65,536 – 8 byte UDP header – 20 byte IP header).

Using IPv6 jumbograms it is possible to have UDP datagrams of size greater than 65,535 bytes. RFC 2675 specifies that the length field is set to zero if the length of the UDP header plus UDP data is greater than 65,535.




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

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

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

    Both return: number of bytes read or written if OK,
                                              -1 on error;
```

A time client-server program using UDP

Server.c

```c
#include<stdio.h>
#include<sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include<stdlib.h>
#include<time.h>
#include<string.h>
int main(int argc, char *argv[])
{
    int n;
    char data[201];
    //create a socket
    int net_socket;
    net_socket = socket(AF_INET, SOCK_DGRAM, 0);
    struct sockaddr_in server_address, client_address;
    server_address.sin_family = AF_INET;
    server_address.sin_port = htons(5600);
    server_address.sin_addr.s_addr = inet_addr(argv[1]);
    bind(net_socket, (struct sockaddr*)&server_address,sizeof(server_address));
    time_t tick;
    char str[100];
    int f=0;
    int len=sizeof(client_address);
    while(1)
    {
      recvfrom(net_socket, &f,4,0,(struct sockaddr*)&client_address,&len);
      tick=time(NULL);
       snprintf(str,sizeof(str),"%s",ctime(&tick));
      sendto(net_socket, str, strlen(str), 0, (struct sockaddr *)&client_address, len);
    }
    close(net_socket);
    return(0);
}
```

Client.c

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

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

#include<string.h>
int main(int argc, char *argv[])
{
    int n;
    char data[201];
    //create a socket
    int net_socket;
    net_socket = socket(AF_INET, SOCK_DGRAM, 0);

    //connect to a server
    //where we want to connect to
    struct sockaddr_in server_address;
    server_address.sin_family = AF_INET;
    server_address.sin_port = htons(5600);
    server_address.sin_addr.s_addr = inet_addr(argv[1]);


    char str[100];
    int f=0;
    int len=sizeof(server_address);
    sendto(net_socket, &f,4,0,(struct sockaddr*)&server_address,len);
    recvfrom(net_socket, str, 100, 0, (struct sockaddr *)&server_address, &len);
    printf("Time is : %s", str);

    return(0);
}
```

Lost Datagrams

UDP client/server example is not reliable.

If a client datagram is lost (say it is dis- carded by some router between the client and server), the client will block forever in its call to recvfrom

Similarly, if the client datagram arrives at the server but the server's reply is lost, the client will again block forever in its call to recvfrom.

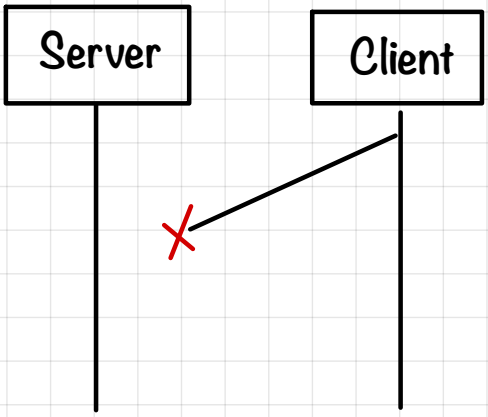
A typical way to prevent this is to place a timeout on the client's call to recvfrom.

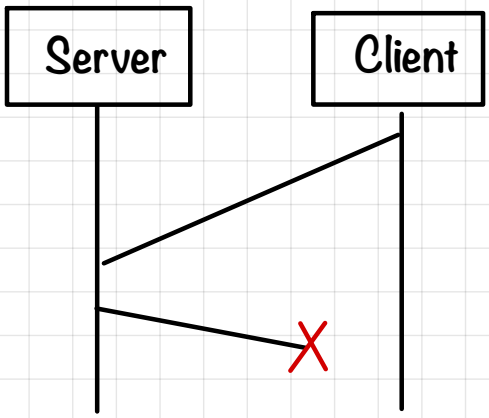
```c
#include<stdio.h>
#include<sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include<stdlib.h>
#include<string.h>
#include<signal.h>
void handel_alarm (int sig)
{
  printf("Waited for maximum time.... no data recived ... exiting\n");
  exit(0);
}
int main(int argc, char *argv[])
{
    int n;
    char data[201];
    int net_socket;
    net_socket = socket(AF_INET, SOCK_DGRAM, 0);
    struct sockaddr_in server_address;
    server_address.sin_family = AF_INET;
    server_address.sin_port = htons(5600);
    server_address.sin_addr.s_addr = inet_addr(argv[1]);
    char str[100];
    int f=0;
    int len=sizeof(server_address);
    signal(SIGALRM, handel_alarm);
    sendto(net_socket, &f,4,0,(struct sockaddr*)&server_address,len);
    alarm(5);
    recvfrom(net_socket, str, 100, 0, (struct sockaddr *)&server_address, &len);
    printf("Time is : %s", str);
    return(0);
}
```

We are going to use SIGALRM signal to make a time out if data is not received within maximum timeout

Set alarm for time out the handle the signal by informing the user about time out

Just placing a timeout on the recvfrom is not the entire solution. For example, if we do time out, we cannot tell whether our datagram never made it to the server, or if the server's reply never made it back.





In both of the above condition there is no way to tell it the daragram reached the server or not.

If the client's request was something like "trans- fer a certain amount of money from account A to account B", it would make a big difference as to whether the request was lost or the reply was lost.

So we need reliability added to UDP communication.

Verifying Received Response

Any process that knows the client's ephemeral port number could send datagrams to our client, and these would be intermixed with the normal server replies.

Problem with recvfrom function is that it sets the address variable, hence we need to compare it the datagram received is from server or some other client

Steps to follow:

- Allocate another socket address structure
- Compare returned address