Select () system call and I/O Multiplexing

Server Concurrency

- For servicing multiple clients there are two main approaches:
  - Forking with fork()
  - Selecting with select()
- The fork () approach creates a new process to handle each incoming client connection.
  - Inefficient (High overhead due to context switching)

- A better approach would be to have a *single* process handle all incoming clients, without having to spawn separate child “server handlers”- select().

I/O Multiplexing

Monitoring multiple descriptors:

- A server that handles both TCP and UDP
- A generic TCP client (like telnet) needs to be able to handle unexpected situations, such as a server that shuts down without warning.
  - **Input from standard input should be sent to a TCP socket.**
  - **Input from a TCP socket should be sent to standard output.**

- Non-determinism and concurrency problem:
  - **How do we know when to check for input from each source?**

Need of I/O Multiplexing

- When a client is handling multiple descriptors (normally interactive input and a network socket), I/O multiplexing should be used.
- Client handling multiple sockets at the same time.
- If a server handles both TCP and UDP, I/O multiplexing is normally used.

Significance of select function

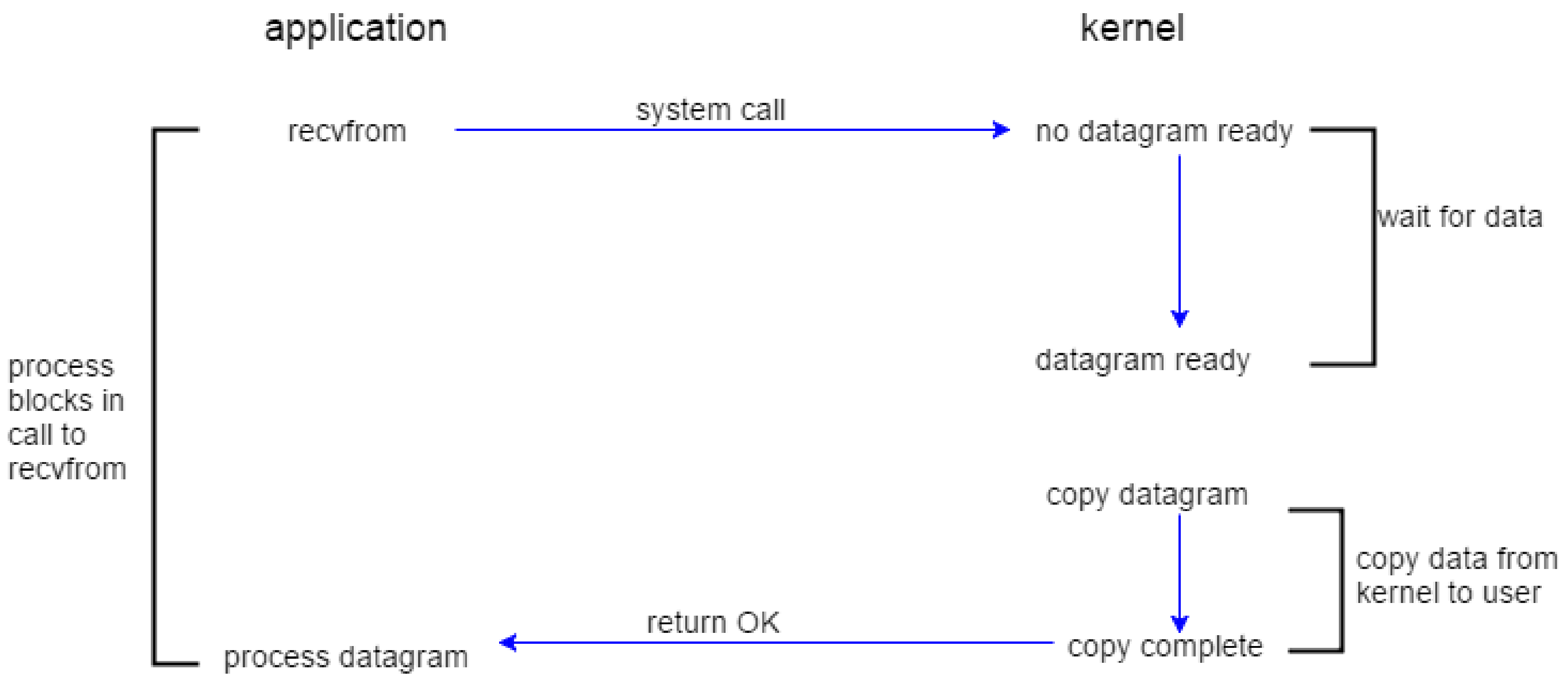
- We encountered a problem when the client was blocked in a call to fgets (on standard input) and the server process was killed.

- The server TCP correctly sent a FIN to the client TCP, but since the client process was blocked reading from standard input, it never saw the EOF until it read from the socket (possibly much later).

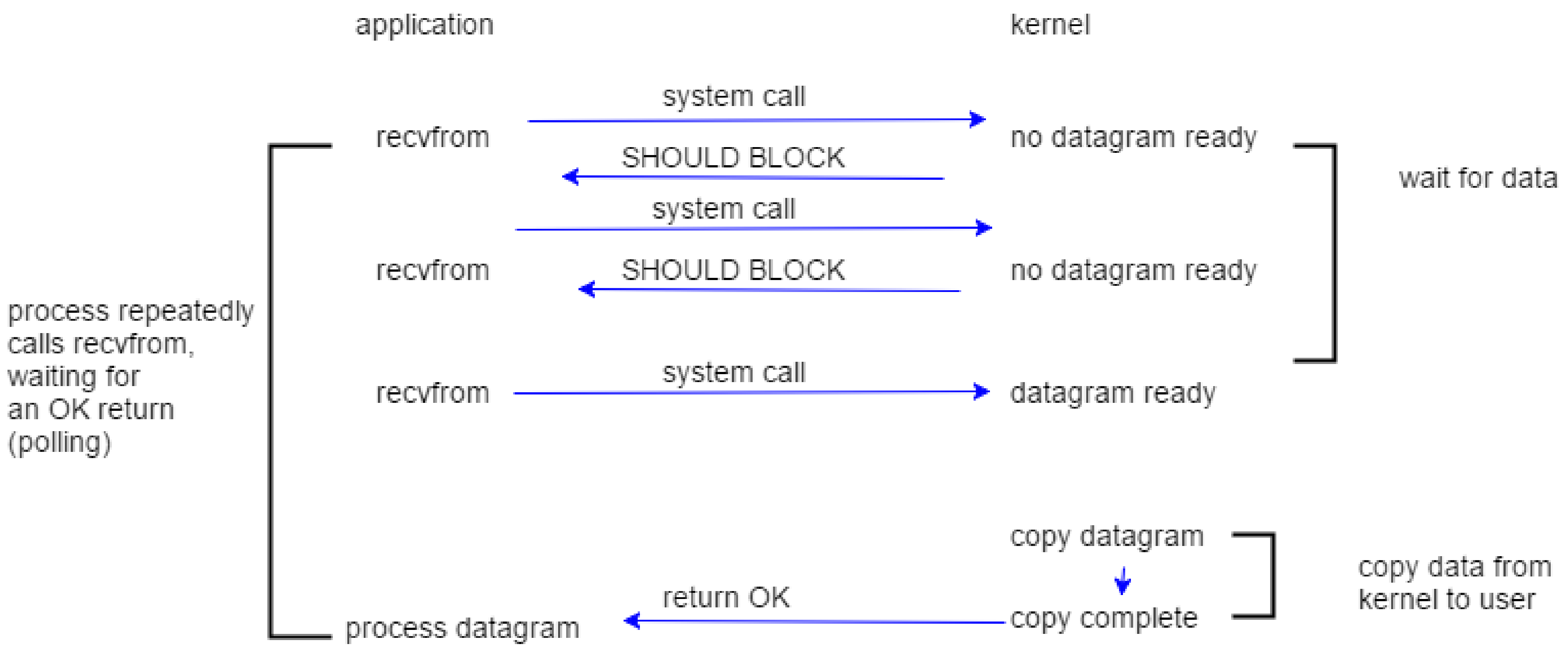
- What we need is the capability to tell the kernel that we want to be notified if one or more I/O conditions are ready.

- This capability is called *I/O multiplexing* and is provided by the select function.

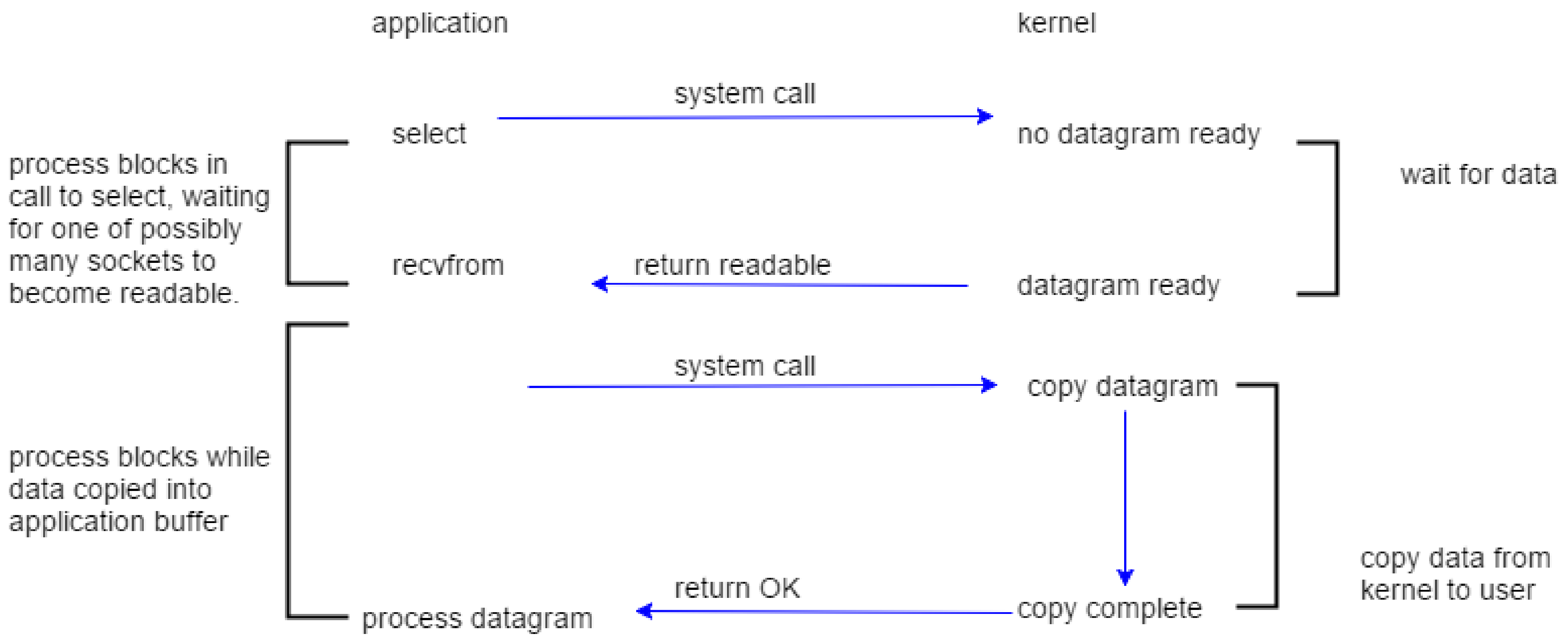
Blocking I/O


application
kernel
recvfrom
system call
no datagram ready
wait for data
datagram ready
process blocks in call to recvfrom
copy datagram
copy data from kernel to user
return OK
copy complete
process datagram

Non-blocking I/O



select()


application
kernel
system call
select
no datagram ready
process blocks in call to select, waiting for one of possibly many sockets to become readable.
wait for data
recvfrom
return readable
datagram ready
system call
copy datagram
process blocks while data copied into application buffer
copy data from kernel to user
return OK
process datagram
copy complete

select()

- The select() system call allows us to use blocking I/O on a set of descriptors (file, socket, …).
- For example, we can ask select to notify us when data is available for reading on either STDIN or a TCP socket.
- The select() system call provides a way for a single server to wait until a set of network connections has data available for reading.

select()

```
#include <sys/time.h>
int select( int maxfd,
            fd_set *readset,
            fd_set *writeset,
            fd_set *excepset,
            const struct timeval *timeout);
```

maxfd:	highest number assigned to a descriptor.
readset:	set of descriptors we want to read from.
writeset:	set of descriptors we want to write to.
excepset:	set of descriptors to watch for exceptions/errors.
timeout:	maximum time select should wait

select()

- **select()** will return if *any* of the descriptors in readset and writeset of file descriptors are ready for reading or writing, respectively, or, if any of the descriptors in exceptset are in an error condition.
- The **FD_SET(int fd, fd_set *set)** function will add the file descriptor *fd* to the set *set.*
- The **FD_ISSET(int fd, fd_set *set)** function will tell you if filedescriptor *fd* is in the modified set *set.*
- select() returns the total number of descriptors in the modified sets.
- If a client closes a socket whose file descriptor is in one of your watched sets, select() will return, and your next recv() will return 0, indicating the socket has been closed

Setting the timeval in select()

- Setting the timeout to 0, select() times out *immediately*
- Setting the timeout to NULL, select() will *never* time out, and will block indefinitely until a file descriptor is modified
- To ignore a particular file descriptor set, just set it to NULL in the call:
  - select (max, &readfds, NULL, NULL, NULL);
  - Here we only care about reading, and we want to block indefinitely until we do have a file descriptor ready to be read.

struct timeval

```
struct timeval {
        long tv_sec;    /* seconds */
        long tv_usec;   /* microseconds */
}

struct timeval max = {1,0};
```

Using select()

- Create fd_set
- Clear the whole set with FD_ZERO
- Add each descriptor you want to watch using FD_SET.
- Call select
- when select returns, use FD_ISSET to see if I/O is possible on each descriptor.

fd_set

Operations to use with fd_set:

- void FD_ZERO(fd_set *fdset);
- void FD_SET(int fd, fd_set *fdset);
- void FD_CLR(int fd, fd_set *fdset);
- int FD_ISSET(int fd, fd_set *fdset);

Example

select( ) System Call

- **We can combine our concurrent TCP echo server and iterative UDP server into a single server that uses select to multiplex TCP and UDP socket.**
- **Select** function is used to select between TCP and UDP sockets. This function gives instructions to the kernel to wait for any of the multiple events to occur and awakens the process only after one or more events occur or a specified time passes.
- **Example** – kernel will return only when one of these conditions occurs
- Any Descriptor from {1, 2} is ready for reading
- Any Descriptor from {3,4} is ready for writing
- Time 2sec have passed
- The entire process can be broken down into the following steps :
- **Server:**
- Create TCP i.e Listening socket
- Create a UDP socket
- Bind both socket to the server address.
- Initialize a descriptor set for select and calculate a maximum of 2 descriptor for which we will wait
- Call select and get the ready descriptor(TCP or UDP)
- Handle new connection if ready descriptor is of TCP OR receive datagram if ready descriptor is of UDP

select( ) System Call

- **UDP Client:**
- Create UDP socket.
- Send message to server.
- Wait until response from server is received.
- Close socket descriptor and exit.
- **TCP Client:**
- Create a TCP scoket.
- Call connect to establish connection with server
- When the connection is accepted write message to server
- Read response of Server
- Close socket descriptor and exit.

- **Necessary functions:**

```
int select(int maxfd, fd_set *readsset,
fd_set *writeset, fd_set *exceptset, const struct timeval *timeout);
Returns: positive count of descriptors ready, 0 on timeout, -1 error
```

- **Arguments:**
  - **maxfd:** maximum number of descriptor ready.
  - **timeout:** How long to wait for select to return.

select( ) System Call

```
struct timeval{ long tv_sec; long tv_usec; };
if timeout==NULL then wait forever if timeout == fixed_amount_time then wait until specified time
if timeout == 0 return immediately.
```

•**readset:** Descriptor set that we want kernel to test for reading.
•**writeset:** Descriptor set that we want kernel to test for writing.
•**exceptset:** Descriptor set that we want kernel to test for exception conditions.

```
int read(int sockfd, void * buff, size_t nbytes);
Returns: number of bytes read from the descriptor. -1 on error
```

Arguments:
1.sockfd: Descriptor which receives data.
2.buff: Application buffer socket descriptor data is copied to this buffer.
3.nbytes: Number of bytes to be copied to application buffer.

Server Program:

```
// Server program
#include <arpa/inet.h>
#include <errno.h>
#include <netinet/in.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>
#define PORT 5000
#define MAXLINE 1024
int max(int x, int y)
{
    if (x > y)
        return x;
    else
        return y;
}
int main()
{
    int listenfd, connfd, udpfd, nready, maxfdp1;
    char buffer[MAXLINE];
    pid_t childpid;
    fd_set rset;
    ssize_t n;
    socklen_t len;
    const int on = 1;
    struct sockaddr_in cliaddr, servaddr;
    char* message = "Hello Client";
```

Server Program:

```
  /* create listening TCP socket */
listenfd = socket(AF_INET, SOCK_STREAM, 0);
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(PORT);

// binding server addr structure to listenfd
bind(listenfd, (struct sockaddr*)&servaddr, sizeof(servaddr));
listen(listenfd, 10);

/* create UDP socket */
udpfd = socket(AF_INET, SOCK_DGRAM, 0);
// binding server addr structure to udp sockfd
bind(udpfd, (struct sockaddr*)&servaddr, sizeof(servaddr));

// clear the descriptor set
FD_ZERO(&rset);

// get maxfd
maxfdp1 = max(listenfd, udpfd) + 1;
```

```
for (;;) {

        // set listenfd and udpfd in readset
        FD_SET(listenfd, &rset);
        FD_SET(udpfd, &rset);

        // select the ready descriptor
        nready = select(maxfdp1, &rset, NULL, NULL, NULL);

        // if tcp socket is readable then handle
        // it by accepting the connection
 if (FD_ISSET(listenfd, &rset)) {
            len = sizeof(cliaddr);
            connfd = accept(listenfd, (struct sockaddr*)&cliaddr, &len);
            if ((childpid = fork()) == 0) {
                close(listenfd);
                while(1)
                {
                bzero(buffer, sizeof(buffer));
                printf("Message From TCP client: ");
                read(connfd, buffer, sizeof(buffer));
                puts(buffer);
                fgets(buffer,100,stdin);
                write(connfd, buffer, sizeof(buffer));
                }
                close(connfd);
                exit(0);
            }

        }
```

Server Program:

```
 // if udp socket is readable receive the message.
        if (FD_ISSET(udpfd, &rset)) {
            len = sizeof(cliaddr);
            bzero(buffer, sizeof(buffer));
            printf("\nMessage from UDP client: ");
            n = recvfrom(udpfd, buffer, sizeof(buffer), 0,
                        (struct sockaddr*)&cliaddr, &len);
            puts(buffer);
            fgets(buffer,100,stdin);
            sendto(udpfd, buffer, sizeof(buffer), 0,
                (struct sockaddr*)&cliaddr, sizeof(cliaddr));
        }
    }
}
```

TCP Client Program:

```
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <arpa/inet.h>
#include <unistd.h>
#define PORT 5000
#define MAXLINE 1024
int main()
{
    int sockfd;
    char buffer[MAXLINE];
    char* message = "Hello Server";
    struct sockaddr_in servaddr;

    int n, len;
    // Creating socket file descriptor
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        printf("socket creation failed");
        exit(0);
    }

    memset(&servaddr, 0, sizeof(servaddr));
```

TCP Client Program:

```
// Filling server information
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(PORT);
    servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");

    if (connect(sockfd, (struct sockaddr*)&servaddr,
                            sizeof(servaddr)) < 0) {
        printf("\n Error : Connect Failed \n");
    }
         while(1)
         {
    memset(buffer, 0, sizeof(buffer));
    fgets(buffer,100,stdin);
    //strcpy(buffer, "Hello Server");
    write(sockfd, buffer, sizeof(buffer));
    printf("Message from server: ");
    read(sockfd, buffer, sizeof(buffer));
    puts(buffer);
    }
    close(sockfd);
}
```

UDP Client Program:

```
// UDP client program
#include <arpa/inet.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>
#define PORT 5000
#define MAXLINE 1024
int main()
{
    int sockfd;
    char buffer[MAXLINE];
    char* message = "Hello Server";
    struct sockaddr_in servaddr;

    int n, len;
    // Creating socket file descriptor
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        printf("socket creation failed");
        exit(0);
    }

    memset(&servaddr, 0, sizeof(servaddr));
```

UDP Client Program:

```
 // Filling server information
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(PORT);
    servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    // send hello message to server
    while(1)
    {
    fgets(buffer,100,stdin);
    sendto(sockfd, buffer, strlen(buffer),
        0, (const struct sockaddr*)&servaddr,
        sizeof(servaddr));

    // receive server's response
    printf("Message from server: ");
    n = recvfrom(sockfd, (char*)buffer, MAXLINE,
                0, (struct sockaddr*)&servaddr,
                &len);
    puts(buffer);
    }
    close(sockfd);
    return 0;
}
```

Steps to compile and run the codes

1. Compile the server program (gcc server.c -o ser)
2. Run server using (./ser)
3. On another terminal, compile tcp client program (gcc tcp_client.c -o tcpcli)
4. Run tcp client (./tcpcli)
5. On another terminal, compile udp client program (gcc udp_client.c -o udpcli)
6. Run udp client (./udpcli)