UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"

Faculdade de Engenharia e Ciências de Guaratinguetá
Departamento de Engenharia Elétrica

Sistemas Microcomputadorizados

Laboratório 6

Comunicação UDP/IP

Sumário

1 Objetivos **2**

2 O Protocolo UDP **2**

3 Programação em C usando Sockets e o protocolo UDP **3**
3.1 Passos para a criação do socket no lado do cliente . . . 3
3.1.1 Passos para a criação do socket no lado do servidor . . . 3

4 Exemplo de Programação **4**
4.1 Arquivo de cabeçalho com definições comuns . . . 4
4.2 Servidor para Linux . . . 4
4.2.1 Arquivo de código-fonte `udpserver.c` . . . 4
4.2.2 Arquivo de código-fonte `echoserver.h` . . . 5
4.2.3 Arquivo de código-fonte `echoserver.c` . . . 5
4.2.4 Arquivo `Makefile` . . . 6
4.3 Cliente para Linux . . . 6
4.3.1 Arquivo de código-fonte `udpclient.c` . . . 6
4.3.2 Arquivo de código-fonte `messageclient.h` . . . 7
4.3.3 Arquivo de código-fonte `messageclient.c` . . . 7
4.3.4 Arquivo `Makefile` . . . 8
4.4 Servidor para Windows . . . 9
4.4.1 Arquivo de código-fonte `udpserver.c` . . . 9
4.4.2 Arquivo de código-fonte `echoserver.h` . . . 10
4.4.3 Arquivo de código-fonte `echoserver.c` . . . 10
4.4.4 Arquivo Makefile . . . 11
4.5 Cliente para Windows . . . 11
4.5.1 Arquivo de código-fonte `udpclient.c` . . . 12
4.5.2 Arquivo de código-fonte `messageclient.h` . . . 12
4.5.3 Arquivo de código-fonte `messageclient.c` . . . 12
4.5.4 Arquivo Makefile . . . 13

1 Objetivos

- Estudar e implementar a comunicação UDP/IP entre diferentes dispositivos de uma rede.

2 O Protocolo UDP

O conjunto de protocolos da Internet admite um protocolo de transporte não orientado a conexões, o protocolo de datagrama do usuário, ou **UDP** (*User Datagram Protocol*). O UDP oferece um meio para as aplicações enviarem datagramas IP encapsulados sem que seja necessário estabelecer uma conexão.

O UDP transmite **segmentos** que consistem em um cabeçalho de 8 bytes, seguido pela carga útil (*payload*). O cabeçalho é mostrado na Figura 1. Os dois números de **portas** servem para identificar processos nas máquinas de origem e destino. Quando um pacote UDP é recebido, sua carga útil é entregue ao processo associado à porta de destino. Essa associação ocorre quando a primitiva `BIND`, ou algo semelhante, é utilizada. Pense nas portas como caixas de correio que as aplicações podem utilizar para receber pacotes. De fato, a principal vantagem em utilizar o UDP em relação ao uso do IP bruto é a adição das portas de origem e destino. Sem os campos de números de portas, a camada de transporte não saberia o que fazer como pacote recebido. Com eles, a camada entrega o segmento encapsulado à aplicação correta.

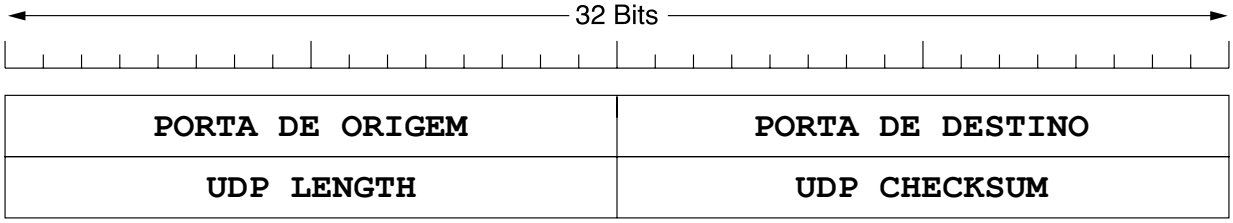


Figura 1: Cabeçalho UDP [1].

A porta de origem é necessária, principalmente, quando uma resposta precisa ser enviada de volta à origem. Copiando o campo `PORTA DE ORIGEM` do segmento de entrada no campo `PORTA DE DESTINO` do segmento de saída, o processo que transmite a resposta pode especificar qual processo na máquina transmissora deve recebê-lo.

O campo `UDP LENGTH` inclui o cabeçalho de 8 bytes e os dados. O comprimento mínimo é de 8 bytes, para incluir o cabeçalho. O comprimento máximo é de 65.515 bytes, que é menor que o maior número que caberá em 16 bits, devido ao limite de tamanho nos pacotes IP.

Um campo opcional de `UDP CHECKSUM` também é fornecido para gerar confiabilidade extra. Ele faz o *checksum* do cabeçalho, dos dados e de um pseudocabeçalho conceitual do IP. Ao realizar um cálculo, o campo de *checksum* é definido como zero e o campo de dados é preenchido com um byte zero adicional se seu comprimento for um número ímpar. O algoritmo de checksum consiste, simplesmente, em somar todas as palavras de 16 bits com complemento de um e apanhar o complemento de um da soma. Por conseguinte, quando o receptor realizar o cálculo sobre o segmento inteiro, incluindo o campo de *Checksum*, o resultado deve ser 0. Se o checksum não for calculado, ele será armazenado como zero, pois, por uma feliz coincidência da aritmética de complemento de um, um valor 0 verdadeiro calculado é armazenado com todos os bits iguais a 1. É tolice desativá-lo, a menos que a qualidade dos dados não tenha importância (por exemplo, no caso de voz digitalizada).

Vale a pena mencionar algumas ações que o UDP **não** realiza. Ele não realiza controle de fluxo, controle de congestionamento ou retransmissão após a recepção de um segmento incorreto. Tudo isso cabe aos processos do usuário. O que ele faz é fornecer uma interface para o protocolo IP com o

recurso adicional de demultiplexação de vários processos que utilizam as portas e detecção opcional de erro fim a fim [1].

3 Programação em C usando Sockets e o protocolo UDP

Os programas cliente e servidor utilizando sockets e o protocolo UDP são bastante similares em relação às suas versões utilizando o protocolo TCP, conforme vimos no laboratório anterior. A grande diferença é que, neste caso, não utilizaremos as intruções para estabelecer uma conexão entre os dois *hosts*, além de alguns parâmetros diferentes que são utilizados durante a chamada ao método `socket()`.

3.1 Passos para a criação do socket no lado do cliente

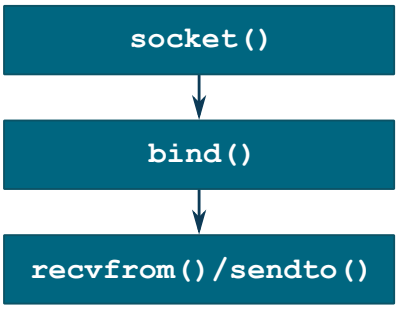
A criação de um socket no lado cliente envolve, basicamente, 2 passos, conforme enumerado abaixo e ilustrado pela Figura 2b.

1. Criar um socket com a chamada ao sistema `socket()` [2].
2. Enviar e receber dados utilizando as funções `sendto()` [3] e `recvfrom()` [4].

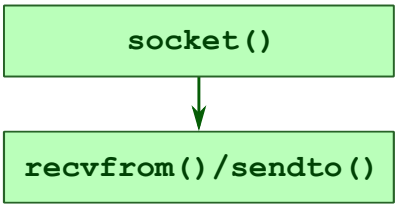
3.1.1 Passos para a criação do socket no lado do servidor

De forma semelhante, a criação do socket no lado servidor emprega os passos enumerados abaixo e ilustrados pela figura 2a.

1. Criar um socket com a chamada ao sistema `socket()` [2].
2. Vincular o socket a um endereço utilizando a função `bind()` [5].
3. Enviar e receber dados usando as funções `sendto()` e `recvfrom()`.



(a) Diagrama para o servidor.



(b) Diagrama para o cliente.

Figura 2: Diagramas para a construção do socket utilizando o protocolo UDP.

4 Exemplo de Programação

Neste exemplo de programação, assim como no laboratório anterior, vamos criar os processos `echoserver` (servidor) e `udpclient` (cliente). A diferença é que, desta vez, faremos uso do protocolo UDP ao invés do protocolo TCP.

4.1 Arquivo de cabeçalho com definições comuns

Um arquivo de cabeçalho, denominado "`defs.h`", que possui algumas definições comuns tanto ao cliente quanto ao servidor, foi criado separadamente dos demais arquivos de forma a facilitar o desenvolvimento. Este arquivo foi salvo em um diretório chamado `include`.

Bloco de código 1: Arquivo de cabeçalho `defs.h`.

```
#ifndef _DEFS_H_
#define _DEFS_H_

#define TRUE 1
#define FALSE 0

#define MAX 256

typedef struct sockaddr_in SockAddrIn;
typedef struct sockaddr SockAddr;

#endif
```

4.2 Servidor para Linux

O código utilizado para implementar o servidor foi dividido em três arquivos.

4.2.1 Arquivo de código-fonte `udpserver.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>

#include <sys/socket.h>
#include <netinet/in.h>

#include "defs.h"
#include "echoserver.h"

int main(int argc, char *argv[]) {
  if (argc < 2) {
    fprintf(stderr, "Usage: %s SERVER_PORT.\n", argv[0]);
    fprintf(stderr, "Received only %d parameters.\n", argc);
    fprintf(stderr, "Execution aborted.\n");
    exit(EXIT_FAILURE);
  }

  int server_socket;
  SockAddrIn server_address;
  unsigned long int port = strtoul(argv[1], NULL, 0);

  // Create socket
  server_socket = socket(AF_INET, SOCK_DGRAM, 0);
  if (server_socket < 0) {
    fprintf(stderr, "Failed to create socket.\n");
    fprintf(stderr, "Error: %s\n", strerror(errno));
    exit(EXIT_FAILURE);
  } else {
```

```
32      fprintf(stdout, "Socket successfully created.\n");
33    }
34
35    memset(&server_address, 0, sizeof(server_address));
36
37    // Configure server IP address and PORT
38    server_address.sin_family = AF_INET;
39    server_address.sin_addr.s_addr = htonl(INADDR_ANY);
40    server_address.sin_port = htons(port);
41
42    // Bind newly created socket to server IP address
43    int bind_result = bind(server_socket, (SockAddr*) &server_address, sizeof(server_address));
44    if (bind_result < 0) {
45      fprintf(stderr, "Failed to bind socket to address.\n");
46      fprintf(stderr, "Error: %s\n", strerror(errno));
47      close(server_socket);
48      exit(EXIT_FAILURE);
49    } else {
50      fprintf(stdout, "Socket successfully bound.\n");
51    }
52
53    // Echo function. Defined in echoserver.h
54    echo(server_socket);
55
56    // Close the socket when the server finishes its execution
57    close(server_socket);
58
59    return EXIT_SUCCESS;
60  }
```

4.2.2 Arquivo de código-fonte `echoserver.h`

```
1  #ifndef _ECHOSERVER_H_
2  #define _ECHOSERVER_H_
3
4  void echo(int socket_handle);
5
6  #endif
```

4.2.3 Arquivo de código-fonte `echoserver.c`

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <errno.h>
4  #include <netinet/in.h>
5
6  #include "defs.h"
7  #include "echoserver.h"
8
9  void echo(int socket_handle) {
10   char buff[MAX];
11   socklen_t len;
12   ssize_t comm_len;
13   SockAddrIn client_address;
14
15   len = sizeof(client_address);
16
17   while(TRUE) {
18     // Clear buffer
19     memset(&buff, 0, sizeof(buff));
20     memset(&client_address, 0, sizeof(client_address));
21
22     // Read message from client and copy it to the buffer
23     comm_len = recvfrom(socket_handle, (char *) &buff, sizeof(buff), 0, (SockAddr *) &client_address
       , &len);
24     if (comm_len < 0) {
25       fprintf(stderr, "An error occurred while receiving data.\n");
26       fprintf(stderr, "Error: %s\n", strerror(errno));
```

```
      break;
    }

    // Print buffer content
    fprintf(stdout, "[Message from client] %s\n", buff);

    // If incoming message contains "exit", finish server execution
    if (strncmp("exit", buff, 4) == 0) {
      fprintf(stdout, "Server execution finished.");
      break;
    }

    // Send incoming message back to client (echo)
    comm_len = sendto(socket_handle, (char *) &buff, sizeof(buff), 0, (SockAddr *) &client_address,
    len);
    if (comm_len < 0) {
      fprintf(stderr, "An error occurred while sending data.\n");
      fprintf(stderr, "Error: %s\n", strerror(errno));
      break;
    }
  }
}
```

4.2.4 Arquivo `Makefile`

```
TARGET=udp_server
TARGET_SOURCES=udpserver.c echoserver.c

BINDIR=../bin

FLAGS=-O2 -Wall -MMD
LIBS=

INCLUDE=-I. -I../include/

CMP = gcc
LDFLAGS=$(LIBS)

all: install clean

install: $(TARGET)
  mkdir -p $(BINDIR)
  mv $(TARGET) $(BINDIR)

$(TARGET): $(TARGET_SOURCES:.c = .o)
  $(CMP) $(FLAGS) $(INCLUDE) -o $@ $^ $(LDFLAGS)

%.o: %.c
  $(CMP) $(FLAGS) $(INCLUDE) -c -o $@ $<

-include $(TARGET_SOURCES:.c=.d)

clean:
  @rm -rf *.o *.d *~

distclean: clean
  @rm $(TARGET)
```

4.3 Cliente para Linux

De forma similar ao servidor, o código para implementar o cliente também foi dividido em três arquivos.

4.3.1 Arquivo de código-fonte `udpclient.c`

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <arpa/inet.h>
#include <sys/socket.h>

#include "defs.h"
#include "messageclient.h"

int main(int argc, char *argv[]) {
  if (argc < 3) {
    fprintf(stderr, "Usage: %s SERVER_IP SERVER_PORT.\n", argv[0]);
    fprintf(stderr, "Received only %d parameters.\n", argc);
    fprintf(stderr, "Execution aborted.\n");
    exit(EXIT_FAILURE);
  }

  int client_socket;
  SockAddrIn server_address;
  char *server_ip = argv[1];
  unsigned long int port = strtoul(argv[2], NULL, 0);

  // Create socket
  client_socket = socket(AF_INET, SOCK_DGRAM, 0);
  if (client_socket < 0) {
    fprintf(stderr, "Failed to create socket.\n");
    fprintf(stderr, "Error: %s\n", strerror(errno));
    exit(EXIT_FAILURE);
  } else {
    fprintf(stdout, "Socket successfully created.\n");
  }

  // Reset server address to 0 before usage
  memset(&server_address, 0, sizeof(server_address));

  // Set server address and port
  server_address.sin_family = AF_INET;
  server_address.sin_addr.s_addr = inet_addr(server_ip);
  server_address.sin_port = htons(port);

  // Function to send messages to server
  msgclient(client_socket, server_address);

  // Close the socket
  close(client_socket);

  return EXIT_SUCCESS;
}
```

4.3.2 Arquivo de código-fonte `messageclient.h`

```c
#ifndef _MESSAGECLIENT_H_
#define _MESSAGECLIENT_H_

void msgclient(int socket_handle, SockAddrIn server_address);

#endif
```

4.3.3 Arquivo de código-fonte `messageclient.c`

```c
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdbool.h>

```

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

#include "defs.h"
#include "messageclient.h"

void msgclient(int socket_handle, SockAddrIn server_address) {
  char buff[MAX];
  char letter;
  int n;
  ssize_t comm_len;
  socklen_t recv_len = 0;

  while (true) {
    // Clear buffer
    memset(&buff, 0, sizeof(buff));
    memset(&letter, 0, sizeof(letter));
    n = 0;

    fprintf(stdout, "Type a message: ");
    while (TRUE) {
      letter = getchar();

      if (letter == '\n') break;

      buff[n] = letter;
      n++;
    }

    // Send message via socket
    comm_len = sendto(socket_handle, (char *) &buff, sizeof(buff), 0, (SockAddr *) &server_address,
    sizeof(server_address));
    if (comm_len < 0) {
      fprintf(stderr, "An error occurred while sending data.\n");
      fprintf(stderr, "Error: %s\n", strerror(errno));
      break;
    }

    int compare_result = strncmp(buff, "exit", 4);
    if (compare_result == 0) {
      fprintf(stdout, "Client exit...\n");
      break;
    }

    // Clear buffer
    memset(&buff, 0, sizeof(buff));

    recv_len = sizeof(server_address);
    // Wait for server response
    comm_len = recvfrom(socket_handle, (char *) &buff, sizeof(buff), 0, (SockAddr *) &server_address
    , &recv_len);
    if (comm_len < 0) {
      fprintf(stderr, "An error occurred while receiving data.\n");
      fprintf(stderr, "Error: %s\n", strerror(errno));
      break;
    }

    fprintf(stdout, "Received from Server: %s\n", buff);
  }
}
```

4.3.4 Arquivo `Makefile`

```
TARGET=udp_client
TARGET_SOURCES=udpclient.c messageclient.c

BINDIR=../bin

FLAGS=-O2 -Wall -MMD
LIBS=

```

```
INCLUDE=-I. -I../include/

CMP = gcc
LDFLAGS=$(LIBS)

all: install clean

install: $(TARGET)
  mkdir -p $(BINDIR)
  mv $(TARGET) $(BINDIR)

$(TARGET): $(TARGET_SOURCES:.c = .o)
  $(CMP) $(FLAGS) $(INCLUDE) -o $@ $^ $(LDFLAGS)

%.o: %.c
  $(CMP) $(FLAGS) $(INCLUDE) -c -o $@ $<

-include $(TARGET_SOURCES:.c=.d)

clean:
  @rm -rf *.o *.d *~

distclean: clean
  @rm $(TARGET)
```

4.4 Servidor para Windows

O código apresentado abaixo é utilizado para implementar um servidor para Windows. Repare que algumas chamadas às funções do sistema são um pouco diferentes daquelas utilizadas no cliente para Linux e refletem a utilização de bibliotecas específicas de cada plataforma.

4.4.1 Arquivo de código-fonte `udpserver.c`

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <winsock2.h>

#include "defs.h"
#include "echoserver.h"

int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s SERVER_PORT.\n", argv[0]);
        fprintf(stderr, "Received only %d parameters.\n", argc);
        fprintf(stderr, "Execution aborted.\n");
        exit(EXIT_FAILURE);
    }

    int server_socket;
    SockAddrIn server_address;
    unsigned long int port = strtoul(argv[1], NULL, 0);
    WSADATA wsadata;

    /* Load Winsock 2.0 DLL */
    int load_result = WSAStartup(MAKEWORD(2, 2), &wsadata);
    if (load_result != 0) {
        fprintf(stderr, "Failed do load Winsock library.\n");
        exit(EXIT_FAILURE);
    }

    // Create socket
    server_socket = socket(AF_INET, SOCK_DGRAM, 0);
    if (server_socket < 0) {
        fprintf(stderr, "Failed to create socket.\n");
        fprintf(stderr, "Error: %s\n", strerror(errno));
        exit(EXIT_FAILURE);
```

```
36      } else {
37          fprintf(stdout, "Socket successfully created.\n");
38      }
39
40      // Reset server address to 0 before usage
41      memset(&server_address, 0, sizeof(server_address));
42
43      // Configure server IP address and PORT
44      server_address.sin_family = AF_INET;
45      server_address.sin_addr.s_addr = htonl(INADDR_ANY);
46      server_address.sin_port = htons(port);
47
48      // Bind newly created socket to IP address
49      int bind_result = bind(server_socket, (SockAddr *) &server_address, sizeof(server_address));
50      if (bind_result < 0) {
51          fprintf(stderr, "Failed to bind socket to address.\n");
52          fprintf(stderr, "Error: %s\n", strerror(errno));
53          closesocket(server_socket);
54          WSACleanup();
55          exit(EXIT_FAILURE);
56      } else {
57          fprintf(stdout, "Socket successfully bound.\n");
58      }
59
60      // Echo function. Defined in echoserver.h
61      echo(server_socket);
62
63      // Close the socket when the server finishes its execution
64      closesocket(server_socket);
65      WSACleanup();
66      return 0;
67  }
```

4.4.2 Arquivo de código-fonte `echoserver.h`

```
1  #ifndef _ECHOSERVER_H_
2  #define _ECHOSERVER_H_
3
4  void echo(int socket_handle);
5
6  #endif /* _ECHOSERVER_H_ */
```

4.4.3 Arquivo de código-fonte `echoserver.c`

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <errno.h>
4  #include <winsock2.h>
5
6  #include "defs.h"
7  #include "echoserver.h"
8
9  void echo(int socket_handle) {
10   char buff[MAX];
11   int len;
12   ssize_t comm_len;
13   SockAddrIn client_address;
14
15   len = sizeof(client_address);
16
17   while(TRUE) {
18     // Clear buffer
19     memset(&buff, 0, sizeof(buff));
20     memset(&client_address, 0, sizeof(client_address));
21
22     // Read message from client and copy it to the buffer
23     comm_len = recvfrom(socket_handle, (char *) &buff, sizeof(buff), 0, (SockAddr *) &client_address
       , &len);
```

```
24      if (comm_len < 0) {
25        fprintf(stderr, "An error occurred while receiving data.\n");
26        fprintf(stderr, "Error: %s\n", strerror(errno));
27        break;
28      }
29
30      // Print buffer content
31      fprintf(stdout, "[Message from client] %s\n", buff);
32
33      // If incoming message contains "exit", finish server execution
34      if (strncmp ("exit", buff, 4) == 0) {
35        fprintf(stdout, "Server execution finished.\n");
36        break;
37      }
38
39      // Send incoming message back to client (echo)
40      comm_len = sendto(socket_handle, (char *) &buff, sizeof(buff), 0, (SockAddr *) &client_address,
        len);
41      if (comm_len < 0) {
42        fprintf(stderr, "An error occurred while sending data.\n");
43        fprintf(stderr, "Error: %s\n", strerror(errno));
44        break;
45      }
46    }
47  }
```

4.4.4 Arquivo Makefile

```
1  TARGET=udpserver
2  TARGET_SOURCES=echoserver.c udpserver.c
3
4  FLAGS=-O2 -Wall -MMD
5  LIBS=-lws2_32
6
7  INCLUDE=-I. -I..\include
8  BINDIR=..\bin
9
10 CMP=gcc
11 LDFLAGS=$(LIBS) -LC:\mingw64\x86_64-w64-mingw32\lib
12
13 .PHONY=clean distclean
14
15 all: install clean
16
17 install: $(TARGET)
18   if not exist $(BINDIR) md $(BINDIR)
19   move $(TARGET).exe $(BINDIR)
20
21 $(TARGET): $(TARGET_SOURCES:.c=.o)
22   $(CMP) $(FLAGS) $(INCLUDE) -o $@ $^ $(LDFLAGS)
23
24 %.o: %.c
25   $(CMP) $(FLAGS) $(INCLUDE) -c -o $@ $<
26
27 -include $(TARGET_SOURCES:.c=.d)
28
29 clean:
30   del *.o *.d *~
31
32 distclean: clean
33   del $(TARGET).exe
```

4.5 Cliente para Windows

De forma similar ao servidor para Windows, o cliente também apresenta algumas diferenças em relação ao código para Linux.

4.5.1 Arquivo de código-fonte `udpclient.c`

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <winsock2.h>

#include "defs.h"
#include "messageclient.h"

int main(int argc, char *argv[]) {
  if (argc < 3) {
    fprintf(stderr, "Usage: %s SERVER_IP SERVER_PORT.\n", argv[0]);
    fprintf(stderr, "Received only %d parameters.\n", argc);
    fprintf(stderr, "Execution aborted.\n");
    exit(EXIT_FAILURE);
  }

  int client_socket;
  SockAddrIn server_address;
  WSADATA wsaData;
  char *server_ip = argv[1];
  unsigned long int port = strtoul(argv[2], NULL, 0);

  /* Load Winsock 2.0 DLL */
  int load_result = WSAStartup(MAKEWORD(2, 2), &wsaData);
  if (load_result != 0) {
    fprintf(stderr, "Failed to load Winsock library.\n");
    exit(EXIT_FAILURE);
  }

  // Create Socket
  client_socket = socket(AF_INET, SOCK_DGRAM, 0);
  if(client_socket < 0) {
    fprintf(stderr, "Failed to create socket.\n");
    fprintf(stderr, "Error: %s\n", strerror(errno));
    exit(EXIT_FAILURE);
  } else {
    fprintf(stdout, "Socket successfully created.\n");
  }

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

  // Set server address and port
  server_address.sin_family = AF_INET;
  server_address.sin_addr.s_addr = inet_addr(server_ip);
  server_address.sin_port = htons(port);

  // Function to send messages to server
  msgclient(client_socket, server_address);

  // Close socket
  closesocket(client_socket);
  WSACleanup();
  return 0;
}
```

4.5.2 Arquivo de código-fonte `messageclient.h`

```c
#ifndef _MESSAGECLIENT_H_
#define _MESSAGECLIENT_H_

void msgclient(int socket_handle, SockAddrIn server_address);

#endif
```

4.5.3 Arquivo de código-fonte `messageclient.c`

```c
#include <stdio.h>
#include <string.h>
#include <errno.h>

#include <winsock.h>

#include "defs.h"
#include "messageclient.h"

void msgclient(int socket_handle, SockAddrIn server_address) {
  char buff[MAX];
  char letter;
  int n;
  ssize_t comm_len;
  int recv_len = 0;

  while (1) {
    // Clear Buffer
    memset(&buff, 0, sizeof(buff));
    memset(&letter, 0, sizeof(letter));
    n = 0;

    fprintf(stdout, "Type a message: ");

    // Save message to buffer
    while (1) {
      letter = getchar();

      if (letter == '\n') break;

      buff[n] = letter;
      n++;
    }

    // Send message via socket
    comm_len = sendto(socket_handle, (char *) &buff, sizeof(buff), 0, (SockAddr *) &server_address,
    sizeof(server_address));
    if (comm_len < 0) {
      fprintf(stderr, "An error occurred while sending data.\n");
      fprintf(stderr, "Error: %s\n", strerror(errno));
      break;
    }

    int compare_result = strncmp(buff, "exit", 4);
    if (compare_result == 0) {
      fprintf(stdout, "Client Exit...\n");
      break;
    }

    // Clear buffer
    memset(&buff, 0, sizeof(buff));

    recv_len = sizeof(server_address);
    // Wait for server response
    comm_len = recvfrom(socket_handle, (char *) &buff, sizeof(buff), 0, (SockAddr *) &server_address
    , &recv_len);
    if (comm_len < 0) {
      fprintf(stderr, "An error occurred while receiving data.\n");
      fprintf(stderr, "Error: %s\n", strerror(errno));
      break;
    }
    fprintf(stdout, "Received from Server: %s\n", buff);
  }
}
```

4.5.4 Arquivo Makefile

```
TARGET=udpclient
TARGET_SOURCES=udpclient.c messageclient.c

```

```
4  FLAGS=-O2 -Wall -MMD
5  LIBS=-lws2_32
6
7  INCLUDE=-I. -I..\include
8  BINDIR=..\bin
9
10 CMP=gcc
11 LDFLAGS=$(LIBS) -LC:\mingw64\x86_64-w64-mingw32\lib
12
13 .PHONY=clean distclean
14
15 all: install clean
16
17 install: $(TARGET)
18   if not exist $(BINDIR) md $(BINDIR)
19   move $(TARGET).exe $(BINDIR)
20
21 $(TARGET): $(TARGET_SOURCES:.c=.o)
22   $(CMP) $(FLAGS) $(INCLUDE) -o $@ $^ $(LDFLAGS)
23
24 %.o: %.c
25   $(CMP) $(FLAGS) $(INCLUDE) -c -o $@ $<
26
27 -include $(TARGET_SOURCES:.c=.d)
28
29 clean:
30   del *.o *.d *~
31
32 distclean: clean
33   del $(TARGET).exe
```

Referências

[1] A. S. Tanembaum, N. Feamster, and D. Wetherall, *Computer Networks*. Pearson Education Limited, 6th ed., 2021.

[2] M. Kerrisk, "socket(2) – linux manual page." https://man7.org/linux/man-pages/man2/socket.2.html. Último acesso em: 22/05/2024.

[3] M. Kerrisk, "send(2) – linux manual page." https://man7.org/linux/man-pages/man2/send.2.html. Último acesso em: 22/05/2024.

[4] M. Kerrisk, "recv(2) – linux manual page." https://man7.org/linux/man-pages/man2/recv.2.html. Último acesso em: 22/05/2024.

[5] M. Kerrisk, "bind(2) – linux manual page." https://man7.org/linux/man-pages/man2/bind.2.html. Último acesso em: 22/05/2024.