UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"

Faculdade de Engenharia e Ciências de Guaratinguetá
Departamento de Engenharia Elétrica

Sistemas Microcomputadorizados

Laboratório 5

Comunicação TCP/IP

Sumário

1 Objetivos

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

2 Introdução

Em conjunto com a camada de rede, a camada de transporte é o núcleo da hierarquia de protocolos utilizados na Internet. A camada de rede oferece os meios para remessas de pacotes fim a fim utilizando datagramas ou circuitos virtuais. A camada de transporte se baseia na camada de rede para oferecer transporte de dados de um processo em uma máquina de origem a um processo em uma máquina de destino com um nível de confiabilidade desejado, independentemente das redes físicas em uso durante a comunicação. Ela provê as abstrações de que as aplicações precisam para usar a rede [1].

A Internet possui dois protocolos principais na camada de transporte, um orientado e outro não-orientado a conexões. Eles complementam um ao outro. O protocolo orientado a conexões é o TCP, enquanto o protocolo não-orientado a conexões é o UDP.

2.1 Serviços orientados e não orientados a conexões

Um serviço orientado a conexão se baseia no conceito do sistema telefônico. Para falar com alguém, você tira o telefone do gancho, digita o número, fala e, em seguida, desliga. Da mesma forma, para utilizar um serviço de rede orientado a conexão, primeiro o usuário do serviço estabelece uma conexão, a utiliza e, depois, a libera. O aspecto essencial de uma conexão é que ela funciona como uma espécie de "tubo" (Figura 1): o transmissor empurra objetos (neste caso, bits) em uma extremidade, enquanto o receptor recebe esses objetos na outra extremidade. Na maioria dos casos, a ordem é preservada, de forma que os bits chegam na sequência em que foram enviados.

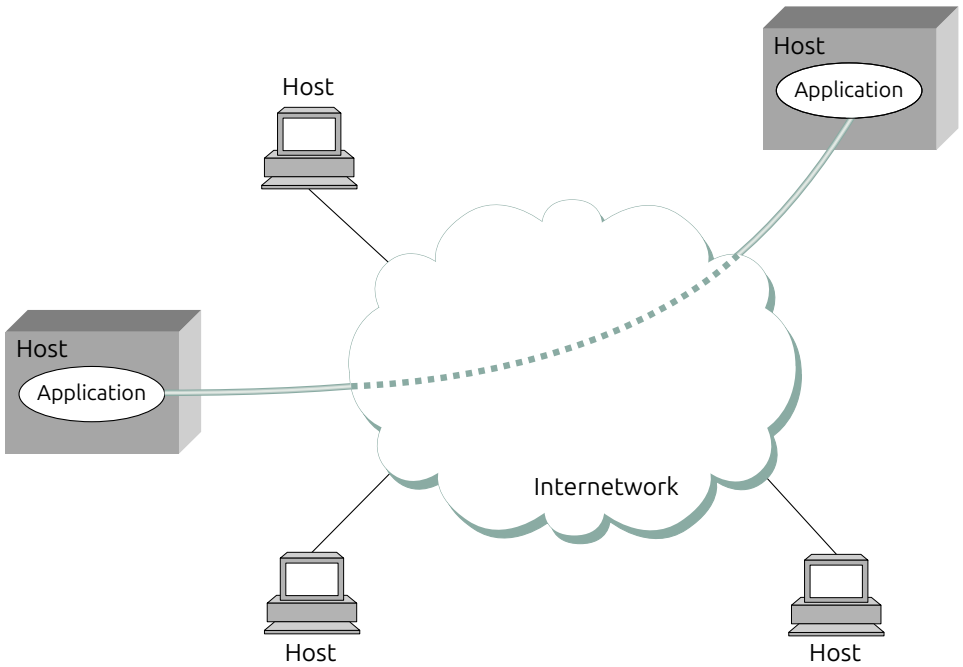


Figura 1: Processos comunicando-se através de uma conexão [2].

Em alguns casos, quando uma conexão é estabelecida, o transmissor, o receptor e a sub-rede conduzem uma negociação sobre os parâmetros a serem utilizados, como o tamanho máximo das

mensagens, a qualidade do serviço exigida, entre outras questões. Em geral, um lado faz uma proposta e a outra parte pode aceitá-la, rejeitá-la ou fazer uma contraproposta.

Um serviço não-orientado a conexão, por sua vez, baseia-se na ideia do sistema postal. Cada mensagem (que seria análogo a uma carta) carrega o endereço completo do destino e é roteada pelos nós intermediários através do sistema, independentemente de todas as outras. Existem diferentes nomes para mensagens em diferentes contextos; um pacote é uma mensagem na camada de rede. Quando os nós intermediários recebem uma mensagem completa antes de enviá-la para o próximo nó, isso é chamado de comutação *store-and-forward*. A alternativa, em que a transmissão de uma mensagem em um nó começa antes de ser completamente recebida por ele, é chamada de comutação *cut-through*. Normalmente, quando duas mensagens são enviadas ao mesmo destino, a primeira a ser enviada é a primeira a chegar. No entanto, é possível que a primeira mensagem a ser enviada esteja atrasada, de modo que a segunda chegue primeiro.

Cada tipo de serviço pode ser caracterizado por sua confiabilidade. Alguns serviços são confiáveis, no sentido de nunca perderem dados. Em geral, um serviço confiável é implementado para que o receptor confirme o recebimento de cada mensagem, de modo que o transmissor se certifique de que ela chegou. O processo de confirmação introduz *overhead* e atrasos, que normalmente compensam, mas às vezes são indesejáveis.

Para algumas aplicações, os atrasos introduzidos pelas confirmações são inaceitáveis. Uma dessas aplicações é o tráfego de voz digital por *Voice over IP* (*VoIP*). Os usuários de telefone preferem ouvir um pouco de ruído na linha ou uma palavra truncada de vez em quando a experimentar um atraso para aguardar confirmações. O mesmo acontece durante a transmissão de uma conferência de vídeo; não haverá problema se aparecerem alguns *pixels* errados. No entanto, é irritante ver uma imagem parada enquanto o fluxo é interrompido e reiniciado para a correção de erros [1].

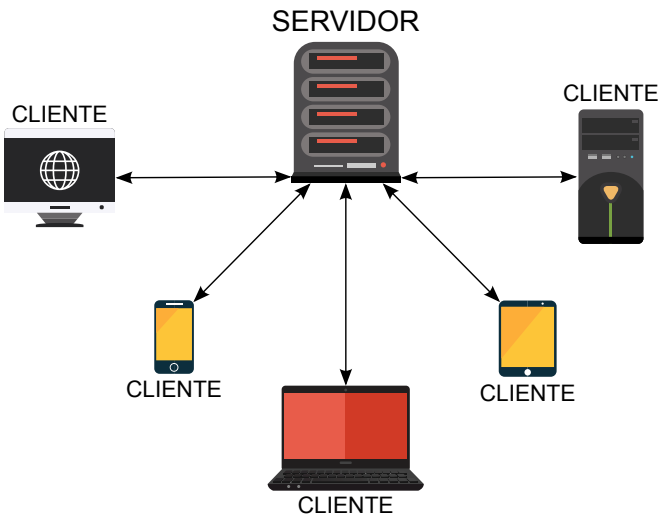
3 O Modelo Cliente-Servidor

O modelo cliente-servidor é um dos paradigmas mais utilizados em sistemas em rede. Processos do tipo cliente, normalmente, se comunicam apenas com um servidor em um determinado instante de tempo. Sob a perspectiva do servidor, em qualquer instante de tempo, é comum que este esteja se comunicando com diversos clientes. Clientes precisam saber da existência e do endereço do servidor. Entretanto, isto não ocorre do lado do servidor, pois este não precisa ter conhecimento da existência do cliente antes que alguma conexão seja estabelecida [3]. A Figura 2a exibe um modelo desta estrutura de rede.

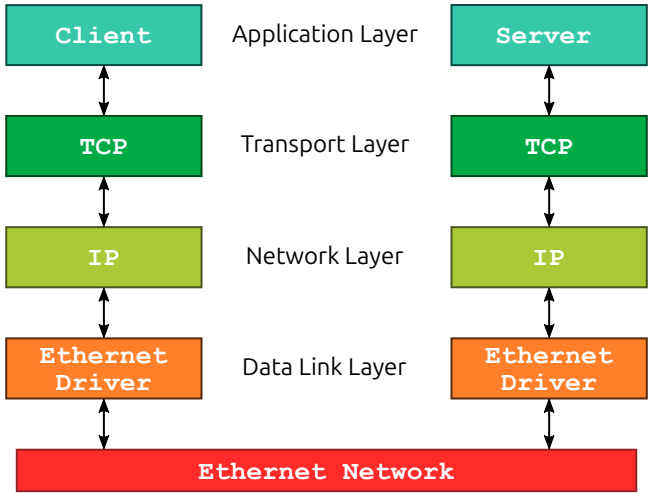
Clientes e servidores comunicam-se através de várias camadas de protocolos de rede. Neste laboratório, utilizaremos a pilha TCP/IP. A Figura 2b exibe um exemplo de uma rede local (LAN) com dois componentes.

4 Sockets

Todo sistema computacional possui uma forma de permitir com que seus aplicativos consigam se comunicar através de uma rede. Nestes sistemas, normalmente, os protocolos de comunicação são implementados como parte de seu sistema operacional, o qual, por sua vez, oferece uma interface a seus aplicativos para que estes acessem seus subsistemas de rede. Com o passar do tempo, algumas interfaces tornaram-se bastante populares e começaram a ser suportadas por diversos sistemas operacionais diferentes. Foi o que aconteceu com a interface chamada *Socket*, a qual, originalmente, foi concebida para o sistema Unix. Esta interface é comumente chamada de API (*Application Programming Interface*) da rede, uma vez que disponibiliza os serviços oferecidos pelo subsistema de rede do



(a) Modelo cliente-servidor.



(b) Exemplo de uma rede LAN Ethernet no modelo cliente servidor. Adaptado de [3].

sistema operacional [2].

Analisando de forma técnica, um *Socket* é um *endpoint* de uma comunicação bidirecional entre dois programas que trocam mensagens através de uma rede. Um socket é ligado a um número de porta[1], de forma que a camada TCP possa identificar o aplicativo para o qual os dados serão enviados. Um *endpoint* é uma combinação de um endereço IP e um número de porta. Cada conexão TCP pode ser identificada exclusivamente por seus dois terminais. Desta forma, é possível ter várias conexões entre seu *host* e o servidor [4].

Uma aplicação do tipo servidor, normalmente, atende a uma porta específica que aguarda solicitações de conexão de um cliente. Quando esta solicitação é recebida, o cliente e o servidor estabelecem uma conexão dedicada através da qual eles podem se comunicar. Durante o processo de conexão, o cliente recebe um número de porta local e liga um *socket* a ele. O cliente envia mensagens para o servidor ao escrever no socket, e recebe mensagens ao ler do socket. Da mesma forma, o servidor obtém um novo número de porta local (ele precisa deste novo número de porta para poder continuar aguardando por solicitações de conexão na porta original). O servidor também vincula um socket a sua porta local e se comunica com o cliente escrevendo e lendo o socket.

4.1 Programação TCP/IP utilizando Sockets em C

A maior parte da comunicação entre processos emprega o modelo cliente – servidor. Como mencionado anteriormente, estes termos referem-se aos dois processos que irão se comunicar. Um dos processos, o cliente, conecta-se ao outro processo, o servidor, para fazer algum tipo de requisição (informações ou serviços, por exemplo). Uma vez que a comunicação seja estabelecida, ambos lados podem enviar e receber informações.

As chamadas ao sistema para estabelecer uma conexão são um pouco diferentes para o cliente e o servidor. Porém, ambas envolvem a construção de um socket.

4.1.1 Passos para a criação de um socket no lado do cliente

A criação de um socket no lado cliente envolve, basicamente, 3 passos, conforme enumerado abaixo e ilustrado pela Figura 3b.

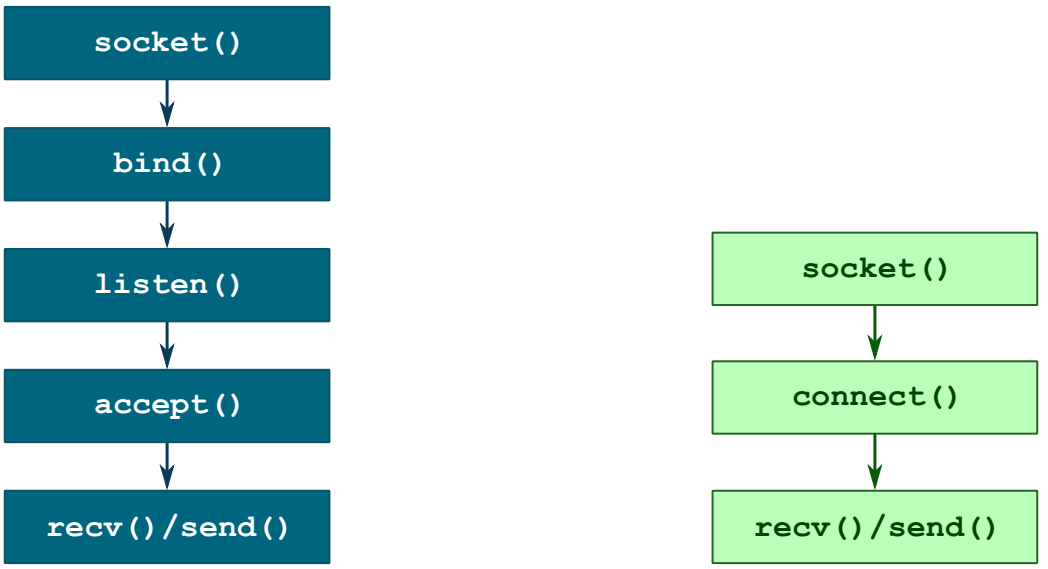
[1]Uma porta é um número que identifica uma aplicação. Algumas aplicações utilizam números de portas "famosas", como, por exemplo, a porta 21, a qual normalmente é utilizada por servidores FTP (*File Transfer Protocol*).

1. Criar um socket com a chamada ao sistema `socket()` [5].
2. Conectar o socket ao endereço do servidor utilizando a função `connect()` [6].
3. Enviar e receber dados. Existem diversas maneiras de se atingir este objetivo, mas a mais simples é utilizando as funções `read()` and `write()`. Entretanto, é aconselhável utilizar as funções `send()` [7] e `recv()` [8], uma vez que estas oferecem maior controle da comunicação.

4.1.2 Passos para a criação de um socket no lado do servidor

De forma semelhante, a criação de um socket no lado servidor emprega os passos enumerados abaixo e ilustrados pela figura 3a.

1. Criar um socket com a chamada ao sistema `socket()` [5].
2. Vincular o socket a um endereço utilizando a função `bind()` [9].
3. Escutar por solicitações de conexão através da chamada ao sistema `listen()` [10].
4. Aceitar a conexão utilizando a função `accept()` [11]. Esta chamada ao sistema normalmente bloqueia a execução do processo até que o cliente se conecte ao servidor.
5. Enviar e receber dados.



(a) Diagrama para o servidor. (b) Diagrama para o cliente.

Figura 3: Diagramas para a construção de um socket.

5 Exemplo de Programação

Neste exemplo, vamos criar dois processos: um servidor e um cliente. O servidor, denominado `echoserver`, possui apenas a função de retransmitir os dados recebidos de volta ao cliente. O cliente, por sua vez, transmitirá ao servidor os dados recebidos através da linha de comando.

5.1 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`.

```
#ifndef _DEFS_H_
#define _DEFS_H_

#define TRUE  1
#define MAX   256

typedef struct sockaddr_in SockAddrIn;
typedef struct sockaddr SockAddr;

#endif
```

5.2 Servidor para Linux

O código utilizado para implementar o servidor foi dividido em 3 arquivos.

5.2.1 Arquivo de código-fonte `server.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.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;
  int connection_socket;
  socklen_t conn_length;
  SockAddrIn server_address;
  SockAddrIn connection_address;
  unsigned long int port = strtoul(argv[1], NULL, 0);

  // Create socket
  server_socket = socket(AF_INET, SOCK_STREAM, 0);
  if (server_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");
  }
```

```
36
37    // Reset server address to 0 before usage
38    memset(&server_address, 0, sizeof(server_address));
39
40    // Configure server IP address and PORT
41    server_address.sin_family = AF_INET;
42    server_address.sin_addr.s_addr = htonl(INADDR_ANY);
43    server_address.sin_port = htons(port);
44
45    // Bind newly created socket to IP address
46    int bind_result = bind(server_socket, (SockAddr *) &server_address, sizeof(server_address));
47    if (bind_result < 0) {
48      fprintf(stderr, "Failed to bind socket to address.\n");
49      fprintf(stderr, "Error: %s\n", strerror(errno));
50      close(server_socket);
51      exit(EXIT_FAILURE);
52    } else {
53      fprintf(stdout, "Socket successfully bound.\n");
54    }
55
56    // Server ready to listen
57    int listen_result = listen(server_socket, 2);
58    if (listen_result < 0) {
59      fprintf(stderr, "Server failed to listen.\n");
60      fprintf(stderr, "Error: %s\n", strerror(errno));
61      close(server_socket);
62      exit(EXIT_FAILURE);
63    } else {
64      fprintf(stdout, "Server listening on port %lu\n", port);
65    }
66
67    conn_length = sizeof(connection_address);
68
69    // Accept data packet from client
70    connection_socket = accept(server_socket, (SockAddr *) &connection_address, &conn_length);
71    if (connection_socket < 0) {
72      fprintf(stderr, "Server connection to client failed.\n");
73      fprintf(stderr, "Error: %s\n", strerror(errno));
74      close(server_socket);
75      exit(EXIT_FAILURE);
76    } else {
77      fprintf(stdout, "Server successfully connected to client.\n");
78    }
79
80    // Echo function. Defined in echoserver.h
81    echo(connection_socket);
82
83    // Close the socket when the server finishes its execution
84    close(connection_socket);
85    close(server_socket);
86
87    return 0;
88  }
```

5.2.2 Arquivo de código-fonte `echoserver.h`

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

5.2.3 Arquivo de código-fonte `echoserver.c`

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdbool.h>
```

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

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

void echo(int socket_handle) {
  char buff[MAX];
  ssize_t comm_len;

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

    // Read message from client and copy it to the buffer
    comm_len = recv(socket_handle, (char *) &buff, sizeof(buff), 0);
    if (comm_len < 0) {
      fprintf(stderr, "An error occurred while receiving data.\n");
      fprintf(stderr, "Error: %s\n", strerror(errno));
      break;
    } else if (comm_len == 0) {
      fprintf(stderr, "Client disconnected. Shutting down.\n");
      break;
    }

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

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

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

5.2.4 Arquivo `Makefile`

```
TARGET=tcpserver
TARGET_SOURCES=server.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)
```

5.3 Cliente para Linux

De forma similar ao servidor, o código para implementar o cliente também foi dividido em 3 arquivos.

5.3.1 Arquivo de código-fonte `tcpclient.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);

  client_socket = socket(AF_INET, SOCK_STREAM, 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));

  server_address.sin_family = AF_INET;
  server_address.sin_addr.s_addr = inet_addr(server_ip);
  server_address.sin_port = htons(port);

  int connect_result = connect(client_socket, (SockAddr *) &server_address, sizeof(server_address));
  if (connect_result < 0) {
    fprintf(stderr, "Failed to connect to server.\n");
    fprintf(stderr, "Error: %s\n", strerror(errno));
    close(client_socket);
    exit(EXIT_FAILURE);
  } else {
    fprintf(stdout, "Connected to server.\n");
  }

  msgclient(client_socket);

```

```
52    close(client_socket);
53
54    return EXIT_SUCCESS;
55  }
```

5.3.2 Arquivo de código-fonte `messageclient.h`

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

5.3.3 Arquivo de código-fonte `messageclient.c`

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdbool.h>
4  #include <unistd.h>
5  #include <errno.h>
6
7  #include <sys/socket.h>
8
9  #include "defs.h"
10 #include "messageclient.h"
11
12 void msgclient(int socket_handle) {
13   char buff[MAX];
14   char letter;
15   int n;
16   ssize_t comm_len;
17
18   while (true) {
19     // Clear buffer
20     memset(&buff, 0, sizeof(buff));
21     memset(&letter, 0, sizeof(letter));
22     n = 0;
23
24     fprintf(stdout, "Type a message: ");
25
26     // Save message to buffer
27     while (true) {
28       letter = getchar();
29       if (letter == '\n') break;
30
31       buff[n] = letter;
32       n++;
33     }
34
35     // Send message via socket
36     comm_len = send(socket_handle, (char *) &buff, sizeof(buff), 0);
37     if (comm_len < 0) {
38       fprintf(stderr, "An error occurred while sending data.\n");
39       fprintf(stderr, "Error: %s\n", strerror(errno));
40       break;
41     }
42
43     int compare_result = strncmp(buff, "exit", 4);
44     if (compare_result == 0) {
45       fprintf(stdout, "Client Exit...\n");
46       break;
47     }
48
49     // Clear buffer
50     memset(&buff, 0, sizeof(buff));
51
52     // Wait for server response
```

```
53      comm_len = recv(socket_handle, (char *) &buff, sizeof(buff), 0);
54      if (comm_len < 0) {
55        fprintf(stderr, "An error occurred while receiving data.\n");
56        fprintf(stderr, "Error: %s\n", strerror(errno));
57        break;
58      }
59
60      fprintf(stdout, "Received from Server: %s\n", buff);
61    }
62  }
```

5.3.4 Arquivo `Makefile`

```
1  TARGET=tcp_client
2  TARGET_SOURCES=tcpclient.c messageclient.c
3
4  BINDIR=../bin
5
6  FLAGS=-O2 -Wall -MMD
7  LIBS=
8
9  INCLUDE=-I. -I../include/
10
11 CMP = gcc
12 LDFLAGS=$(LIBS)
13
14 all: install clean
15
16 install: $(TARGET)
17   mkdir -p $(BINDIR)
18   mv $(TARGET) $(BINDIR)
19
20 $(TARGET): $(TARGET_SOURCES:.c = .o)
21   $(CMP) $(FLAGS) $(INCLUDE) -o $@ $^ $(LDFLAGS)
22
23 %.o: %.c
24   $(CMP) $(FLAGS) $(INCLUDE) -c -o $@ $<
25
26 -include $(TARGET_SOURCES:.c=.d)
27
28 clean:
29   @rm -rf *.o *.d *~
30
31 distclean: clean
32   @rm $(TARGET)
```

5.4 Servidor para Windows

O código apresentado abaixo é utilizado para implementar o 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.

5.4.1 Arquivo de código-fonte `tcpserver.c`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <errno.h>
5  #include <winsock2.h>
6
7  #include "defs.h"
8  #include "echoserver.h"
9
10 int main(int argc, char *argv[]) {
11   if (argc > 2) {
```

```
12      fprintf(stderr, "Usage: %s SERVER_PORT.\n", argv[0]);
13      fprintf(stderr, "Received only %d parameters.\n", argc);
14      fprintf(stderr, "Execution aborted.\n");
15      exit(EXIT_FAILURE);
16    }
17
18    int server_socket, connection_socket, conn_length;
19    SockAddrIn server_address, connection_address;
20    unsigned long int port = strtoul(argv[1], NULL, 0);
21    WSADATA wsadata;
22
23    if (WSAStartup(MAKEWORD(2, 2), &wsadata) != 0) {
24      fprintf(stderr, "Failed to load Winsock library.\n");
25      fprintf(stderr, "Error: %s\n", strerror(errno));
26      exit(EXIT_FAILURE);
27    }
28
29    server_socket = socket(AF_INET, SOCK_STREAM, 0);
30    if (server_socket < 0) {
31      fprintf(stderr, "Failed to create socket.\n");
32      fprintf(stderr, "Error: %s\n", strerror(errno));
33      exit(EXIT_FAILURE);
34    } else {
35      fprintf(stdout, "Socket successfully created.\n");
36    }
37
38    memset(&server_address, 0, sizeof(server_address));
39    server_address.sin_family = AF_INET;
40    server_address.sin_addr.s_addr = htonl(INADDR_ANY);
41    server_address.sin_port = htons(port);
42
43    if (bind(server_socket, (SockAddr *) &server_address, sizeof(server_address)) < 0) {
44      fprintf(stderr, "Failed to bind socket to address.\n");
45      fprintf(stderr, "Error: %s\n", strerror(errno));
46      closesocket(server_socket);
47      WSACleanup();
48      exit(EXIT_FAILURE);
49    } else {
50      fprintf(stdout, "Socket successfully bound.\n");
51    }
52
53    if (listen(server_socket, 2) < 0) {
54      fprintf(stderr, "Server failed to listen.\n");
55      fprintf(stderr, "Error: %s\n", strerror(errno));
56      closesocket(server_socket);
57      WSACleanup();
58      exit(EXIT_FAILURE);
59    } else {
60      fprintf(stdout, "Server listening on port %lu.\n", port);
61    }
62
63    conn_length = sizeof(connection_address);
64    connection_socket = accept(server_socket, (SockAddr *) &connection_address, &conn_length);
65    if (connection_socket < 0) {
66      fprintf(stderr, "Server failed to connect to client.\n");
67      fprintf(stderr, "Error: %s\n", strerror(errno));
68      closesocket(server_socket);
69      WSACleanup();
70      exit(EXIT_FAILURE);
71    } else {
72      fprintf(stdout, "Server successfully connected to client.\n");
73    }
74
75    echo(connection_socket);
76    closesocket(connection_socket);
77    closesocket(server_socket);
78    WSACleanup();
79
80    return 0;
81  }
```

5.4.2 Arquivo de código-fonte `echoserver.h`

```
#ifndef _ECHOSERVER_H_
#define _ECHOSERVER_H_

void echo (int socket_handle);

#endif  /* _ECHOSERVER_H_ */
```

5.4.3 Arquivo de código-fonte `echoserver.c`

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

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

void echo (int socket_handle) {
  char buff[MAX];
  ssize_t comm_len;

  while (TRUE) {
    memset(&buff, 0, sizeof(buff));

    comm_len = recv(socket_handle, (char *) &buff, sizeof(buff), 0);
    if (comm_len < 0) {
      fprintf(stderr, "An error occurred while receiving data.\n");
      fprintf(stderr, "Error: %s\n", strerror(errno));
      break;
    } else if (comm_len == 0) {
      fprintf(stderr, "Client disconnected. Shutting down.\n");
      break;
    }

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

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

    comm_len = send(socket_handle, (char *) &buff, sizeof(buff), 0);
    if (comm_len < 0) {
      fprintf(stderr, "An error occurred while sending data.\n");
      fprintf(stderr, "Error: %s\n", strerror(errno));
      break;
    }
  }
}
```

5.4.4 Arquivo `Makefile`

```
TARGET=tcpserver
TARGET_SOURCES=tcpserver.c echoserver.c

FLAGS=-O2 -Wall -MMD
LIBS=-lws2_32

INCLUDE=-I. -I..\include
BINDIR=..\bin

CMP=gcc
LDFLAGS=$(LIBS) -LC:\mingw64\x86_64-w64-mingw32\lib

```

```
.PHONY=clean distclean

all: install clean

install: $(TARGET)
  if not exist $(BINDIR) md $(BINDIR)
  move $(TARGET).exe $(BINDIR)

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

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

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

clean:
  del *.o *.d *~

distclean: clean
  del $(TARGET).exe
```

5.5 Cliente para Windows

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

5.5.1 Arquivo de código-fonte `tcpclient.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 Library */
  if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
    fprintf(stderr, "Failed to load Winsock library.\n");
    exit(EXIT_FAILURE);
  }

  client_socket = socket(AF_INET, SOCK_STREAM, 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));
```

```
  server_address.sin_family = AF_INET;
  server_address.sin_addr.s_addr = inet_addr(server_ip);
  server_address.sin_port = htons(port);

  if (connect(client_socket, (SockAddr *) &server_address, sizeof(server_address)) < 0) {
    fprintf(stderr, "Failed to connect to server.\n");
    fprintf(stderr, "Error: %s\n", strerror(errno));
    closesocket(client_socket);
    WSACleanup();
    exit(EXIT_FAILURE);
  } else {
    fprintf(stdout, "Connected to server.\n");
  }

  msgclient(client_socket);

  closesocket(client_socket);
  WSACleanup();

  return 0;
}
```

5.5.2 Arquivo de código-fonte `messageclient.h`

```
#ifndef _MESSAGECLIENT_H_
#define _MESSAGECLIENT_H_

void msgclient(int socket_handle);

#endif  /* _MESSAGECLIENT_H_ */
```

5.5.3 Arquivo de código-fonte `messageclient.c`

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

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

void msgclient(int socket_handle) {
  char buff[MAX];
  char letter;
  int n;

  while (TRUE) {
    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++;
    }

    if (send(socket_handle, (char *) &buff, sizeof(buff), 0) < 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) {
```

```
36        fprintf(stdout, "Client exit...\n");
37        break;
38      }
39
40      memset(&buff, 0, sizeof(buff));
41      if (recv(socket_handle, (char *) &buff, sizeof(buff), 0) < 0) {
42        fprintf(stderr, "An error occurred while receiving data.\n");
43        fprintf(stderr, "Error: %s\n", strerror(errno));
44        break;
45      }
46
47      fprintf(stdout, "Received from server: %s\n", buff);
48    }
49  }
```

5.5.4 Arquivo `Makefile`

```
1  TARGET=tcpclient
2  TARGET_SOURCES=tcpclient.c messageclient.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
```

6 Processo de compilação para Windows

6.1 Utilizando a ferramenta `make`

Ao instalarmos o conjunto de ferramentas para compilação de programas em C, conforme o tutorial disponibilizado no Google Classroom da disciplina, conseguimos utilizar a ferramenta `make` para nos auxiliar no processo de compilação.

Para utilizar esta ferramenta, basta abrir o prompt de comando do Windows (`cmd`), navegar até a pasta onde se encontram os arquivos de código-fonte do programa e o `Makefile` e, por fim, executar o comando `mingw32-make.exe`. Este comando executará os passos para compilação de acordo com as instruções contidas no arquivo `Makefile`. Ao final, um arquivo executável deve ser gerado e o programa estará pronto para execução.

6.2 Utilizando uma IDE

Neste roteiro, para a elaboração e compilação dos programas para Windows, utilizaremos como referência o IDE Eclipse. Para a compilação tanto do servidor quanto do cliente, precisamos informar ao IDE onde estão localizados a biblioteca `ws2_32` e o arquivo de cabeçalho `defs.h`. A primeira contém a implementação da interface socket (arquivo `libws2_32.a`), enquanto o segundo possui as definições que mencionamos anteriormente. Este passo é necessário porque ambos arquivos encontram-se fora do escopo do contexto de cada um dos projetos (cliente e servidor).

Primeiramente, vamos informar ao IDE onde encontrar o arquivo `defs.h`. Para isso, siga os seguintes passos:

1. Clique no menu `Project > Properties`;
2. Na janela que se abrir, selecione `C/C++ Build > Settings`;
3. No menu que se abrir, clique na aba `Tool Settings`;
4. Por fim, clique em `Includes`;
5. No painel "`Include paths (-I)`", clique no botão para adicionar um novo local;
6. Digite o caminho completo da pasta onde se encontra o arquivo `defs.h`;
7. A Figura 4 ilustra o resultado final;
8. Com a configuração concluída, clique em `Apply and Close`.

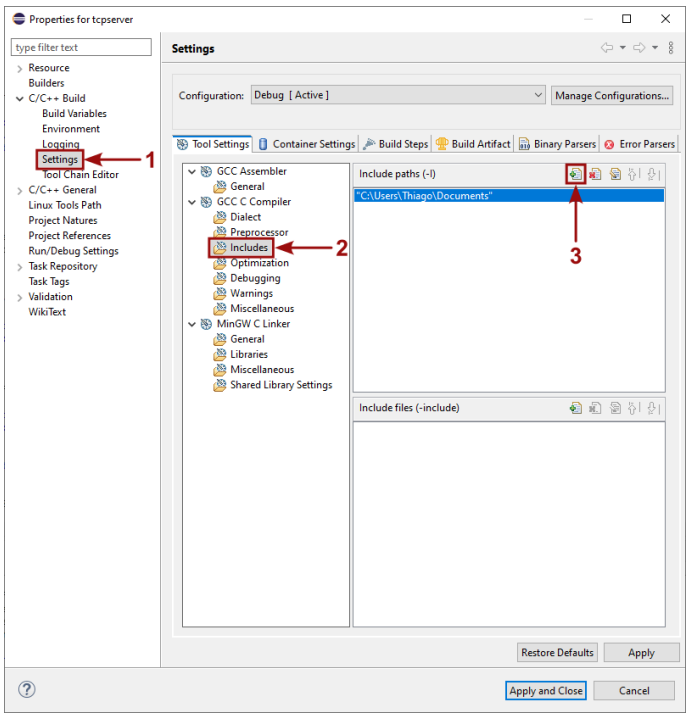


Figura 4: Configuração para inclusão dos arquivos de cabeçalho no Eclipse.

Por fim, vamos informar onde se encontra a biblioteca `ws2_32`. Para isso, siga os passos abaixo:

1. Clique no menu `Project > Properties`;

2. Na janela que se abrir, selecione `C/C++ Build > Settings`;
3. No menu que se abrir, clique na aba `Tool Settings`;
4. Por fim, clique em `Libraries` (Figura 5);
5. No painel "`Library search path (-L)`", clique no botão para adicionar um novo local;
6. Informe o caminho completo do diretório onde se encontra a biblioteca `ws2_32` (no caso deste tutorial, o caminho completo é `C:\mingw-w64\x86_64-w64-mingw32\lib`);
7. No painel "`Libraries (-l)`", clique no botão para adicionar uma nova biblioteca;
8. Na janela que se abrir, digite o nome da biblioteca (neste caso, o nome da biblioteca é `ws2_32`);
7. Com a configuração concluída, clique em `Apply and Close`.

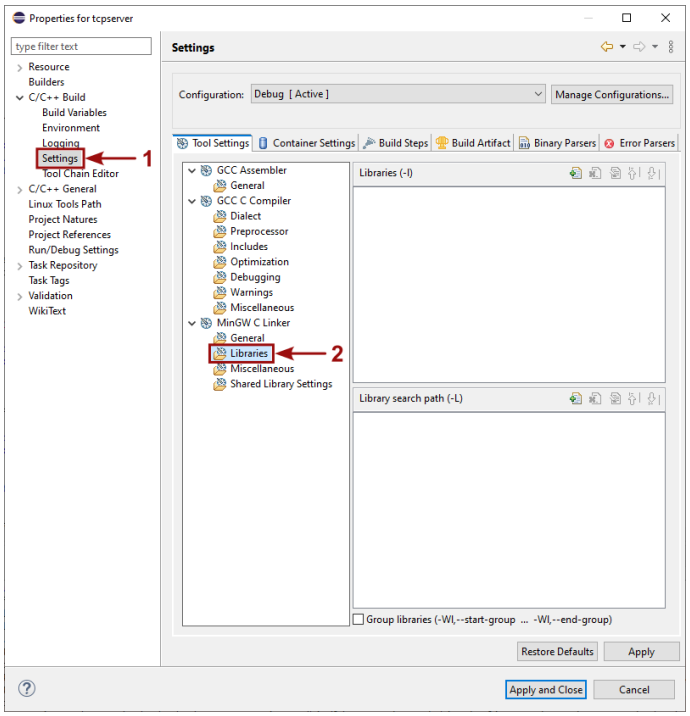


Figura 5: Tela de configuração para inclusão de bibliotecas externas.

Após concluir estas configurações, basta ir até o menu `Project > Build` para compilar o programa.

Referências

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

[2] L. L. Peterson and B. S. Davie, *Computer networks: a systems approach*. Elsevier, 2007.

[3] A. T. Campbell, "Socket programming." https://www.cs.dartmouth.edu/~campbell/cs60/socketprogramming.html. Último acesso em: 22/05/2024.

[4] Oracle, "What is a socket?." https://docs.oracle.com/javase/tutorial/networking/sockets/definition.html. Último acesso em: 22/05/2024.

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

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

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

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

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

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

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