Ficha 7 – Sockets TCP

Tópicos abordados:

– *Sockets* TCP
– Servidores TCP iterativos
– Servidores TCP concorrentes
– Exemplo de um servidor e cliente TCP
– Exercícios

Duração prevista: 3 aulas

1. *Sockets* TCP

Por ser um protocolo orientado à ligação, o TCP utiliza um esquema de funcionamento diferente do UDP, conforme ilustrado na Figura 1:

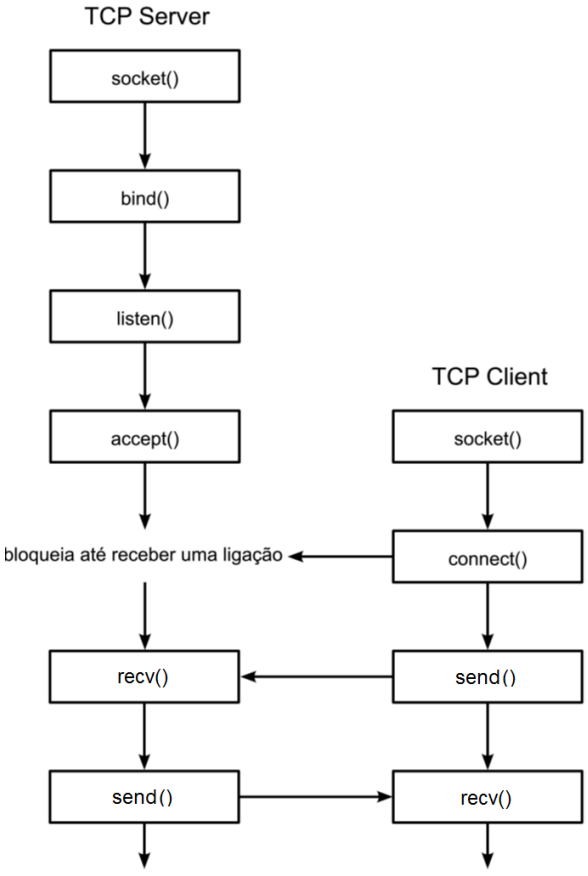

Figura 1 - Esquema típico de uma ligação TCP/IP

De salientar que, apesar de na Figura 1 o servidor esperar pela receção de dados após a aceitação de uma ligação do cliente, esta sequência não é obrigatória. Assim, e em alternativa, o servidor pode enviar dados ao cliente logo após a aceitação de um cliente.

Os protótipos das funções empregues no diagrama da Figura 1 estão na Tabela 1:

```
int socket(int family, int type, int protocol);
int bind(int sockfd, const struct sockaddr *my_addr, socklen_t addrlen);
int listen (int sockfd, int backlog);
int accept(int sockfd, struct sockaddr* cli_addr, socklen_t * addrlen);
int connect(int sockfd, const struct sockaddr *svc_addr, socklen_t addrlen);

ssize_t recv(int sockfd, void *buf, size_t len, int flags);
ssize_t send(int sockfd, const void *buf, size_t len, int flags);

int close(int sockfd);
```

Tabela 1 – Funções de criação e manipulação de sockets

1.1. Função socket ()

```
int socket(int family, int type, int protocol);
```

Como já foi referido na Ficha 6 (*Sockets* UDP), a função `socket()` cria e devolve um descritor para um *socket* da família *family* (por exemplo `AF_INET ou AF_INET6`) e do tipo *type* (`SOCK_STREAM`, `SOCK_DGRAM`, ou `SOCK_RAW`). Para se usar o protocolo definido por omissão (TCP para `SOCK_STREAM`, UDP para `SOCK_DGRM`, ...), o valor de *protocol* deve ser zero.

Valores de retorno

Sucesso – identificador do *socket*.

Insucesso – devolve -1 e a variável *errno* é preenchida com o código de erro correspondente.

1.2. Servidor TCP

Após a criação do *socket*, para as aplicações servidoras TCP são empregues as funções `bind()`, `listen()` e `accept()`. Estas funções, à semelhança de outras da API *socket*, devolvem -1 em caso de erro, sendo atribuindo à variável `errno` um código numérico descritivo do erro ocorrido. Através do uso da função `strerror` (`strerror(errno)`) obtém-se uma *string* descritiva do erro. Em alternativa poderá utilizar a macro ERROR disponibilizada nos ficheiros “debug.c” e “debug.h”.

1.2.1. Função bind ()

```
int bind(int sockfd, const struct sockaddr *my_addr,
         socklen_t addrlen);
```

Tal como sucede nas aplicações servidoras UDP, a função `bind()` efetua o mapeamento do *socket sockfd* para um determinado porto e interface local. Note-se que esta função também pode ser chamada pelas aplicações clientes se existir a necessidade de estas usarem um porto (e/ou interface) específico (dado tratar-se de uma situação pouco frequente, esta situação não está representada na Figura 1).

Valores de retorno

Sucesso – devolve 0 .

Insucesso – devolve -1 e a variável *errno* é preenchida com o código de erro correspondente.

1.2.2. Função listen()

```
int listen (int sockfd, int backlog);
```

A função `listen()` serve para indicar que o *socket* servidor já está disponível para receber pedidos de ligações no *socket sockfd* através da função `accept()`. Para além disso, o parâmetro *backlog* permite definir o tamanho da fila de espera de ligações aceites para o *socket sockfd*. Apesar de este parâmetro permitir definir esse limite, a sua interpretação poderá ser diferente entre diferentes plataformas e sistemas operativos[1] [2].

Valores de retorno

Sucesso – devolve 0.

Insucesso – devolve -1 e a variável *errno* é preenchida com o código de erro correspondente.

[1] http://veithen.github.io/2014/01/01/how-tcp-backlog-works-in-linux.html

[2] http://stackoverflow.com/questions/5111040/listen-ignores-the-backlog-argument

1.2.3. Função accept ()

```
int accept(int sockfd, struct sockaddr* cli_addr,
           socklen_t * addrlen);
```

A função `accept()`, tal como o nome sugere, aceita as ligações dos clientes que se encontram na lista de espera. Esta função cria e devolve um descritor para um **novo** *socket* conectado que passará a ser utilizado para comunicar com o cliente que iniciou a ligação. Para além disso, a função preenche a estrutura indicada com o endereço e o porto do cliente (*cli_addr*), caso os valores passados não sejam NULL. Este descritor será utilizado para comunicar com o cliente que iniciou a ligação, sendo um canal de comunicação um para um.

Valores de retorno

Sucesso – número >= 0, que é um identificador para o *socket* do cliente.

Insucesso – devolve -1 e a variável *errno* é preenchida com o código de erro correspondente.

Lab 1

Crie um servidor TCP (*server.c*) que fique à escuta apenas de um cliente no porto 1234. Assim que um cliente se ligar ao servidor, este deve mostrar o endereço IP e porto do cliente e, de seguida, fechar o *socket* e terminar.

Nota: use o programa da seguinte forma: *./server --porto 1234*

1.3. Cliente TCP

As aplicações cliente TCP, depois de criarem um descritor com a função `socket()`, chamam a função `connect()` para solicitar uma ligação ao servidor.

1.3.1. Função connect ()

```
int connect(int sockfd, const struct sockaddr
            *svc_addr, socklen_t addrlen);
```

Esta função liga um cliente ao servidor indicado através do endereço e porto definidos na variável *svc_addr*.

Valores de retorno

Sucesso – devolve 0.

Insucesso – devolve -1 e a variável *errno* é preenchida com o código de erro correspondente.

Lab 2

Crie um cliente TCP (*client.c*) que faça a ligação ao servidor do *Lab 1* e termine logo de seguida. Execute, em consolas distintas, o servidor (*Lab 1*) e o cliente (*Lab 2*) para efetuar testes.

Nota: use o programa cliente da seguinte forma: *./client --ip 127.0.0.1 --porto 1234*

1.4. Comunicação entre cliente/servidor TCP

Para enviar e receber dados, as aplicações (servidoras e clientes) podem utilizar as funções `recv()` e `send()`.

1.4.1. Função recv ()

```
ssize_t recv(int sockfd, void *buf, size_t len, int
             flags);
```

A função `recv()` permite a uma aplicação, servidora ou cliente, receber dados a partir de um *socket*. Tal como a função `recvfrom()`, esta função, por omissão, bloqueia a aplicação até receber dados.

Como primeiro parâmetro a função recebe o descritor do *socket* do qual se quer receber dados. O segundo e o terceiro parâmetro *buf* e *len* especificam, respetivamente, o endereço e tamanho da zona de memória (*buffer*) no qual os dados irão ser escritos.

Finalmente, o parâmetro *flags* permite aceder a modos alternativos da função. Por exemplo, a opção MSG_DONTWAIT pode ser usada na função `recv()` para que esta deixe de ser bloqueante.

Valores de retorno

Sucesso – devolve o número de bytes recebidos.

Insucesso – devolve -1 e a variável *errno* é preenchida com o código de erro correspondente.

1.4.2. Função send ()

```
ssize_t send(int sockfd, const void *buf, size_t len,
             int flags);
```

A função `send()` permite a uma aplicação enviar dados a partir de um *socket*. O socket tem de estar no estado dito "conectado".

Como primeiro parâmetro, a função recebe o descritor do *socket* do qual se quer enviar dados. O parâmetro *buf* especifica o endereço da zona de memória (*buffer*) onde se

encontram os dados que se pretende enviar. Por sua vez, o parâmetro *len* define o número de bytes a ser enviados. O parâmetro *flags* permite aceder a modos alternativos da função.

Valores de retorno

Sucesso – devolve o número de bytes enviados.

Insucesso – devolve -1 e a variável *errno* é preenchida com o código de erro correspondente.

Nota: caso se efetue uma operação de *send* num *socket* que, entretanto, tenha sido fechado pela outra entidade comunicante, o sistema operativo envia o *signal* SIGPIPE ("Broken Pipe") para assinalar o fecho da ligação. O uso da flag MSG_NOSIGNAL desativa esse comportamento, sendo que nesse caso, a notificação de ligação remota fechada é feita através do retorno do valor -1 pela função `send()` e da atribuição do valor EPIPE à variável `errno`.

1.4.3. Função close ()

```
int close(int sockfd);
```

Quando empregue sobre um descritor de *socket* TCP, a função `close()` encerra a ligação.

Valores de retorno

Sucesso – devolve 0.

Insucesso – devolve -1 e a variável *errno* é preenchida com o código de erro correspondente.

Lab 3

Utilize os programas do *Lab 1* (*server.c*) *e Lab 2* (*cliente.c*) de modo a que o servidor fique à espera de receber uma *string* (número entre 1 e 9999) e responda com esse valor convertido para inteiro. Se não for possível converter a *string*, devolve-se o valor 0.

Nota: não se esqueça de converter os dados para o formato de rede.

1.4.4. Sincronização do envio/receção de dados

Ao contrário do UDP, o TCP utiliza uma fila para armazenar os dados recebidos. Isto faz com que, seja necessário ter em atenção, a forma como são enviados e recebidos os dados, de modo a ler apenas o que se pretende. Para lidar com este aspeto são implementados protocolos sobre a camada de dados do TCP (ex.: HTTP, FTP, etc.). Com estes protocolos é possível saber onde começa e termina determinado "pacote de dados" que uma

aplicação envia à outra. Assim, é possível enviar pacotes consecutivos sem ter que receber a confirmação da receção dos mesmos. ***Lab 4***

Modifique os programas do *Lab 3* (*server.c*, *cliente.c* e *common.h*) de modo a implementar o protocolo que é mostrado na Figura 2.

a) O *output* do programa foi o esperado? Porquê?

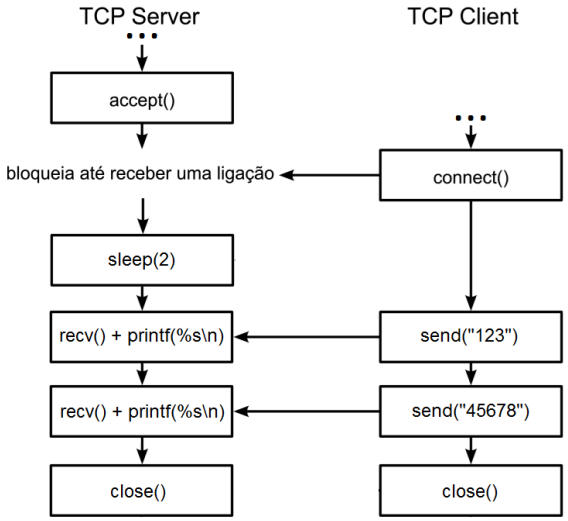

Figura 2 - Exemplo de cliente e servidor sem sincronização recv/send

Assim, para evitar este problema, nesta UC, recorre-se à sincronização da comunicação, utilizando a seguinte técnica: para cada envio tem que haver sempre uma receção de dados. A Figura 3 mostra um exemplo para resolver o problema anterior:

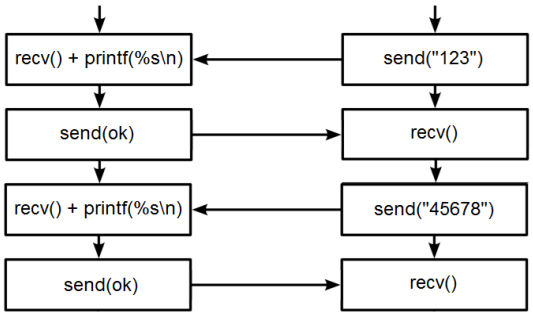

Figura 3 - Sincronização do envio/receção de dados sem protocolos auxiliares

Lab 5

Modifique os programas do *Lab 4* (*server.c*, *cliente.c* e *common.h*) de modo a implementar a técnica de sincronização da comunicação.

2. Servidor TCP iterativo

Num servidor TCP iterativo (Figura 4), o servidor é executado num único processo que atende os pedidos de cada cliente de forma sequencial. Os outros clientes que queiram comunicar têm que esperar que o servidor termine de responder ao cliente atual.

Este tipo de servidor só se deve utilizar para operações de curta duração, para que os clientes não tenham que esperar em demasia pela sua vez.

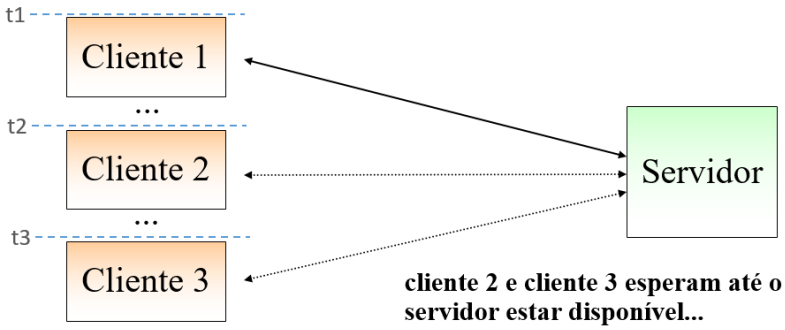

Figura 4 - Servidor iterativo: gere os vários clientes de forma sequencial

2.1. Servidor

A tarefa principal do processo servidor está representada na Figura 5:

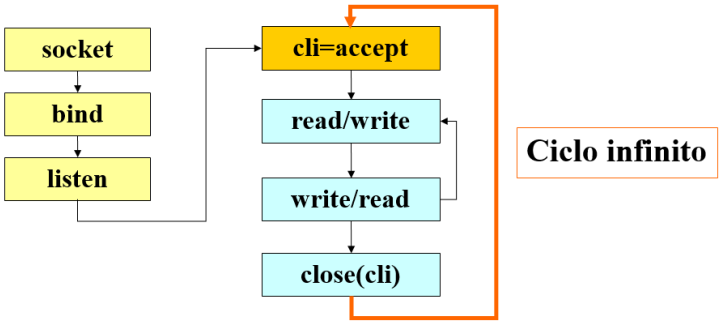

Figura 5 - Modo de funcionamento do servidor iterativo

Depois de criar o *socket*, fazer *bind* e configurar a lista de espera de clientes (*listen*), o servidor bloqueia (*accept*) à espera que um cliente se ligue (*connect*). De seguida, "trata" de responder a esse cliente e, para finalizar a ligação, fecha o *socket* com o mesmo. Depois, inicia uma nova iteração, bloqueando à espera de um novo cliente.

Lab 6

Crie um servidor TCP iterativo (fila de espera == 1) com o propósito de implementar um serviço que disponibiliza a data e hora atual. O formato da resposta depende de, no pedido, ser recebido o inteiro: 0 – data e hora | 1 – apenas data | 2 – apenas hora. O inteiro é representado pelo tipo de dados **uint8_t**. Por sua vez, a string contendo a

data/hora deve ter o seguinte formato: **YYYYMMDD_HHhmm:ss**, em que Y representa o ano (quatro dígitos), MM o mês (dois dígitos), DD o dia (dois dígitos), HH a hora (formato 00 a 23), mm os minutos (dois dígitos) e ss os segundos (dois dígitos). A string a ser enviada pelo servidor tem um máximo de 64 caracteres e não é terminada com o '\0'.

Nota: use o programa da seguinte forma: *./server --porto 1234*

2.2. Cliente

O cliente tem que ser uma aplicação "normal" cliente TCP. Para isso terá apenas que conhecer o IP e porto do servidor.

Lab 7

Elabore o cliente TCP que se ligue ao servidor iterativo do *Lab 6* e que use o serviço para obter a data e hora atuais.

Nota: use o programa da seguinte forma: *./client --ip 127.0.0.1 --porto 1234*

Lab 8

Utilize os programas do *Lab 6* (*server.c / common.h*) e *Lab 7* (*client.c*) de modo a que:

a) *Server*: demore 3 segundos (*sleep(3)*) a responder aos clientes;

b) *Client*: crie 5 clientes (*fork()*) para testar um acesso simultâneo ao serviço;

c) Qual o resultado obtido? Explique.

3. Servidor TCP concorrente

Num servidor TCP concorrente (Figura 6), o servidor é executado num processo e atende os pedidos criando um processo (ou *thread*) para cada cliente. Entretanto, o processo pai (ou *main thread*) regressa ao `accept()` para tratamento dos próximos pedidos de ligação. A limitação máxima de clientes a atender em simultâneo é definida através do tamanho máximo da fila de espera indicado como segundo parâmetro da função `listen()`.

Este tipo de servidor deve ser utilizado para operações de média/longa duração.

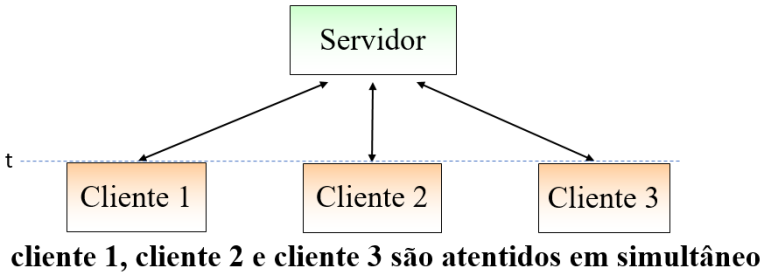

Figura 6 - Servidor TCP concorrente

3.1. Servidor

A tarefa principal do processo/*thread* servidor (processo pai ou *main thread*) está representada na Figura 7:

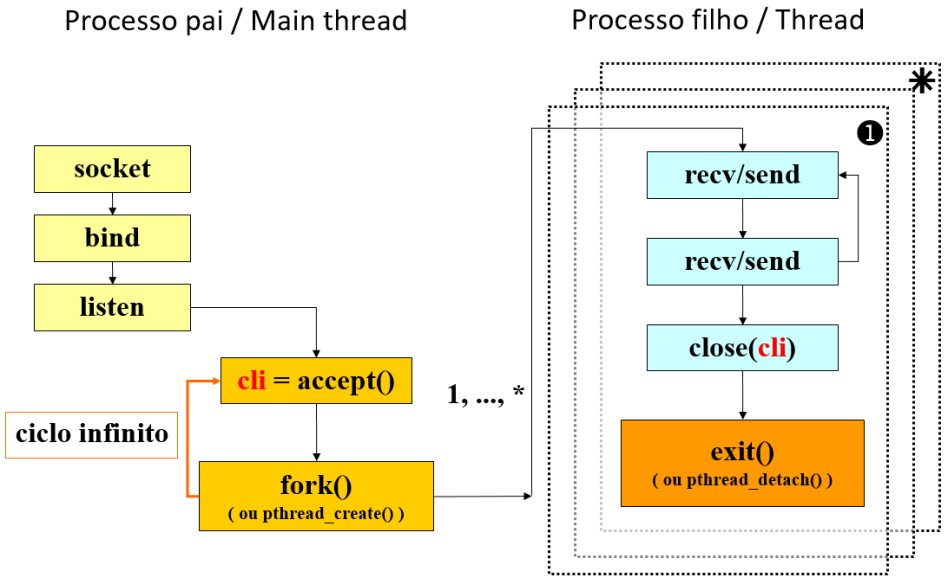

Figura 7 - Servidor TCP concorrente e dois clientes

Depois de criar o *socket*, fazer *bind* e configurar a lista de espera de clientes (*listen*), o servidor bloqueia (*accept*) à espera que um cliente se ligue (*connect*). Ao receber a ligação de um cliente, cria um processo (ou *thread*), que tem como objetivo tratar de responder ao cliente. O processo pai (*main thread*), que está em ciclo infinito, depois de libertar todos os recursos que não vai utilizar, volta a bloquear à espera de novos clientes (*accept*).

Quanto ao processo que foi criado, faz a comunicação com o cliente e, quando terminar, fecha o *socket* criado para comunicar com o cliente e termina a sua execução.

Lab 9

Crie um servidor TCP concorrente (fila de espera == 8) com o propósito de implementar um serviço que disponibiliza a data e hora atual. O formato da resposta depende de, no pedido, ser recebido o inteiro: 0 – data e hora | 1 – apenas data | 2 – apenas hora. O servidor deve demorar 5 segundos (*sleep(5)*) a responder a cada cliente.

Nota: use o programa da seguinte forma: *./server --porto 1234*

3.2. Cliente

O cliente tem que ser uma aplicação "normal" cliente TCP. Para isso, terá apenas que conhecer o IP e o porto do servidor.

Lab 10

Elabore o cliente TCP, que deve criar 5 processos (*fork()*) para testar um acesso simultâneo ao servidor concorrente do *Lab 9* e que use o serviço para obter apenas a data atual.

Nota: use o programa da seguinte forma: *./client --ip 127.0.0.1 --porto 1234*

4. Exemplo

Segue-se um exemplo que ilustra a implementação do jogo "Adivinha Número" usando *sockets* TCP. O servidor recebe números de um cliente e compara-os com um número aleatório que escolheu (entre 1 e 100). Se o número do cliente for menor que o número escolhido, o servidor envia a constante MENOR, se for maior envia a constante MAIOR e se for igual envia a constante IGUAL. O cliente pede valores ao utilizador e envia-os ao servidor, depois lê a resposta e indica ao utilizador se deve escolher um número maior, menor, ou se acertou.

Nota: foram omitidos do código apresentado os #defines dos erros ("comum.h").

Ficheiro comum.h

Ficheiro a ser incluído no código do servidor e no código do cliente.

```
#define IGUAL   0
#define MENOR   1
#define MAIOR   2
```

Servidor

```
#include <stdio.h>
#include <sys/types.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <stdlib.h>
#include "debug.h"
#include "comum.h"
#include "servidor_opt.h"

void processaCliente(int fd);

int main(int argc, char *argv[])
{
    int ser_fd, cli_fd;
    socklen_t cli_len;
    struct sockaddr_in ser_addr, cli_addr;

    /* Processa os parâmetros da linha de comando */
    struct gengetopt_args_info args_info;
    if (cmdline_parser(argc, argv, &args_info) != 0)
        ERROR(C_ERRO_CMDLINE, "cmdline_parser");

    /* cria um socket */
    if ((ser_fd = socket(AF_INET, SOCK_STREAM, 0)) == 1)
        ERROR(C_ERRO_SOCKET, "socket");

    /* preenche estrutura: ip/porto do servidor */
    memset(&ser_addr, 0, sizeof(ser_addr));
    ser_addr.sin_family = AF_INET;
    ser_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    ser_addr.sin_port = htons(args_info.porto_arg);
```

```
    /* disponibiliza o porto para escuta */
    if (bind(ser_fd, (struct sockaddr *) &ser_addr, sizeof(ser_addr)) == -1)
        ERROR(C_ERRO_BIND, "bind");
    if (listen(ser_fd, 5) == 1)
        ERROR(C_ERRO_LISTEN, "Listen");
    printf("Servidor %s no porto %d\n", argv[0], args_info.porto_arg);

    /* ciclo infinito para atender todos os clientes */
    while (1) {
       cli_len = sizeof(struct sockaddr);
        /* accept - bloqueante */
        cli_fd = accept(ser_fd, (struct sockaddr *) &cli_addr, &cli_len);
        if (cli_fd < 0)
            ERROR(C_ERRO_ACCEPT, "Accept");
        /* mostra informação sobre o cliente e processa pedido */
        char ip[20];
        DEBUG("cliente[%s@%d]",inet_ntop(AF_INET,&cli_addr.sin_addr,
                 ip, sizeof(ip)), ntohs(cli_addr.sin_port));
        processaCliente(cli_fd);

        /* liberta recursos utilizados com este cliente*/
        close(cli_fd);
    }

    return (0);
}

void processaCliente(int fd)
{
    uint16_t  n_cli, n_serv, res;

    srand(time(NULL));
    n_serv = 1 + (uint16_t) (100.0 * rand() / (RAND_MAX + 1.0));
    DEBUG("número escolhido: %d\n", n_serv);

    do {
        /* recebe dados do cliente - chamada bloqueante */
        if (recv(fd, &n_cli, sizeof(uint16_t), 0) == -1)
            ERROR(C_ERRO_RECV, "recv");

        n_cli = ntohs(n_cli);

        if (n_cli == n_serv)
            res = IGUAL;
        else if (n_cli < n_serv)
            res = MENOR;
        else
            res = MAIOR;

        res = htons(res);

        /* envia resposta ao cliente */
        if (send(fd, &res, sizeof(uint16_t), 0) == -1)
            ERROR(C_ERRO_SEND, "send");

    } while (n_cli != n_serv);
}
```

Cliente

```
#include <stdio.h>
#include <sys/types.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <unistd.h>
#include <string.h>
#include "debug.h"
#include "comum.h"
#include "cliente_opt.h"

void adivinhaNum(int fd);

int main(int argc, char *argv[])
{
    int sock_fd;
    struct sockaddr_in ser_addr;

    /* Processa os parâmetros da linha de comando */
    struct gengetopt_args_info args_info;
    if (cmdline_parser(argc, argv, &args_info) != 0)
        ERROR(C_ERRO_CMDLINE, "cmdline_parser");

    /* preenche estrutura: ip/porto do servidor */
    memset(&ser_addr, 0, sizeof(ser_addr));
    ser_addr.sin_family = AF_INET;
    ser_addr.sin_port = htons(args_info.porto_arg);
    if (inet_pton(AF_INET, args_info.ip_arg, &ser_addr.sin_addr) <= 0)
        ERROR(C_ERRO_INET_PTON, "inet_pton");

    /* cria um socket */
    if ((sock_fd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
        ERROR(C_ERRO_SOCKET, "socket");

    /* faz ligação ao servidor */
    if(connect(sock_fd, (struct sockaddr *)&ser_addr, sizeof(ser_addr))==-1)
        ERROR(C_ERRO_CONNECT, "Nao conseguiu ligaçao");

   /* jogo */
    adivinhaNum(sock_fd);

   /* liberta recursos utilizados */
   close(sock_fd);

   return (0);
}


void adivinhaNum(int fd)
{
    uint16_t num, res;

    do {
        printf("\nIntroduza um número entre 1 e 100:  ");
        scanf("%hu", &num);  // hu - unsigned short (half) int

        num=htons(num);

     /* envia número ao servidor */
        if (send(fd, &num, sizeof(uint16_t), 0) < 0)
            ERROR(C_ERRO_SEND, "write");
```

```
     /* recebe a resposta do servidor - chamada bloqueante */
        if (recv(fd, &res, sizeof(uint16_t), 0) < 0)
            ERROR(C_ERRO_RECV, "read");

        res=ntohs(res);
        if (res == IGUAL)
            printf("Parabens! Acertou\n\n");
        else if (res == MENOR)
            printf("O número do servidor é MAIOR\n");
        else
            printf("O número do servidor é MENOR\n");

    } while (res != IGUAL);
}
```

Lab 11

Compile o código, do servidor e do cliente, mostrado no exemplo anterior e:

a) Execute o servidor da seguinte forma: ./servidor --porto 1234

b) Execute o cliente da seguinte forma: ./cliente --ip 127.0.0.1 --porto 1234

c) Termine abruptamente o cliente, por exemplo, recorrendo ao CTRL+C, ou fechando o terminal onde corre o cliente. O que é que sucede ao servidor?

d) Modifique o código por forma a que o término abrupto do cliente não leve ao término do servidor.

Solução: i) detetar o “recv==0”; ii)

Lab 12

Compile o código, do servidor e do cliente, mostrado no exemplo anterior e:

a) Execute o servidor da seguinte forma: ./servidor --porto 1234

b) Execute o cliente da seguinte forma: ./cliente --ip 127.0.0.1 --porto 1234

Nota: Jogue o jogo até acertar no número.

c) Com o servidor ainda em execução (está em ciclo infinito):

Execute 2 vezes o cliente e explique os resultados obtidos.

d) Modifique o código do servidor para que obtenha o resultado esperado.

5. Exercícios

Nota: na resolução de cada exercício deverá utilizar o *template* de exercícios cliente/servidor, que inclui uma *makefile* bem como todas os ficheiros para as aplicações cliente e servidor, bem como as dependências necessárias à compilação.

5.1. Para a aula

1. Elabore, em C com *sockets* TCP, o programa servidor **iterativo** "**servidorEco**", que recebe como parâmetro de entrada o porto onde vai ficar à escuta. O servidor deve receber uma mensagem enviada por um cliente, mostrá-la no *stdout* e enviá-la novamente ao cliente, até que receba a mensagem "fim". Implemente também o cliente "clienteEco", que recebe como parâmetros de entrada o endereço IP e o porto do servidor. O cliente deve pedir ao utilizador as mensagens a enviar e terminar quando a mensagem for igual a "fim".
2. Implemente agora a versão **concorrente** do "**servidorEco2**", recorrendo à utilização de *threads*.

5.2. Extra-aula

3. Elabore, em linguagem C e recorrendo à API *socket*, a aplicação cliente **tinyHTTP**. Esta deve permitir efetuar pedidos GET do protocolo HTTP/1.1 à página inicial de um servidor Web HTTP (ex: www.example.com), indicado através do seu endereço na linha de comandos.

   Utilize o comando GET, e envie o pedido a ser enviado é, respetivamente:

   ```
   GET / HTTP/1.1
   Host: <nome_do_domínio>
   ```

   Para garantir que o servidor fecha o *socket* no fim da resposta, utilize também:

   ```
   Connection: close
   ```

   Nota 1: na mudança de linha utilizam-se os carateres: '\r' e '\n'.

   Nota 2: todos os pormenores omissos sobre o protocolo HTTP devem ser alvo de pesquisa.

4. Elabore a aplicação servidora "**lsRemoto**" que deverá simular o comando ls remoto. O cliente especifica qual o diretório que pretende listar e o servidor

responde com a informação resultante da execução do comando `ls` nesse diretório, ou uma mensagem apropriada na ocorrência de um erro. A comunicação deve ser feita através de *sockets streams*. Utilize um cliente *telnet* (*putty*) para testar a aplicação.

5. Implemente, em C, o servidor "**daFicheiro**" que recebe como parâmetros de entrada o porto onde vai ficar à escuta e o caminho para um ficheiro. Sempre que um cliente se ligar, o servidor envia-lhe o conteúdo do ficheiro e termina a ligação.

Implemente o cliente "recebeFicheiro" que tem como parâmetros de entrada o endereço IP do servidor, o porto onde está à escuta e nome do ficheiro a criar com o conteúdo enviado pelo servidor.

6. Pretende-se que implemente um par de aplicações cliente/servidor rudimentar que permita realizar reservas para um evento com uma determinada capacidade máxima de lugares. A comunicação entre a aplicação servidor e a aplicação cliente faz-se através do protocolo de transporte TCP, sendo empregue um protocolo aplicacional baseado em mensagens de dois bytes, definidas do seguinte modo:

```
#define OK                  0x0001
#define NOT_ENOUGH_PLACES   0x0002
#define FULL                0x0003
```

O servidor recebe como argumentos da linha de comandos o porto (`-p/--porto <int>`) a partir do qual vai receber mensagens TCP e o número de lugares do evento (`-l/--lugares <int>`). O número de lugares deve estar compreendido entre 1 e 9999. Os valores passados através dos argumentos --porto e `--lugares` devem ser convenientemente validados.

Quando recebe, por parte de um cliente, um pedido de reserva de um determinado número de lugares para o evento, o servidor decrementa o número de lugares disponíveis e envia a mensagem OK para o cliente. Caso ainda existam lugares disponíveis, mas não em número suficiente para satisfazer o pedido do cliente, o servidor responde com a mensagem NOT_ENOUGH_SPACE. Caso já não existam lugares disponíveis, o servidor responde com a mensagem FULL.

Por sua vez, o cliente recebe como parâmetros da linha de comandos o IP do servidor (`-i/--ip`), o porto do servidor (`-p/--porto`) e o número de thr*e*ads (`-t/--threads`) a utilizar para fazer reservas. Os valores passados através de parâmetros da linha de comando devem ser convenientemente validados.

Cada *thread* deve fazer pedidos de reserva até que não existam mais lugares disponíveis. O número de lugares a reservar em cada pedido é um valor aleatório entre 1 e 5. Para efeitos de teste, as *threads* devem contabilizar o número total de reservas feitas e o programa cliente deve mostrar esse valor (que deve corresponder sempre ao número de lugares do evento) depois de todas as *threads* terem terminado. Os pedidos de reserva devem ser realizados à vez pelas *threads* de modo a que o acesso à informação partilhada entre as *threads* (número total de reservas já realizadas por todas as *threads* e indicação sobre se ainda existem lugares disponíveis para o evento) seja sincronizado. Quando, após realizar um pedido de reserva, uma *thread* recebe do servidor:

- a mensagem OK, deve incrementar o número total de reservas já realizadas pelas *threads* e passar a vez;
- a mensagem NOT_ENOUGH_PLACES, deve continuar a efetuar pedidos de reserva;
- a mensagem FULL deve registar que já não existem lugares disponíveis para que as outras *threads* possam terminar sem terem que fazer mais pedidos, e terminar.

Considere o exemplo descrito de seguida:

./servidor --porto 1234 --lugares 10	**./cliente --ip 127.0.0.1 -p 1234 --threads 2**
Pronto a receber pedidos de reserva / porto 1234	[a thread 1 envia um pedido ao servidor para reserva de 4 lugares]
[recebe um pedido de reserva de 4 lugares e responde **OK**]	[a thread 1 recebe do servidor a mensagem **OK**]
[recebe um pedido de reserva de 5 lugares e responde **OK**]	[a thread 2 envia um pedido ao servidor para reserva de 5 lugares]
	[a thread 2 recebe do servidor a mensagem OK]
[recebe um pedido de reserva de 2 lugares e responde **NOT_ENOUGH_PLACES**]	[a thread 1 envia um pedido ao servidor para reserva de 2 lugares]
	[a thread 1 recebe do servidor a mensagem **NOT_ENOUGH_PLACES**]

[recebe um pedido de reserva de 1 lugar e responde OK] [recebe um pedido de reserva de 3 lugares e responde FULL]	[a thread 1 envia um pedido ao servidor para reserva de 1 lugar] [a thread 1 recebe do servidor a mensagem OK] [a thread 2 envia um pedido ao servidor para reserva de 3 lugares] [a thread 2 recebe do servidor a mensagem FULL] **Foram reservados 10 lugares**

7. Elabore, recorrendo à linguagem C, a aplicação cliente/servidor `inverte_palavra`. A aplicação é composta por um programa i) servidor e por um programa ii) cliente, sendo que as aplicações devem comunicar através do protocolo TCP. Como o nome sugere, o objetivo da aplicação é o de devolver uma *string* invertida. Assim, o cliente recebe uma *string* da linha de comando e tenta, de seguida, enviá-la ao servidor. O servidor, por sua vez, devolve a versão invertida da *string* (por exemplo: "abcd" leva à devolução de "dcba"). Finalmente, o cliente mostra o resultado obtido do servidor, caso toda a operação tenha sido bem-sucedida. Considere como tamanho máximo para uma palavra o valor 256.

   A aplicação **cliente** recebe os seguintes parâmetros da linha de comando:

   `--ip / -i <IP>`: endereço IPv4 do servidor no formato dotted decimal. Se não for especificada a opção, deve assumir-se o valor 127.0.0.1;

   `--port / -p <int>`: porto TCP remoto do servidor. Se não for especificada a opção, deve assumir-se o valor 9999;

   `--word / -w <string>`: palavra da qual se pretende obter a versão invertida.

   A aplicação **servidora** recebe os seguintes parâmetros da linha de comando:

   `--port / -p <int>`: porto TCP de escuta do servidor. Se não for especificada a opção, deve assumir-se o valor 9999;

   `--failure /-f <float>`: taxa de falha na resposta a pedidos TCP do cliente. Por omissão, o valor é `0` (isto é, não há falhas propositadas pela aplicação servidora).

   A opção **`--failure/-f`** <**float**> da aplicação servidora permite simular falhas ao nível da resposta do servidor para o cliente. O valor <float> corresponde à

taxa de falhas. Por exemplo, se for especificado o valor `0.25`, isso significa que o servidor não responde a `25%` dos pedidos do cliente, ou seja, `1` em cada `4` pedido do cliente são descartados pelo servidor. Note-se que a aplicação cliente deve estar preparada para lidar com as falhas ao nível da resposta da aplicação servidora, esperando no máximo dois segundos por uma resposta da aplicação servidora. Alguns exemplos de execução são mostrados de seguida:

Exemplo #1

SERVIDOR	CLIENTE
./servidor –p 1234 [servidor] PID: 8812 [servidor]: aguarda pedido TCP/1234	
	./cliente --remoto 127.0.0.1 --port 1234 –w abcde [cliente]: a contactar servidor TCP@127.0.0.1:1234
[servidor]: pedido recebido (abcde). [servidor]: inversão: ‘abcde’ para ‘edcba’ [servidor]: a enviar ‘edcba para cliente	
	[cliente] inversao recebida: ‘abcde’ ➔ ‘edcba’
[servidor] Finito.	
[servidor] aguarda pedido TCP/1234	

Exemplo #2

SERVIDOR	CLIENTE
./servidor –p 2000 –f 0.5 [servidor] PID: 7902 [servidor]: aguarda pedido TCP/2000	
	./cliente --remoto 127.0.0.1 --port 2000 --word hello [cliente]: a contactar servidor TCP@127.0.0.1:2000
[servidor]: pedido recebido (hello). [servidor]: a simular falha (taxa=0.5) [servidor]: a descartar pedido do cliente	
	[cliente] sem resposta do servidor (esgotado: 2 segundos) [cliente] ERRO na comunicação – Operação terminada
[servidor] aguarda pedido TCP/2000	

Exemplo #3

```
./servidor –p 712020
ERRO: porto inválido
```

6. Bibliografia

[1] “Unix Network Programming: The Sockets Networking API”, Volume 1, 3rd edition, 2003, 1024 pages, Addison-Wesley. ISBN: 0-13-141155-1

[2] Transparências das aulas teórico-práticas de Programação Avançada

[3] Páginas do manual (man pages – man <nome da função>)

[4] “Beej's Guide to Network Programming - Using Internet Sockets”, Brian “Beej Jorgensen” Hall, 2016 (http://beej.us/guide/bgnet/)