utad UNIVERSIDADE DE TRÁS-OS-MONTES E ALTO DOURO

Relatório do Trabalho Prático 1

Sistemas Distribuídos
Licenciatura de Engenharia Informática

AUTOR

DANIEL FILIPE MACEDO DOS ANJOS	Nº 58010
FRANCISCO JOSÉ FÉLIX MEIRELES	Nº 69174
JOSÉ EDUARDO CARDOSO PINTO	Nº 69330

RELATÓRIO APRESENTADO À ESCOLA DE CIÊNCIAS E TECNOLOGIA DA UNIVERSIDADE DE TRÁS-OS-MONTES E ALTO DOURO

VILA REAL, 2021

ÍNDICE

Secção 1 : Protocolo de Comunicação

O protocolo de comunicação definido contempla 4 comandos que o cliente pode usar para interagir com o servidor:

- **Date:** O servidor devolve a data e horas no momento em que recebe o pedido;
- **Get:** Ao receber este comando, o servidor gera uma chave do euro-milhões (5 números entre 1 e 50 e 2 estrelas entre 1 e 12) garantindo que não existem números ou estrelas repetidas entre si e também que a chave final é única. Por fim, a chave gerada é guardada num ficheiro e enviada na mensagem de resposta ao cliente;
- **Help:** O servidor informa o cliente em relação ao protocolo de comunicação, expondo todos os comandos disponíveis;
- **Quit:** Permite ao cliente cessar a comunicação com o servidor.

Secção 2 : Implementação

2.1. Atendimento dos clientes

Para o servidor poder receber e gerir os pedidos que recebe, recorremos à biblioteca WinSock2 que permite a criação e gestão de Windows Sockets. No início do funcionamento do servidor é criada uma Socket à qual foi associada a porta 68000 (descrita na constante DS_TEST_PORT) e, para o IP, a constante INADDR_ANY que permite associar todas as interfaces de rede disponíveis à Socket. Finalmente, a Socket é colocada à "escuta".

Quando um cliente se tenta ligar ao servidor, o pedido é capturado pela Socket criada que, posteriormente, aceita o pedido, estabelecendo a comunicação através do objeto SOCKET retornado pela função accept. Sempre que uma conexão é estabelecida, é criada uma Thread através da função CreateThread, que recebe como parâmetros o apontador para a Socket clientSocket (onde a conexão foi estabelecida), o nome da função que irá executar (handleconnection) e retorna o seu id. No final da execução, as Sockets são fechadas e limpas (através da função WSACleanup).

2.2. Comunicação

Quando a conexão é estabelecida com sucesso, o servidor envia uma mensagem de boas vindas, entrando depois num ciclo while onde lida com os pedidos do cliente e onde se mantém até que este se desconecte. Dentro do ciclo, o espaço de memória dedicado a armazenar a resposta do servidor é limpo e o comando utilizado pelo cliente é armazenado na variável strRec.

Através de uma cadeia de declarações IF ELSE, os conteúdos de strRec são comparados com cada um dos comandos que fazem parte do protocolo de comunicação. Cada comando realiza a função pretendida e, no fim, envia uma resposta para o cliente com o resultado dessa função.

2.3. Chaves Únicas

Para garantir que cada chave gerada é única e que não existem números repetidos na sua composição, foram implementadas as seguintes soluções:

- **Eliminar números repetidos dentro da mesma chave:** Na função random_key, é criado um array de inteiros que irá armazenar a chave. Existem depois dois ciclos while semelhantes que são responsáveis por chamar random_number (gera um número aleatório). O ciclo com início na linha 267 é responsável por gerar os números e o seguinte as estrelas. Cada um é executado até que sejam gerados elementos suficientes para criar a chave) e, sempre que um novo valor é gerado, é feita uma pesquisa pelo array correspondente para verificar se o novo valor já existe. Caso sim, o processo volta ao início, caso não, o vlaor é adicionado ao array.

- **Garantir que cada chave é única:** Sempre que uma chave é gerada com sucesso, é registada no ficheiro keys.txt. A função random_key é executada dentro de um ciclo do while de forma a que, quando termina, a chave retornada é testada contra o conteúdo desse ficheiro através da função keyExists. Caso exista, continua o processo de criação de keys. Caso não, é então escrita no ficheiro através da função save_key_to_file e enviada na resposta do servidor para o cliente

2.4. Atendimento simultâneo

Para lidar com pedidos de vários clientes em simultâneo, recorremos a Threads (como mencionado na Secção 1) sendo que cada um fica responsável por um cliente, permitindo estabelecer comunicação com vários clientes ao mesmo tempo.

No caso da gestão de acesso ao ficheiro keys.txt, foi usado um Mutex que permite bloquear o ficheiro antes de efetuar leitura ou escrita de dados, assegurando que apenas a thread que o bloqueou consegue manipulá-lo. No fim da operação (seja ela a função keyExists, save_key_to_file ou countSuppliedKeys) o mutex é libertado para que outra thread possa utilizar o recurso

Cliente

```c
/*
    Simple winsock client
*/

#include<stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include<winsock2.h>

#pragma comment(lib,"ws2_32.lib")
#pragma warning(disable : 4996)

int octetIsNumber(char* octet);
int isValidIP(char* IP);
char** str_split(char* a_str, const char a_delim);

int main(int argc, char* argv[])
{
    WSADATA wsa;
    SOCKET s;
    struct sockaddr_in server;
    char server_reply[2000];
    char message[2000] = "date";
    char serverIP[13];
    int recv_size;
    int ws_result;
    int i, count = 0;
    int flag = 0;
    char question;
    // Initialise winsock
    printf("\nInitialising Winsock...");
    if (WSAStartup(MAKEWORD(2, 2), &wsa) != 0)
    {
        printf("Failed. Error Code : %d", WSAGetLastError());
        return 1;
    }

    printf("Initialised.\n");

    //Create a socket
    s = socket(AF_INET, SOCK_STREAM, 0);
    if (s == INVALID_SOCKET)
    {
        printf("Could not create socket : %d", WSAGetLastError());
    }
```

```
    printf("Socket created.\n");

    //Ask user for IP
    do {
        printf("Insert the IP of the server: \n");
        scanf("%s", serverIP);
    } while (!isValidIP(serverIP));

    // create the socket  address (ip address and port)
    server.sin_addr.s_addr = inet_addr(serverIP);
    server.sin_family = AF_INET;
    server.sin_port = htons(68000);

    //Connect to remote server
    ws_result = connect(s, (struct sockaddr*)&server, sizeof(server));
    if (ws_result < 0)
    {
        puts("connect error");
        return 1;
    }

    puts("Connected");
    char command[2000];

    char** tokens;
    // Queremos que o client receba a mensagem de boas-vindas
    recv_size = recv(s, server_reply, 2000, 0);
    if (recv_size == SOCKET_ERROR)
    {
        puts("recv failed");
    }
    //Add a NULL terminating character to make it a proper string before
printing
    server_reply[recv_size] = '\0';

    printf("%s\n", server_reply);

    do {
        printf("Pressione 'g' para gerar chaves euromilhões, 'h' para aju
da ou 'q' para sair.\n");
        scanf(" %c", &question);
        flag = 1;

        switch (question)
        {
        case 'g':
```

```
            printf("Indique quantas chaves de euromilhões quer gerar: \n"
);

            scanf("%d", &i);

            do {
                // Efetuar limpeza no espa□o de mem□ria reservado para o
 command
                memset(command, 0, sizeof(command));

                strncpy(command, "get", 4);

                ws_result = send(s, command, strlen(command), 0);

                // Na ocorr□ncia de erro
                if (ws_result < 0)
                {
                    puts("Send failed");
                    return 1;
                }
                // Apagar o que est□ escrito no vector de resposta do se
rvidor
                memset(server_reply, 0, sizeof(server_reply));

                //Receive a reply from the server
                recv_size = recv(s, server_reply, 2000, 0);
                if (recv_size == SOCKET_ERROR)
                {
                    puts("recv failed");
                }
                //Add a NULL terminating character to make it a proper st
ring before printing
                server_reply[recv_size] = '\0';

                tokens = str_split(server_reply, ',');

                if (tokens)
                {
                    int i;
                    printf("Numeros= ");
                    for (i = 0; i < 5; i++)
                    {
                        printf("|%s", *(tokens + i));
                        free(*(tokens + i));
                    }
                    printf("Estrelas= ");
                    for (i; i < *(tokens + i); i++)
                    {
                        printf("|%s", *(tokens + i));
```

```
                        free(*(tokens + i));
                    }
                    printf("\n");
                    free(tokens);
                }

                // Imprime para a consola o Resultado da Resposta do serv
idor
                printf("%s\n", server_reply);

                count++;
            } while (count < i);

            flag = 0;

            break;

        case 'h':

            memset(command, 0, sizeof(command));

            strncpy(command, "help", 2000);

            ws_result = send(s, command, strlen(command), 0);

            if (ws_result < 0)
            {
                puts("Send failed");
                return 1;
            }

            memset(server_reply, 0, sizeof(server_reply));

            recv_size = recv(s, server_reply, 2000, 0);
            if (recv_size == SOCKET_ERROR)
            {
                puts("recv failed");
            }

            server_reply[recv_size] = '\0';

            printf("%s\n", server_reply);

            flag = 0;

            break;

        case 'q':
```

```
            memset(command, 0, sizeof(command));

            strncpy(command, "quit", 2000);

            ws_result = send(s, command, strlen(command), 0);

            if (ws_result < 0)
            {
                puts("Send failed");
                return 1;
            }

            memset(server_reply, 0, sizeof(server_reply));

            recv_size = recv(s, server_reply, 2000, 0);
            if (recv_size == SOCKET_ERROR)
            {
                puts("recv failed");
            }

            server_reply[recv_size] = '\0';

            printf("%s\n", server_reply);
            flag = 0;

            break;

        default:

            printf("Erro!\n");

            break;
        }
    } while (flag == 0);

    // Close the socket
    closesocket(s);

    //Cleanup winsock
    WSACleanup();

    return 0;
}

//Check if an octet of the IP given by the user is a number
int octetIsNumber(char* octet) {
    while (*octet != '\0') {
        if (!isdigit(*octet)) {
            return 0;
```

```
        }
        octet++;
    }
    return 1;
}

//Check if the IP given by the user is valid
int isValidIP(char* IP) {
    char aux[13];
    strcpy(aux, IP);
    char* ptr;
    int octetAsInt, IPdots = 0;

    if (aux == NULL) {
        return 0;
    }

    //Break the string into tokens based on the provided delimiter
    //If IP = "192.231.1.0" tokens are "192", "231", "1" and "0"
    ptr = strtok(aux, ".");

    //If the delimiter is not found on the string, there are no tokens re
turned my strtok and thus is not a valid IP
    if (ptr == NULL) {
        return 0;
    }

    while (ptr != '\0') {
        if (!octetIsNumber(ptr)) {
            return 0;
        }

        octetAsInt = atoi(ptr);

        if (octetAsInt >= 0 && octetAsInt <= 255) {
            //NULL tells strtok to continue tokenizing the string passed
in the first call
            ptr = strtok(NULL, ".");
            if (ptr != NULL) {
                IPdots++;
            }
        }
        else {
            return 0;
        }
    }

    if (IPdots != 3) {
        return 0;
```

```
    }

    return 1;
}

//char* strremove(char* str, const char* sub) {
//  char* p, * q, * r;
//  if ((q = r = strstr(str, sub)) != NULL) {
//      size_t len = strlen(sub);
//      while ((r = strstr(p = r + len, sub)) != NULL) {
//          while (p < r)
//              *q++ = *p++;
//      }
//      while ((*q++ = *p++) != '\0')
//          continue;
//  }
//  return str;
//}

char** str_split(char* a_str, const char a_delim)
{
    char** result = 0;
    size_t count = 0;
    char* tmp = a_str;
    char* last_comma = 0;
    char delim[2];
    delim[0] = a_delim;
    delim[1] = 0;

    /* Count how many elements will be extracted. */
    while (*tmp)
    {
        if (a_delim == *tmp)
        {
            count++;
            last_comma = tmp;
        }
        tmp++;
    }

    /* Add space for trailing token. */
    count += last_comma < (a_str + strlen(a_str) - 1);

    /* Add space for terminating null string so caller
       knows where the list of returned strings ends. */
    count++;

    result = malloc(sizeof(char*) * count);
```

```
    if (result)
    {
        size_t idx = 0;
        char* token = strtok(a_str, delim);

        while (token)
        {
            *(result + idx++) = strdup(token);
            token = strtok(0, delim);
        }
        if (idx == count - 1) {
        }
        *(result + idx) = 0;
    }

    return result;
}
```

Servidor

```c
/*
Simple winsock Server
*/

#include <stdio.h>
#include <stdlib.h>
#include <winsock2.h>
#include <time.h>
#include <windows.h>
#define TRUE 1
#define DS_TEST_PORT 68000
#define MAX_NUMBER 50
#define MIN_NUMBER 1
#define MAX_STAR 12
#define MIN_STAR 1
#define KEY_SIZE 7
#define KEY_FILE "keys.txt"
#define mutLabel "mutexLabel"

#pragma comment (lib, "ws2_32.lib")
#pragma warning(disable : 4996)

// function ot handle the incoming connection
//  param: the socket of the calling client

DWORD WINAPI handleconnection(LPVOID lpParam);
int* random_key(int (*random_number)(int, int));
int random_number(int min_num, int max_num);
void save_key_to_file(int* key);
char* convert_array_to_string(int array[], int n);
void print_array(int* a, int len);
int find_elem(int value, const int a[], int m, int n);
char* countSuppliedKeys();
int keyExists(int* keyGenerated);

HANDLE sharedMutex;

int main()
{

    // Initialise winsock
    WSADATA wsData;
    WORD ver = MAKEWORD(2, 2);

    printf("\nInitialising Winsock...");
    int wsResult = WSAStartup(ver, &wsData);
```

```
    if (wsResult != 0) {
        fprintf(stderr, "\nWinsock setup fail! Error Code : %d\n", WSAGet
LastError());
        return 1;
    }

    // Create a socket
    SOCKET listening = socket(AF_INET, SOCK_STREAM, 0);
    if (listening == INVALID_SOCKET) {
        fprintf(stderr, "\nSocket creationg fail! Error Code : %d\n", WSA
GetLastError());
        return 1;
    }

    printf("\nSocket created.");

    // Bind the socket (ip address and port)
    struct sockaddr_in hint;
    hint.sin_family = AF_INET;
    hint.sin_port = htons(DS_TEST_PORT);
    hint.sin_addr.S_un.S_addr = INADDR_ANY;

    bind(listening, (struct sockaddr*)&hint, sizeof(hint));
    printf("\nSocket binded.");

    // Setup the socket for listening
    listen(listening, SOMAXCONN);
    printf("\nServer listening.");

    // Wait for connection
    struct sockaddr_in client;
    int clientSize;
    SOCKET clientSocket;
    SOCKET* ptclientSocket;
    DWORD dwThreadId;
    HANDLE  hThread;
    int conresult = 0;
    HANDLE keyFileMutex;

    // Create a mutex with no initial owner
    sharedMutex = CreateMutex(
        NULL,              // default security attributes
        FALSE,             // initially not owned
        mutLabel);             // unnamed mutex

    if (sharedMutex == NULL)
    {
        printf("CreateMutex error: %d\n", GetLastError());
        return 1;
```

```
    }


    while (TRUE)
    {
        clientSize = sizeof(client);
        clientSocket = accept(listening, (struct sockaddr*)&client, &clie
ntSize);

        ptclientSocket = &clientSocket;

        printf("\nHandling a new connection.");

        // Handle the communication with the client

        hThread = CreateThread(
            NULL,                   // default security attributes
            0,                      // use default stack size
            handleconnection,       // thread function name
            ptclientSocket,          // argument to thread function
            0,                      // use default creation flags
            &dwThreadId);   // returns the thread identifier

        // Check the return value for success.
        // If CreateThread fails, terminate execution.

        if (hThread == NULL)
        {
            printf("\nThread Creation error.");
            ExitProcess(3);
        }
    }

    // Close the socket
    closesocket(clientSocket);

    // Close listening socket
    closesocket(listening);

    CloseHandle(sharedMutex);

    //Cleanup winsock
    WSACleanup();
}

DWORD WINAPI handleconnection(LPVOID lpParam)
{
    srand(time(NULL));
```

```
    char strMsg[1024];
    char strRec[1024];

    int i = 1;
    SOCKET cs;
    SOCKET* ptCs;

    HANDLE mutex;

    ptCs = (SOCKET*)lpParam;
    cs = *ptCs;

    strcpy(strMsg, "\nHello! welcome to the server...\n");
    printf("\n%s\n", strMsg);
    send(cs, strMsg, strlen(strMsg) + 1, 0);

    while (TRUE) {
        ZeroMemory(strRec, 1024);
        int bytesReceived = recv(cs, strRec, 1024, 0);
        if (bytesReceived == SOCKET_ERROR) {
            printf("\nReceive error!\n");
            break;
        }
        if (bytesReceived == 0) {
            printf("\nClient disconnected!\n");
            break;
        }

        printf("%i : %s\n", i++, strRec);
        //send(cs, strRec, bytesReceived + 1, 0);

        if (strcmp(strRec, "date") == 0) {
            // current date/time based on current system
            time_t now = time(0);
            // convert now to string form
            char* dt = ctime(&now);

            strcpy(strMsg, "\n\nThe local date and time is: ");
            strcat(strMsg, dt);
            strcat(strMsg, "\n");

            send(cs, strMsg, strlen(strMsg) + 1, 0);


        }
        else if (strcmp(strRec, "get") == 0) {

            mutex = OpenMutex(MUTEX_ALL_ACCESS, FALSE, mutLabel);
```

```
            if (mutex == NULL) {
                printf("Mutex error: %d\n", GetLastError());
            }
            // Gerar a Chave

            int* key_generated;

            do {

                key_generated = random_key(random_number);

            } while (keyExists(key_generated));

            if (!key_generated) {
                printf("ERROR: N□o foi poss□vel gerar a chave do euromil
h□es");
            }

            print_array(key_generated, KEY_SIZE);

            // Abrir o ficheiro para efetuar o registo
            WaitForSingleObject(mutex, INFINITE);

            save_key_to_file(key_generated);

            ReleaseMutex(mutex);

            // Enviar para o cliente a chave gerada

            //strcpy(strMsg, "\n\nChave do euromilhoes: ");
            //strcat(strMsg, convert_array_to_string(key_generated, 7));
            //strcat(strMsg, "\n Chaves fornecidas: ");

            WaitForSingleObject(mutex, INFINITE);

            //strcat(strMsg, countSuppliedKeys());

            ReleaseMutex(mutex);

            //strcat(strMsg, convert_array_to_string(int array[], int n))
;
            strcat(strMsg, "\n");

            send(cs, strMsg, strlen(strMsg) + 1, 0);

            // Libertar mem□ria
            free(key_generated);
            CloseHandle(mutex);
            key_generated = NULL;
```

```
        }
        else if (strcmp(strRec, "help") == 0)
        {
            strcpy(strMsg, "\n\nHelp: ");
            strcat(strMsg, "O servidor utiliza os comandos get, help e qu
it.\n");


            send(cs, strMsg, strlen(strMsg) + 1, 0);
        }
        else if (strcmp(strRec, "quit") == 0) {
            strcpy(strMsg, "\n400 BYE\n");
            send(cs, strMsg, strlen(strMsg) + 1, 0);

            // Close the socket
            closesocket(cs);
            return 0;
        }

    }
}
// Fun��o que gera uma chave do euromilhoes de forma aleat�ria
int* random_key(int (*random_number)(int, int)) {
    // Array que vai armazenar a chave do euromilhoes
    int* key = (int*)malloc(KEY_SIZE * sizeof(int));
    if (!key)
        return NULL;

    int temp = 0;
    int generatedNums = 0;
    // Gerar os 5 n�meros
    while (generatedNums != 5) {

        temp = (*random_number)(MIN_NUMBER, MAX_NUMBER);

        if (find_elem(temp, key, 0, generatedNums) == -1) {
            key[generatedNums] = temp;
            generatedNums++;
        }

    }

    while (generatedNums != 7) {

        temp = (*random_number)(MIN_STAR, MAX_STAR);

        //Stars start on the 5th position fo the array
        if (find_elem(temp, key, 5, generatedNums) == -1) {
            key[generatedNums] = temp;
```

```
            generatedNums++;
        }
    }


    // Retorna a chave
    return key;
}

/*Retorna um random n?mero inteiro, no intervalo [min_num, max_num]*/
int random_number(int min_num, int max_num)
{
    int result = 0, low_num = 0, hi_num = 0;

    if (min_num < max_num)
    {
        low_num = min_num;
        hi_num = max_num + 1; // include max_num in output
    }
    else {
        low_num = max_num + 1; // include max_num in output
        hi_num = min_num;
    }


    result = (rand() % (hi_num - low_num)) + low_num;
    return result;
}

void save_key_to_file(int* key) {

    int fileWriteError = 0;
    int* endOfArray = key + KEY_SIZE;
    DWORD waitResult;

    FILE* fp;
    fp = fopen(KEY_FILE, "a+");

    if ( fp == NULL) {
        perror("Erro a abrir ficheiro!\n");
    }



    // Regista o ficheiro, a chave gerada
    while (key != endOfArray && fileWriteError == 0) {

        if (fprintf(fp, "%d ", *key) > 0) {
            key++;
```

```
        }
        else {
            printf("Erro ao escrever no ficheiro!\n");
            fileWriteError = 1;
        }
    }
    time_t currentTime = time(0);
    char* cT = ctime(&currentTime);

    fprintf(fp, "- %s", cT);

    // Fecha ficheiro
    fclose(fp);
}

int keyExists(int* keyGenerated) {

    FILE* fp;
    fp = fopen(KEY_FILE, "r");
    int exists = 0;
    HANDLE mutex;

    if (fp == NULL) {
        perror("Erro a abrir ficheiro!\n");
    }

    char line[255];

    mutex = OpenMutex(MUTEX_ALL_ACCESS, FALSE, mutLabel);

    if (mutex == NULL) {
        printf("Mutex error: %d\n", GetLastError());
    }


    while (!feof(fp)) {

        int keyPlace = 0;
        int key[KEY_SIZE];
        char* numbers;
        char lineSeparation[255];

        WaitForSingleObject(mutex, INFINITE);
        fgets(line, 255, fp);
        ReleaseMutex(mutex);
        strcpy(lineSeparation, line);

        //Separate key from date emmited
        strtok(lineSeparation, "-");
```

```
        //Separate each number
        numbers = strtok(lineSeparation, " ");


        while (numbers != NULL) {

            key[keyPlace] = atoi(numbers);
            numbers = strtok(NULL, " ");
            keyPlace++;

        }

        for (int i = 0; i < KEY_SIZE; i++) {

            if (key[i] == keyGenerated[i]) {
                exists = 1;
            }
        }

    }

    CloseHandle(mutex);



    return exists;

}

// Converte um array de inteiros para uma string
char* convert_array_to_string(int array[], int n) {
    int i;
    char* output = (char*)malloc(128);
    char* point = output + 1;
    *output = '[';
    for (i = 0; i != n; ++i)
        point += sprintf(point, i + 1 != n ? "%d," : "%d]", array[i]);

    return output;
}

// Imprime array para efeitos de debug
void print_array(int* a, int len) {
    for (int i = 0; i < len; i++) printf("%d ", a[i]);
}

// Procura elemento no array
int find_elem(int value, const int a[], int m, int n)
{
```

```
    while (m < n && a[m] != value) m++;

    return m == n ? -1 : m;
}

char* countSuppliedKeys() {

    FILE* fp;
    char keyCountString[1024];
    fp = fopen(KEY_FILE, "r");
    HANDLE mutex;

    mutex = OpenMutex(MUTEX_ALL_ACCESS, FALSE, mutLabel);

    if (mutex == NULL) {
        printf("Mutex error: %d\n", GetLastError());
    }

    if ( fp == NULL) {
        perror("fopen");
    }

    int keyCount = 0;
    int ch = 0;

    while (!feof(fp)) {
        WaitForSingleObject(mutex, INFINITE);
        ch = fgetc(fp);
        ReleaseMutex(mutex);

        if (ch == '\n') {
            keyCount++;
        }
    }

    sprintf(keyCountString, "%d", keyCount);

    CloseHandle(mutex);

    return keyCountString;
}
```