UNIVERSIDADE DE TRÁS-OS-MONTES E ALTO DOURO utad

Sistemas Distribuídos

Trabalho prático 1

Servidor de Chaves do Euromilhões

Licenciatura em Engenharia Informática

Trabalho realizado por:

Vítor Neto – 68717

João Leal – 68719

Hugo Anes – 68578

Abril de 2021

Protocolo

Lista de comandos:

- “CHAVE X” – Gera um número (X) de chaves do Euromilhões;
- “CHAVE” – Gera uma chave do Euromilhões;
- “QUIT” – Encerra a comunicação com o servidor.

Respostas possíveis:

- “100 OK” – Ligação estabelecida
- “200 OK” – Recebido com sucesso
- “421 MISSDIRECTED” - Comando desconhecido
- “404 NOT FOUND” – Comando incorreto
- “400 BYE” – Ligação terminada
- “N N N N N + E E” - Chave
- “Generated X keys so far this session”
- “Generated Y keys so far in total”

Estados:

- Conectado
- Conexão falhada
- Desligado
- A aguardar pedido
- Erro no pedido
- A processar pedido
- À espera de conexões
- Mutex ocupado

Implementação

Neste trabalho, serão implementados um cliente e um servidor, com recurso a winsocks 2.2 no Visual Studio C.

Nota: o ficheiro "chavesTotais.txt", tem que estar presente e com o valor '0' escrito.

Para o atendimento aos clientes, será necessário inicializar uma socket, bem como um apontador para a socket, visto que vamos efetuar atendimento concurrente, ou seja, atender vários clientes ao mesmo tempo, com recurso a threads e mutexes, porque será necessário aceder a ficheiros e os mutexes são cruciais para esta função. Começamos então por inicializar as sockets e o mutex como pode ser visto nas figuras 1 e 2:

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

printf("\nSocket created. Waiting...");

//Dar bind da socket, ou seja, atribuir os valores do IP e porto
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));

//Colocar a socket em modo listen, para que possa "escutar" os novos clientes que se conectem
listen(listening, SOMAXCONN);

//Esperar a conexão do cliente
struct sockaddr_in client;
int clientSize;

//Definição de uma socket, um ponteiro para uma socket, uma HANDLE e uma DWORD, para que possa ser atribuído um ID a cada uma das threads que são criadas concurrentemente
SOCKET clientSocket;
SOCKET* ptclientSocket;
DWORD dwThreadId;
HANDLE  hThread;
```

Figura 1 – Inicialização das sockets e atribuição do IP e porto do servidor

```
//Criação do mutex
ghMutex = CreateMutex(
    NULL,   //Não utilizamos parâmetros de segurança
    FALSE,  //O mutex não possui utilizador inicial, daí o parâmetro FALSE
    NULL);  //Não está atribuído a nenhum lpParam inicialmente

//Verifica a existência de erros na inicialização do mutex
if (ghMutex == NULL) {
    printf("CreateMutex error: %d\n", GetLastError());
    return 1;
}
```

Figura 2 – Criação do mutex

Para que possa ser utilizado o mutex e para que possam ser atendidos vários clientes concorrentemente, será necessário que sejam criadas threads por cada novo cliente se decida conectar ao servidor. Este passo é de relevante importância, visto que sem este passo, o servidor apenas conseguiria atender um cliente de cada vez. As threads possibilitam que a execução do programa se subdivida em linhas de execução individuais e independentes, permitindo assim que cada cliente "pense" que é o único que está conectado ao servidor, visto que não consegue saber nem perceber o número de clientes que também estão conectados a esse mesmo servidor. Na Figura 3, pode-se verificar todos os passos necessários para a criação de novas threads, bem como a sua atribuição a cada novo cliente que se conecta ao servidor, atribuindo também um ID único a cada nova thread que é criada.

```
//Ciclo infinito que percorre sempre até que o servidor se desligue, aguardando novas conexões e criando e atribuindo threads para cada uma delas
while (TRUE)
{
    //Descobre o tamanho do client que se está a tentar conectar ao server e aceita-o baseando-se no seu tamanho
    clientSize = sizeof(client);
    clientSocket = accept(listening, (struct sockaddr*)&client, &clientSize);
    ptclientSocket = &clientSocket; //Atribui-se o novo cliente ao ponteiro
    printf("\nHandling a new connection.");

    //Lida com a comunicação de novos clientes ao server, criando uma thread para cada um, com diferentes thread IDs e redirecionando cada uma para a função handleconnection
    hThread = CreateThread(
        NULL,                   //Atributos de segurança por defeito (NULL)
        0,                      //Tamanho da stack por defeito (0)
        handleconnection,       //Nome da função que vai "tratar" das threads
        ptclientSocket,         //Argumento que é passado como lpParam à função handleconnection
        0,                      //Número de flags por defeito (0)
        &dwThreadId);           //ID atribuído à thread que está a ser criada

    //Testar se a criação da nova thread deu erro ou não, se der erro termina a conexão
    if (hThread == NULL)
    {
        printf("\nThread Creation error.");
        ExitProcess(3);
    }
}
```

Figura 3 – Criação e atribuição das threads a cada novo cliente

O cliente conecta-se ao servidor ao fornecer o IP desejado.

De volta ao servidor, na função handleconnection, para onde a criação das novas threads serão reencaminhadas, são criados sockets para cada um dos novos clientes, com o ID passado pelo lpParam, da função handleconnection. É também inicializado o dwWaiResult para usar com o mutex.

```
DWORD dwWaitResult;
SOCKET cs;
SOCKET* ptCs;

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

Figura 4 – Atribuição da socket e ID da thread

O mutex vai ser de extrema importância, visto que é necessário aceder a ficheiros. O que o mutex faz é funcionar como uma "flag", isto é, por exemplo, se estiverem vários clientes ligados ao servidor a enviar-lhe pedidos, estes terão de aceder ao mesmo ficheiro, visto que este está armazenado no servidor e é igual para todos. Ao tentarem aceder, por mais pequena que seja a probabilidade, existe a possibilidade de dois clientes tentarem abrir o ficheiro ao mesmo tempo, o que vai produzir resultados indesejados. Para resolver este problema, o mutex funciona como um aviso para os clientes que tentem aceder ao ficheiro, isto é, caso um cliente esteja a utilizar o ficheiro, se outros tentarem aceder a esse mesmo ficheiro ao mesmo tempo, serão colocados numa lista de espera, até que o mutex seja libertado e um dos clientes que está à espera fique com a posse do mutex e possa aceder ao ficheiro, visto que só a thread que possui o mutex tem acesso ao ficheiro. A Figura 7 demonstra o uso do mutex desenvolvido:

```
dwWaitResult = WaitForSingleObject(
    ghMutex,
    INFINITE);
switch(dwWaitResult){
case WAIT_OBJECT_0:
    __try {
            for (int k = 0; k < numChaves; k++)
            {
                again1:
                GerarChave(chave);
                Sleep(1);
                if (VerificarChaveRepetida(chave) == 1) {
                    goto again1;
                }
                else {
                    intTostring = StringFromKey(chave);
                    strcat_s(strMsg, 4096, intTostring);
                    strcat_s(strMsg, 4096, "\n");
                    chavesGeradas++;
                    chavesGeradasTotal = SomarChavesTotais();
                }
            }
    }
    __finally {
        if (!ReleaseMutex(ghMutex)) {
            printf("Erro");
            return 0;
        }
    }
    break;
case WAIT_ABANDONED:
    return FALSE;
}
```

Figura 5 – Atendimento simultâneo com recurso a mutex

Para que haja a garantia de que não são geradas chaves repetidas, o servidor acede ao ficheiro, onde são guardadas todas as chaves que são geradas ao longo do tempo e passa-as, linha a linha, para uma variável que irá ser utilizada para comparar com a última chave gerada, que é passada como parâmetro à função VerificarChaveRepetida. Esta função verifica todas as chaves presentes no ficheiro e compara com esta chave, caso os números sejam iguais, incrementa 1 valor à variável "flag", que irá retornar 1 caso seja igual ou maior a 7, ou seja, caso os números de uma dada chave do ficheiro sejam todas iguais à chave em questão e irá guardar a nova chave no ficheiro e retornar 0 caso verifique que não existem chaves no ficheiro iguais à que está a ser gerada.

```
//Para verificarmos se existem chaves repetidas, temos de passar a última chave gerada como parâmetro para ser comparada
int VerificarChaveRepetida(int* chave) {
    FILE* fp = fopen("chaves.txt", "r"); //Abrir o ficheiro em modo de leitura
    int flag = 0;                        //Inicializar a flag em 0, para sinalizar se existe ou não chave repetida posteriormente
    int chavesTotais[7] = { 0, 0, 0, 0, 0, 0, 0 };

    if (fp == NULL) {
        printf("Erro a ler o ficheiro!");
    }
    while (!feof(fp)) { //Percorrer o ficheiro e armazenar na cada um dos valores encontrados por linha em cada uma das posições do array chavesTotais
        flag = 0;
        fscanf(fp, "%d %d %d %d %d %d %d\n", &chavesTotais[0], &chavesTotais[1], &chavesTotais[2], &chavesTotais[3], &chavesTotais[4], &chavesTotais[5], &chavesTotais[6]);

        for (int i = 0; i < 7; i++) { //Percorrer ambos os arrays da última chave gerada e das chaves encontradas no ficheiro para comparar número a número
            if (chavesTotais[i] == chave[i]) //Caso existam números iguais, soma 1 à flag
                flag += 1;                   //Nota: Esta comparação é possível porque todas as chaves já estão organizadas por ordem crescente
        }
        if (flag >= 7) { //Caso a flag seja maior ou igual a 7, significa que a última chave gerada tem todos os 7 números iguais a uma das chaves presentes no ficheiro e informamos o main disso ao retornar 1
            fclose(fp);
            printf("\nFound a repeated key!\n");
            return 1;
        }
    }
    //Caso a chave não seja repetida, dá return de 0, a sinalizar o main de que a chave não é repetida e guarda a chave no ficheiro de chaves totais em modo de "append"
    FILE* fpw = fopen("chaves.txt", "a");
    for (int i = 0; i < 6; i++)
    {
        fprintf(fpw, "%d ", chave[i]);
    }

    fprintf(fpw, "%d\n", chave[6]);

    fclose(fpw);
    return 0;
}
```

Figura 6 – Garantia de geração de chaves únicas

Anexo

Conteúdo

Código do server:

```
//Código do server
//Código desenvolvido por Vítor Neto 68717, João Leal 68719, Hugo Anes 68571
#include <stdio.h>
#include <stdlib.h>
#include <winsock2.h>
#include <time.h>
#include <Windows.h>
#include <cstdint>

#define TRUE 1
#define DS_TEST_PORT 68000 //Definir o porto do servidor
#define N_CHAVE 7          //Número de algarismos que uma chave tem

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

//Prototipagem das funções
DWORD WINAPI handleconnection(LPVOID lpParam);  //Esta função servirá para tratar
das conexões concorrentes, que irá ser possível através do uso de threads e
mutexes
HANDLE ghMutex;
       //Definição de uma handle (número) que irá servir para implementação do
mutex
const char* StringFromKey(int* chave);                //Função que converte a
chave para um array de caracteres
void GerarChave(int* chave);                                    //Função que gera a
chave aleatória com 5 números de 1-50 e 2 estrelas (números) de 1-12
int* OrdenarChave(int* chave);                                  //Função que ordena
a chave por ordem crescente em 2 etapas, entre os números e depois entre as
estrelas
int SomarChavesTotais();                                        //Calcula o número
de chaves que já foram geradas no total pelo server
int VerificarChaveRepetida(int* chave);               //Verifica se existem
chaves repetidas alguma vez geradas pelo servidor

//Programa principal
int main()
{
       //Inicializar o winsock
       WSADATA wsData;
       WORD ver = MAKEWORD(2, 2);

       //Criação do mutex
       ghMutex = CreateMutex(
              NULL,  //Não utilizamos parâmetros de segurança
```

```
		FALSE, //O mutex não possui utilizador inicial, daí o parâmetro FALSE
		NULL); //Não está atribuído a nenhum lpParam inicialmente

	//Verifica a existência de erros na inicialização do mutex
	if (ghMutex == NULL) {
		printf("CreateMutex error: %d\n", GetLastError());
		return 1;
	}

	//Verifica a existência de erros na inicialização da winsock, caso wsResult seja diferente de 0, dá erro
	printf("\nInitialising Winsock...");
	int wsResult = WSAStartup(ver, &wsData);
	if (wsResult != 0) {
		fprintf(stderr, "\nWinsock setup fail! Error Code : %d\n", WSAGetLastError());
		return 1;
	}

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

	printf("\nSocket created. Waiting...");

	//Dar bind da socket, ou seja, atribuir os valores do IP e porto
	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));

	//Colocar a socket em modo listen, para que possa "escutar" os novos clientes que se conectem
	listen(listening, SOMAXCONN);

	//Esperar a conexão do cliente
	struct sockaddr_in client;
	int clientSize;

	//Definição de uma socket, um ponteiro para uma socket, uma HANDLE e uma DWORD, para que possa ser atribuído um ID a cada uma das threads que são criadas concurrentemente
	SOCKET clientSocket;
	SOCKET* ptclientSocket;
	DWORD dwThreadId;
	HANDLE  hThread;

	//Ciclo infinito que percorre sempre até que o servidor se desligue, aguardando novas conexões e criando e atribuindo threads para cada uma delas
	while (TRUE)
	{
		//Descobre o tamanho do client que se está a tentar conectar ao server e aceita-o baseando-se no seu tamanho
		clientSize = sizeof(client);
		clientSocket = accept(listening, (struct sockaddr*)&client, &clientSize);
```

```
			ptclientSocket = &clientSocket; //Atribui-se o novo cliente ao
ponteiro
			printf("\nHandling a new connection.");

			//Lida com a comunicação de novos clientes ao server, criando uma
thread para cada um, com diferentes thread IDs e redirenciondo cada uma para a
função handleconnection
			hThread = CreateThread(
				NULL,			//Atributos de segurança por defeito
(NULL)
				0,			//Tamanho da stack por defeito (0)
				handleconnection,	//Nome da função que vai "tratar" das
threads
				ptclientSocket,		//Argumento que é passado como
lpParam à função handleconnection
				0,			//Número de flags por defeito (0)
				&dwThreadId);		//ID atribuído à thread que está
a ser criada

			//Testar se a criação da nova thread deu erro ou não, se der erro
termina a conexão
			if (hThread == NULL)
			{
				printf("\nThread Creation error.");
				ExitProcess(3);
			}
		}
		//Fechar as sockets listening e dos clientes bem como o mutex que foi
criado
		closesocket(clientSocket);
		closesocket(listening);
		CloseHandle(ghMutex);

		//Limpeza da winsock
		WSACleanup();
		return 1;
}

//Função que trata das conexões com os novos clientes que se conectam ao
servidor, ou seja das threads
DWORD WINAPI handleconnection(LPVOID lpParam) {
		srand(time(NULL));					//Dar
seed ao rand, para que se baseie no tempo do pc
		const char* intTostring;				//Variável
que armazena o valor que é convertido de inteiro para vetor de chars
		char* halfStrRec;
		//Mensagem received dividida por delimitadores
		int numChaves = 0;
		//Número de chaves a serem geradas a cada iteração, fornecido pelo cliente
		int chavesGeradas = 0;
		//Variável que irá ser utilizada para calcular o número de chaves que
foram geradas durante a sessão, ou seja, enquanto o servidor está ligado,
reiniciando sempre que o servidor reinicia
		int chavesGeradasTotal = 0;				//Esta
variável irá servir para armazenar o número de chaves que já foram geradas no
total pelo servidor
		int chave[N_CHAVE] = { 0, 0, 0, 0, 0, 0, 0 };	//Armazena as chaves
geradas a cada iteração
		char strMsg[4096];
		//Array que armazena as mensagens que serão encaminhadas para o cliente
		char strRec[4096];
		//Array que armazena as mensagens que são recebidas pelo cliente
```

```
	DWORD dwWaitResult;
	//Double word que servirá para verificar se o mutex tem dono ou se a thread precisa de aguardar a sua vez
	SOCKET cs;
	//Socket criada que servirá para armazenar a socket passada como lpParam na função handleconnection
	SOCKET* ptCs;
	//Ponteiro que aponta para a socket criada acima

	ptCs = (SOCKET*)lpParam;					//Atribuição da socket passada como parâmetro à função
	cs = *ptCs;
	//Apontador para a socket

	//Se tudo correr bem até aqui, o servidor envia o status code "100 OK" a indicar que o cliente se conseguiu conectar com sucesso
	strcpy(strMsg, "100 OK");
	send(cs, strMsg, strlen(strMsg) + 1, 0);

	//Ciclo infinito que fica a aguardar comandos do cliente até que este se decida desconectar, ou que algum erro ocorra
	while (1) {
		//A cada iteração, será necessário "limpar" a troca de mensagens entre o servidor e o cliente
		ZeroMemory(strRec, 4096);
		ZeroMemory(strMsg, 4096);

		//Recebimento e verificação de ocorrimento de erros da mensagem recebida do cliente
		int bytesReceived = recv(cs, strRec, 4096, 0);
		if (bytesReceived == SOCKET_ERROR) {
			printf("\nReceive error!\n");
			break;
		}
		if (bytesReceived == 0) {
			printf("\nClient disconnected!\n");
			break;
		}
		printf("\n Client: %s\n", strRec);

		if (strcmp(strRec, "\r\n") != 0)
		{
			//Se a mensagem recebida for igual a "QUIT", então o server envia o status code "400 BYE" ao cliente e fecha o socket, visto que esta mssg pede para terminar a conexão
			if (strcmp(strRec, "QUIT") == 0 || strcmp(strRec, "quit") == 0) {
				strcpy(strMsg, "400 BYE");
				send(cs, strMsg, strlen(strMsg) + 1, 0);
				printf("Response: %s\n", strMsg);
				closesocket(cs);
				return 0;
			}

			//Dividir a mensagem recebida até aparecer o primeiro espaço, ou seja " "
			halfStrRec = strtok(strRec, " ");
			if (strcmp(halfStrRec, "CHAVE") == 0 || strcmp(halfStrRec, "chave") == 0) //Se a mensagem dividida for igual a "CHAVE", passaremos a comparar a seguinte parte da mensagem
			{
```

```
                        halfStrRec = strtok(NULL, " "); //Voltamos a dividir a
mensagem de forma a obtermos a seguinte parte da mensagem, ou seja, o número de
chaves, por exemplo, se for "CHAVE 10", a próxima parte da mensagem a ser
analisada é o 10
                        if (halfStrRec != NULL) { //Se não for fornecido
nenhum número, passamos este if à frente e será visto à frente o que acontece
                                numChaves = atoi(halfStrRec); //Se for
fornecido um número, convertemos este número para inteiro para que seja possível
efetuar um ciclo for, conforme o número de chaves a serem geradas
                                if (numChaves != NULL) { //Se o número de
chaves passado for um número, passamos para este if, senão, se for, por exemplo,
um caracter, este if é ignorado

                                        //O server envia a mensagem 200 OK para
o cliente caso o comando fornecido seja correto, para que este aguarde a geração
da chave
                                        strcpy(strMsg, "200 OK");
                                        send(cs, strMsg, strlen(strMsg) + 1, 0);
                                        ZeroMemory(strMsg, 4096);

                                                //O dwWaitResult verifica se o
mutex está ocupado, e indica quanto tempo a thread deve aguardar no máximo para
que lhe seja disponibilizado este mutex
                                                dwWaitResult =
WaitForSingleObject(
                                                ghMutex,
                                                INFINITE);
                                                switch (dwWaitResult) { //Caso o
resultado seja WAIT_OBJECT_0, o mutex não está ocupado e vai ser atribuído à
thread
                                                case WAIT_OBJECT_0:
                                                        __try {
                                                                for (int k =
0; k < numChaves; k++) //Enquanto o mutex estiver a ser utilizado, será gerado um
número de chaves igual ao fornecido pelo cliente
                                                                        {

        again:

        GerarChave(chave); //A função GerarChave irá ser analisada com mais
pormenor no final, esta serve para gerar chaves

        if (VerificarChaveRepetida(chave) == 1) { //Caso a função
VerificarChaveRepetida retorne 1, ou seja, encontrou uma chave repetida, a chave
atual é descartada e salta para o ponto "repetir", para que seja gerada uma nova
chave

        goto again;

        }

        else { //Se não se verificar que a chave foi repetida, passamos à
conversão da chave de inteiro para array de carateres com a função StringFromKey
(analisada posteriormente)

        intTostring = StringFromKey(chave);

        strcat_s(strMsg, 4096, intTostring); //A chave, já convertida em array de
carateres, é adicionada à mensagem a ser enviada ao cliente

        strcat_s(strMsg, 4096, "\n");
```

```
		chavesGeradas++; //chavesGeradas inrementa 1 valor, para sabermos quantas chaves já foram geradas nesta sessão

		chavesGeradasTotal = SomarChavesTotais(); //E a função SomarChavesTotais, também analisada posteriormente, calcula o total de chaves geradas no total desde sempre

		}
                                                                        }
                                                                }
                                                                __finally { //Após a thread concluir a sua tarefa com o mutex que lhe foi fornecido, esta liberta-o, para que outras o possam utilizar
                                                                        if (!ReleaseMutex(ghMutex)) {

		printf("Erro");
                                                                                return 0;
                                                                        }
                                                                }
                                                                break;

                                                        case WAIT_ABANDONED: //Se o mutex já não existir, ocorre um erro
                                                                return FALSE;
                                                        }

                                                //Caso todos os passos anteriores se tenham verificado, é ainda anexado à mensagem que vai ser enviada ao cliente o número de chaves geradas esta sessão e no total
                                                strcat_s(strMsg, 4096, "Number of keys generated this session: ");
                                                sprintf_s(&strMsg[strlen(strMsg)], sizeof(int), "%d", chavesGeradas);
                                                strcat_s(strMsg, 4096, "\n");
                                                strcat_s(strMsg, 4096, "Number of keys generated in total: ");
                                                sprintf_s(&strMsg[strlen(strMsg)], sizeof(int), "%d", chavesGeradasTotal);
                                                strcat_s(strMsg, 4096, "\n");
                                        }
                                        else
                                        {
                                                strcpy(strMsg, "404 NOT FOUND"); //O status code 404 ocorre quando o número de chaves a ser gerado é inválido
                                        }
                                }
                                else //Se o número de chaves não for fornecido pelo cliente, é assumido que o cliente deseja gerar apenas 1 chave e executa exatamente as mesmas funções como se fosse para várias chaves mas com o numChaves igualado a 1
                                {
                                        numChaves = 1;
                                        strcpy(strMsg, "200 OK");
                                        send(cs, strMsg, strlen(strMsg) + 1, 0);
                                        ZeroMemory(strMsg, 4096);

                                                dwWaitResult = WaitForSingleObject(
                                                        ghMutex,
                                                        INFINITE);
                                                switch(dwWaitResult){
```

```
                                        case WAIT_OBJECT_0:
                                                __try {
                                                        for (int k = 0; k <
numChaves; k++)
                                                        {
                                                                again1:

                GerarChave(chave);
                                                                Sleep(1);
                                                                if
(VerificarChaveRepetida(chave) == 1) {
                                                                        goto
again1;
                                                                }
                                                                else {

                intTostring = StringFromKey(chave);

                strcat_s(strMsg, 4096, intTostring);

                strcat_s(strMsg, 4096, "\n");

                chavesGeradas++;

                chavesGeradasTotal = SomarChavesTotais();
                                                                }
                                                        }
                                                __finally {
                                                        if (!ReleaseMutex(ghMutex))
{
                                                                printf("Erro");
                                                                return 0;
                                                        }
                                                }
                                                break;
                                        case WAIT_ABANDONED:
                                                return FALSE;
                                        }

                                        strcat_s(strMsg, 4096, "Number of keys
generated this session: ");
                                        sprintf_s(&strMsg[strlen(strMsg)], sizeof(int),
"%d", chavesGeradas);
                                        strcat_s(strMsg, 4096, "\n");
                                        strcat_s(strMsg, 4096, "Number of keys
generated in total: ");
                                        sprintf_s(&strMsg[strlen(strMsg)], sizeof(int),
"%d", chavesGeradasTotal);
                                        strcat_s(strMsg, 4096, "\n");
                                }
                        }
                        else
                        {
                                strcpy(strMsg, "421 MISSDIRECTED"); //Com o status
code 421, o servidor informa o cliente de que o comando enviado não é reconhecido
pelo mesmo
                        }
                        //Finalmente, caso não ocorram erros, a mensagem é
encaminhada para o cliente e o servidor apresenta também a mensagem bem como a
data e hora em que esta foi enviada
                        send(cs, strMsg, strlen(strMsg) + 1, 0);
```

```
                        printf("Server response: %s", strMsg);
                        time_t now = time(0);
                        char* dt = ctime(&now);
                        printf("Key sent on: %s\n", dt);
                }
        }
        return 1;
}

//A função GerarChave recebe como parâmetro o array de inteiros "chave" para que sejam armazenados lá dentro os números da chave
void GerarChave(int* chave) {

        for (int i = 0; i < 5; i++) //Neste for, são gerados 5 números de 1 a 50 e são armazenados nas 5 primeiras posições do array "chave"
        {
                chave[i] = rand() % 50 + 1;
                for (int j = 0; j < i; j++) //Como não podem sair números repetidos na mesma chave, caso um dos números seja igual ao novo gerado, o número tem de ser gerado novamente
                {
                        if (chave[i] == chave[j])
                        {
                                i--;
                                break;
                        }
                }
        }

        //Para serem geradas as estrelas, de 1 a 12, é seguida uma ordem de ideias parecida com a dos números, mas será mais fácil utilizar um ciclo While, visto que são apenas 2 estrelas
        int estrela1, estrela2;
        estrela1 = rand() % 12 + 1;
        estrela2 = estrela1;
        while (estrela1 == estrela2) {

                estrela2 = rand() % 12 + 1;

        }

        //No final, com a certeza de que as 2 estrelas não são repetidas, são armazenadas nas 2 últimas posições do array "chave"
        chave[5] = estrela1;
        chave[6] = estrela2;

        chave = OrdenarChave(chave); //Para finalizar esta função, o array "chave" vai passar pela função que ordena a chave por ordem crescente (função explicada posteriormente)
}
//Esta função converte a chave para um array de carateres
const char* StringFromKey(int* chave) {

        char* str = (char*)malloc(2000);
        if (str == NULL)
        {
                return "";
        }
        strcpy_s(str, 2000, "");

        for (int i = 0; i < N_CHAVE; i++) //Para todos os números da chave, converte-os para caracter e armazena-os na variável str
        {
```

```
			if (chave[i] < 10) //Estes ifs colocam um "+" entre os números e as
estrelas
			{
				sprintf_s(&str[strlen(str)], (sizeof(int) + (3 *
sizeof(char))), "%d   ", chave[i]);
			}
			else
			{
				sprintf_s(&str[strlen(str)], (sizeof(int) + (3 *
sizeof(char))), "%d  ", chave[i]);
			}
			if (i == 4)
			{
				strcat_s(str, sizeof(char) * (strlen(str) + 7), " +   ");
			}
		}
		return str; //No final, retorna a variável str com todos os ints
convertidos para char
}
//A função OrdenarChave, recebe como parâmetro a chave que irá ser ordenada por
ordem crescente
int* OrdenarChave(int* chave) {

		int temp = 0;
		int temp2 = 0;

		//Serão necessários 2 ciclos for, um para os números e um para as
estrelas, visto que as estrelas têm que ficar obrigatoriamente nas 2 últimas
posições da chave
		//Os ciclos for serão semelhantes para os números como para as estrelas,
armazenando o número a ser trocado temporariamente caso o número seguinte seja
menor que o atual
		for (int i = 0; i < 5; i++)
			for (int j = i + 1; j < 5; j++) {
				if (chave[i] > chave[j]) {
					temp = chave[j];
					chave[j] = chave[i];
					chave[i] = temp;
				}
			}
		for (int k = 5; k < 7; k++)
			for (int l = k + 1; l < 7; l++) {
				if (chave[k] > chave[l]) {
					temp2 = chave[l];
					chave[l] = chave[k];
					chave[k] = temp2;
				}
			}
		return chave; //Retorna a chave já ordenada
}
//Esta função soma o total de chaves que já foram geradas pelo servidor ao aceder
a um ficheiro .txt com um inteiro armazenado que representa o número de chaves
geradas
int SomarChavesTotais() {
		int chavesGeradasTotal = 0;
		FILE* chavesTotais = fopen("chavesTotais.txt", "r"); //Abrimos o ficheiro
em modo de leitura

		if (chavesTotais == NULL) {
			printf("Error!");
			exit(-1);
		}
```

```
	fscanf(chavesTotais, "%d", &chavesGeradasTotal); //Passamos o inteiro que está armazenado no ficheiro para a variável chavesGeradasTotal e incrementamos 1 valor
	chavesGeradasTotal += 1;

	fclose(chavesTotais); //Fechamos o ficheiro em modo de leitura

	FILE* chavesTotaisW = fopen("chavesTotais.txt", "w"); //E abrimos em modo de escrita para o atualizarmos com o novo número de chaves geradas no total
	fprintf(chavesTotaisW, "%d", chavesGeradasTotal);
	fclose(chavesTotaisW); //Fechamos o ficheiro em modo de escrita
	return chavesGeradasTotal; //E retornamos o número descoberto
}
//Para verificarmos se existem chaves repetidas, temos de passar a última chave gerada como parâmetro para ser comparada
int VerificarChaveRepetida(int* chave) {
	FILE* fp = fopen("chaves.txt", "r"); //Abrir o ficheiro em modo de leitura
	int flag = 0;                                    //Inicializar a flag em 0, para sinalizar se existe ou não chave repetida posteriormente
	int chavesTotais[7] = { 0, 0, 0, 0, 0, 0, 0 };

	if (fp == NULL) {
		printf("Erro a ler o ficheiro!");
	}
	while (!feof(fp)) { //Percorrer o ficheiro e armazenar na cada um dos valores encontrados por linha em cada uma das posições do array chavesTotais
		flag = 0;
		fscanf(fp, "%d %d %d %d %d %d %d\n", &chavesTotais[0], &chavesTotais[1], &chavesTotais[2], &chavesTotais[3], &chavesTotais[4], &chavesTotais[5], &chavesTotais[6]);

		for (int i = 0; i < 7; i++) { //Percorrer ambos os arrays da última chave gerada e das chaves encontradas no ficheiro para comparar número a número
			if (chavesTotais[i] == chave[i]) //Caso existam números iguais, soma 1 à flag
				flag += 1;                                //Nota: Esta comparação é possível porque todas as chaves já estão organizadas por ordem crescente
		}
		if (flag >= 7) { //Caso a flag seja maior ou igual a 7, significa que a última chave gerada tem todos os 7 números iguais a uma das chaves presentes no ficheiro e informamos o main disso ao retornar 1
			fclose(fp);
			printf("\nFound a repeated key!\n");
			return 1;
		}
	}
	//Caso a chave não seja repetida, dá return de 0, a sinalizar o main de que a chave não é repetida e guarda a chave no ficheiro de chaves totais em modo de "append"
	FILE* fpw = fopen("chaves.txt", "a");
	for (int i = 0; i < 6; i++)
	{
		fprintf(fpw, "%d ", chave[i]);
	}

	fprintf(fpw, "%d\n", chave[6]);

	fclose(fpw);
	return 0;
}
```

Código do client:

```
/*
	Simple winsock client
	Código por Vítor Neto 68717, João Leal 68719, Hugo Anes 68571
*/

//Inclusão das bibliotecas necessárias
#include<stdio.h>
#include<winsock2.h>
#include<string.h>
#include <time.h>

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

//Programa principal
int main(int argc, char* argv[])
{
	//Inicialização das variáveis
	WSADATA wsa;					//Variável que armazena as informações do socket
	SOCKET s;					//Socket utilizada para a conexão
	struct sockaddr_in server;			//Estrutura que vai armazenar os dados do servidor ao qual o cliente se vai conectar, como o IP e o porto
	char* message = (char*)malloc(4096);	//Mensagem que vai ser enviada ao servidor
	char server_reply[4096];			//Variável que armazena a resposta do servidor
	int recv_size;					//Tamanho da resposta do servidor, irá ser também verificada para resolver problemas de troca de informações
	int ws_result;					//Resultado da tentativa de conexão com o servidor, se for menor que 0 dá erro
	char ip[20];					//Vetor que irá armazenar o IP ao qual o utilizador se tentará conectar
	int connected = 0;				//Estado da conexão (0 = desconectado, 1 = conectado)
	char* codeString = (char*)malloc(4096);//Vetor de caracteres que irá armazenar a resposta do servidor para futura comparação dos status codes
	int codeInt = 0;				//Variável que armazena os status codes passados pela codeString, para o cliente decidir a ação necessária a tomar

	//Inicializar a socket
	printf("\nInitialising Winsock...");
	if (WSAStartup(MAKEWORD(2, 2), &wsa) != 0) //Caso haja problemas na inicialização da socket, dá erro
	{
		printf("Failed. Error Code : %d", WSAGetLastError());
		return 1;
	}
	printf("Initialised.\n");

	//Criação da socket
	s = socket(AF_INET, SOCK_STREAM, 0);
	if (s == INVALID_SOCKET) //Caso a socket seja inválida, dá erro
	{
		printf("Could not create socket : %d", WSAGetLastError());
	}
	printf("Socket created.\n");
```

```
	//Loop que pede continuamente ao utilizador do cliente que forneça um IP para que este se conecte com um servidor, que só acaba quando a variável connected passa a 1
	//ou seja, quando o cliente encontra um servidor válido com o qual se pode conectar
	while (connected == 0) {

		here:
		printf("Insert server IP to which you would like to connect => ");
		//scanf("%s", &ip); //Scanf não estava a funcionar então usamos o fgets
		fgets(ip, 20, stdin);

		//Criar o endereço da socket, atribuindo o IP e o porto (68000)
		server.sin_addr.s_addr = inet_addr(ip);
		server.sin_family = AF_INET;
		server.sin_port = htons(68000);

		//Conectar ao servidor remoto
		ws_result = connect(s, (struct sockaddr*)&server, sizeof(server));
		if (ws_result < 0) //Se o resultado da conexão for menor que 0, significa que o IP inserido é unreachable, e dá o seguinte erro:
		{
			puts("Connection error");
			goto here; //Ao dar este erro, salta-mos para o ponto do código "here", voltando ao início do ciclo while
		}

		//Receber a resposta do servidor
		recv_size = recv(s, server_reply, 4096, 0);
		if (recv_size == SOCKET_ERROR) //Verificação de erros no recebimento da resposta do servidor
		{
			puts("recv failed");
		}

		//Se o IP fornecido for válido, o servidor irá responder com o status code "100 OK", fazendo com que o estado da conexão, o connected, passe a 1, fazendo
		//com que o ciclo quebre
		if (strcmp(server_reply, "100 OK") == 0) {
			connected = 1;
			printf("Connected\n");
		}
	}

	//Ciclo infinito para que o cliente consiga enviar comandos ao servidor continuamente
	while (1)
	{
		//Limpeza das variáveis que armazenam as mensagens que são enviadas ao e recebidas do servidor
		ZeroMemory(message, 4096);
		ZeroMemory(server_reply, 4096);
		ZeroMemory(codeString, 4096);

		//Informa o utilizador que comandos pode utilizar e aguarda que este os insira
		fputs("\nCommands: CHAVE X => Generates random keys\nQUIT => Terminates the connection\nWrite something =>", stdout);
		fgets(message, 4096, stdin);
```

```
		ws_result = send(s, message, strlen(message) - 1, 0); //Envia a
mensagem ao servidor e verifica foi enviado com sucesso
		if (ws_result < 0)
		{
			puts("Send failed");
			return 1;
		}

		recv_size = recv(s, server_reply, 4096, 0); //Recebe a resposta do
servidor e verifica se foi recebido com sucesso
		if (recv_size == SOCKET_ERROR)
		{
			puts("recv failed");
		}

		//Nestas 3 linhas de código, pegamos na resposta do servidor e
dividimos a resposta em 2 partes.
		//Visto que a resposta do servidor será sempre um status code deste
género: NNN MSSG, dividimos a mensagem em 2 partes, delimitada pelo espaço em
branco.
		//Com isto, ficamos só com o número do código de estado,
convertendo-o posteriormente para inteiro, visto que este estava armazenado num
vetor de carateres.
		codeString = server_reply;
		codeString = strtok(codeString, " ");
		codeInt = atoi(codeString);

		//Nesta sequência de ifs encadeados, verifica-se o status code
recebido pelo servidor, para que o cliente apresente uma resposta ao utilizador
conforme o status code recebido
		//Caso o status code seja 200, significa que o servidor enviou a
resposta com sucesso e irá apresentá-la no ecrã, neste caso a chave
		if (codeInt == 200) {
			recv_size = recv(s, server_reply, 4096, 0);
			server_reply[recv_size] = '\0';
			printf("\nServer => \n%s", server_reply);
			time_t now = time(0); //Apresenta também a data e hora em que
a chave foi recebida
			char* dt = ctime(&now);
			printf("Key received on: %s\n", dt);
		}
		else if (codeInt == 421) //Com o status code 421, o servidor
informa o cliente de que o comando enviado não é reconhecido pelo mesmo
		{

			printf("Server=> Error: Unrecognised command\n");

		}
		else if (codeInt == 404) //O status code 404 ocorre quando o número
de chaves a ser gerado é inválido
		{

			printf("Server=> Error: Can't generate N number of keys\n");

		}
		else if (codeInt == 400) //O código 400 ocorre quando o cliente se
desconecta do servidor
		{

			printf("Server=> Connection terminated\n");
			break;
```

```
            }
        }

        //Fecha a socket do cliente
        closesocket(s);
        //Limpa a socket
        WSACleanup();

        system("pause");
        return 0;
}
```