PROGRAMAÇÃO DISTRIBUÍDA

Ficha de exercícios

REVISÃO – SOCKETS WINDOWS

1. Desenvolva uma aplicação cliente/servidor Win32, baseada no protocolo UDP, que permita obter a hora atual no sistema que alberga a aplicação servidora. Os clientes recebem a localização do servidor através da linha de comando e terminam depois do resultado ser apresentado na saída *standard*. Antes de serem apresentadas, as horas obtidas (pode recorrer à função *GetLocalTime*) devem ser acertadas em função dos respetivos tempos de resposta.
2. Desenvolva uma aplicação cliente/servidor Win32, baseada no protocolo TCP, com funcionalidades semelhantes às da aplicação realizada no âmbito do exercício 1. O servidor é do tipo iterativo.
3. Transforme o servidor desenvolvido no exercício 2 num servidor concorrente.
4. Desenvolva uma aplicação cliente/servidor Win32, baseada no protocolo TCP, que permita obter um ficheiro armazenado no servidor. A aplicação cliente deve ser lançada passando, na linha de comando, a localização do servidor, o nome do ficheiro pretendido e o nome com que este ficará guardado localmente. O servidor deve ser lançado passando, na linha de comando, o porto de escuta. Deve ser possível transferir ficheiros com qualquer dimensão. Para o efeito, defina um tamanho máximo para os blocos transferidos (por exemplo, MAX_DATA = 512 bytes), sendo a conclusão de transferência de um determinado ficheiro assinalada através do encerramento da ligação TCP pelo servidor. Qualquer problema que surja durante a transferência de um ficheiro, incluindo situações de *timeout*, leva a que esta seja abortada.

SOCKETS JAVA

5. Desenvolva uma aplicação cliente/servidor em Java, baseada no protocolo UDP, compatível com a versão Win32 realizada no exercício 1.
6. Desenvolva uma aplicação cliente/servidor em Java, baseada no protocolo UDP, que permita obter um ficheiro armazenado no servidor. A aplicação cliente deve ser lançada passando na linha de comando a localização do servidor, o nome do ficheiro pretendido e a diretoria local onde a cópia obtida deve ser armazenada. O servidor deve ser lançado passando, na linha de comando, o porto de escuta e a diretoria local onde se encontram os ficheiros suscetíveis de serem carregados pelos clientes. O servidor deve apenas permitir o acesso a ficheiros localizados na diretoria indicada ou subdiretorias. Deve ser possível transferir ficheiros com qualquer dimensão. Para o efeito, defina um tamanho máximo para os blocos transferidos (por exemplo, MAX_DATA = 4000 bytes), correspondendo o último bloco de um ficheiro a um datagrama UDP com conteúdo de tamanho igual a zero bytes. O cliente começa por enviar ao servidor um datagrama com conteúdo correspondente ao nome do ficheiro pretendido. Qualquer problema que surja durante a transferência de um ficheiro, incluindo situações de *timeout*, leva a que esta seja abortada.

7. Desenvolva uma aplicação cliente/servidor em Java, baseada no protocolo TCP, compatível com a versão Win32 realizada no exercício 2.
8. Desenvolva uma aplicação cliente/servidor em Java, baseada no protocolo TCP, com funcionalidades semelhantes às da aplicação realizada no âmbito do exercício 6. Neste caso, o servidor assinala a conclusão de transferência de um ficheiro encerrando a respetiva ligação TCP.
9. Altere as aplicações cliente/servidor UDP e TCP desenvolvidas nos exercícios 5, 6, 7 e 8 de modo a que sejam trocados objetos serializados (do tipo *String*) em vez de cadeias de caracteres. Analise o tráfego gerado e compare-o com aquele que é gerado recorrendo às versões originais.
10. Transforme os servidores TCP desenvolvidos em Java, no âmbito dos exercícios 7 e 8, em servidores concorrentes.
11. Desenvolva, na linguagem Java, uma aplicação (*peer-to-peer*) elementar que permita a troca de mensagens no seio de um grupo de utilizadores através da utilização de endereços IP do tipo *multicast* (i.e., pertencentes à classe D). As aplicações são lançadas passando, na linha de comando, o nome do utilizador, o endereço de grupo e o porto de escuta pretendidos. Estas ficam continuamente à espera de mensagens introduzidas na entrada standard, as quais são enviadas para o grupo de conversa, e de mensagens recebidas via rede, as quais são mostradas na saída standard. As mensagens são trocadas sob a forma de cadeias de caracteres. Uma mensagem recebida com a sequência "LIST" faz com que a aplicação reenvie o nome do utilizador local à origem.
12. Altere a aplicação desenvolvida no âmbito da pergunta anterior de modo a que sejam trocadas instância da classe *Msg*, sendo esta constituída pelos atributos *protected String* ***nickname*** e *protected String* ***msg***, bem como pelos métodos *public* ***Msg****(String nickname, String msg)*, *public String* ***getNickname****()* e *public String* ***getMsg****()*. A resposta a uma sequência de caracteres "LIST" é dada através de um objeto serializado to tipo *String*.
13. Desenvolva, na linguagem Java, um servidor TCP concorrente do tipo *proxy*, ou seja, que faça a ponte entre clientes TCP e um servidor TCP específico. A localização deste último é fornecida ao *proxy* através da linha de comando.
14. Desenvolva, na linguagem Java, uma aplicação cliente/servidor TCP que permita usar uma rede de computadores para calcular o valor de Π por integração numérica de um modo paralelo. Na equação seguinte, *n* corresponde ao número de intervalos e x_i à abcissa do centro do i-ésimo intervalo.

$$\Pi = \int_0^1 \frac{4}{1 + x^2}\, dx \cong \frac{1}{n}\sum_{i=1}^{n} \frac{4}{1 + x_i^2}$$

O objetivo é transformar um conjunto (*cluster*) de computadores numa espécie de computador paralelo. A aplicação cliente (i.e., *master*) recebe, através da linha de comando, o nome de um ficheiro de texto onde se encontram definidas as localizações dos servidores (i.e., *workers*), bem como o número de intervalos pretendidos (i.e., a variável *n*). Ao arrancar, o *master* começa por enviar, a cada *worker*, um objeto serializado com os seguintes atributos encapsulados: número de intervalos, número de *workers* e respetivo índice (i.e., se é o 1º *worker*, o 2º *worker*, etc.). Cada *worker* calcula a sua parte do valor de Π, com base nos dados que lhe foram fornecidos, e envia-a ao *master* através da respetiva ligação TCP, sob a forma de uma instância da classe

Double. O *master*, depois de distribuir as tarefas, aguarda pela receção de todos os resultados parciais, mostra-os na saída standard, soma-os e apresenta o resultado final (i.e., o valor aproximado de Π obtido por integração numérica).

15. a) Desenvolva uma variação da aplicação desenvolvida no exercício 14 para que o *master* obtenha a lista de *workers* (endereços IP e portos de escuta TCP) através de uma base de dados MySQL. O endereço do servidor de base de dados, o nome da base de dados e as credenciais de acesso são fornecidos através da linha de comando.

    b) Faça uma variação da aplicação desenvolvida no exercício 14 para que o *master* descubra automaticamente os endereços IP e os portos de escuta dos *workers* presentes na rede local enviando um único *datagrama* UDP para o endereço de difusão 255.255.255.255 e porto 5001.

    c) Faça uma variação da aplicação desenvolvida no exercício 14 de modo a que o *master* comece por entrar num ciclo de aceitação de ligações de *workers* depois de ter enviado um único *datagrama* UDP, que transporta a representação serializada do valor do porto de escuta associado ao seu socket TCP, para o endereço 255.255.255.255 e porto 5001. O porto associado ao *socket* de escuta do servidor deve ser atribuído de forma automática.

JAVA RMI

16. Desenvolva, em Java RMI, uma aplicação cliente/servidor que permita obter a hora atual no sistema que alberga o serviço remoto. Os clientes recebem a localização do RMI *registry* onde se encontra registado o serviço através da linha de comando e terminam depois do resultado ser apresentado na saída *standard*. O serviço remoto e o *registry* onde este se encontra registado sob o nome ***timeserver*** correm na mesma máquina. O resultado é fornecido através de uma instância da classe ***Hora***, sendo esta constituída pelos atributos *static final long* ***serialVersionUID*** *= 1L*, *protected int* ***horas***, *protected int* ***minutos*** e *protected int* ***segundos***, bem como pelos métodos públicos ***Hora****(int hora, int minuto, int segundos)*, *getters* e ***toString***. O serviço remoto implementa a interface remota ***RemoteTimeInterface***, sendo esta constituída apenas pelo método ***Hora getHora()***.

17. Desenvolva uma aplicação cliente/servidor em Java RMI que permita obter um ficheiro armazenado no computador que aloja o serviço remoto. A aplicação cliente deve ser lançada passando, na linha de comando, a localização do RMI *registry* onde se encontra registado o serviço bem como o nome do ficheiro pretendido. O serviço remoto deve ser lançado passando, na linha de comando, a diretoria local onde se encontram os ficheiros suscetíveis de serem carregados pelos clientes. O serviço remoto e o *registry* onde este se encontra registado correm na mesma máquina. Deve ser possível transferir ficheiros com qualquer dimensão. Sendo assim, a interface remota do serviço deve possibilitar a obtenção de blocos em vez de ficheiros completos (e.g., *byte []* ***getFileChunk****(String fileName, long offset)*). Para o efeito, defina um tamanho máximo para os blocos solicitados (por exemplo, MAX_CHUNK_LENGTH = 512 bytes). Qualquer problema que surja durante a obtenção de um ficheiro leva a que esta operação seja cancelada.

18. Desenvolva, em Java RMI, uma versão do exercício 17 que permita aos clientes solicitarem um ficheiro de tamanho qualquer invocando uma única vez um determinado método (e.g., *void* ***getFile****(String fileName, GetRemoteFileClientInterface cliRef) throws java.io.IOException*) no serviço remoto. Para o efeito, deve recorrer a um mecanismo de *callback*. Em concreto, um

cliente, depois de abrir o ficheiro local para escrita, invoca o método *getFile* na interface remota passando-lhe o nome do ficheiro pretendido e a referência para a interface remota que ele próprio inclui. É o método *getFile* do serviço remoto que, para cada bloco do ficheiro pretendido, invoca o método *writeFileChunk* na interface remota do cliente. Este método possui apenas um *array* de *bytes* e um inteiro como argumentos (e.g., *void* ***writeFileChunk****(byte [] fileChunk, int nbytes) throws java.io.IOException*).

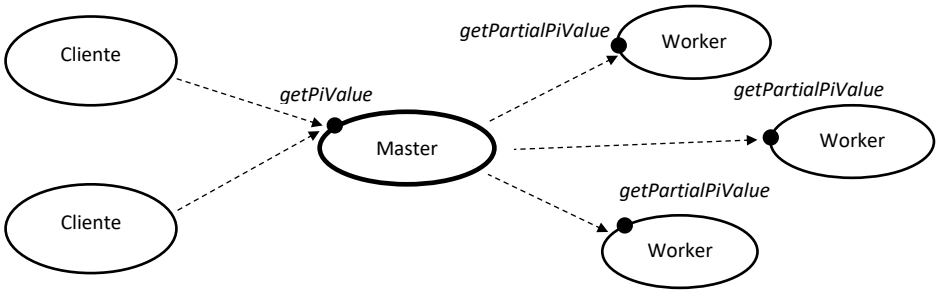
19. Desenvolva, em Java RMI e com recurso a *callbacks*, um observador remoto que é, de um modo assíncrono, notificado das operações realizadas pelo serviço remoto desenvolvido no âmbito do exercício 18. Este deve incluir a interface remota ***GetRemoteFileObserverInterface*** constituída apenas pelo método ***notifyNewOperationConcluded****(String description)*.

20. Desenvolva, em Java RMI, uma aplicação que permita calcular o valor de Π de forma paralela num *cluster* de computadores. A solução deve incluir três tipos de aplicações:

    a. os ***workers***, destinados a participar no cálculo do valor de Π e que incluem um serviço RMI: interface remota ***PartialPiValueInterface*** constituída pelo método único ***getPartialPiValue****(long nIntervals, int nWorkers, int index)*;

    b. o ***master***, que inclui um serviço RMI: interface remota ***PiValueInterface*** constituída pelo método único *double* ***getPiValue****(long nIntervals)*;

    c. os **clientes** que invocam o método *getPiValue* no serviço remoto do ***master*** para obter o valor de Π.



O método *getPiValue* recorre a *threads* adicionais para poder invocar o método *getPartialPiValue* nos vários *workers* em simultâneo (ver sugestões mais abaixo).

Cada *worker* regista o seu serviço RMI no *registry* local sob o nome ***piWorker***. O *master* faz o mesmo recorrendo ao nome ***piFrontEnd***.

O *master* recebe, através da linha de comando, o nome de um ficheiro de texto que inclui as localizações dos RMI *registries* onde se encontram registados os serviços remotos dos *workers*.

Sugestões:

- Depois de processar o ficheiro com os endereços IP/nomes dos *workers*, o *front-end* pode armazenar as referências remotas correspondentes numa lista ou conjunto;
- Como o método *getPartialPiValue* de cada *worker* é bloqueante, o método *getPiValue* pode recorrer a *threads* adicionais para que se obtenha o efeito de concorrência pretendido. A referência remota de um *worker* pode ser passada como argumento no

construtor da respetiva *thread* e o resultado obtido pode ser armazenado num dos membros desta *thread*.

21. Desenvolva, em Java RMI e com recurso a *callbacks*, um observador remoto que é, de um modo assíncrono, notificado dos cálculos efetuados pelo master desenvolvido no âmbito do exercício 20. Este deve incluir a interface remota ***RemotePiObserver*** constituída apenas pelo método ***notifyNewCalculation****(CalculationDetails cd)*. A classe ***CalculationDetails*** inclui os atributos *nIntervals* e *result*, os respetivos *getters* e um construtor que recebe os valores dos dois atributos.
22. Desenvolva uma variante da aplicação distribuída especificada no exercício 20 em que são os próprios *workers* que se registam no *master*.