Network Programming - TCP

SCOMRED, October 2017

Desenvolvimento de aplicações com o protocolo TCP

Numa primeira análise, desenvolver aplicações de rede que utilizam o protocolo TCP devia ser mais simples do que aplicações que utilizam UDP.

Esta análise deriva do facto de o TCP ser *connection-oriented* e fiável, logo as aplicações não têm de se preocupar com a correção de eventuais erros ou falhas nas comunicações.

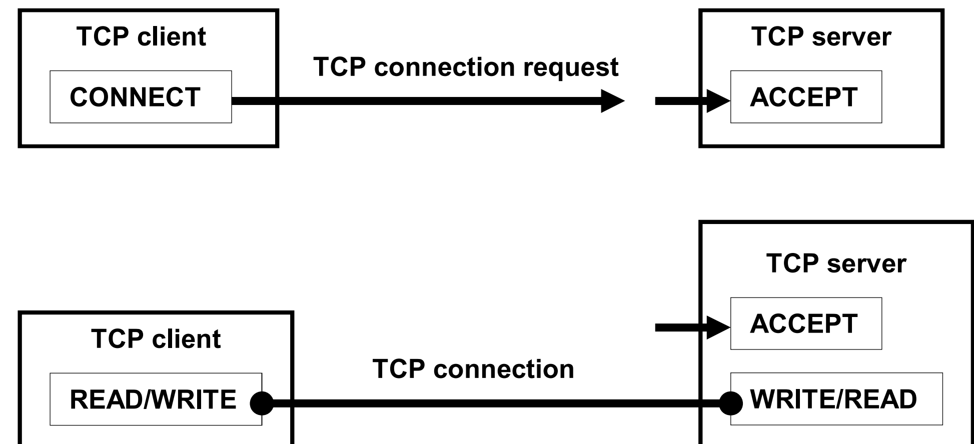
A validade desta análise é indiscutível sob o ponto de vista da implementação de aplicações de rede com um mínimo de qualidade e sofisticação.

Sob o ponto de vista de aplicações de rede muito simples, os benefícios indiscutíveis das ligações TCP entre pares de aplicações trazem consigo alguns desafios.

Algo que desde logo torna o TCP substancialmente diferente do UDP é o facto de ser *connection-oriented*, isto significa que antes de poder haver qualquer transferência de dados é necessário **estabelecer uma conexão TCP entre as duas aplicações envolvidas.**

Desde logo isto significa que na API será necessário usar métodos adicionais.

O estabelecimento de uma conexão TCP usa obrigatoriamente o modelo cliente-servidor. O cliente toma a iniciativa de solicitar ao servidor o estabelecimento da conexão TCP.



Para enviar o pedido de estabelecimento de conexão TCP ao servidor o cliente necessita de conhecer: o endereço IP do nó onde a aplicação servidora está em execução e o número de porto TCP que a aplicação servidora está a utilizar.

Se a aplicação servidora se está a comportar corretamente como servidor que é, deve estar permanentemente a aguardar por contactos de clientes. A aplicação servidora deverá então aceitar (***accept***) o pedido de estabelecimento de ligação TCP que lhe chegou do cliente.

Aceite o pedido de ligação pelo servidor, passamos a ter uma nova ligação TCP entre as duas aplicações envolvidas.

A ligação TCP tem as seguintes caraterísticas:

- É um canal bidirecional de transferência de bytes, de utilização exclusiva pelas duas aplicações envolvidas.
- Tem duas extremidades, uma em cada aplicação. Quando um byte é escrito numa extremidade, fica disponível para leitura na outra extremidade.
- A ordem de escrita dos bytes numa extremidade é exatamente a mesma que se encontra na leitura na outra extremidade.
- Na medida do possível é isenta de erros. O TCP corrige todos os erros pontuais detetados através da retransmissão dos dados. Se houver uma falha de rede prolongada, não havendo possibilidade de manter a conexão TCP, ela será terminada.

Apesar de todos os benefícios de uma ligação TCP, dela também resultam algumas dificuldades ao nível da implementação das aplicações de rede:

- **Receção assíncrona.** Este é um problema que afeta apenas o servidor TCP. À medida que vai aceitando pedidos de ligações de novos clientes vai tendo em mãos um número crescente de ligações TCP (uma para cada cliente). O problema é que, de entre as várias ligações, ele não sabe onde irão surgir dados em primeiro lugar. Este é um problema de receção assíncrona, o servidor não pode ficar parado à espera de dados numa das ligações porque os dados podem surgir noutra ligação. No UDP este problema não se coloca porque não existem ligações.

- **Sincronização de byte.** A um determinado número de bytes cuja leitura se solicita numa extremidade tem de corresponder uma operação de escrita do mesmo número de bytes na outra extremidade. Quando se solicita a leitura de um determinado número de bytes a operação vai bloquear a aplicação/thread até que esse número de bytes esteja disponível (tenha sido escrito na outra extremidade). No UDP este problema não se coloca porque as operações são orientadas por pacotes e não bytes.

Receção assíncrona

Este é um problema clássico em que uma aplicação tem várias fontes distintas de leitura de dados e não é capaz de pré-determinar em qual das fontes os dados vão surgir primeiro.

No caso do servidor TCP as fontes de dados são: novos clientes (pedidos de estabelecimento de ligação) e um conjunto de várias ligações já estabelecidas com clientes.

Uma solução desaconselhada é ***polling***, para ser implementada primeiro será necessário definir um *timeout* reduzido para as operações de leitura de cada uma das várias fontes de dados. O *polling* consiste em percorrer ciclicamente a lista de fontes de dados e tentar efetuar uma leitura em cada uma delas. Graças ao *timeout*, não havendo dados para ler a operação não bloqueia, gera um erro, a aplicação pode então passar à fonte de dados seguinte. Esta solução consome desnecessariamente recursos de CPU.

A solução mais aconselhada é usar ***threads***, um *thread* é uma unidade de execução sequencial independente. A solução consiste em criar um *thread* para cada fonte de dados e manter esse *thread* parado na operação de leitura à espera dos dados.

Sincronização de byte

Esta é uma questão com que ambos os lados da ligação TCP têm de lidar de forma coordenada. É necessário que exista sempre uma correspondência exata entre o número de bytes escritos numa extremidade e o número de bytes lidos na outra extremidade.

Se tentarmos ler mais bytes do que foram escritos, a leitura bloqueia à espera de bytes restantes. Se lermos menos bytes do que foram escritos ficarão bytes não lidos que surgirão na operação de leitura seguinte.

A forma como é realizada a coordenação entre leituras e escritas deve estar definida no protocolo de aplicação que rege o diálogo entre as aplicações.

Existem três abordagens possíveis para este problema:

1 – **Mensagens de tamanho fixo** – as mensagens possuem um tamanho fixo estabelecido no protocolo de aplicação. Todas as operações de escrita e leitura correspondem sempre a esse número de bytes fixo. Esta é uma solução pouco flexível que leva tipicamente a situações em que a informação útil em cada mensagem seja apenas uma fração do tamanho total.

Sincronização de byte

2 – **Indicador de fim de mensagem** – as mensagens possuem um tamanho variável, um byte (ou vários) com valor especial é usado para assinalar o fim da mensagem. Esta abordagem tem dois problemas: a aplicação que está a ler a mensagem só pode ler um byte de cada vez; o dito byte com valor especial não pode ocorrer na mensagem. Esta técnica pode ser usada facilmente para conteúdos limitados como texto. É usada no protocolo HTTP na transmissão do cabeçalho da mensagem que se encontra em formato de texto.

3 – **Pré indicação do tamanho da mensagem** – quando uma aplicação pretende escrever uma mensagem começa por anunciar ao parceiro o número de bytes que a mensagem contém, dessa forma o leitor fica desde logo a saber o tamanho da mensagem e sabe exatamente quantos byte deve ler depois do indicador do tamanho da mensagem. Esta técnica é também usada no protocolo HTTP quando é enviado o atributo ***Content-length*** que indica o número de bytes que constituem o corpo da mensagem que se segue.

Classe Socket

Em Java um cliente TCP pode solicitar o estabelecimento de uma ligação TCP através da instanciação da classe Socket. O construtor recebe como argumentos o endereço IP do nó de rede em que se encontra a aplicação servidora TCP e o número de porto TCP que a aplicação servidora está a utilizar.

public Socket(InetAddress address, int port) throws IOException

O construtor gera uma exceção se o estabelecimento da ligação TCP falha.

Para ter sucesso é necessário que a aplicação servidora esteja no local indicado (IP e número de porto) e aceite o pedido de ligação.

Em caso de sucesso o *socket* criado fica ligado à aplicação servidora através de uma conexão TCP, constituindo uma das extremidades dessa ligação.

Classe ServerSocket

A aplicação servidora TCP deve criar um objeto da classe ServerSocket. O construtor recebe como argumento o número de porto onde a aplicação vai receber pedidos de ligação TCP dos clientes:

public ServerSocket(int port) throws IOException

O construtor gera uma exceção em caso de erro, por exemplo se o número de porto pretendido estiver já a ser usado por outra aplicação.

Depois de criado o ServerSocket, pedidos de estabelecimento de ligação de clientes ficam em estado pendente até serem explicitamente aceites pela aplicação servidora.

Método accept() da classe ServerSocket

Depois de criado o **ServerSocket**, a aplicação tem de aceitar explicitamente o pedido de ligação do cliente. Isso é conseguido através do método **accept()**.

public Socket accept()   throws IOException

Este método aceita o pedido de ligação seguinte na fila de espera de pedidos pendentes, se não existir nenhum, bloqueia até que surja um pedido.

Em caso de sucesso, a ligação TCP com o cliente fica estabelecida, nesse caso o método accept() **devolve um novo objeto da classe Socket** que corresponde à extremidade do lado do servidor da ligação TCP.

A aplicação servidora terá de lidar com múltiplos *sockets*, desde logo o **ServerSocket** onde recebe pedidos de ligação de novos clientes e também tem de lidar com os vários **Socket** que vão sendo devolvidos pelo método accept() e estão ligados a cada um dos clientes.

Ler e escrever bytes através de uma ligação TCP

Quando o servidor invoca com sucesso o método accept() fica estabelecida a ligação TCP. Em cada extremidade desta ligação encontra-se um objeto da classe Socket.

As operações de leitura e escrita sobre a conexão TCP são realizadas através, respetivamente, do **InputStream** e **OutputStream** dos sockets, podem ser obtidos através dos métodos seguintes:

```
public OutputStream getOutputStream()     throws IOException
public InputStream getInputStream()       throws IOException
```

Os objetos devolvidos podem se usados para criar objetos de classes Stream apropriadas, para manusear bytes diretamente as classes indicadas são DataOutputStream e DataInputStream. Por exemplo, se sock é um objeto da classe Socket, correspondendo a uma ligação TCP estabelecida podem ser criados através de:

```
DataOutputStream sOut = new DataOutputStream(sock.getOutputStream());
DataInputStream sIn = new DataInputStream(sock.getInputStream());
```

Os objetos da classe **InputStream** definem entre outros os seguintes métodos de leitura de bytes:

int read() throws IOException

int read(byte[] b, int off, int len) throws IOException

O primeiro lê apenas um byte e devolve o seu valor sob a forma de um inteiro. O segunde lê uma sequência de **len** bytes e coloca-os no vetor **b** a partir da posição **off**.

Os objetos da classe **OutputStream** definem entre outros os seguintes métodos de escrita de bytes:

void write(int b) throws IOException

void write(byte[] b, int off, int len) throws IOException

O primeiro escreve apenas um byte cujo valor é fornecido como argumento sob a forma de um inteiro. O segundo escreve uma sequência de **len** bytes existentes no vetor **b** a partir da posição **off**.

Threads em Java

Um servidor TCP é uma aplicação que tem de lidar com eventos de receção assíncrona, tendo um **ServerSocket** e vários **Socket** ligados a clientes não consegue pré determinar em qual deve fazer primeiro a leitura de dados.

Em Java a solução recomendada é usar *threads* de tal forma que para cada socket exista um *thread* a aguardar a chegada de dados. Cada *thread* fica bloqueado, mas como são entidades de execução sequencial independentes, não se bloqueiam uns aos outros.

Em Java um *thread* é implementado através de uma classe que implementa a interface **Runnable**. A nova classe deve definir o método **run()**. É o método run() que será invocado para colocar o thread em execução quando for chamado o método start() da classe Thread. A classe pode ser definida de uma de duas formas:

- Definindo uma subclasse da classe **Thread** (*extends the Thread class*). A classe Thread implementa a interface Runnable. Neste caso, como a classe Thread já define o método run(), terá de ser declarado que o queremos substituir (anotação **@Override**).

- Definindo uma classe e declarando que ela implementa a interface Runnable.

Exemplo de uma classe para implementar um thread de um servidor TCP, neste exemplo optou-se por declarar que a classe implementa a interface Runnable:

```
public class AttendClient  implements Runnable {
        private Socket cliSock;
        public AttendClient(Socket s) { cliSock=s;}
        public void run() {                  // thread execution starts
                // get cliSock input and output streams
                // read requests and write replies
                cliSock.close();
        }                                    // thread execution ends
}
```

O construtor AttendClient() é usado apenas para guardar o Socket quando o objeto é criado. O método run() é usado para lançar a execução do thread em paralelo.

Se tivesse sido declarada como subclasse de Thread (extends Thread), a declaração do método run() teria de precedida da anotação **@Override** porque ele já está definido na classe Thread.

A classe contendo a aplicação servidora que lança um thread para cada cliente que se liga poderá ser algo como:

```
public class TcpServer {
    public static void main(String args[]) {
        static ServerSocket sock = new ServerSocket(9999);
        static Socket nSock;
        while(true) {
            nSock=sock.accept();            // wait for a new connection
            Thread cliConn = new Thread(new AttendClient(nSock));
            cliConn.start();       // start running the thread in background
        }
    }
}
```

Quando um novo cliente se liga ao servidor (accept), é definido um novo Thread através da instanciação da classe AttendClient, o socket ligado ao cliente é passado ao thread através do construtor. O novo thread inicia a sua execução em paralelo quando é invocado o método start() da classe Thread que provoca a execução do método run() da classe AttendClient.

Concorrência e *Locking*

Quando se usam *threads*, o *thread* inicial e os *threads* criados têm acesso aos mesmos objetos e métodos. Isto levanta problemas ao nível de acesso concorrencial, como todos os *threads* estão em execução ao mesmo tempo (em paralelo) se vários acedem aos mesmos objetos e em especial se alteram o seu conteúdo, os resultados são imprevisíveis e podem mesmo levar ao bloqueio dos *threads* (*deadlock*),

As linguagens de programação proporcionam mecanismos para evitar estas situações, um deles é o *lock* ou *mutex*.

Um *lock* pode estar livre (***released***) ou capturado por um *thread* (***acquired***), é implementado de tal forma que só pode ser capturado se estiver livre. Se dois ou mais *threads* tentam capturar o mesmo *lock* simultaneamente, apenas um terá sucesso imediato, os restantes ficam bloqueados à espera que o *thread* que conseguiu capturar o *lock* o liberte.

É importante entender que o *lock* por si não condiciona o acesso a objetos, apenas surte o efeito desejado se todos os *thread* fizerem a captura do *lock* antes de acederem ao objeto pretendido.

Declaração synchronized

Em Java cada objeto e cada classe possui um ***intrinsic lock*** associado, o *intrinsic lock* pode ser utilizado através da declaração *synchronized*. A forma mais simples de utilização é ao definir métodos.

Se um método não estático é declarado *synchronized*, quando é invocado será realizada a captura do *intrinsic lock* associado ao **objeto** (instância da classe), quando o método termina o *intrinsic lock* será libertado. Consequentemente, se na classe vários métodos não estáticos são declarados *synchronized*, numa instancia, em cada instante, apenas um *thread* poderá estar a executar um deles (exclusão mútua).

Se um método estático é declarado *synchronized*, quando é invocado será realizada a captura do *intrinsic lock* associado à **classe** a que pertence. Se numa classe vários métodos estáticos são *synchronized*, em cada instante, apenas um *thread* poderá estar a executar um deles (exclusão mútua).

Ao declarar métodos *synchronized*, quem estabelece a necessidade de usar *locking* é a própria classe e não quem que invoca os métodos da classe.

Declaração synchronized - exemplo

No exemplo seguinte é declarada uma classe totalmente estática em que todos os métodos são *synchronized*, isso torna impossível mais do que um *thread* executar simultaneamente qualquer destes métodos.

```
public class Contador {
    private static Integer valor;
    public static synchronized int get() { return valor; }
    public static synchronized void inc() { valor++; }
    public static synchronized void dec() { valor--; }
    public static synchronized void set(int v) { valor=v; }
    public static synchronized void reset() { valor=0; }
}
```

Em métodos mais complexos poderá não fazer muito sentido manter o objeto ou classe totalmente bloqueados durante toda a execução do método. Isso pode resultar de dois fatores: porque apenas algumas parte do código do método manipulam dados para os quais há necessidade de garantir a exclusão mútua; porque alguns métodos manipulam alguns objetos e outros métodos manipulam objetos diferentes.

Para uma definição mais detalhada do *locking* podemos declarar um **bloco de código *synchronized* com um dado objeto.** O efeito é que antes de se iniciar a execução desse bloco o *intrinsic lock* associado ao objeto referido será capturado, terminada a execução do bloco será libertado, se o método terminar (*return*) a meio do bloco, o *lock* é igualmente libertado.

Se desejável, num método estático, podemos fazer o *lock* da classe através de **NOME.class**, onde NOME é o nome da classe.

Exemplo:

```
public class Contador {
    private static Integer valor;
    public static synchronized void inc() { valor++; }
    public static synchronized void dec() { valor--; }
    public static void readVal() throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Indique o novo valor: ");
        int v = Integer.parseInt(in.readLine());
         synchronized(Counter.class) { valor=v; }
    } }
```

Blocos synchronized e métodos synchronized

Num método estático declarar o método *synchronized* é o mesmo que definir todo o código do método dentro de um bloco *synchronized* com a classe (**NOME.class**).

Num método não estático podemos referir a instância atual da classe (objeto) através do nome **this.** Assim, num método não estático declarar o método *synchronized* é o mesmo que definir todo o código do método dentro de um bloco *synchronized* com **this.**

A declaração de blocos *synchronized* com a classe (**NOME.class**) num método estático ou com o objeto (**this**) num método não estático funciona em conjunto com as declarações *synchronized* dos métodos porque os *intrinsic lock* usados são os mesmos.

Métodos estáticos e não estáticos

Um método estático apenas tem acesso direto a outros métodos estáticos e objetos estáticos da classe (métodos não estáticos e objetos não estáticos existem apenas em instâncias da classe).

Um métodos não estático tem acesso direto a todos os métodos e objetos, estáticos ou não estáticos definidos na classe. Quando usa métodos e objetos não estáticos eles pertencem ao objeto (instância da classe), quando usa métodos e objetos estáticos eles pertencem à classe.

O problema de que devemos estar conscientes é o seguinte: um método *synchronized* não estático faz o *lock* do objeto e não da classe. Consequentemente não serve para garantir a exclusividade de acesso à classe relativamente a métodos *synchronized* estáticos.

Exemplo de utilização incorreta

```
public class Contador {
    private static Integer valor;
    public static synchronized int get() { return valor; }
    public synchronized void inc() { valor++; }
    public synchronized void dec() { valor--; }
    public static synchronized void set(int v) { valor=v; }
    public static synchronized void reset() { valor=0; }
}
```

Neste exemplo, por alguma razão, os métodos inc() e dec() não são estáticos, isso significa que não existem na classe, apenas nas instâncias da classe.

O problema é que o facto de eles serem declarados *synchronized* vai levar ao *lock* da instância e não da classe, ao contrário do que acontece com os outros métodos *synchronized* estáticos.

Consequentemente a classe está mal implementada sob o ponto de vista de *locking* porque não garante a exclusividade de acesso ao objeto estático **valor**.

Quando métodos estáticos e não estáticos acedem a métodos, e objetos estáticos da classe, declarar todos os métodos *synchronized* não é suficiente.

Uma solução possível é manter os métodos estáticos *synchronized* e nos métodos não estáticos fazer o *lock* da classe, aplicado ao exemplo anterior seria:

```
public class Contador {
    private static Integer valor;
    public static synchronized int get() { return valor; }
    public void inc() {synchronized(Contador.class) {valor++;} }
    public void dec() {synchronized(Contador.class) {valor--;} }
    public static synchronized void set(int v) { valor=v; }
    public static synchronized void reset() { valor=0; }
}
```

Uma outra solução seria não declarar nenhum método *synchronized* e em todos eles executar as operações sobre o objeto **valor** dentro de um bloco *synchronized* com o objeto **valor**.

Locking em aplicações de rede

Em termos gerais, a exclusão mútua através de *locking* tem de ser usada sempre que temos vários *threads* a aceder aos mesmos objetos.

As aplicações de rede não são exceção, em particular no acesso a sockets ou streams de sockets TCP (conexões TCP).

Se dois ou mais *threads* escrevem (write) simultaneamente sobre um mesmo OutputStream, ou se dois *threads* leem (read) simultaneamente de um mesmo InputStream, os resultados serão totalmente imprevisíveis pois poderão ser uma mistura de dados.

O nível a que se implementa o *locking* para evitar esta situação pode ser diverso. Por exemplo em operações de escrita sobre um OutputStream:

Se existir um único método a executar operações de escrita podemos declarar o mesmo *synchronized*, mas isso vai bloquear todos os métodos *synchronized* da classe ou objeto.

Podemos evitar um bloqueio tão generalizado realizando todas as operações de escrita dentro de blocos *synchronized* com o OutputStream sobre o qual se vai escrever.

Projetos – TcpChatSrv e TcpChatCli

Pretende-se o desenvolvimento de duas aplicações de rede que deverão comunicar entre si através do **protocolo TCP** segundo o modelo **cliente-servidor**.

O objetivo geral destas aplicações é a conversação livre entre um grupo de utilizadores através de linhas de texto.

Cada utilizador executa localmente no seu posto de trabalho a aplicação **TcpChatCli** e solicita uma ligação à aplicação **TcpChatSrv** pretendida através do fornecimento do correspondente endereço IP ou nome DNS.

A conversação ocorre apenas entre os vários utilizadores que estão ligados à mesma aplicação TcpChatSrv.

Protocolo de aplicação – números de porto TCP

Por omissão, a aplicação **TcpChatSrv** aceita pedidos de ligação TCP de clientes no porto 8099. Deve no entanto ser possível ao administrador colocar a aplicação em funcionamento em outro número de porto.

Por omissão, a aplicação **TcpChatCli** solicita o estabelecimento de uma ligação TCP com o número de porto 8099 do nó de rede indicado pelo utilizador através de um endereço IP ou nome DNS. Deve no entanto ser possível ao utilizador indicar um número de porto diferente.

Protocolo de aplicação – mensagens

Todas as mensagens trocadas entre as aplicações possuem o seguinte formato, doravante referido como mensagem:

CONTENT-LEN: é **um byte** que deve ser interpretado como um valor inteiro sem sinal (*unsigned*). O CONTENT-LEN é o primeiro elemento enviado, quando uma aplicação pretende enviar uma mensagem escreve primeiro este byte, quando um aplicação pretende receber uma mensagem lê primeiro este byte. O valor do CONTENT-LEN indica por quantos bytes é constituído o conteúdo da mensagem (CONTENT). O valor zero no CONTENT-LEN indica uma mensagem vazia e deve ser interpretado como uma solicitação ou confirmação de fim de ligação.

CONTENT: é uma sequência de 0 a 255 bytes, o conteúdo pode ser de qualquer tipo, mas normalmente será interpretado como carateres ASCII (texto).

Protocolo de aplicação – diálogo de fecho de ligação

Qualquer diálogo pode ser interrompido através do diálogo de fecho de ligação.

O diálogo de fecho de ligação é sempre iniciado pela aplicação **TCPChatCli** de acordo com uma solicitação do utilizador.

Este diálogo é iniciado através do envio de uma mensagem vazia (solicitação de fecho da ligação).

Quando a aplicação **TCPChatSrv** a recebe deve remover esse cliente da sua lista clientes (se estiver na lista), devolver à aplicação **TCPChatCli** uma mensagem também vazia (confirmação de fecho da ligação) e de seguida termina a ligação TCP correspondente.

Após a aplicação **TCPChatCli** ter solicitado o fecho da ligação deve aguardar a confirmação do **TCPChatSrv**, após o que deve também terminar a ligação TCP.

Protocolo de aplicação – diálogo de nickname

Os diálogos iniciam-se após o estabelecimento bem sucedido da ligação TCP entre a aplicação **TCPChatCli** e a aplicação **TcpChatSrv** (solicitada pela primeira).

O primeiro diálogo é obrigatório e destina-se a definir um **nickname** único para o utilizador. O nickname é um texto arbitrário solicitado ao utilizador da aplicação **TCPChatCli.**

Após o estabelecimento da ligação TCP a aplicação **TCPChatCli** envia uma mensagem com o **nickname.** O **TcpChatSrv** verifica se o **nickname** está livre, em caso afirmativo responde com uma mensagem com conteúdo **OK** (em maiúsculas), adiciona o cliente à lista de clientes e passa ao **diálogo de chat.**

Se o nickname já está a ser usado, a aplicação **TcpChatSrv** responde com uma mensagem com conteúdo **KO** (em maiúsculas) e mantem-se no diálogo de nickname aguardando um novo nickname da aplicação cliente.

Se o utilizador pretender desistir de definir um nickname e terminar a ligação, a aplicação **TCPChatCli** deve passar para o diálogo de fecho de ligação.

Protocolo de aplicação - diálogo de chat

O diálogo de chat só se inicia após a conclusão bem sucedida do diálogo de nickname, estando o cliente já registado na lista de clientes gerida pelo **TcpChatSrv**.

Ao contrário dos anteriores, este diálogo é assíncrono, em qualquer momento o **TcpChatSrv** pode receber uma mensagem da aplicação **TcpChatCli** e em qualquer momento a aplicação **TcpChatCli** pode receber uma mensagem para aplicação **TcpChatSrv**.

Nesta fase o **TcpChatCli** recebe do utilizador mensagens de texto livre a enviar ao **TcpChatSrv**.

Quando o **TcpChatSrv** recebe uma mensagem durante este diálogo verifica se é um comando (mensagem começada por um ponto), nesta fase não vamos implementar comandos e consequentemente os comandos serão ignorados.

Não sendo um comando, a missão do **TcpChatSrv** é fazer chegar a mensagem a todos os clientes (incluindo o de origem da mensagem).

Protocolo de aplicação – diálogo de chat

Neste diálogo, quando o **TcpChatSrv** recebe uma mensagem de uma aplicação **TcpChatCli** deve envia-la a todos os clientes (na lista de clientes, incluindo o de origem da mensagem), antes de o fazer insere no início da mensagem recebida o nickname correspondente ao cliente de origem da mensagem, na forma:

[nickname] MENSAGEM-RECEBIDA

Durante o diálogo de chat, em qualquer momento a aplicação **TcpChatCli** pode receber uma mensagem do **TcpChatSrv.** Essa mensagem deve ser apresentada ao utilizador.

Durante o diálogo de chat, a aplicação **TcpChatCli** pode iniciar o **diálogo de fecho de ligação** caso o utilizador pretenda sair da sua aplicação **TcpChatCli.**

Comentários gerais sobre a implementação

Receção assíncrona - threads

TcpChatSrv – pedidos de ligação TCP de novos clientes e mensagens enviadas por clientes constituem os eventos de receção assíncrona com que a aplicação terá de lidar e como é típico de qualquer servidor TCP.

TcpChatCli – na fase de diálogo de chat existem dois eventos de receção assíncrona: leitura de mensagens enviadas pelo TcpChatSrv através da ligação TCP; leitura de mensagens digitadas localmente pelo utilizador para serem enviadas ao TcpChatSrv.

Desta forma, no TcpChatSrv terão de existir pelo menos: um thread disponível para aceitar ligações de novos clientes (accept) e, para cada ligação aceite, um thread a ler mensagens enviadas pelo cliente.

No TcpChatCli terão de existir pelo menos dois threads: um thread a ler mensagens vindas do servidor e outro thread a ler mensagens digitadas pelo utilizador.

Estruturas de dados

TcpChatCli – embora necessite de dois threads, a aplicação cliente pode ser bastante simples e não necessita de gerir qualquer quantidade significativa de dados.

TcpChatSrv – por seu lado o servidor terá de gerir uma lista de clientes, cada cliente terá determinadas propriedades, como por exemplo o nickname. Será de ponderar neste caso a definição de uma classe para representar e gerir um objeto do tipo cliente e uma outra classe que usa a anterior para representar e gerir a lista de clientes.

Locking

TcpChatCli – embora existam dois threads, não são de prever aqui problemas de concorrência porque cada thread utiliza recursos diferentes, um lê o texto digitado e escreve-o no OutputStream, o outro lê mensagens do InputStream e apresenta-as ao utilizador.

TcpChatSrv – existem pelo menos dois aspetos de concorrência a considerar:

- Escrita nos OutputStream – para cada cliente existe um OutputStream, se forem recebidas mensagens simultâneas de vários clientes temos de evitar que essas mensagens sejam escritas ao mesmo tempo no mesmo OutputStream de um dado cliente.

- Gestão da lista de clientes – as operações de acesso a clientes na lista, alteração de clientes na lista, remoção ou adição terão de ser realizadas em exclusão mútua.

Java - Objetos

Uma vantagem da programação orientada a objetos (OOP) é que os objetos são adequados para representar entidades do mundo real (da especificação geral do problema).

Um objeto contém dados e métodos para interagir com esses dados, os dados de um objeto **devem ser privados** (apenas acessíveis aos métodos do próprio objeto).

Alguns dados de um objeto serão acessíveis externamente a outros objetos, de forma indireta, através da invocação de **métodos públicos** definidos pelo objeto.

O conjunto dos métodos públicos disponibilizados por um objeto pode ser designado a interface do objeto.

Note-se que quando nos referimos a dados de um objeto (muitas vezes designados **campos**/**fields**, **atributos**, ou mesmo **variáveis**), eles próprios são muitas vezes objetos, mas podem ser também tipos primitivos como int, boolean, byte, float, etc.

Java - Classes e construtores

Um tipo de objeto é definido através de uma declaração de classe.

Uma vez declarada uma classe podem ser criados vários objetos (instanciação) dessa classe (tipo).

No Java a instanciação de um objeto realiza-se através da declaração **new** aplicada ao método construtor da classe.

O método construtor da classe **tem o mesmo nome da classe** e está implicitamente definido, no entanto também podemos definir um, deve ser público, não estático e não devolver qualquer valor (implicitamente devolve uma instancia da classe).

Tipicamente o construtor será utilizado para realizar operações de inicialização sobre o objeto assim que é criado.

Java - elementos estáticos da classe

Os vários objetos criados a partir de uma classe (instâncias ou exemplares) são totalmente autónomos. Cada um tem os seus dados e métodos totalmente independentes dos restantes objetos criados por instanciação da mesma classe.

Ao declarar uma classe podemos declarar alguns dos seus dados e métodos como sendo estáticos (**static**), por oposição aos restantes que serão não estáticos por omissão. **Os elementos estáticos apenas existem na classe, não nos objetos criados a partir da classe (instâncias).**

Os elementos estáticos definidos na classe constituem um **objeto especial, único e global, referenciado pelo nome da classe.** Neste objeto único e global apenas existem os elementos estático da classe.

Java - elementos estáticos da classe

Na classe (objeto único e global referenciado pelo nome da classe) não existem elementos não estáticos, logo em métodos estáticos apenas podem ser usados dados estáticos da classe e métodos estáticos da classe.

Numa instância da classe (objeto) apenas existem dados e métodos não estáticos, mas nos métodos não estáticos de um objeto podemos usar dados e métodos estáticos da classe, isso significa que estamos a interagir diretamente com o mencionado objeto único e global referenciado pelo nome da classe.

Este aspeto foi anteriormente referido a propósito da declaração de métodos *synchronized* estáticos e não estáticos uma vez que os primeiros fazem o *lock* à classe e os segundos fazem *lock* ao objeto.