UNIVERSIDADE DO MINHO
LEI 3º ANO

Gestor de requisições

Trabalho Prático de Sistemas Distribuídos

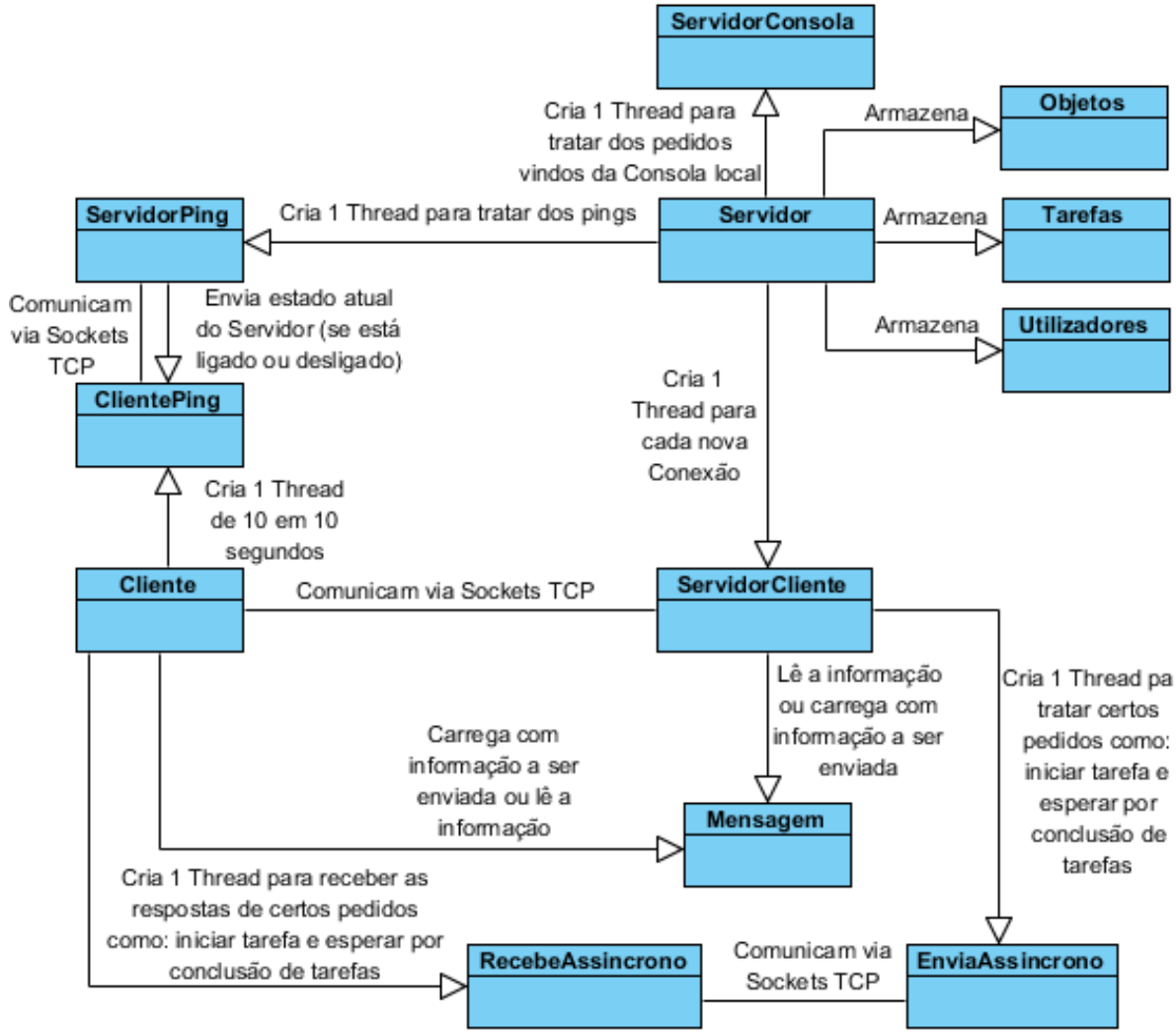
04/01/2015

Carlos Silva – A66553 Daniel Coelho – A53979
João Nicolau – A64358 Tiago Santos – A64381

Introdução

Foi proposto aos alunos de Sistemas Distribuídos a realização de um servidor ***multi-threaded*** que permitisse receber conexões de vários clientes através de ***sockets TCP***. Esse servidor deveria ser um Gestor de requisições / tarefas que deveria suportar várias funcionalidades, entre as quais deveria: permitir o abastecimento do armazém com objetos, definir tarefas com uma lista de objetos necessários, iniciar e concluir tarefas e ainda esperar por conclusão de tarefas. Deveria ainda ter mecanismos de Exclusão Mútua aprendidos nas aulas e tudo deveria ser escrito em Java.

Modelo do servidor implementado



O grupo realizou o servidor da seguinte forma: o **Cliente** liga-se ao **Servidor**, este cria uma ***Thread*** para atender o novo **Cliente** e volta a esperar por novas conexões; após a ligação estabelecida, o **Cliente** fica à espera de pedidos vindos da consola local. Cada pedido é carregado num **Objeto** da classe ou subclasse de **Mensagem** com a respetiva informação e enviado ao **ServidorCliente** pelo ***Socket TCP***.

Cliente

Um Cliente possui as seguintes informações em variáveis:

- ***Socket*** socket – possibilita a comunicação entre o **ServidorCliente** e o **Cliente**. Criado na porta 12345.
- ***ObjectInputStream*** inputServidorOb – criado a partir de um ***InputStream*** do ***Socket***. Possibilita a transmissão **ServidorCliente** -> **Cliente** de objetos da classe **Mensagem**.
- ***ObjectOutputStream*** outputServidorOb – criado a partir de um ***OutputStream*** do ***Socket***. Possibilita a transmissão **Cliente** -> **ServidorCliente** de objetos da classe **Mensagem**.
- ***BufferedReader*** inputServidor – permite ler ***strings*** que o ***ServidorCliente*** envia.
- ***BufferedReader*** inputConsola – permite ler o ***input*** do utilizador na consola local.
- ***String*** utilizador – nome do **Utilizador** após efetuar *login*. Apenas serve para indicar no menu principal o nome de **Utilizador** após o *login*.
- ***Boolean*** ServidorLigado – estado do **Servidor** (*true* = ligado | *false* = desligado). É a ***Thread*** de **ClientePing** que muda estes valores.

O **Cliente** é composto por 2 **Threads**, estando uma a atender o utilizador e enviar os respetivos pedidos ao **Servidor** e outra num ciclo que, enquanto o **Servidor** não se desligar ou o **Cliente** não fechar a conexão cria, de 10 em 10 segundos, 1 ***Thread*** **ClientePing** que se liga ao **ServidorPing** e espera que este envie o estado do **Servidor** (***true*** se estiver a trabalhar ou ***false*** se for desligado) e caso seja ***false*** o **ClientePing** mudará a variável **ServidorLigado** do **Cliente** para ***false.*** O **Cliente** ficará ciente do estado do Servidor no momento em que estiver à espera que o utilizador digite algo na consola, pois em vez de se fazer apenas **inputConsola.readLine()**, decidiu-se utilizar o método ***ready***() que indica se algo foi escrito na consola e enquanto nada for escrito o **Cliente** verifica a variável ***ServidorLigado*** e caso seja ***true*** adormece 200 milissegundos, voltando depois a repetir o ciclo; caso seja ***false*** o Cliente se desligará sozinho avisando o utilizador que o **Servidor** foi desligado.

Quando um **Cliente** é executado, é apresentado o menu de *login*. Após efetuar o login é apresentado o menu principal.

```
CLIENTE
Utilizador : 'xpto'
1 -> Abastecer armazém
2 -> Definir tarefa
3 -> Iniciar tarefa (requisitar objetos)
4 -> Finalizar tarefa (devolver objetos)
5 -> Espera por conclusão de tarefas
6 -> Listagem de tipos de tarefas / tarefas em curso
9 -> Logout
Q -> Fechar conexão
```

Neste menu o utilizador tem ao seu dispor as funcionalidades pedidas no enunciado do projeto, tendo cada opção pedidos de informações correspondentes. Por exemplo, a opção “5 -> Espera por conclusão de tarefas” pede os nomes das tarefas que se espera concluir, enquanto a opção “4 -> Finalizar tarefa (devolver objetos) ” apenas pede o número de identificação da tarefa.

Vamos analisar o código do método **finalizarTarefa()**. Responsável por enviar ao **Servidor** o pedido de conclusão de tarefa, a título de exemplo, uma vez que as funcionalidades pedidas foram implementadas de forma semelhante.

```
	private void finalizarTarefa () throws ServidorDesligado, IOException {
		int id = lerNumeroConsola ("Especifique o identificador da tarefa que
finalizou", "");
		Mensagem m = new MensagemId (6, id);
		enviarMensagem (m);
		String r = receberString ();
		switch (r) {
		case "1"://Sucesso
			System.out.println ("Finalizou a tarefa com id = '" + id +
"'");
			break;

		case "-1"://Tarefa não existe
			System.err.println("Não há nenhuma tarefa iniciada com esse
Identificador!");
			break;

		case "-2"://Tarefa exige algum objeto que não existe
			System.err.println("Essa tarefa exige objetos que o servidor
não dispõe!");
			break;

		default://Erro
			System.err.println("Ocorreu um erro a efetuar o pedido!");
			break;
		}
	}
```

- Começa-se por ler da consola local o que o utilizador deve fornecer. O método **lerNumeroConsola(String mensagem, String erro)** escreve a **mensagem** na consola e espera que o utilizador escreva o pedido garantindo que seja um número (pois verifica e escreve a mensagem de **erro** enquanto o utilizador não o fizer). É utilizado um ***BufferedReader*** para receber o *input* da consola.
- Após recolher a informação do utilizador é criado um objeto da classe **Mensagem** (Nota: MensagemId, MensagemLista, etc são subclasses de **Mensagem**) que implementa ***Serializable*** e é enviado ao **ServidorCliente** utilizando um **ObjectOutputStream** e invocando o método ***writeObject(Object o)*** seguido de um ***flush()***.
- Feito o pedido, o **Cliente** espera pela resposta, sendo neste caso uma ***string*** (por isso foi utilizado o método **receberString**() que espera pela resposta do servidor em forma de ***string*** e devolve-a). Existe também o método **receberMensagem**() que faz o mesmo que **receberString**() mas espera por um **objeto** (que é utilizado para respostas do servidor em forma de **objetos** da classe **Mensagem**.
- Quando é recebida a resposta do **Servidor** é então feito um **switch**(), pois de forma a enviar o mínimo de dados possíveis pelos ***Sockets*** decidiu-se dar números para os diferentes resultados possíveis, sendo em geral "1" como concluído com sucesso, e por fim notificar o utilizador de tal resposta.

O **Cliente** tem métodos que enviam mensagens ao **ServidorCliente** em que é criado 1 ***Thread*** para receber a respetiva resposta. Esses métodos são: **iniciarTarefa**() e **esperaTarefas**() que correspondem às funcionalidades de "Requisição de objetos" e "Espera por conclusão de tarefas" respetivamente. Isto foi implementado desta forma porque estas funcionalidades nunca garantem uma resposta rápida uma vez que só se pode iniciar tarefas se houver os respetivos objetos disponíveis assim como não se sabe quando as tarefas serão concluídas. Desta forma o utilizador

poderá realizar mais pedidos enquanto espera pela resposta destes. Uma nota importante: apenas pode fazer dois pedidos deste tipo (1 para iniciar tarefa e 1 para esperar por conclusão de tarefas) em simultâneo uma vez que é utilizado a porta 12346 e 12347 nos Sockets criados para obter a resposta.

Mensagem

A classe **Mensagem** é responsável pela transmissão de informação entre o Cliente e o Servidor. Possui uma variável ***int* tipo** que indica o tipo de pedido efetuado (ex.: 0 -> fechar conexão, 1 -> registar utilizador, 8 -> listagem tarefas / tarefas em curso, etc.) e todos os objetos da classe ou subclasse de Mensagem têm esta variável como preenchimento obrigatório.

As subclasses de **Mensagem** são:

- **MensagemId** – para a funcionalidade de finalizar tarefa.
- **MensagemLista** - para as funcionalidades de espera por conclusão de tarefas e listagem de tarefas.
- **MensagemObjeto** – para a funcionalidade de abastecimento do armazém.
- **MensagemTarefa** – para as funcionalidades de definição de tipo de tarefa e início de tarefa.
- **MensagemUtilizador** – para as funcionalidades de registar utilizadores e de efetuar *login*.

Todas estas classes têm um construtor que pede a respetiva informação e um ou vários métodos que permitem ler essa informação, não contêm métodos que permitam alterar a informação pois o objetivo dos objetos destas classes é juntar informação necessário em um local e enviar ao Servidor ou ao Cliente.

Objeto

A classe **Objeto** guarda o nome do objeto e a quantidade disponível. Tem ainda o ***Lock*** do próprio objeto e duas variáveis de condição: uma condição para os casos de não haver quantidade suficiente para a realização de tarefas e outra para o caso de a quantidade disponível for igual à quantidade máxima (sendo a máxima igual a ***Integer.MAX_VALUE***) para evitar ***overflow***.

Tanto o ***Lock*** como as variáveis de condição são utilizados nos métodos de abastecer e de requisitar do próprio objeto.

O método de abastecer tem o seguinte ciclo que faz a ***Thread*** que quer abastecer esperar que seja libertado alguns objetos do mesmo tipo:

```
while (this.quantidade > (Integer.MAX_VALUE - n)) this.naoCheio.await ();
```

Sendo "n" o número de objetos que se quer abastecer do mesmo tipo.

O método de requisitar tem o seguinte ciclo que obriga a ***Thread*** a esperar que haja objetos suficientes:

```
while (this.quantidade < n) this.naoSuficiente.await ();
```

Sendo "n" o número de objetos que se quer requisitar do mesmo tipo.

Tarefa

A classe **Tarefa** guarda o nome da tarefa e um ***Map*** com os objetos (chaves) e respetivas quantidades necessárias (valores). Tem ainda o ***Lock*** do próprio objeto e um método que permite adicionar objetos e respetivas quantidades ao ***Map*** com os objetos necessários.

Utilizador

A classe **Utilizador** guarda o nome do utilizador, a respetiva palavra-chave, o conjunto de nomes de tarefas criadas por este utilizador e um ***Map*** com as tarefas iniciadas pelo mesmo sendo a chave o nome da tarefa e o valor o identificador (número) da respetiva tarefa. Tem ainda o ***Lock*** das tarefas criadas e o ***Lock*** das tarefas iniciadas. Tem métodos que permitem adicionar tarefas criadas e tarefas iniciadas.

Servidor

O Servidor possui as seguintes informações em variáveis:

- ***Map <String, Utilizador>*** utilizadores – guarda o nome de utilizador e a respetiva palavra-chave.
- ***Set <Objeto>*** objetos – guarda os objetos.
- ***Map <String, Tarefa>*** tarefas – guarda as tarefas sendo a chave o nome da tarefa.
- ***Map <Integer, Tarefa>*** tarefasIniciadas – guarda as tarefas iniciadas sendo a chave o identificador atribuído aquando a sua iniciação.
- ***Map <String, Integer>*** tarefasConcluidas – guarda as tarefas concluídas sendo a chave o nome da tarefa e o valor o número de instâncias concluídas.
- Cada variável acima descrita tem um ***Lock*** associado para garantir exclusão mútua aquando nova inserção / remoção ou leitura.
- Cada variável acima descrita tem um ***boolean*** associado para registar se houve alguma modificação e caso sim armazenar em disco a nova versão quando desligarem o servidor.
- ***Condition*** naoConcluido – variável de condição que serve para garantir que a ***Thread*** que faz o pedido de esperar por conclusão de tarefas espere sempre que não haja tarefas concluídas da lista pedida.
- ***Int*** idTarefInic – variável que serve para contar e atribuir identificadores às tarefas iniciadas.

O **Servidor** é composto por 3 ***Threads***, o **ServidorConsola** que fica a atender os pedidos vindos da consola local, o **ServidorPing** que abre um ***ServerSocket*** na porta 12349 e fica à espera que os **ClientePing** se liguem para lhes enviar o estado atual do **Servidor** e a ***Thread*** principal que após ligar estas duas ***Threads*** abre um ***ServerSocket*** na porta 12345 e fica à escuta de novas ligações de clientes. Após uma ligação ser aceite, é criada uma nova ***Thread*** com **ServidorCliente** que irá atender o novo cliente e o **Servidor** volta à escuta de ligações.

É nesta classe que se encontra a maioria dos métodos que implementam as funcionalidades pedidas no enunciado. À entrada de cada método faz-se o ***lock*** do objeto que se pretende aceder, faz-se o que se tem a fazer e no fim é feito o ***unlock*** desse objeto.

A título de exemplo, vai-se analisar o código do método responsável por abastecer o armazém.

```
void abastecer (String nome, int quantidade) {
	this.lockObjs.lock ();
	try {
		Objeto o = getObjeto (nome);
		o.abastecer (quantidade);
	} catch (ObjetoNaoExiste e) {
		Objeto o = new Objeto (nome, quantidade);
		this.objetos.add (o);
	} finally {
		this.objModificado = true;
		this.lockObjs.unlock ();
	}
}
```

-Começa-se por fazer ***lock*** do ***Set*** que contem todos os objetos e procura-se se o objeto pretendido já existe com o método "`Objeto getObjeto (String nome) throws ObjetoNaoExiste`" que procura e retorna o **Objeto** pretendido (fazendo **lock** do Set no inicio e **unlock** no fim) e caso não exista retorna uma exceção.
-Se existir basta chamar o método "`public void abastecer (int n)`" desse objeto que trata de abastecer (fazendo ***lock*** e ***unlock*** do objeto).
-Se não existir então cria-se um objeto novo e adiciona-se ao ***Set***.
-Por fim faz-se ***unlock*** do ***Set*** dos objetos.

ServidorCliente

Esta é a classe responsável por assegurar a comunicação entre o servidor e o cliente por isso implementa ***Runnable***. Esta classe e as restantes que são executadas numa ***Thread*** implementam ***Runnable*** em vez de estender ***Thread*** para poder-se, caso necessário, estender essas classes a outras. Possui as seguintes variáveis de instância:

- ***Socket*** socket – ***socket*** criado no Servidor aquando o inicio da ligação ao cliente.
- ***ObjectInputStream*** inputClienteOb – serve para receber objetos do cliente.
- ***ObjectOutputStream*** outputClienteob – serve para enviar objetos ao cliente.
- ***PrintWriter*** outputCliente – serve para enviar ***strings*** ao cliente pois há casos em que a resposta é apenas um número ou nome, e assim envia-se menos ***bytes*** de dados.
- ***Servidor*** servidor – para permitir efetuar as operações pretendidas.
- ***ServidorPing*** servidorPing – para no fim da comunicação informar o **servidorPing** que o cliente fechou a conexão.
- **Utilizador** utilizador – para permitir efetuar as operações exclusivas desse utilizador como adicionar / remover tarefas iniciadas e adicionar tarefas criadas.

As ***Threads*** que tratam desta classe ficam num ciclo à espera de receber **objetos** da classe **Mensagem** vindo do cliente, e à medida que chegam, a Thread extrai o tipo de pedido da **Mensagem** mais a informação relevante e executa os métodos do **Servidor** apropriados.

O ServidorCliente tem dois métodos de enviar dados para o cliente: o "**enviarMensagem**" que permite enviar objetos da classe **Mensagem** e o "**enviarString**" que permite enviar ***strings***.

Aqui está um pequeno excerto do código do ciclo principal.

```
while (this.socket.isClosed () == false) {

	Mensagem input = receberMensagem ();
	switch (input.getTipo ()) {
		...
		case 2://Login
			mensUt = (MensagemUtilizador) input;
			nome = mensUt.getNome ();
			pass = mensUt.getPass ();
			try {
				this.utilizador = this.servidor.login (nome, pass);
				enviarString ("1");
			} catch (UtilizadorNaoExiste e) {
				enviarString ("-1");
			} catch (PasswordErrada e) {
				enviarString ("-2");
			}
			break;
			...
```

Conclusão

Este trabalho prático permitiu-nos perceber e aplicar mecanismos de exclusão mútua como o uso de ***Locks*** ou ***synchronized*** e as variáveis de condição do Java e ainda o uso de ***Sockets*** para a comunicação entre duas ***Threads*** diferentes.

Tivemos de tomar várias decisões durante a implementação, entre as quais destaca-se:

- O envio dos pedidos do cliente para o servidor são feitos com apenas um envio de um objeto da classe **Mensagem** para garantir que o servidor recebesse o pedido com toda a informação necessária duma vez.
- As respostas do servidor são em forma de **string** (exceto os pedidos de listagem de tarefas que são em forma de um objeto da classe **Mensagem** de forma a incluir todos os elementos) para enviar, pelo ***Socket***, o mínimo de ***bytes*** possível. Podia-se colocar o envio em forma de ***bytes*** em vez de ***strings*** mas tivemos problemas ao implementar, uma vez que o cliente nunca conseguia receber os ***bytes***.
- A implementação do armazenamento em disco das tarefas, dos objetos e dos utilizadores apenas quando o servidor é desligado. Optamos por esta solução pois assim não tivemos problemas com os ***Locks.*** Uma vez que a ***Thread*** que fez ***lock*** já não existiria para fazer ***unlock***.
- Decidimos implementar um mecanismo de ***ping***, em que o **Cliente** cria de 10 em 10 segundos uma ***Thread*** para verificar se o servidor continua vivo apenas para tentar tornar a aplicação mais realista. Assim se o utilizador do lado do servidor desligar o servidor, todos os clientes acabariam por saber e fechariam a conexão. Originalmente, tínhamos uma ***Thread*** em paralelo com a principal do lado do Cliente a fazer o seguinte: acordava, pedia o estado do servidor e voltava a adormecer, mas tivemos dificuldades a encerrar graciosamente esta ***Thread*** sempre que o próprio utilizador mandava fechar a conexão.