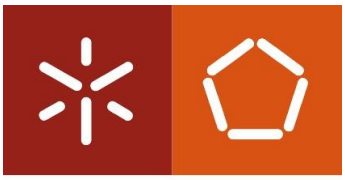

Universidade do Minho

Departamento de Informática

Trabalho Prático
Sistemas Distribuídos

Catarina Machado (a81047) Gonçalo Faria (a86264)
João Vilaça (a82339) José Fernandes (a82467)

1 de Julho de 2019

(a) Catarina.

(b) Gonçalo.

(c) João.

(d) José.

Conteúdo

1 Descrição do enunciado proposto **2**

2 Descrição dos objetivos do relatório **2**

3 Descrição do trabalho realizado **2**
3.1 Modelo de comunicação 2
3.2 Servidor 3
3.2.1 Worker 3
3.2.2 Estado partilhado 3
3.2.3 Escalonador de recursos 3
3.2.4 Escalonador de recursos aplicado ao problema de reservas e licitações 4
3.2.5 Fila de trabalho 5
3.3 Cliente 5
3.4 Controlo de concorrência 6

4 Análise de desempenho **6**

5 Análise Crítica do Resultado Obtido **7**

6 Correspondência entre as entidades mencionadas e classes no nosso código **7**

1 Descrição do enunciado proposto

O enunciado propõe o desenvolvimento de uma plataforma de gestão de sistemas de computação na nuvem que permite a obtenção de um servidor disponível para uso por parte dos utilizadores. Esta plataforma será um sistema distribuído, programado em Java, com a capacidade de receber e processar requisições de alocação de servidores, quer sejam "a pedido" ou através de "leilão", na nuvem e de contabilização do custo incorrido pelos utilizadores. Deverão, segundo o enunciado, ser criadas duas aplicações distintas, "Servidor" e "Cliente". O "Servidor" será o elemento fundamental da plataforma e deverá, por isso, suportar as funcionalidades de requisição de servidores acima descritas. As requisições deverão ser efetuadas pelo clientes através de um protocolo em texto orientado à linha. Por sua vez o "Cliente" deverá assumir o papel de interface entre o utilizador e a plataforma, convertendo os pedidos do utilizador na interface de texto para o protocolo implementado, e fazendo a ligação com o "Servidor".

2 Descrição dos objetivos do relatório

O presente relatório tem como principal objetivo a descrição e justificação das decisões tomadas pela equipa durante o desenvolvimento das várias fases da plataforma. Ao longo das próximas páginas será apresentada em detalhe a arquitetura da plataforma, em termos de classes utilizadas e estratégias implementadas, quer para o "Servidor" quer para o "Cliente". Será também feita uma breve descrição e análise do protocolo de comunicação entre ambos.

3 Descrição do trabalho realizado

3.1 Modelo de comunicação

Por forma a simplificar a complexidade associada à conceção do sistema distribuído proposto [2], tal como indicado no enunciado, implementamos uma arquitetura **cliente-servidor**. Neste tipo de arquitetura existem essencialmente dois conjuntos de processos, processos **cliente** e, embora com apenas um elemento nesta aplicação, processos **servidor**. O servidor é caracterizado pela implementação de serviços e o cliente por requerer estes por via de envio de pedidos.

A interação entre estes dois grupos de processos é uma em que é dada prioridade ao servidor pois, após o envio de cada pedido, o cliente espera até obter a resposta do servidor. Este tipo de interação, no entanto, não é ótima para todo o tipo de aplicações, pois ao cliente não é possível saber se ao não obter respostas do servidor esta é devida à perda dos seus pedidos ou à perda das respostas do servidor. Adicionalmente, este tipo de arquitetura é por vezes difícil de aplicar quando não existe uma clara separação entre quais processos são os clientes e quais são os servidores.

Apesar disto, esta arquitetura foi escolhida pois o protocolo de comunicação proposto no enunciado é o TCP/IP, que é um protocolo de transporte confiável e por essa razão assegura a comunicação de falhas nas transmissão.

Tendo como objetivo incluir no sistema distribuído a desenvolver transparência no acesso, seguindo [1], decidimos conceber a comunicação entre cliente-servidor usando **RPC** (Remote Procedure Call). Esta forma de estruturação de software faz com que a execução de funcionalidades do servidor, possivelmente numa localização diferente do cliente, seja o mais próximo possível da execução local. Desta forma, a integração do software desenvolvido em soluções de software já existentes é facilitada e é também introduzida a possibilidade de separação das tarefas de desenvolvimento do projeto em blocos isolados que podem facilmente ser independentemente testadas.

Inicialmente, decidimos desenvolver o sistema usando **RPC** assíncrono, isto contribuiu para que a grande maioria dos procedimentos correspondentes aos pedidos não devolvessem resultados. De uma forma semelhante a aplicações de email, a nossa solução não requeria que o cliente ficasse à espera que os pedidos fossem respondidos antes que novos fossem enviados. Mesmo assim, como no enunciado foi especificado que o cliente tem de esperar pela atribuição servidores a reservas e a licitações, introduzimos no sistema um mecanismo de sincronização que, através de espera de

novas notificações, permite ao cliente esperar pelas respostas aos seus pedidos, satisfazendo assim o requisito especificado.

3.2 Servidor

O processo servidor está continuamente ativo com um socket servidor à espera que clientes a este se liguem. Após ocorrer uma ligação, o resultado do método *connect* do socket servidor é enviado a um novo *thread* **worker** que inicia a sua execução concorrentemente com o *thread* original que volta repetir o mesmo procedimento..

3.2.1 Worker

A classe **worker** é dedicada a estabelecer comunicação com o cliente por via de um protocolo de comunicação por texto. Neste protocolo, o cliente, após identificado, indica, através de mensagens de texto, o nome e os parâmetros dos serviços que este pretende. A tarefa do *thread* **worker** é a de, sistematicamente, traduzir cada uma das mensagens, na respetiva invocação dos métodos que estas representam no sistema.

Adicionalmente, em alternância com a tradução das mensagens do cliente, este *thread*, traduz também a resposta correspondente ao resultado das invocações dos serviços.

Ao ser instanciada, **worker**, sendo que recebe como parâmetro a referência para o **estado partilhado**, cria um objeto da classe **ServerCMS**, a classe responsável por assegurar a segurança dos acessos dos diferentes *threads* ao **estado partilhado** e, principalmente, de implementar os serviços que o servidor disponibiliza.

Os serviços disponibilizados são *login*, *logout*, *register*, *reserve*, *bid*, *release*, b*alance*, *listActive-Machines*, *listActiveBids*, *listActiveReservationRequests* e *getNotifications*.

3.2.2 Estado partilhado

No estado partilhado, em essência, encontra-se, embora hierarquicamente distribuída, toda a informação relativa ao sistema que nós desenvolvemos. As suas variáveis são o dicionário que faz a correspondência entre o e-mail e a **conta** de utilizadores registados, um *array* com os diferentes **tipos de máquina** e também um conjunto de **trabalhadores do servidor**.

3.2.3 Escalonador de recursos

Cada **tipo de máquina**, contém um **escalonador de recursos** assim como os métodos para adequadamente interagir com este. Este escalonador é a estrutura da dados, por nós concebida, que avalia e atribui um conjunto de recursos a diferentes pedidos de diferentes prioridades.

Esta estrutura é parametrizada pela classes *resource* e *partaker* que, representam respetivamente a classe recurso e a classe que pretende adquirir esse recurso. O escalonador é composto por uma pilha de *resource*, uma fila de prioridade de **pedidos** e também uma fila de prioridade de **pedidos realizados**. Adicionalmente, existem dois dicionários que fazem a correspondência entre as chaves e os **pedidos** e **pedidos realizados** presentes na estrutura.

Na figura 2 encontra-se apresentado o diagrama UML correspondente à hierarquia de classes do escalonador de recursos.

Dado que a classe **pedidos** é uma classe abstrata, a sua ordem natural é apenas parcialmente especificada para assegurar a ordenação das prioridades dos **pedidos**. Para que classes possam usar o escalonador é necessário que estas ampliem(*extends*) esta classe abstrata e incluam a regra para a ordenação de pedidos com o mesmo grau de prioridade, fazendo assim desta ordem natural uma ordenação lexicográfica. Esta classe, assim como a classe pedido realizado, são, tal como o escalonador, parametrizadas pelas classes *partaker* e *resource*.

A classe **pedido realizado** é de igual forma uma classe abstrata que, contém como ordem natural, como lhe é sempre associada um **pedido**, uma ordem correspondente ao inverso da ordem de **pedido**. A intuição desta decisão estrutural provém da necessidade de obter na frente das duas filas de prioridades, uma que contém os melhores **pedidos**, e outra que contém os piores **pedidos**

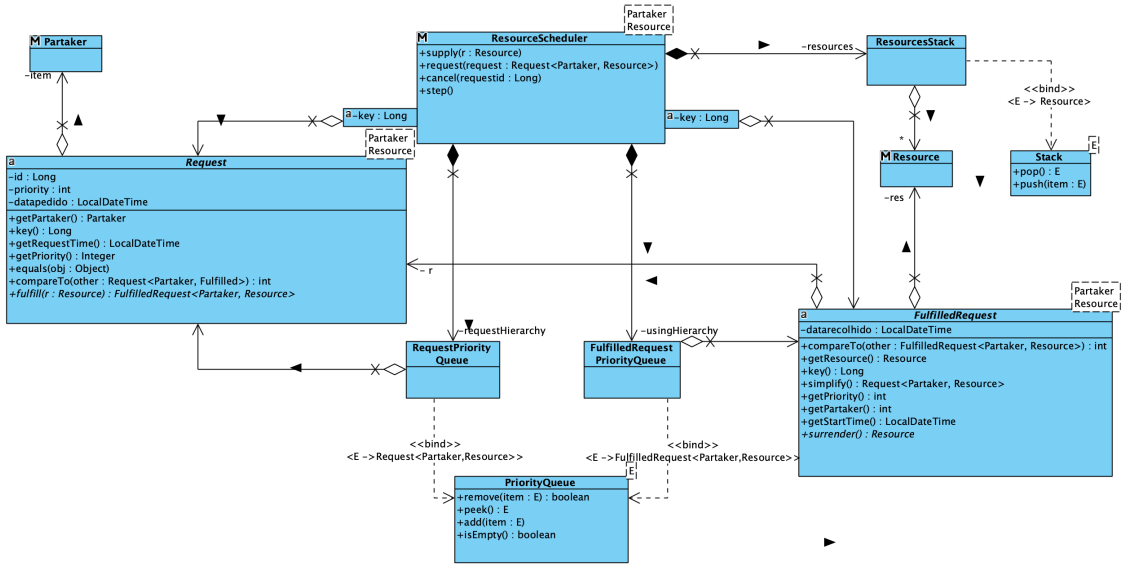

Figura 2: Diagrama UML da hierarquia de classes do escalonador de recursos.

realizados. Desta forma, sempre que não houver recursos livres, o pior **pedido realizado** e o melhor **pedido**, são comparados e se o **pedido** tiver uma ordenação inferior ele é tornado num **pedido realizado** e o **pedido realizado** original abdica do seu recurso sendo, desta forma, convertido de novo apenas a **pedido**.

Para além do já mencionado, a classe que ampliar pedido terá que, necessariamente, implementar o método *fulfill*. Este método, recebe um *resource* e devolve um **pedido realizado**. De uma forma dual, a classe que ampliar pedido realizado deve implementar o método *surrender*. Este método, devolve essencialmente o recurso previamente adquirido por esta.

Os métodos disponibilizados pelo escalonador são *supply*, que é dedicado a introduzir recursos, *request*, dedicado a receber **pedidos**, *cancel*, para remover o **pedido** ou **pedido realizado** correspondente à indicada chave, e por fim, *step*, que é o método que assegura a contínua atualização interna desta estrutura. O método *step*, sendo que por cada operação na estrutura é necessário no máximo uma vez, será iterativamente chamado, por um arbitrário conjunto de *threads*, à medida que as diferentes operações do escalonador forem sendo executadas.

3.2.4 Escalonador de recursos aplicado ao problema de reservas e licitações

Para usar o escalonador de recursos no nosso projeto, criamos as classes **bilhete**, **recibo**, **bilhete de leilão** e **bilhete de reserva**.

A classe **bilhete** amplia a classe **pedido** e introduz funcionalidades úteis, no contexto do sistema de alocação de servidores na nuvem, mais precisamente, obter a chave do **tipo de máquina** correspondente a este e também o preço. Este preço, corresponde, efetivamente, ora à oferta em leilão ou ao valor de reserva.

As classes **bilhete de leilão** e **bilhete de reserva**, ampliam **bilhete** e, servem essencialmente para especificar a prioridade de **pedido**. Esta prioridade, é 0 para os pedidos que correspondem a reservas e 1 para os correspondentes a ofertas em leilão. Adicionalmente, em **bilhete de leilão** e **bilhete de reserva** são especificados também os critérios de ordenação para os **pedidos** que instanciam cada uma destas classes. Por fim, foi introduzida a classe **recibo**, que amplia **pedido realizado** e contém funcionalidades adicionais dedicadas a calcular a dívida da **conta** de utilizador.

Na figura 3 encontra-se apresentado o diagrama UML correspondente à hierarquia de classes de **pedidos** e **pedidos realizados**.

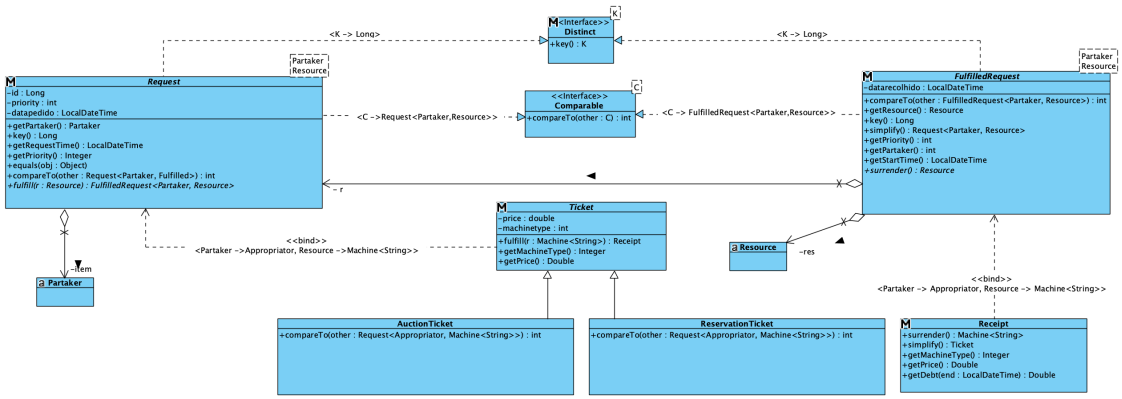

Figura 3: Diagrama UML da hierarquia de classes de pedidos e pedidos realizados

3.2.5 Fila de trabalho

Tal como foi especificado, o escalonador, para que se mantenha organizado, requer que o método *step* seja chamado, à medida que **pedidos** e recursos vão sendo adicionados e removidos desta estrutura. Em alternativa a chamar este método após o término das operações *supply*, *request* e *cancel* que, essencialmente, desencadearia uma maior variabilidade no tempo de resposta aos diferentes pedidos, este é feito num conjunto de *threads* à parte, através de uma fila de **carga de trabalho** partilhada.

À medida que operações do escalonador vão sendo executadas, ao longo do tempo de vida desta estrutura, as diferentes instâncias da classe **tipo de máquina** vão adicionando objetos da classe **carga de trabalho** a uma fila. Esta fila, é na verdade, uma variável de classe de **carga de trabalho**.

Nas instâncias de **carga de trabalho**, estão apenas compreendidas duas funcionalidades, a de obter o identificador do tipo de servidor a que esta carga corresponde, e a de obter o número de iterações que o respetivo escalonador precisa. Como o acesso a cada escalonar é feito em exclusão mútua, ao usar um conjunto de *threads* trabalhadores, em vez de apenas um, é criada a oportunidade de atualizar diferentes escalonadores em simultâneo.

A fila de **carga de trabalho** desenvolvida foi feita para que se duas cargas seguidas correspondem ao mesmo **tipo de máquina**, então, estas são fundidas, obtendo assim como resultado, apenas uma carga com o número de iterações correspondente à soma das iterações das cargas intervenientes. Desta forma, é evitada a luta por vários *threads* para o acesso ao mesmo escalonador, apenas para fazer uma iteração.

Os *threads* referidos, são correspondentes à classe **trabalhador do servidor** e são criados no momento que o programa servidor inicia. Desta forma, mantêm-se em espera passiva para remover elementos da fila de **carga de trabalho**.

3.3 Cliente

O Cliente funciona como um interpretador de comandos Unix, recebendo os comandos do utilizador, realizando o processamento destes mesmos e executando-os. Antes de poderem ser inseridos os comandos o cliente tenta estabelecer conexão com o servidor, ficando num *loop* caso esta não seja possível. Sempre que o cliente perder a ligação, este processo repete-se, voltando este a tentar estabelecer ligação com o servidor.

Após ser feito o *login* com sucesso, o cliente lança um *thread*, em *background*, para escutar notificações. Este *thread* utiliza um *socket* à parte de forma a não bloquear o socket do *thread* principal que o usa para chamar os métodos no servidor.

No que diz respeito à comunicação do cliente com o servidor esta foi encapsulada numa classe **ClienteCMS** que, em concordância com **RPC**, com intuito de ver realizados os pedidos locais num sistema remoto, implementa uma interface em comum com **ServerCMS**. Devido a este encapsulamento, o restante programa não lida com o facto que estão a ser chamados métodos noutro sistema.

3.4 Controlo de concorrência

Nos diagramas apresentados, assim como, ao longo do relatório, não foi abordado em detalhe questões associadas ao controlo de concorrência. No entanto, para impedir corridas aos objetos partilhados, implementamos e usamos um conjunto de mecanismos de *lock*. Por um lado a classe conta, escalonador e fila partilhada, contém *locks* reentrantes em todos os métodos. Por outro lados as classes **carga de trabalho** e dicionário concorrente usam locks de leitura e escrita.

4 Análise de desempenho

Para validar algumas das hipóteses apresentadas, em particular a que sugere que usar vários **trabalhadores do servidor** resultará num melhor desempenho, realizamos alguns testes. Tal como foi indicado anteriormente, nós conjeturamos que, devido a exclusão mútua ser aplicada a nível do escalonador, ao introduzir vários **trabalhadores do servidor**, abriríamos a possibilidade de vários escalonadores serem atualizados em simultâneo.

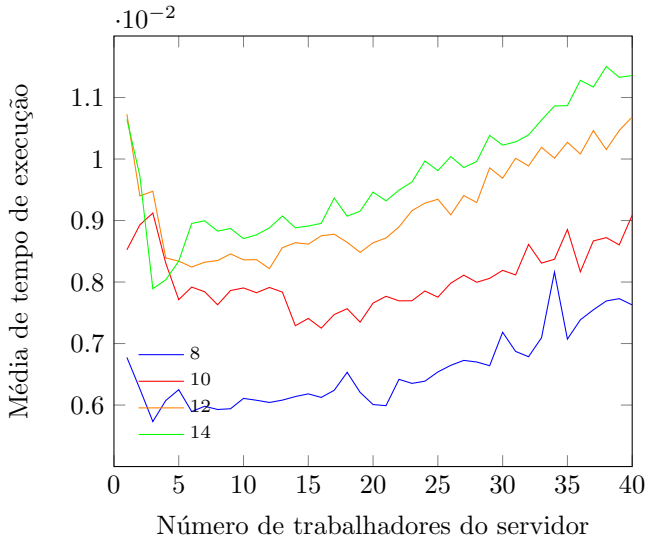

Figura 4: Cada linha é correspondente ao mesmo teste realizado com o número de tipos de máquina indicado. Os testes foram realizados num MacBook Pro com um processador 2,3 GHz dual-core, Intel Core i5 7360U.

Os testes realizados foram feitos com 20 clientes, cada um deles contribuindo com 10 reservas e 10 licitações por tipo de máquina e 60 máquinas por tipo de máquina. Os valores apresentados são a média de 500 iterações.

O gráfico apresentado em 4, mostra a média de tempo de execução à medida que aumentamos o número de **trabalhadores do servidor** para diferentes números de **tipos de máquinas**.

Como é possível verificar, o aumento de **trabalhadores de servidor** resulta numa melhoria dos tempos de execução. No entanto, quando o número **trabalhadores de servidor** excede, em larga medida, o número de **tipos de máquina** o desempenho piora.

Embora precisa-se-mos de mais testes para o concluir, o decremento do desempenho, à medida que o número de **trabalhadores do servidor** excede o número de **tipos de máquina**, aparente ser causado pela espera no acesso ao escalonador pelos vários **trabalhadores do servidor**.

5 Análise Crítica do Resultado Obtido

Neste projeto, desenvolvemos um solução de software, na forma de um sistema distribuído, que implementa um protocolo cliente-servidor. Este sistema, permite a vários clientes ligarem-se a um servidor com o intuito de ver realizados pedidos a serviços relativos à alocação de servidores na nuvem.

O projeto realizado, embora satisfaça com distinção os requisitos propostos, podia ser posteriormente refinado. Uma das alternativas de desenvolvimento, seria a de, em contraste à abordagem que nós seguimos de criar classes que ampliam as capacidades de suportar concorrência das coleções de *java*, implementar de raiz estruturas de dados que limitam as operações de exclusão mútua a pequenos elementos atómicos. Estas estruturas, sem de forma alguma perder as vantagens associadas à consistências de dados adjacentes às classes alinhadas à nossa abordagem, permitiriam melhorar significativamente os tempos de resposta ao diferentes serviços que o servidor proporciona.

Embora nós tenhamos considerado, já numa fase final do projeto, a criação e posterior inclusão destas estruturas, como o desenho conceptual inicial do sistema não as teve em consideração, não conseguimos adequadamente obter o esperado benefício destas.

6 Correspondência entre as entidades mencionadas e classes no nosso código

Entiade mencionada	**Classe no código**
Pedido	Request
Pedido realizado	FulfilledRequest
Bilhete	Ticket
Bilhete de leilão	AuctionTicket
Bilhete de reserva	ReservationTicket
Recibo	Receipt
Conta	Account
Fila partilhada	SharedQueue
Fila de prioridade	PriorityQueue
Carga de trabalho	WorkLoad
Estado partilhado	SharedServerState
Escalonador de recursos	ResourceScheduler
Trabalhador do servidor	ServerEmployee
Tipo de máquina	InstanceType
Máquina	Instance
Dicionário concorrente	ConcurrentMap
Worker	Worker

Referências

[1] Birrell A. and Nelson B., *Implementing Remote Procedure Calls*, ACM Transactions on Computer Systems, 2(1):39–59, Feb. 1984.

[2] Saltzer J. and Kaashoek M., *Principles of Computer System Design, An Introduction.*, Morgan Kaufman, San Mateo, CA., 2009.