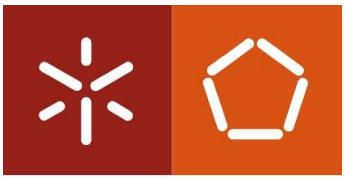

Universidade do Minho

Departamento de Informática

Sistemas Distribuídos
Grupo 16
Alocação de servidores na cloud

Eduardo Barbosa (a83344) Bárbara Cardoso (a80453)
Pedro Mendes (a79003)

6 de Janeiro de 2019

Conteúdo

1 Introdução

No âmbito da disciplina de Sistemas Distribuídos foi proposta a implementação de uma aplicação para gestão de servidores (requisitar/leiloar) na *cloud*, semelhante às já existentes *Google Cloud* e *Amazon EC2* . Esta aplicação deve suportar registo e autenticação por parte dos utilizadores bem como a compra de servidores, licitar em leilões e criação dos mesmos.

Esta implementação baseia-se na arquitectura *cliente-servidor*.

2 Descrição do Problema

O serviço oferecido pela aplicação baseia-se em:

- Registo e autenticação de clientes;
- Reserva de servidores;
- Serviço de leilões.

Um cliente para se registar necessita de fornecer um `email`, uma `password` e um `username`.

De forma a adquirir uma *droplet*, servidor disponivel no catálogo, um utilizador tem duas opções, ou compra a *droplet* ao preço a que esta aparece no catálogo de servidores ou vai a leilão. Um leilão é criado com o valor que o utilizador propor e decorre durante `40s`. Durante este tempo todos os utilizadores podem licitar de forma a tentarem arrecadar a *droplet*. Quando deixa de haver stock de um tipo de *droplet* e um utilizador tenta adquirir uma *droplet* desse tipo, dependendo do pedido ser uma reserva normal ou um leilão, uma de duas coisas acontece:

$$\text{Reservar} \rightarrow \text{Tenta roubar}$$

$$\text{Leilão} \rightarrow \text{Fila de espera}$$

A reserva de um tipo de *droplet* sem stock resulta na tentativa de roubo de uma *droplet* desse tipo que foi adquirida por leilão.

A tentativa de criar um leilão com uma *droplet* cujo tipo se encontra esgotado resulta em o utilizador ir para uma fila de espera. Quando uma *droplet* desse tipo é libertada o utilizador com a maior oferta nessa fila, ganha o servidor.

O utilizador tem uma divida associada, resultante do tempo que mantêm as suas *droplets*.

Em qualquer momento o utilizador pode escolher prescindir de uma *droplet* sendo que o `stock` daquele tipo de servidor é aumentado e a divida proveniente daquela *droplet* é guardada para efeitos económicos e estatísticos.

3 Proposta de Resolução

O primeiro passo foi identificar as entidades, leia-se classes, envolvidas como `Bid` que representa uma licitação de um utilizador, `Auction` que representa um leilão a decorrer e `User` que representa um utilizador do serviço, entre outras.

Tendo identificado estas entidades identificamos as que têm estado mutável e as que seriam *thread safe* pela sua natureza imutável. Esta separação de classes pelo seu estado, mutável ou imutável, facilita muito o raciocínio e estruturação da arquitectura incluindo a identificação de onde podem surgir problemas relacionados com a componente *multi threaded* do código.

Decidiu-se juntar todos os componentes mutáveis na mesma classe criando assim um único ponto com estado mutável, ou seja, um único ponto de falha. A esta classe foi dada o nome de `AuctionHouse`. A `AuctionHouse` tem a responsabilidade de guardar os utilizadores registados e a sua dívida, as *droplets* reservadas, as disponíveis, os leilões a decorrer e as filas de espera.

Esta aplicação divide-se em 2 componentes principais, o `Client` que é um programa *a la telnet* que é responsável por comunicar com o servidor que, por sua vez, mantêm todo o estado e lógica de

negócio (`AuctionHouse`). Por forma a ter um servidor *multi threaded* recorremos a um *middleware* que faz a integração da *AuctionHouse* com o cliente. Toda a comunicação é feita via *sockets* TCP.

4 Arquitetura

4.1 Cliente

Para a implementação deste sistema foi criada uma classe `Client`, que permite interagir com o servidor a partir de comandos de texto, usando uma única socket TCP. As leituras e escritas feitas sobre esta socket são por si assíncronas para suportar o serviço de notificações do Servidor.

4.2 Servidor

A lógica do servidor está dividida em 4 *packages*:

`server`	Lógica de negócio;
`server/middleware`	Classes responsáveis para criar contexto comum;
`server/exceptions`	*Exceptions* lançadas pela lógica de negócio;
`util`	Classes utilitárias.

4.2.1 Reaproveitamento de Threads

De forma a reaproveitar *threads* utilizamos uma *Thread Pool*. Quando uma *thread* deixa de ser necessária, esta fica em estado *idle* para que possa ser reutilizada para a proxima conexão por parte de outro cliente.

4.2.2 util

`Pair<F,S>`: Esta classe implementa um par visto que Java não implementa uma classe Pair na sua *standard library*. De notar que esta classe é um *BiFunctor* e oferece métodos com esse poder expressivo, como por exemplo:

```
public <F2> Pair<F2, S> mapFirst(final Function<? super F, ? extends F2> function) {
    return of(function.apply(first), second);
}
```

`AtomicInt` e `AtomicFloat`: Estas classes implementam uma versão *thread safe* e mutável de um `Integer` ou `Float` disponibilizando uma API restrita que apenas permite alterações atómicas do seu valor. Esta é útil para garantir, por exemplo, que os IDs auto-incrementados dos `Droplets` nunca se repetem caso dois fossem instanciados em simultâneo.

`ThreadSafeMap`: Esta implementação da interface `Map<K,V>` garante que todas as operações de escrita feitas sobre ele são atómicas e sequenciais, já as operações de leitura podem ser feitas concorrentemente devido à utilização de um *Read Write Lock*.

`ThreadSafeMutMap`: Esta extensão do `ThreadSafeMap` é mais restrita quanto aos objectos que permite guardar, estes tem de implementar a interface `Lockable` que define objectos com a possibilidade de ser bloqueados por uma *thread*. Esta restrição é necessária para ser possível codificar o seguinte padrão, dentro de cada metodo.

```
this.lock.lock();
V v = this.map.get(k);
v.lock();
```

```
this.lock.unlock();
return v;
```

Assim a API desta estrutura de dados disponibiliza os seguintes métodos para que seja explicito que o objecto retornado esta *locked* e terá de ser *unlocked* para que não sejam criadas situações de *deadlock*.

- `public V getLocked(K k)`
- `public Collection<V> getLocked(Collection<K> keys)`
- `public V putLocked(K k, V v)`
- `public Collection<V> valuesLocked()`
- `public Set<Entry<K, V>> entrySetLocked()`

`Lockable:` Esta interface obriga as classes implementantes a definir os metodos `void lock()` e `void unlock()` para que possam ser guardadas no `ThreadSafeMutMap`.

`ThreadSafeInbox:` Esta classe actua como um *buffer* de mensagens *thread safe* e serve de *inbox* para cada utilizador. Esta implementação utiliza uma *double endeded queue* e variáveis de condição de forma a garantir a atomicidade das suas operações.

4.2.3 server.middleware

`Session:` *Middleware* entre a ligação TCP e a `AuctionHouse`. É instanciada uma nova `Session` para cada nova ligação. Esta recebe a socket da ligação e a `AuctionHouse`. Com isto, a `Session` lê os comandos do utilizador, interpreta-os e comunica-os ao *backend*. Inversamente, a `Session` comunica as respostas do *backend* ao cliente. Para isto é necessário que o cliente permita comunicação assíncrona, pois nem todas as mensagens que o servidor envia são como resposta imediata a um comando.

`CTT:` Classe responsável por ler as mensagens enviadas para um utilizador. É criado um novo `Ctt` para cada nova ligação sempre que um utilizador faz *login*. É instanciado utilizando o utilizador da sessão e o *PrintWritter* da mesma. Esta instância corre na sua própria *thread* e limita-se a ler a `Inbox` do utilizador e mandar o conteúdo para o *buffer*. Desta forma a comunicação por parte do Servidor é assíncrona.

4.2.4 server

`AuctionHouse:` Esta classe guarda no seu estado interno os utilizadores, a divida destes, os leilões a decorrer, as filas de espera, as *droplets* reservadas separadas pelo método pelo qual foram adquiridas e o stock atual.

Esta classe disponibiliza a principal API da aplicação, registos, *logins*, listagem do catalogo, compra de servidores, etc. Todas as operações de leitura não precisam de controlo de concorrência explicito devido a este ser assegurado pelo `ThreadSafeMap` e `ThreadSafeMutMap`. O mesmo se sucede para as operações de escrita que mexem com apenas um *Map*. Por outro lado, as operações de escrita que envolvem mais do que um `Map` bloqueiam-nos de forma a que a operação seja uma transação. Estes *locks* múltiplos são efetuados sempre pela mesma ordem de prioridade para que não seja criados *deadlocks* causados por dependências circulares.

Leilões → Stock → Reservas por Leilão → Reservas por Compra

Figura 1: Ordem pela qual são adquiridos os locks

`Auction:` Classe que coordena uma leilão. O leilão dura 40 segundos e durante este tempo pode receber novas licitações paralelamente. Quando o tempo termina a *callback* que lhe foi passada no construtor é executada e de seguida notifica todos os utilizadores envolvidos para os informar do seu fim e se ganharam ou perderam a *droplet*.

`UniqueBidQueue:` Classe que coordena uma fila de espera para quando não existir stock suficiente para efetuar um leilão. Esta permite duas operações: *enqueue*, que adiciona uma `Bid` à fila, e *serve* que remove a entrada que possui a maior oferta executando uma *callback* que foi passada no seu construtor. Esta irá alterar o stock e fazer as reservas necessárias.

`Bid:` Classe que representa uma intenção de licitação. Tem como variáveis de instância o utilizador que a criou e o montante.

`Droplet:` Classe que representa um servidor do catálogo. Cada servidor tem como atributos o seu id, dono, tipo do servidor, custo e data de atribuição.

`User:` Classe que representa um utilizador da aplicação, guardando o seu email, nome, password e a sua `ThreadSafeInbox`.

`ServerType:` Classe que representa os tipos dos servidores. Cada servidor possui um nome e um preço. É um *enum*.

5 Testes

Para assegurar que as funcionalidades críticas do programa funcionam correctamente foi feita uma tentativa de desenvolver alguns testes automáticos sobre o `AtomicInt`, o `ThreadSafeMap`, o `ThreadSafeMutMap` e o Servidor em si. Estes consistem em criar número desmedido de *threads* que realizam imensas operações sobre as classes em questão e verificam se o estado destas se mantém consistente durante o processo. Em anexo seguem alguns exemplos de testes.

Dado que este projecto foi desenvolvido com o auxílio da ferramenta de controlo de versões *GitHub* tentamos também implementar *continuous integration* usando o serviço *Travis*.

6 Trabalho Futuro

Como complemento do projecto deixamos duas sugestões. Em primeiro lugar seria interessante implementar um pseudo-sistema de pagamentos visto que neste momento apenas é acumulada divida. Por fim deixamos a nota que alguma verificação formal de certos invariantes seria interessante.

7 Conclusões

Sistemas Distribuídos têm como objectivo melhorar o desempenho de um sistema informático dividindo responsabilidades e repartindo tarefas. É esperado que este processamento em paralelo leve a uma redução do tempo de execução. Existem várias formas de implementar paralelismo sendo que a arquitectura usada neste trabalho foi a de *cliente-servidor* tendo por base *threads* da *JVM*.

É de notar que o paralelismo tem os seus custos podendo não ser de todo proveitoso em programas com uma escala mais reduzida. A introdução de paralelismo costuma ser acompanhada pela introdução de concorrência que por sua vez traz problemas de difícil identificação e resolução. É preciso garantir as propriedades de segurança, isto é nenhum estado indesejado é atingido pelo programa, e as propriedades de *liveness*, propriedades que devem ser verdade durante a execução do programa.

Figura 3: Comandos disponibilizados pelo `Session`

8 Anexo

```
public void apply() throws InterruptedException {
    final AtomicInt aInt = new AtomicInt(0);
    List<Thread> threads = IntStream.range(0, 1000)
            .mapToObj(i -> new Thread(() -> aInt.apply(x -> x + 1)))
            .collect(Collectors.toList());
    threads.forEach(Thread::start);
    for (Thread thread : threads) {
        thread.join();
    }
    Assert.assertEquals(aInt.load(), 1000);
}
```

Figura 2: Teste Unitario do metodos `AtomicInt::apply`