Universidade do Minho – 2016/2017

Relatório de Sistemas Distribuídos

LEIlões

Grupo:
Bruno Pereira, a75135
Carlos Pereira, a61887
Diogo Silva, a76407
Maria Ana de Brito, a73580

Introdução

Este projeto tem como objetivo criar uma aplicação distribuída com estrutura cliente/servidor para fazer a gestão de um serviço de leilões, em que existem dois tipos de atores: vendedores e compradores.

Os primeiros são os que leiloam um item, ou seja, iniciam um leilão. Os segundos, compradores, podem licitar sobre qualquer item que esteja num leilão ativo. O leilão é ganho por quem detiver a licitação mais alta (que seja acima ou igual ao preço base estabelecido previamente pelo vendedor). Um vendedor pode licitar no próprio item e, assim, ganhar o próprio leilão. Este também tem o poder de finalizar o leilão, isto é, assim que entender, pode acabar o leilão. Ambos podem, a qualquer momento, listar os leilões em curso. Assim, os dois atores são apenas distinguidos pelo facto de ter ou não inicializado um leilão, pois um utilizador pode ter leilões em curso e ter licitado noutros (ou no próprio).

Trata-se de uma aplicação distribuída e concorrente, portanto é dada uma atenção particular a situações que podem levar a deadlocks, em que pode ocorrer um bloqueio de threads e controlo de acesso a regiões partilhadas, também conhecidas por regiões críticas, pois temos de manter a coerência dos dados nas estruturas usadas e evitar possíveis anomalias.

Cliente/Servidor

A aplicação distribuída tem a seguinte estrutura: um servidor para vários clientes. Quando existe um servidor ativo, este tem que gerir todos os clientes que se queiram ligar. Por cada cliente, o servidor cria um novo socket que servirá como canal de comunicação individual entre o servidor e o cliente, isto é, sempre que uma ligação é estabelecida, uma nova thread para esse cliente é criada e esta apenas escreverá num socket apenas. Desta maneira, por cada cliente existem duas threads (a segunda thread será criada mais tarde) e um socket.

Desta maneira, garantimos que um cliente que, por alguma eventualidade, seja mais lento, não prejudica o desempenho dos outros, pois o sistema deixa de estar vulnerável a esta situação.

Sempre que um cliente decide que quer interromper a ligação ou é forçado a terminá-la (situação explicada mais adiante), o servidor trata desse caso e continua a sua execução, sempre pronto a aceitar novos clientes.

Registar e Login

Para implementar o registo e o login de um utilizador, criamos uma classe Utilizadores.

Para o registo de um utilizador no sistema funcionar corretamente foi necessário colocar o método addUser como synchronized. Assim, só poderá ocorrer um registo de cada vez, impedindo que dois utilizadores com o mesmo username se registem ao mesmo tempo, o que resultaria num erro de escrita.

Relativamente ao login, este não pode ter synchronized, pois tem de permitir que vários utilizadores com usernames diferentes se consigam conectar concorrentemente. No entanto, foi imposta por nós uma restrição que proíbe um cliente de ter duas janelas suas abertas ao mesmo tempo. Para garantir isto, criamos dois Maps. O primeiro, de booleanos, que indica se o utilizador tem uma sessão iniciada, e o segundo, de ReentrantLocks. Para verificar se o cliente tem sessão iniciada, é necessário fazer lock à posição do Map cuja chave é o username. Depois verifica se tem a sessão iniciada. Se tiver, lança uma exceção, se não, faz o login corretamente. A exclusão mútua é necessária, pois, se não acontecesse, era possível que dois clientes, com o mesmo username e password, alterassem o valor da variável (que indica se o cliente já tem a sessão iniciada) ao mesmo tempo e conseguissem fazer ambos o login. O benefício de usar um Map de ReentrantLocks surge do facto de não ser necessário bloquear a estrutura toda para verificar um elemento dela, em vez disso, faz-se exclusão mútua para o cliente correspondente.

Após o login ter sido efetuado com sucesso, a classe LeilaoThread trata da inicialização das estruturas necessárias.

No fim da aplicação, quando o cliente fizer logout, a variável (posição do cliente no Map de booleanos) é alterada e é possível fazer-se login novamente.

Leilões

Um leilão refere-se a algum item que esteja a ser leiloado, que aceite licitações. Desta forma, todos os leilões ativos são guardados numa estrutura de dados Map, na qual se faz a associação entre o seu número identificador único e o item a ser leiloado. Sempre que um leilão é iniciado é devolvido ao cliente o número de identificação e quando se lista os leilões aparecem todos os número identificadores dos leilões ativos, logo é preciso ter um número que nunca seja repetido durante a execução da aplicação. Na classe InfoLeiloes, que armazena o Map de leilões, existe uma variável que representa o número de leilões inicializados até ao momento (quer estejam ativos ou não), que garante que o número identificador atribuído a cada leilão é único, evitando possíveis erros que colocariam em causa toda a execução do serviço de leilões.

A informação de cada item a ser leiloado é guardada na classe Item. Aqui sempre que se leiloa um novo item, é criada uma nova instância que armazena a designação e o preço base, que são desde logo atribuídos pelo vendedor. Denote-se que o preço base de um item, a partir da sua criação, não pode ser mudado, pois criaria uma injustiça a nível do conceito de leilão, uma vez que um vendedor poderia alterar o preço base sempre que quisesse, desrespeitando todas as licitações que anteriormente eram válidas e, agora, são abaixo do novo preço base.

Além da informação do item, também é armazenado o username do comprador com a licitação mais alta até ao momento, bem como o valor dessa licitação. Com o objetivo de enviar uma notificação a cada comprador que fez uma licitação nesse item, são também guardados todos os usernames dos licitadores.

Como o Map que armazena os leilões em curso é uma zona partilhada por todas as threads criadas a cada ligação de um cliente aceite, é necessário garantir que o seu acesso é feito de forma controlada. Desta maneira, na classe InfoLeiloes é armazenada uma estrutura Map que associa a cada número de identificação de um leilão, um ReentrantLock. Assim, se um leilão possui o número 1, o valor desta chave neste novo Map diz respeito ao ReentrantLock associado a este leilão.

Todos os métodos que tenham como fim alterar a estrutura que armazena os leilões, têm que passar pelos locks primeiro. Criaram-se assim locks hierárquicos que permitem um melhor desempenho e controlo de concorrência. Sempre que se adiciona ou retira um leilão, é preciso obter os locks de todos os leilões, visto que é preciso garantir a coerência dos dados armazenados. Quando se efetua uma licitação, apenas o lock do leilão em questão é adquirido, uma vez que não faria sentido parar toda a estrutura sempre que um comprador deseja licitar num item. Se assim fosse, o desempenho da aplicação degenerava muito rapidamente. Assim, sempre que uma licitação é feita, o leilão em que se está a licitar é bloqueado a outras threads, de modo a evitar que licitações simultâneas sejam feitas e situações de erros ocorram.

Por sua vez, quando se lista os leilões pretende-se obter um resultado o mais verdadeiro possível, sem que toda a estrutura fosse afetada. Assim, o grupo considerou duas resoluções, optando pela segunda. A primeira baseia-se em adquirir os locks de todos os leilões, ficando toda a estrutura bloqueada a outras threads por uns breves momentos, uma vez que sempre que se adquiria a informação necessária sobre um leilão (verificar se o cliente que requereu a lista é o vendedor do item e se detém a licitação mais alta), era libertado o seu lock associado. Basicamente era feito inicialmente um lock a todos os leilões e, depois, sucessivamente era feito um unlock à medida que se fazia a travessia do Map. A segunda alternativa, a que foi efetivamente escolhida, baseia-se na ideia de fazer lock a um leilão de cada vez, isto é, à medida que se atravessa o Map, faz-se lock a um leilão, adquire-se a informação necessária e, logo de seguida, procede-se à libertação do lock. O grupo achou que esta solução seria a que teria um melhor desempenho, embora ambas fossem soluções válidas.

LeilaoThread e NotifyThread

A thread LeilaoThread encarrega-se de executar todas as funções propostas da aplicação. Se o cliente se conseguir autenticar, a LeilaoThread cria três novos objetos para esse cliente: SharedCondition, CounterNotes, e um Array de Strings que representa uma OutBox. Cada LeilaoThread tem um Map com o utilizador (chave) e cada um destes objetos (valor). Com o login, cada novo objeto é colocado no map correspondente. Além disto, o login ainda inicia uma nova thread, a NotifyThread, capaz de enviar mensagens ao cliente.

O cliente recebe dois tipos de mensagens destas duas threads. A LeilaoThread envia respostas do sistema, enquanto que a NotifyThread envia as mensagens guardadas na OutBox.

A OutBox consiste num Array de Strings que contém as mensagens correspondentes ao fim de um leilão. Quando um vendedor encerra um leilão, tem de ser enviada uma notificação a todos os compradores que licitaram nele. No entanto, como esta mensagem não é uma resposta do sistema, mas só ocorre quando o vendedor desejar, o cliente não está à espera dela. Assim, essa mensagem é enviada para a OutBox do cliente e será lá armazenada até o cliente a desejar ler. É daqui que surge a necessidade da criação de uma thread auxiliar que só será acordada quando for necessário ler mensagens da OutBox.

A classe CounterNotes possui um número que representa o número de mensagens que o cliente tem na sua OutBox. Esta classe vai ser importante para o tratamento de clientes lentos.

A classe SharedCondition possui um ReentrantLock e uma Condition, que servirão para o cliente acordar a NotifyThread correspondente, quando quiser ler as notificações. Era possível não implementar esta classe, no entanto o funcionamento da aplicação seria muito ineficiente, pois quando fosse necessário acordar a thread certa, ele teria de acordar todas. Assim, a Condition é necessária para o cliente se certificar que acorda a sua NotifyThread.

Assim, quando a NotifyThread é iniciada, ela verifica se há mensagens na OutBox para enviar. Como no início não há mensagens, a thread vai adormecer. Através do método dumpNotifications será enviado um sinal à NotifyThread do cliente, de modo a acordá-la para ela enviar as mensagens da OutBox ao cliente. Quando é acordada, verifica novamente se há mensagens para ler na OutBox. Se não houver, volta a adormecer. Se existirem, é necessário verificar que o cliente não está lento. Se não for lento, começa a

imprimir as mensagens contidas na OutBox (para o socket do cliente) e coloca o valor da CounterNotes a zero, significando que não há mensagens que possam ser lidas. Como não há mensagens para ler, a thread adormecerá até ser acordada novamente. É importante realçar que no método dumpNotifications da LeilaoThread foi necessário fazer o lock da variável contida na SharedCondtition, pois, de outra maneira, o sinal não seria enviado, colocando a aplicação num deadlock.

As mensagens que surgirão na OutBox vão ser inseridas através do método terminarLeilao da LeilaoThread. Quando o vendedor termina o leilão, as mensagens de término têm de ser enviadas para o utilizador correto, por exemplo, o vencedor tem de receber uma notificação indicando que ganhou o leilão, enquanto que qualquer outro licitador receberá uma simples notificação indicativa do fim do leilão. Para colocar a mensagem na OutBox do licitador é necessário fazer lock através da SharedCondition, pois, de outra maneira era possível que a terminarLeilao (em duas threads diferentes) estivesse a escrever duas mensagens diferentes ao mesmo tempo na mesma OutBox. Fazendo o lock, obrigamos a que seja escrita uma mensagem de cada vez, prevenindo erros de escrita. Entre o lock e o unlock, além de se inserir a mensagem na OutBox, também é necessário incrementar o valor da CounterNotes, para o cliente ter conhecimento do número de mensagens que tem para ler.

Relativamente à comunicação com o cliente, ambas as threads, ao serem inicializadas, receberão o Socket do cliente, de modo a conseguirem comunicar com este. A NotifyThread só irá enviar mensagens ao cliente, enquanto que a LeilaoThread tem de ler e escrever no socket do cliente. Assim, a função principal do servidor é inicializar as estruturas partilhadas e inicializar a LeilaoThread quando um cliente entra no sistema. A partir daí, o servidor não interage mais com o cliente, sendo que é a LeilaoThread que vai ler e executar os comandos inseridos pelo cliente.

OutBox

Durante o desenvolvimento da aplicação, surgiu a necessidade de notificar cada um dos licitadores de um leilão sobre o seu término. Assim, criou-se a distinção entre respostas do servidor e notificações.

As primeiras são imediatamente enviadas ao cliente pela LeilaoThread, não sendo, então, armazenadas na OutBox do cliente para serem enviadas pela NotifyThread, encarregue de enviar as notificações para o cliente, sempre que solicitado. As segundas distinguem-se por serem avisos de algo que ocorreu nos leilões, independente do que é solicitado diretamente pelo cliente. Estas só são enviadas ao cliente, quando este o pedir, sendo, até então, armazenadas na sua OutBox.

Desta forma, cada cliente tem a si associado a sua OutBox, armazenada numa estrutura comum às threads de outros clientes, pois são estas que, caso o seu respetivo cliente termine um leilão, ficam responsáveis por adicionar a nova notificação na OutBox de cada um dos licitadores.

Distinguem-se aqui dois tipos de mensagens a serem colocadas na OutBox: uma mensagem de vitória, caso o licitador possua a licitação mais alta e uma simples mensagem de término caso o leilão termine e o licitador não detenha a melhor licitação.

Lembrete

No decorrimento das OutBoxes, surgiu a ideia de relembrar o cliente de quantas notificações armazenadas na OutBox tinha ainda por ler, pois, uma vez criada a opção de ler as notificações no menu principal, o cliente não tinha possibilidade de saber se estava ou não a receber notificações sobre os leilões em que tinha licitado. Assim, no fim de cada método é feito um rastreio do número de mensagens que estão na OutBox do cliente, que é, de seguida, enviado-lhe sob a forma de lembrete, mensagem com uma forma especial, sendo cercada por “---”, de forma a destacar-se das respostas do servidor e das notificações.

Foi decidido que a OutBox iria ter um tamanho máximo, ou seja, só poderia armazenar um determinado número de mensagens até o sistema decidir fechar a ligação do cliente. Assim, o lembrete funciona, igualmente, de forma a notificar o cliente do número de mensagens armazenadas, devendo este ter o cuidado de não deixar esse número aumentar demasiado, lendo, preferencialmente, de forma periódica as notificações Assume-se, assim, que um cliente que não leia um determinado número de notificações, definido como 15 mensagens, seja lento, sendo forçado a terminar a sua execução.

Desta forma, a cada opção escolhida pelo cliente no menu principal, no fim da visualização dos

resultados requisitados, é-lhe mostrado um lembrete sobre quantas mensagens estão, até ao momento, armazenadas na sua OutBox.

Conclusão

Este projeto prático permitiu ao grupo arranjar soluções para cada problema que surgiu durante o desenvolvimento do mesmo. No entanto, naturalmente, reconhecemos algumas limitações na nossa aplicação, tais como a impossibilidade do cliente fazer o login com o mesmo username em máquinas diferentes e as notificações não surgirem espontaneamente ao cliente, sem este ter que solicitar a sua leitura. A solução escolhida para o problema anterior foi, apesar de tudo, considerada correta pelo grupo.

Contudo, apesar destas limitações, o grupo implementou todas as funcionalidades propostas, de forma completa e robusta, considerando todas as situações em que poderia ocorrer algum comportamento anormal, tais como, deadlocks, race conditions, entre outros.