

Sistemas Distribuídos

**Relatório do Projeto**

**“IdeaBroker – Idea Management and Trading”**



**Trabalho realizado por:**

Andreia Sofia Oliveira Cruz, nº2011154038

Bruno José Borges Madureira, nº2011161942

# Introdução

Uma vez que a forma como as ideias estão organizadas na Web não é a melhor, este projeto pretende organizá-las e permitir aos utilizadores ter uma visão geral num tópico específico assim como uma indicação de quanto vale cada ideia.

Para tema do nosso gestor de ideias decidimos escolher **Música** porque esta é universal, portanto cada pessoa tem o seu gosto e sua opinião, o que faz com que existam as mais diversificadas ideias à volta deste tema.

Aos utilizadores será dada a possibilidade de discutir e trocar ideias entre si, sejam elas concordantes ou contrárias, bem como transacioná-las como se de um mercado de ações se tratasse.

Para a implementação deste gestor criámos uma base de dados onde serão guardados todos os dados: a informação sobre as ideias, o histórico das transações feitas e os utilizadores que estão registados.

# Arquitetura Interna

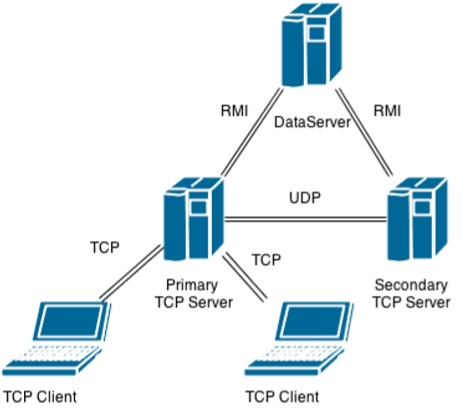


Figura 1

A fig.1 demonstra a solução encontrada para a implementação do gestor de ideias. Como se pode observar existe um servidor RMI, dois servidores TCP, um primário e um secundário, e dois clientes, como exemplo porque podem-se ligar mais.

* Servidor RMI: Neste servidor encontram-se as implementações de acesso à base de dados. Este servidor recebe os pedidos dos clientes, através do servidor TCP que tem implementado um cliente RMI, processa-os acedendo à base de dados e envia a resposta de volta.
* Servidor TCP: Este servidor vai ser uma espécie de interface entre o cliente e o servidor RMI. Vai receber os pedidos do cliente, enviá-los para o servidor RMI tratar, receber a resposta/dados deste e encaminhá-los para o cliente. O servidor secundário vai estar “à escuta” do servidor primário, fazendo-lhe pings através de um socket UDP, para tomar o lugar deste assim que ele deixe de funcionar.
* Cliente TCP: Contém a interface que permite ao utilizador interagir com o gestor de ideias, recebendo os pedidos deste e mostrando a resposta da operação realizada.

# Estruturas de dados

Achámos que a melhor solução para guardar os dados de forma persistente era uma base de dados Oracle, uma vez que uma das características de uma base de dados é oferecer paralelismo. O acesso à base de dados é feito à vez, ou seja enquanto alguém está a aceder mais ninguém o consegue fazer, o que nos assegura a impossibilidade de certos dados serem escritos ao mesmo tempo que outros, impedindo que dados fiquem incompletos ou corrompidos.

A base de dados é constituída por 7 tabelas criadas da seguinte maneira:

1. CREATE TABLE login(name VARCHAR(50), password VARCHAR(50), PRIMARY KEY(name))
2. CREATE TABLE saldos(name VARCHAR(50), creditos NUMERIC(10,3), PRIMARY KEY(name))
3. CREATE TABLE idea\_table(acções\_totais Numeric(10,0), tipo Numeric(2,0),texto VARCHAR(500), idea\_key NUMERIC(15,0),PRIMARY KEY (idea\_key,texto))
4. CREATE TABLE topicos (hashtag VARCHAR(500),topic\_key NUMERIC(15,0),PRIMARY KEY(topic\_key))
5. CREATE TABLE i\_t (idea\_key NUMERIC(15,0),topic\_key NUMERIC(15,0), PRIMARY KEY(idea\_key, topic\_key))
6. CREATE TABLE shares\_table (nome Varchar(500),acções Numeric (10,0),idea\_key NUMERIC(15,0), preco NUMERIC(10,3),Primary key(nome,idea\_key))
7. CREATE TABLE historico(vendedor VARCHAR(500),comprador VARCHAR(500),acções numeric(10,0),preco Numeric(10,3),idea\_key NUMERIC(15,0),n\_compra NUMERIC(15,0), PRIMARY KEY(vendedor,comprador,idea\_key,n\_compra) )

A tabela **login** guarda o registo do utilizador (nome e password de login) tendo como chave primária o nome.

A tabela **saldos** guarda informação sobre a quantidade de créditos de cada utilizador, sendo o nome a chave primária.

A tabela **idea\_table** guarda toda a informação de cada ideia introduzida: nº total de ações para venda, tipo da ideia – favorável (1), contrária (-1) ou neutra (0) – em comparação com outra, texto explicativo da ideia e uma chave identificadora, única a cada uma. Neste caso a chave primária é a chave identificadora, pois esta não pode ser repetida, seguida da descrição da ideia.

A tabela **topicos** guarda todos os tópicos criados usando uma hashtag e, de maneira semelhante às ideias, uma chave identificadora única para cada tópico que será usada como chave primária.

A tabela **i\_t** guarda todas as correspondências entre tópicos e ideias através das chaves identificadoras de ambos. Isto porque uma ideia pode ter vários tópicos e um tópico pode ter várias ideias.

A tabela **shares\_table** guarda informação sobre todas as ações de cada ideia compradas por um determinado utilizador, incluindo o preço pago. A chave primária é o nome do utilizador seguido da chave da ideia.

A tabela **historico** guarda o histórico de todas as transações efetuadas: quem vendeu, a quem vendeu, quantas ações vendeu, por quanto vendeu cada ação, a que ideia pertencem as ações compradas e o número da compra para diferenciar todas as compras do mesmo utilizador isto para, caso o utilizador compre exatamente as mesmas ações da mesma ideia conte como uma nova compra.

Em ambos os servidores, TCP e RMI, temos as classes **Bd**, **Entrada\_historico**, **Topico**, **Idea**, **SharesTableCell** que estão implementadas da mesma maneira em ambos porque uma vez que a chamada ao servidor RMI é feita remotamente este necessita de saber o que retornar. Estas classes, que têm informação sobre tópicos, ideias, ações, utilizadores e histórico de transações, implementam o **Serializable** para que esses objetos possam ser guardados e por conseguinte enviados para outra máquina através da rede.

No servidor TCP temos também as classes **Servidor\_TCP**, **UDP** e **Elementtokenizer**. Esta últimafoi criada para distinguir os vários tópicos a que uma ideia pode pertencer, utilizando **# (hashtags)**, sendo assim cada token será um tópico.

O RMI tem ainda a classe **Rmi\_teste\_server** e, como dito anteriormente, a implementação da base de dados na classe **Bd\_implementation**.

# Especificação de protocolos

**UDP**

* O UDP é um protocolo bastante simples que procura oferecer às aplicações um serviço básico de entrega de pacotes. Este serviço consiste em colocar a mensagem recebida dentro de um segmento, utilizando os números das portas para identificar as aplicações. Este protocolo é pouco fiável não sendo orientado para conexão. Não existem técnicas no protocolo para confirmar que os dados chegaram ao destino corretamente, assim como a entrega pode ser feita fora de ordem e os dados perdidos. Por outro lado, a ausência de estruturas de controlo complexas, como no TCP, garante ao UDP alta eficiência, já que cada pacote é, praticamente, composto só pelos dados. O UDP é ideal para fazer a transmissão de dados poucos sensíveis ou para comunicação sem conexão ou tradução de endereços por DNS. É por estas razões que é usado o UDP, em vez do TCP, entre os servidores primário e secundário pois este é mais rápido e não há necessidade de controlos de fluxo ou de erros para uma operação tão simples como a de enviar um datagram pelo socket de tempos a tempos.
* A classe **UDP** cria um socket UDP que liga os dois servidores TCP. Sendo assim um servidor envia um datagram por esse mesmo socket e depois fica à escuta durante um determinado tempo (no nosso caso 1 segundo). Se não estiver nenhum servidor a trabalhar o que enviou o datagram torna-se no servidor principal, se já estiver um a trabalhar então este espera até o outro deixar de funcionar e passa a ser o servidor principal.

**TCP**

* O protocolo TCP é o protocolo de transporte mais utilizado na Internet. Este protocolo procura tornar a rede, por onde os dados são transmitidos, fiável. Os dados, quando transmitidos, podem ser perdidos e/ou chegar fora de ordem ao destino. Por esta razão, as funções do TCP são: garantir que todos os dados transmitidos são recebidos e que os pacotes, ao chegar ao recetor, fiquem pela ordem correta. Para que estas funções se possam realizar, o protocolo precisa de se “manter informado”. Assim, vai criar uma conexão entre a máquina emissora e a máquina recetora antes de serem transmitidos os dados.

Uma conexão TCP é “full-duplex”, ou seja, podem-se transmitir dados nos dois sentidos. Assim que a máquina recetora receber o pacote da emissora envia a esta um “acknowledge” que tem como principal função avisar o emissor que o pacote de informação que enviou chegou sem alterações. Caso se verifiquem modificações o emissor reenvia o pacote. Como existe a possibilidade de os pacotes chegarem ao destino desordenados o TCP resolve o problema enumerando os pacotes que transmite permitindo ao recetor ordenar os pacotes quando os recebe. Ainda correndo o risco de os pacotes ficarem perdidos na rede e não serem entregues ao destinatário, o TCP contém um controlo de erros. Para cada pacote transmitido o TCP inicia um temporizador e cada pacote tem de ser confirmado como recebido antes que este expire. Se o tempo expirar e o pacote não for recebido este é retransmitido pelo próprio TCP. Ou seja, enquanto não existe a confirmação de entrega, o TCP guarda em “buffer” todos os pacotes prontos a serem reenviados caso necessário. Uma vez que na nossa aplicação não convém haver perdas ou corrupção de dados entre o cliente e o servidor o TCP é o melhor protocolo a utilizar.

* A classe **Servidor\_TCP** implementa, em grande parte, a interface que vai ser apresentada ao utilizador, assim como é responsável pela ligação TCP entre si e o cliente e por chamar a classe **UDP** para a criação do socket UDP.

**RMI**

* Como o próprio nome indica, o RMI permite ao programador invocar  métodos de objetos remotos, ou seja que estão alojados em máquinas virtuais Java distintas. De certa forma o programador pode desenvolver aplicações totalmente distribuídas como se de aplicações locais se tratassem, sendo quase toda a comunicação entre máquinas virtuais Java assegurada transparentemente pelo próprio RMI.

O sistema RMI esconde do programador um grande conjunto de operações e recursos que estão envolvidos no processo de invocação. Decompondo o sistema RMI é possível identificar 3 camadas que vamos listar por ordem de proximidade ao programador:

* A camada de *stub*/*skeleton*, responsável por receber as chamadas da aplicação cliente feitas à interface e por reencaminhá-las para o objeto remoto.
* A camada de Referências Remotas (Remote Reference Layer), que lida com a gestão e com a interpretação das referências remotas.
* A camada de Transporte, que assegura a ligação entre as máquinas virtuais através de TCP/IP.
* **Camada *Stubs* e *Skeletons***

A camada mais próxima do programador, ou seja da aplicação cliente e do objeto remoto, é a camada *Stubs*/Skeletons. Os *Stubs* são classes usadas do lado da aplicação cliente e funcionam como proxies entre a aplicação cliente e o objeto remoto. Os *Stubs* recebem os parâmetros dos métodos exportados pelo objeto remoto (definidas pela interface da classe remota) e reencaminham-nos para o servidor onde serão interpretados por uma instância de uma classe Skeleton. O Skeleton recebe os parâmetros enviados pelo Stub e executa as respetivas chamadas no objeto remoto. Em sentido inverso, os *Skeletons* são também responsáveis por receber o valor de retorno do método remoto (local na sua perspetiva) e direcioná-los para os *Stubs* dos clientes correspondentes.

* **Camada de Referências Remotas**

Esta camada mantém as referências entre os clientes e os objetos remotos e estabelece a semântica da ligação RMI. As referências mantidas referem-se a ligação unicast, ou seja um proxy para um objeto remoto (i.e. um *Stub* para um *Skeleton*). Esta camada funciona como um router entre o cliente e os (eventuais) vários objetos remotos. Por enquanto, o RMI não suporta diretamente outras semânticas de comunicação como por exemplo multicast.

* **Camada de Transporte**

Esta camada lida diretamente com a comunicação entre as várias JVM's, usando TCP/IP. É importante referir que mesmo que as JVM's sejam executadas no mesmo computador, o RMI recorre ***sempre*** à comunicação TCP/IP. Isto significa que é sempre necessário possuir uma interface de rede funcional para se poder utilizar RMI (mesmo tendo a aplicação cliente e a aplicação servidora a executar no mesmo computador).

## **Serviço de Registo / Naming**

## Para que seja possível a uma aplicação cliente a utilização de um determinado serviço remoto é necessário que esse mesmo serviço seja de alguma forma encontrado na rede. O RMI utiliza um sistema relativamente simples para isso, recorrendo a um Serviço de Registo para esse efeito. Todas as aplicações servidores que desejem tornar alguns dos seus objetos acessíveis remotamente deverão registar esses mesmos objetos no serviço de registo com um nome conhecido das aplicações clientes.

## O Serviço de Registo deverá estar disponível numa localização  preestabelecida (i.e. endereço da máquina e porta de rede) de forma a ser conhecido por todos os eventuais clientes. Qualquer aplicação cliente que pretenda aceder a um determinado objeto remoto (já registado) terá de consultar um ou mais Serviços de Registos em busca do nome com o qual o serviço remoto foi registado. No caso de esse serviço (i.e. nome) ser encontrado o Serviço de Registos retorna um objeto Stub através do qual se torna possível chamar os métodos do objeto remoto. O RMI inclui no conjunto de aplicações que lhe está associado uma implementação simples de um Serviço de Registos, o *rmi registry*.

* A classe **Rmi\_teste\_server** é responsável por implementar a base de dados e por todas as operações relacionadas com esta. É também aqui que se chama (lookup) o *rmi registry* para registar todos os métodos que irão ser chamados pelo cliente RMI que se encontra implementado no servidor TCP.

# Tratamento de exceções

A ligação ao servidor TCP pode cair, sem que o servidor deixe de trabalhar, ou até mesmo o servidor deixar de responder durante uns segundos mas depois resumir trabalho, são portanto falhas transientes. Estas falhas são transparentes para o cliente pois a aplicação não é fechada, ou seja, o cliente não necessita de fazer o login outra vez para continuar a utilizar a aplicação. A solução faz uso das exceções que acontecem no java aquando da criação do socket de ligação na classe L\_Rclient. O cliente tenta ligar-se ao servidor TCP, se ocorrer alguma exceção tenta ligar-se outra vez ao mesmo servidor criando um novo socket mas se continuar a não conseguir vai tentar ligar-se ao servidor secundário.

As ideias nunca poderão estar duplicadas, mesmo que ocorra uma falha no servidor TCP primário e o secundário resuma trabalho enquanto um cliente está a introduzir uma ideia. Isto é assegurado pela própria base de dados e pelo servidor RMI.

No entanto, não conseguimos implementar a tempo o mesmo processo para quando a ligação entre o servidor TCP e o servidor RMI “caia”, utilizando as exceções aquando o lookup do RMI registry.

# Solução de Fail-Over

A solução encontrada para possíveis falhas, como por exemplo o servidor primário deixar de trabalhar, foi ter um servidor secundário a trabalhar paralelamente que irá tomar o lugar do servidor primário assim que ele deixe de responder.

Esta solução passa por o servidor secundário estar constantemente a enviar pings, usando datagrams UDP, através do socket que liga os dois servidores ficando depois à escuta de uma resposta do servidor primário, outro ping, durante um determinado período de tempo, no nosso caso 1 segundo. Este período de tempo é executado com a ajuda de um temporizador implementado na função timeout(), na classe UDP, que se reinicia enquanto tiver resposta do outro servidor. Portanto, assim que esse período de tempo chegar ao fim e o servidor secundário não tiver recebido nenhuma resposta do primário é porque este deixou de funcionar logo o secundário terá de tomar o seu lugar, continuando as operações que os clientes estavam a realizar.

Depois do servidor secundário tomar o lugar de primário qualquer cliente novo que se queira conectar ao gestor de ideias irá tentar ligar-se ao antigo servidor mas como isso não funcionará, este será ligado ao novo servidor primário.

Se entretanto o servidor primário voltar a funcionar irá tomar o lugar do servidor secundário, ficando à espera que ocorra alguma falha utilizando o mesmo método anteriormente explicado, uma vez que o outro servidor estará a funcionar como primário.

Enquanto estas falhas no servidor acontecem o cliente não é informado, ou seja não sabe que elas ocorreram, nunca está demasiado tempo à espera nem é obrigado a realizar o login outra vez para continuar a utilizar a aplicação. É também assegurada a ordem para compra de ações mesmo que entretanto haja uma falha no servidor TCP primário, isto porque estas operações são asseguradas pelo servidor RMI, que comunica com a base de dados, e uma vez que este é só um a ordem de compra fica assegurada, o primeiro a fazer a compra é o primeiro a ser servido.

A nossa solução não está absolutamente correta pois aquando uma falha no servidor TCP primário o cliente, para se ligar ao servidor secundário, terá de colocar o input duas vezes.

# Descrição de testes

|  |  |
| --- | --- |
| Teste | Passou/Falhou |
| Servidor secundário toma o lugar de primário se servidor primário for a baixo |  |
| Servidor TCP tenta reconectar-se ao RMI quando este deixar de funcionar |  |
| Proteção contra SQL injection |  |
| Cliente liga-se ao servidor secundário quando primário deixa de funcionar |  |
| Utilizador consegue registar-se |  |
| Utilizador consegue fazer login com os seus dados |  |
| Utilizador consegue criar um tópico |  |
| Utilizador consegue ver uma listagem de tópicos previamente criados |  |
| Utilizador consegue inserir uma ideia, com relação a outra |  |
| Utilizador consegue apagar uma ideia |  |
| Utilizador consegue visualizar uma listagem com um histórico de transações realizadas |  |
| Utilizador consegue mudar o preço das ações de uma ideia sua |  |
| Utilizador consegue comprar ações de uma ideia |  |
| Servidores e clientes correm em máquinas diferentes |  |
| Não há ideias duplicadas |  |
| Servidores conseguem fazer ping um ao outro através de socket UDP |  |
| Ordem de compras assegurada, mesmo em caso de falha no servidor TCP |  |

Tabela 1