Universidade de Coimbra

Faculdade de Ciências e Tecnologia

Licenciatura em Engenharia Informática

*1º Trabalho Prático – Sockets, RMI e Sincronização de Servidores*

*Sistemas Distribuídos, 1º Semestre*



**Trabalho realizado por:**

* **Ivo Correia nº 2008110814**
* **João Barbosa nº 2008111830**

Coimbra, 25 de Outubro de 2010

# Índice

[Índice 1](#_Toc275360176)

[Introdução 2](#_Toc275360177)

[Principais Estruturas do Sistema 4](#_Toc275360178)

[Clientes (TCP e RMI) 4](#_Toc275360179)

[Servidores 5](#_Toc275360180)

[Suporte à Persistência de Dados 6](#_Toc275360181)

[Comunicação Entre Servidores 8](#_Toc275360182)

[Esquema de Troca de Mensagens 8](#_Toc275360183)

[Interacção com os Clientes Activos 11](#_Toc275360184)

[Distinção entre Clientes Online e Offline 11](#_Toc275360185)

[Envio de Mensagens para Clientes 12](#_Toc275360186)

[Envio de Comandos para o Servidor 13](#_Toc275360187)

[Iniciar uma Nova Sessão 15](#_Toc275360188)

[Registar um Novo Cliente 15](#_Toc275360189)

[Efectuar o Login 15](#_Toc275360190)

[Realização de Apostas 17](#_Toc275360191)

[Gerar uma Nova Ronda 17](#_Toc275360192)

[Apostar em Partidas 17](#_Toc275360193)

[Especificações de Testes 19](#_Toc275360194)

[Cenário de Duplo Servidor Primário 19](#_Toc275360195)

[Efectuar o Login 19](#_Toc275360196)

[Conclusão 20](#_Toc275360197)

[Cenário de Duplo Servidor Primário 20](#_Toc275360198)

[Bibliografia 21](#_Toc275360199)

# Introdução

O primeiro trabalho prático da disciplina de Sistemas Distribuídos consistia na implementação de uma aplicação distribuída cliente-servidor, o qual permitia gerar jogos de futebol, os seus resultados, permitir que os clientes façam as suas apostas e distribuir prémios dos apostadores vencedores.

De modo a alcançar tal objectivo, são instalados dois servidores que têm como uma das funções comunicar entre si de modo a evitar que nem todos os clientes estejam ligados ao mesmo servidor. Por isso, trocam mensagens entre si para eleger um servidor primário e em caso de falha deste, o servidor secundário tem de obrigatoriamente tomar esta posição para que os clientes possam continuar a fazer as suas apostas.

Do lado dos clientes, temos dois tipos de aplicações: ligações através de *sockets* TCP e Java RMI. Em termos de interface com o utilizador, ambas as abordagens são em tudo semelhantes. Quanto às suas aplicações, logicamente que as duas são internamente bastante diferentes. Contudo, é da responsabilidade do servidor conseguir atender simultaneamente os dois tipos de clientes sem que estes criem conflitos entre si.

De modo a auxiliar a execução do trabalho, foi planificado no enunciado quatro etapas para a realização do projecto, a saber: implementação dos *sockets* TCP, implementação do Java RMI, tratamento de excepções e falhas de comunicação entre servidor e clientes e por último, criação de mecanismos de protecção contra falhas num dos servidores.

No entanto, não seguimos exactamente esta ordem, uma vez que apenas sabíamos trabalhar de antemão com *sockets* TCP e ligação UDP. Assim, começámos pela implementação destes dois pontos, de modo a ligar um cliente ao servidor. Depois, não tendo ainda leccionado Java RMI nas aulas teóricas, partimos para o estabelecimento de mecanismos de modo a seleccionar o servidor primário e garantir que este fosse substituído no caso de falhar da máquina. Por fim, para concluir, partimos para o Java RMI.

Em paralelo a cada passo, fomos implementando o controlo de excepções pelo cliente sempre que possível para as operações que íamos gradualmente implementando.

Antes de iniciar o projecto em concreto, foram desenhados diagramas e apontados os principais problemas com que nos iríamos deparar. Nesta fase, observámos que as maiores dificuldades seriam a coordenação entre servidores e garantir a persistência dos dados; assegurar que nenhuma informação fosse perdida e no caso de ser impossibilitado de trocar a mesma, avisar o cliente da falta de ligação; permitir a interoperabilidade dos diferentes tipos de clientes.

(Depois do resto concluído, fazer uma breve análise de como se vai dividir o relatório).

Como nota final, esclarecer um ponto. Muitas vezes referimo-nos rapidamente aos clientes como clientes TCP ou RMI. Assim, os primeiros usam directamente como implementação interna a API de *sockets* TCP disponibilizada pela linguagem Java enquanto os segundos, estão num nível de implementação superior e recorrem ao Java RMI, não esquecendo que apesar de também usarem *sockets* TCP, tal procedimento é transparente para o programador.

Estas denominações têm apenas em vista tornar a leitura mais fluida e leve, não querendo nós de deixar de ser rigorosos na redacção deste relatório.

# Principais Estruturas do Sistema

Neste capítulo, vamos fazer uma breve apresentação das principais estruturas do sistema, para posteriormente ser possível explicar como se interligam de modo a garantir a comunicação entre clientes e servidores.

Dividimos o capítulo principal em três subcapítulos, nomeadamente os clientes, servidores, concluindo com uma breve referência ao suporte para manutenção dos dados utilizado.

## Clientes (TCP e RMI)

Tal como requerido pelo enunciado, era necessário implementar dois tipos de clientes no nosso sistema, os que se ligavam por *sockets* TCP e os que utilizavam RMI. Como começámos a implementação pelos clientes TCP, também serão eles o primeiro tipo de clientes que iremos discutir.

Um cliente TCP é constituído por dois tipos de *threads* (designadamente as classes *TCPClient* e *ClientWriteTCP*), cada uma com uma função distinta mas que trabalham em conjunto e dependem do contributo da outr.

Inicialmente, é lançada a *thread TCPClient* que tem como objectivo estabelecer e manter a ligação com o servidor, sendo que também fica responsável por efectuar as operações de registo e *login* (a discutir mais tarde, TODO: no capítulo x), e mais tarde, ler a partir do *socket*.

Após arrancar, esta *thread* fica encarregue de lançar a outra, mantendo-a em espera com recurso a uma terceira classe, *ConnectionLock*, que apenas funciona como um semáforo, informando se a ligação está activa ou não. A *threads* já lançada fica à espera que a *thread* principal consiga ligar-se ao servidor, recorrendo ao *wait*.

Durante esta espera, *TCPClient* liga-se através de um *socket* TCP ao servidor, tendo como resultado dois necessários possíveis:

* A ligação é estabelecida com sucesso e depois de passar a referência do *socket* aberto para a *thread* de escrita, acorda-a e inicia o processo de registo ou *login*. Se este estiver concluído, então verifica se há mensagens em ficheiro de uma anterior sessão que precisam de ser enviadas e findo o processo, começa a ler o *socket*, estando protegida no caso da ligação falhar.
* Se houver problemas com a ligação, então vai ser gerada uma excepção e a *thread* filha continua adormecida (partindo do princípio que esta é a primeira ligação efectuada). Neste momento, a *thread* principal espera um dado período de tempo, antes de tentar de novo.

Se ao fim de um dado número de tentativas (configurável) o servidor continuar a não dar resposta, então o cliente irá mudar-se para o segundo endereço que tem em espera e tentar ligar-se ao segundo servidor.

Como notas finais, falta acrescentar que se o segundo servidor também não der resposta, a sessão é terminada. Por outro lado, se o cliente ainda não tiver tentado ligar-se a nenhum servidor e a resposta é logo negativa, ele tenta imediatamente ligar-se ao segundo servidor antes de voltar ao primeiro.

Uma vez a funcionar, *TCPClient* tem como função ler toda a informação do *socket* que é naturalmente enviada pelo servidor, imprimindo mensagens caso seja necessário.

Por seu lado, *ClientWriteTCP* tem a função complementar, isto é, escrever no *socket* toda a informação que a aplicação cliente deseja enviar para o servidor. Mesmo quando o servidor está em baixo, esta *thread* tem de se manter activa, pois um dos requisitos do enunciado é que todas a mensagens que sejam enviadas utilizando o comando ‘*send*’ num período *offline*, sejam guardadas para um *buffer* e enviadas mais tarde, quando a ligação for restabelecida.

Inicialmente, tínhamos três *threads* para o cliente. No entanto, apercebemo-nos que teríamos sempre uma *thread* inactiva. Deste modo, juntámos a *thread* responsável pela manutenção da ligação e a *thread* de leitura do *socket*, pois estas duas actividades nunca se sobrepõe.

Com o cliente TCP apresentado, passemos então à versão RMI.

A versão RMI naturalmente que partilha muitos dos conceitos dos clientes TCP, mas, como seria de esperar, acaba por ser uma versão simplificada e naturalmente, requer um menor número de *threads* e menos linhas de código.

Assim sendo, construímos as classes *RMIClient* (que implementa a interface *ServerOperations*) e *RMIWriter*.

Em analogia com o cliente TCP, *RMIClient* fica responsável pelo estabelecimento da ligação com o servidor (seguindo um método em tudo igual ao anteriormente descrito para os clientes TCP), pelos métodos de registo e *login* de clientes e pela verificação da existência de mensagens guardadas em ficheiro que necessitem de ser enviadas assim que possível.

Por seu lado, o *RMIWriter* é quase como um agregado das *threads* *ClientWriteTCP* e *ClientReadTCP*. Tal como *ClientWriteTCP*, esta *thread* fica responsável por ler os comandos inseridos pelo utilizador e dos processar para o servidor. No entanto, como no Java RMI a chamada de métodos remotos é muito semelhante à chamada de métodos locais, a resposta do servidor vem como um objecto devolvido pelo método invocado. Deste modo, não é necessário criar uma nova *thread* para ler as respostas do servidor, sendo que *RMIWriter* vai ler essas mesmas mensagens imprimi-las na consola.

Como nota de rodapé, se o leitor estiver interessado em saber quais os comandos que são trocados entre clientes e servidor, aconselhamos vivamente a consultar o manual de utilizador que acompanha este relatório. Analisá-los um a um de novo aqui seria redundante e por isso, optamos pelos restringir ao manual.

## Servidores

No nosso sistema, temos dois servidores que executam código igual, apenas diferindo nos valores que são dados aos portos de comunicação. Como é requerido pelo Java RMI, os servidores implementam a interface *ClientOperations* que permite saber quais os métodos disponibilizados por um cliente RMI.

Quando um servidor inicia, começa uma troca de mensagens com o seu companheiro (TODO: ver capítulo x) e findo o processo de eleição de qual dos dois deve tomar a posição de servidor primário, são aceites ligações por parte dos clientes.

Analisando o código, podemos observar que a classe fulcral em todo o processo é a classe *Server*, uma vez que é ela que fica responsável em iniciar todas as outras *threads* que a irão auxiliar no atendimento aos clientes e manutenção da conectividade.

Depois de preenchidas as variáveis relativas aos diferentes portos utilizando os parâmetros passados como argumentos, a *thread* principal lança a *thread* *ConnectionWithServerManager* que, tal como explicamos no TODO capítulo x, inicia ‘conversações’ com o outro servidor, esperando num *lock* (*ConnectionLock*) em tudo igual ao utilizado pelos clientes de modo a sincronizarem as suas *threads*. Depois de concluído o processo, temos duas possibilidade que analisamos separadamente:

* Este servidor foi eleito como servidor primário. Neste caso, pode então abrir o *socket* TCP ao qual os clientes se podem ligar; faz os registos dos objectos remotos necessários para o funcionamento do RMI e por fim, fica à escuta no porto definido para o estabelecimento de ligações TCP. Sempre que um cliente se liga, é criada uma nova *thread*, *TCPClientThread*, que toma conta da interacção com este cliente e liberta o servidor principal para que este possa estabelecer contacto com novos clientes.
* No caso de ser nomeado servidor secundário, fica preso num *wait* e deste modo, impossibilitado de atender clientes que se tentem ligar. Apesar de aparentemente inactivo, nem todas as *threads* ficam adormecidas. Mais concretamente, as *threads* responsáveis pela comunicação entre servidores nunca param, de modo a poderem avisar a *thread* principal que houve algum tipo de falhas e este servidor é agora o primário.

Em qualquer um dos casos, quando é criado o objecto a partir da classe *Server*, são também inicializados os objectos relativos às classes *ActiveClients* (responsável pela manutenção das listas de clientes activos no sistema) e *GlobalDataBase* (que gere a informação que necessita de ser guardada de forma persistente, como a lista de todos os clientes registados no sistema ou jogos que estão a decorrer).

O objecto relativo à classe *BetScheduler* (responsável pela criação dos jogos, análise das apostas e por guardar, comunicando com *GlobalDataBase*, toda esta informação imprescindível) só é criado se existir confirmação de que se trata do servidor primário.

# Suporte à Persistência de Dados

Como já foi referido brevemente na secção anterior, a classe *GlobalDataBase* é a responsável de guardar toda a informação vital de forma persistente, de modo a não haver inconsistências e possibilitar ao servidor secundário ter toda a informação actualizada se eventualmente for promovido.

Deste modo, *GlobalDataBase* é constituído por uma *hash table* que guarda todos os clientes registados no sistema. Cada entrada na tabela é constituída por um objecto da classe *ClientInfo*, onde guardamos as informações relativas ao nome de utilizador, palavra-chave e endereço electrónico, assim como o número de créditos que o cliente possui.

Optámos por esta estrutura de dados porque em nenhuma ocasião nos é pedido que listemos todos os clientes presentes no sistema. Deste modo, podemos fazer uma procura rápida e eficaz sempre que necessitemos de actualizar o número de créditos de um dado utilizador ou se umnome de utilizador já está ocupado. Esta *hash table* é sempre inicialmente lida a partir de um ficheiro e sempre que se regista uma alteração, é novamente escrita para o mesmo ficheiro.

Em relação aos jogos a decorrer, sempre que é criado uma nova ronda, o *BetScheduler* transmite essa informação à *GlobalDataBase* e esta escreve em ficheiro a listagem de todos os desafios que estão a decorrer, assim como o número correspondente ao primeiro jogo da jornada. Não guarda contudo, o tempo que já passou desde o início da ronda e desse modo, quando o servidor secundário ler as informações relativas a estes mesmos jogos, vai ter de partir do pressuposto que a ronda tinha começado precisamente naquele momento. Daí, no limite e em caso de falha, uma ronda poderá durar o dobro do tempo atribuído a uma ronda regular.

As apostas feitas até ao momentos pelos clientes para uma dada ronda também são guardadas em memória, mas neste caso, fica o *BetScheduler* directamente responsável por essa tarefa.

# Comunicação Entre Servidores

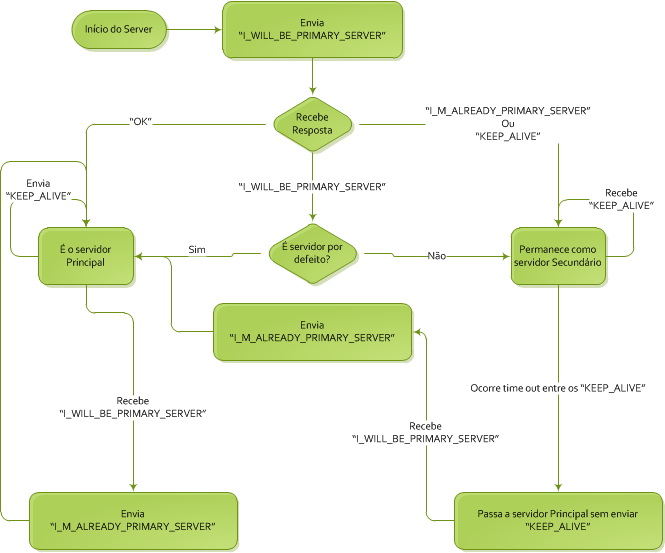
Num sistema distribuído, é essencial garantir que os serviços não fiquem indisponíveis apenas porque o servidor se encontra parado. Deste modo, as máquinas são replicadas de modo a garantir um maior grau de confiança.

Contudo, é preciso que haja uma coordenação exímia entre todos os servidores para não haver competição entre os mesmos, partilha de informação errada ou inconsistente ou mesmo levar a uma falha geral na rede. É desta comunicação entre servidores que discutimos no capítulo seguinte.

Como nota antes de começar a discussão, convém esclarecer que os servidores comunicam entre si recorrendo a uma ligação UDP de modo a não sobrecarregar a rede. Contudo, há o revés da medalha que não são feitas garantias de entrega de mensagens e daí, terem de ser seguidos alguns passos em baixo descritos.

## Esquema de Troca de Mensagens

O diagrama apresentado em seguida foi o esquema desenhado por nós para representar a troca de mensagens entre servidores. Apesar de tentar ser o mais explícito e compreensível possível, achámos por bem fazer uma breve descrição do protocolo que ele pretende representar. Este mesmo protocolo está construído de modo a não ser necessário guardar o estado do sistema antes da ocorrência de qualquer tipo de falha no lado do servidor.



*Diagrama 1 - Protocolo de eleição do servidor primário.*

Quando o servidor é lançado, podemos observar a a criação de *thread*, mas concretamente o objecto da classe *ConnectionWithServerManager*. Por seu lado, esta *thread* cria um outro fluxo, *ReceiveServerMessages*, objecto que fica responsável por escutar no porto indicado para a chegada de mensagens vindas do outro servidor e informar do seu conteúdo à *thread* que a criou. De acordo com o tipo de mensagens recebidas, o *ConnectionWithServerManager* pode actuar de acordo com a situação.

De modo a desencadear todo o processo, o *ConnectionWithServerManager* envia uma mensagem inicial ‘I\_WILL\_BE\_PRIMARY\_SERVER’ para o outro servidor e é accionado um temporizador. Se ocorrer um *timeout*, então o processo é repetido um dado número de vezes, de modo a permitir falhar temporárias na rede.

Se o servidor receber como resposta ‘I\_M\_ALREADY\_PRIMARY\_SERVER’, então concluímos que o outro servidor já está operacional e somos então nomeados servidor secundário.

Se por seu lado recebermos um ‘OK’, encontramo-nos numa situação que eventualmente vai ocorrer poucas vezes: significa que este servidor caiu, mas conseguiu recuperar a tempo, de modo que o servidor secundário nunca tenha chegado a detectar esta falha. Neste caso, continuamos como servidor primário.

Se por seu lado receber como resposta a mesma mensagem que acabou de enviar, ‘I\_WILL\_BE\_PRIMARY\_SERVER’, quer dizer que os servidores estão a ser iniciados quase simultaneamente. De modo a garantir que apenas um seja o servidor principal, é definido anteriormente por defeito, qual dos dois deve tomar a posição de servidor primário nesta situação.

Pode ainda ocorrer o caso do servidor secundário falhar. Nesse caso, como resposta, receberemos um ‘KEEP\_ALIVE’. Assim, apenas temos de tomar consciência que o outro servidor se encontra activo e daí, ficamos de novo como servidor secundário.

Por fim, se ao fim de todas as tentativas, o companheiro não der sinais de vida, então assumimos que se encontra inactivo e tomamos a posição de servidor primário (contudo, prestar atenção ao TODO capítulo STONITH, onde encontrámos a solução para o problema de termos dois servidores primários).

Terminado este processo, deparamo-nos uma vez mais, com duas possibilidades distintas:

* Se formos nomeados servidores primários, então entramos num ciclo infinito em que a função do gestor de mensagens é simplesmente enviar mensagens do tipo ‘KEEP\_ALIVE’. Como neste ponto é indiferente o que o outro servidor nos comunica, a *thread ReceiveServerMessages* é terminada.

Se o servidor principal detectar que o outro servidor não se encontra activo, para de enviar qualquer tipo de mensagens para não sobrecarregar a rede desnecessariamente.

Quando receber algum tipo de mensagens vindos do outro servidor, retoma a comunicação de forma normal.

* No caso de sermos nomeados como servidores secundários, temos apenas de ficar à escuta num dado porto, à espera de mensagens provenientes do colega, accionando temporizadores que em caso de *timeout*, accionam os dados mecanismos para que este servidor ocupe a posição de destaque. Uma vez mais, tomar em atenção os casos que apenas existem falhas na rede e nos encontramos num cenário em que é necessário garantir que não existem dois servidores primários.

Quando o processo de eleição descrito estiver terminado, este *thread* pode informar o servidor acerca do seu estado actual, desbloqueando-o caso tenha sido considerado o servidor principal.

# Interacção com os Clientes Activos

A interacção entre clientes e servidor é sem dúvida o objectivo máximo do sistema. Sem ele, qualquer das estruturas descritas nos capítulos anteriores, perdem significado se forem vistas isoladamente.

A manutenção das listas de clientes activos e do envio de informação para os mesmos fica a cargo do objecto instanciado a partir da classe *ActiveClients*.

Por seu turno, existe uma *thread* do lado do servidor (*TCPClientThread*) que fica responsável de receber os comandos dos clientes TCP e actuar de acordo com os mesmos.

Os clientes RMI são mais autónomos (pelo menos, do ponto de vista do programador) e não precisam de componentes adicionais para além do *RMI Registry*.

## Distinção entre Clientes Online e Offline

Como já foi referido anteriormente, os servidores têm uma lista de todos os clientes registados no sistema em forma de *hash table*. Contundo, nem todos os clientes estarão activos durante todo o tempo de funcionamento dos servidores e assim, torna-se necessário distinguir entre clientes *online* e *offline*.

Para guardar e garantir pesquisas eficientes de clientes *online*, usamos duas estruturas:

* Uma lista ligada, onde os clientes que se acabaram de ligar são adicionados à cauda. Esta estrutura torna-se útil quando queremos percorrer todos os clientes activos.

Cada nó da lista é constituído por uma classe privada designada por *ClientListElement*. Nesta classe, registamos apenas o nome de utilizador e o tipo de ligação (TCP ou RMI) do cliente, de modo a poder actuar de diferentes modos para os diferentes tipos de clientes passíveis de alterar o sistema.

* Contudo, quando queremos procurar um dado cliente, procurar numa lista ligada pode-se tornar incompatível com a eficiência pedida. Assim, recorremos a outra *hash table* que recebendo o nome do utilizador, devolve um ponteiro para um dado nó da lista ligada onde é guardada a informação de um cliente *online*.

Existia a possibilidade de percorrer toda a *hash table* para obter a lista completa de clientes. Contudo, como é sabido, esse não é o propósito desta estrutura e listagens completas podem ser demasiado dispendiosas.

Assim, optámos por requerer mais memória do sistema para obter uma maior eficiência em tempos de resposta.

Todos os clientes que não se encontram listados nestas estruturas, são naturalmente considerados *offline*. Tantos estes como os últimos, terão sempre uma entrada na *hash table* que funciona como base de dados do sistema, de modo a ser possível fazer essa mesma transição entre *offline* e *online*.

## Envio de Mensagens para Clientes

O envio de mensagens só pode ser feito para clientes que estejam activos, sendo que qualquer utilizador que tenta enviar uma mensagem para um outro utilizador *offline*, é notificado de tal ocorrência.

Podemos ter várias situações em que é necessário enviar mensagens para um cliente.

No primeiro cenário, temos um utilizador que deseja enviar um mensagem para um outro utilizador específico. Transmite tal desejo ao servidor que por sua vez invoca um método de *ActiveClients*. Nesse momento, é feita uma pesquisa na *hash table*, sendo que se o cliente estiver activo, é escrito no *socket* a mensagem ou invocado um método em RMI. Se por outro lado estiver inactivo, então o cliente que iniciou todo o processo é informado de tal.

Podemos ter, por outro lado, mensagens para todos os clientes (e.g. o anunciado de resultados finais ou mensagens de um utilizador para todos os outros utilizadores activos). Neste caso, é invocado outro método que apenas têm de percorrer a lista de clientes activos e enviar a mensagem para cada entrada.

Notar neste ponto que um cliente RMI apresenta muitas mais dificuldades ao sistema do que um cliente TCP quando é necessário manter o registo dos clientes activos.

Todos os clientes, na sua aplicação, têm um comando ‘*exit*’ que simplesmente informa que o cliente vai sair e permite o servidor tomar todas as providências necessárias para manter o sistema consistente.

Contudo, existe sempre a possibilidade do cliente ir abaixo de uma forma inesperada. Se for um cliente TCP, não há problema de maior. Do lado do servidor, é recebida uma excepção e então, o cliente poderá ser removido da lista de clientes activos.

Em oposição, um cliente RMI não possibilita esta notificação de forma espontânea. Se sair sem avisar o servidor, o seu registo irá permanecer na lista de clientes activos, potenciando o surgimento de sérios problemas.

De modo a evitar que tal aconteça, adoptámos um simples mecanismo. Sempre que precisamos de invocar o registo de um cliente RMI activo (i.e. enviar mensagens ou listar os clientes), invocamos primeiro um pequeno método remoto do lado do cliente que apenas devolve ‘*true*’. Se o cliente RMI estiver realmente activo, continuamos a nossa tarefa. Se contudo esta invocação gerar alguma excepção, significa que o cliente não se encontra *online* e consequentemente, podemos removê-lo da lista e prosseguir de uma forma segura.

Foi sugerido em conversa com outros colegas que poderia ser feito um método de ‘*ping*’ regular para limpar eventuais clientes RMI que se encontravam inactivos. Apesar desta outra possibilidade, continuamos a defender que o nosso método traz maiores vantagens que o opositor, uma vez que, mesmo tendo eventualmente clientes inactivos registados como activos por um maior espaço de tempo, estaremos a efectuar um menor número de verificações e deste modo, atrasar menos o sistema.

Por outro lado, a interacção dos clientes activos vão funcionar quase como este ‘*ping*’ regular. Num caso real, a troca de mensagens irá ser frequente e assim, efectuar uma segunda ronda de verificações seria extremamente despropositado.

## Envio de Comandos para o Servidor

A parte essencial da interacção entre servidor e cliente assenta sobre o envio de comandos por parte do cliente e da sua interpretação por parte do servidor. A lista de comandos e suas funcionalidades pode ser vista, relembrando outra vez, no manual do utilizador, sendo que nesta secção apenas iremos explicar como é que são trocados os comandos de uma forma genérica.

Tal como se ligam de forma diferente, também o envio de comandos por clientes TCP ou RMI também se processa de forma diferente. Desse modo, vamos começar pela explicação do processo que envolve clientes TCP passando depois para os de RMI. Em qualquer um dos casos, os comandos são enviados sobre forma de String.

Depois de ler os dados inseridos pelo utilizador, é verificado o estado da ligação e se estiver activo, então apenas fazemos duas comparações com o *input*, que representam os casos excepcionais.

Se o *input* do utilizador for constituído pela String *‘reset’*, então fazemos primeiro uma simples verificação da quantidade de créditos que o cliente possui. Enviamos um pedido ao servidor, avisando de antemão a *thread* que está a ler a partir do *socket* não é necessário imprimir na consola o resultado que for devolvido. Quando existir uma resposta, essa mesma *thread* avisa *ClientWriteTCP* do saldo do cliente. Se o valor for inferior ao número por defeito de créditos, o saldo do utilizador é reajustado sem problemas.

Se no entanto, o cliente tiver um valor superior, então perguntamos ao utilizador se pretende realmente baixar o seu saldo para o número por defeito, informando o número de créditos que irão perder. Dependendo da resposta do cliente, a operação pode ser concluída ou abortada.

A segunda comparação prende-se com o comando *‘exit’*. Se o utilizador imprimir este comando, a aplicação simplesmente termina e do lado do servidor será gerada uma excepção no *socket*, permitindo a *thread* responsável eliminar este cliente da lista de utilizadores activos.

Para todos os outros casos, a String é enviada para o servidor tal e qual for recebida pela aplicação. Este modelo de ‘*thin client*’ foi usado porque eventualmente, poderíamos verificar a validade dos comandos no lado do cliente e só depois enviar para o servidor. Contudo, no servidor, teríamos novamente de efectuar o *parsing* da String e seria apenas repetir o trabalho já feito.

Alternativamente, poderíamos enviar objectos através do *socket*, só que desse modo, continuaríamos a ter de realizar várias comparações do lado do servidor, para além de que teríamos de dividir a String também do lado do cliente. Daí optarmos por enviar apenas Strings, o que simplifica todo o processo.

A *thread* do lado do servidor responsável por este cliente fica encarregue de dividir o comando lido e caso seja válido, proceder de forma adequada e informar o cliente do sucesso ou não da operação. Se por seu lado for um comando inválido, o servidor apenas ignora o comando enviado, informando o cliente de tal opção.

Quanto aos clientes RMI, é mesmo necessário fazer o *parsing* do lado do cliente e essa é a forma mais eficiente de escolher os comandos adequados. Como em Java RMI os métodos remotos são chamados como fosse métodos locais, precisamos de saber previamente qual o método a executar para um dado *input*.

Assim, o cliente apenas tem de dividir a String nas suas componentes e se for um comando válido, chamar o método adequado preenchendo os devidos argumentos. De modo a poder informar o cliente da validade da operação, o servidor apenas precisa de devolver outra String que será lida pelo cliente.

Sendo que no cliente RMI só há uma *thread* responsável pela comunicação entre cliente e servidor, não se levanta os problemas que encontrámos com o TCP devido ao comando ‘*reset*’, em que tínhamos de avisar *ClientReadTCP* que o valor dos créditos do utilizador tinha sido pedido pelo próprio sistema e não espontaneamente pelo utilizador.

Quando executado o ‘*exit*’, não podemos simplesmente sair da aplicação, como no caso do TCP. A aplicação, antes de terminar, executa um método remoto que simplesmente avisa o servidor do término desta secção. Idealmente, todos os clientes RMI terminariam deste modo, mas como tal não acontece, tivemos de implementar vários mecanismos para evitar clientes ‘fantasma’, tal como foi descrito na secção anterior.

Em ambos os casos, temos sempre de lidar com a comunicação *offline*. Sempre que um cliente (tanto TCP como RMI) encontra ligação em baixo, mas já tinha efectuado um *login*, damos a possibilidade ao utilizador de executar um conjunto restrito de comandos, nomeadamente os da família *‘send*’. Para todos os outros, apenas informamos o cliente que a ligação se encontra em baixo e que volte a tentar mais tarde.

Quando nos encontramos neste cenário de falha na ligação, todos os clientes fazem o *parsing* do *input*. Se for um comando *‘send*’, avisa o cliente que a mensagem vai ser guardada (num *buffer* temporário e num ficheiro) para ser enviada mais tarde. Como as mensagens também são guardadas em ficheiro, possibilitamos ao utilizador sair da aplicação. Quando voltar a tentar, a aplicação simplesmente lê o ficheiro se aí encontrar alguma mensagem pendente, envia de imediato para o servidor.

# Iniciar uma Nova Sessão

Antes poder haver comunicação entre o cliente e servidor, é preciso autenticar qualquer utilizador que se tente ligar ao sistema. Desse modo, quando iniciada a aplicação cliente, damos duas possibilidades ao utilizador: ou se regista no sistema ou então insere a informação correcta para poder usar uma conta já registada. Qualquer que seja a opção, esta troca de mensagens efectua-se como uma normal troca de comandos, descrita no capítulo anterior.

## Registar um Novo Cliente

Quando um utilizador é novo no sistema, dados a possibilidade de criar uma conta nova. Para tal, apenas necessita de executar o comando *‘register’*, passando os argumentos adequados. Depois, fica à espera da resposta do servidor.

Após receber um comando, o servidor primeiro verifica se o nome do utilizador não é *‘all’*. Se for, temos de o rejeitar, porque a *keyword ‘all’* é usada pelo comando *‘send’* quando queremos enviar uma mensagem para todos os utilizadores.

De seguida, o servidor efectua uma pesquisa na *hash table* que guarda todos os clientes de modo a verificar se o nome do utilizador requerido já não se encontra escolhido.

Se por fim, passar em todos estes testes, o sistema regista este novo cliente, informa a aplicação que efectuou o pedido de registo a mesma automaticamente efectua o *login*, estando agora o cliente apto a fazer as suas apostas.

## Efectuar o Login

Se um cliente já efectuou um registo anteriormente, poderá então realizar um *login* assim que inicia a aplicação. Para tal, apenas precisa de executar o comando *‘login’* acompanhado dos devidos parâmetros (designadamente o seu nome de utilizador e palavra-chave), ficando à espera do resultado vindo do servidor.

No lado no servidor, este apenas precisa de fazer uma pesquisa na *hash table* que contém todos os clientes registados e verifica se há uma entrada existente para o nome passado. Se não houver, apenas tem de retornar uma resposta negativa. Se pelo contrário, houver um nome de utilizador válido, necessita de comparar as duas palavras-chave e apenas no caso das duas coincidirem é que permite que o utilizador entre no sistema.

No entanto, é de notar que o processo de *login* não é apenas realizado quando um cliente se liga pela primeira vez a um dado servidor. Imaginemos, como exemplo, o seguinte cenário:

O cliente efectua um *login* válido e entra no sistema. As suas credenciais (nome de utilizador e palavra-chave) são salvaguardadas na aplicação cliente, para que esta possa lidar com falhas de uma forma transparente para o utilizador final.

Passado algum tempo, ocorre um dado tipo de falha (e.g. uma quebra na ligação, ocorreu um erro fatal no lado do servidor) que impossibilite a comunicação entre o cliente e o servidor ao qual o cliente estava originalmente ligado. Neste caso, é executado o protocolo de recuperação da ligação descrito no capítulo “Principais Estruturas do Sistema – Clientes (TCP e RMI)” e se bem sucedido, teremos de novamente efectuar o *login*. Como os clientes necessitam restabelecer a ligação, não é preciso em nenhum momento, guardar de modo persistente os clientes activos para que o servidor secundário toma conhecimento desta lista.

Apesar desta imposição, como já dissemos, é fundamental que não obriguemos o utilizador a introduzir de novo os dados já validados. Assim, como a aplicação guardou previamente as credenciais, simplesmente as envia para o servidor, sabendo que elas serão autenticadas.

No entanto, há sempre uma excepção à regra neste simples processo. Sempre que é efectuado o *login*, o servidor não precisa de apenas verificar se as credenciais estão correctas, mas também se aquele nome de utilizador já não se encontra activo numa outra máquina (apenas precisa de procurar na *hash table* gerida pelo *ActiveClients*).

Assim, aquando da nova conexão, poderá dar-se o caso de que naquele curto espaço de tempo, um outro utilizador tenha usado as mesmas credenciais para entrar no sistema, sendo que então é barrada a entrada ao cliente. Apesar de improvável, poderá acontecer, especialmente se uma conta for partilhada por mais do que um utilizador.

# Realização de Apostas

Recorrendo a todos os processos e mecanismos descritos nos capítulos antecedentes, chegamos finalmente ao propósito de toda a aplicação, a criação de rondas de jogos e a possibilidade dos clientes fazerem as suas apostas, naturalmente esperando que acertem e os palpites se traduzam em multiplicações do seu saldo.

Como ressalva, convém relembrar que nem todo o código usado para a realização destas apostas é da nossa autoria, tendo sido usado material disponibilizado pelos docentes da disciplina. No entanto, tivemos de efectuar algumas alterações aos ficheiros originais de modo a poder responder a todos os pontos incluídos no enunciado (especificamente em questões ligadas com a persistência de dados ou possibilidade de modificação do número de jogos por ronda). A classe *BetScheduler* funciona como uma camada superior que ‘esconde’ todo esse código, sendo que o servidor apenas interage directamente com o objecto desta classe.

## Gerar uma Nova Ronda

Sempre que um objecto da classe *BetScheduler* é instanciado, a primeira operação que faz é uma leitura de ficheiro. Se este não se encontrar vazio, então encontramo-nos no caso em que um servidor já iniciou trabalho anteriormente mas, por algum motivo desconhecido, deixou de o fazer.

Assim, para este caso, o novo servidor primário apenas necessita de retomar a partir daquele ponto, criando novas rondas como normalmente. Se por outro lado, o ficheiro se encontrar em branco ou invalidado, *BetScheduler* precisa de fazer um *reset* ao contador e criar uma ronda inicial.

Criada ou lida a ronda, o *BetScheduler* entra num estado de repouso correspondente ao valor definido como o tempo de cada ronda. Findo este período, o gestor de apostas apenas precisa de percorrer a lista de apostas e verificar se existem apostas vencedoras. Em caso afirmativo, envia uma mensagem recorrendo ao *ActiveClients*, que fica responsável de entregar a mensagem caso o cliente esteja *online* e comunicar com o *GlobalDataBaseManager* para actualizar o número de créditos do cliente.

Assim que acaba de percorrer toda a lista, é necessário gerar uma nova ronda de jogos. Apaga os registos anteriores e começa o ciclo desde o início. Guarda a ronda em ficheiro, fica inactivo e faz as actualizações requeridas ao sistema no fim da nova ronda.

## Apostar em Partidas

Quando se encontram jogos disponíveis, é permitido aos clientes realizarem apostas para jogos de uma dada ronda. No mesmo momento que o *BetScheduler* procede à leitura dos jogos a partir de um ficheiro, também necessita de fazer a leitura de um outro ficheiro com propósito semelhante, mas que contém as apostas realizadas para a ronda activa.

Se este ficheiro se encontrar em branco, ainda não forem feitas quaisquer apostas. Se pelo contrário, estiver escrito, é necessário ler todas as apostas e guardá-las num vector que se encontra na memória principal.

Sempre que é feita uma nova aposta, o *BetScheduler* apenas precisa de a adicionar a um vector, sendo que cada entrada nesse vector contém uma classe que guarda diversas informações, mais concretamente o nome de utilizador que realizou a aposta, a quantidade de créditos apostados, o número do jogo e o resultado final. Em cada momento de actualização deste vector, temos de guardar para memória persistente o vector de apostas.

Quando ocorre o *timeout* que indica o fim de uma ronda, a lista tem de ser percorrida, como já foi brevemente referido na secção anterior.

Acabado este procedimento, é necessário limpar o vector de todas as apostas e o ficheiro que guarda as apostas, de modo a não causar conflitos posteriormente.

Os clientes apenas podem fazer apostas nos jogos que estão activos, sendo que sempre que introduzam números de jogos fora dos limites da ronda, é sempre devolvida uma mensagem de erro. Por outro lado, o sistema de apostas também está protegido para o caso de clientes tentarem efectuar apostas de um valor superior ao que têm na conta ou então, fazeres apostas de valor nulo.

Sempre que uma aposta é completa de modo correcto, o sistema retira automaticamente o número de créditos apostados na conta do utilizador. Deste modo, se um cliente apostar num resultado errado, não é preciso tomar nenhuma acção para além de, caso o utilizador ainda esteja *online*, informá-lo da sua derrota.

# Especificações de Testes

De modo a confirmar que toda a arquitectura funciona em pleno e responde às situações de falha mais relevantes, realizamos diversos testes, entre os quais incluímos o cenário de duplo servidor primário (STONITH).

## Efectuar o Login

Como já foi rasdasd

## Cenário de Duplo Servidor Primário

O cenário de duplo servidor primário ocorre quando temos uma quebra na ligação entre os dois servidores. Como se recorre a tráfego UDP entre servidores para o envio destas mensagens, nunca chegamos a ter confirmação explícita de que uma dada mensagem conseguiu com sucesso atravessar toda a rede que separa os dois servidores.

Assim, imaginemos a seguinte situação. Há um servidor primário que envia a um ritmo constante as mensagens de ‘KEEP\_ALIVE’, enquanto o servidor secundário simplesmente as recebe e faz o *reset* dos temporizadores.

A dado momento, há uma falha na rede, e as mensagens deixam de atravessar o canal. Neste momento, temos o servidor primário que não vê nenhuma razão para deixar de ser servidor primário. Por seu lado, o servidor secundário deixa de receber mensagens e, quando ocorrer um *timeout*, legitimamente pode assumir o papel de servidor secundário, pois o seu superior deixou de enviar as mensagens como era da sua responsabilidade. Quando o servidor secundário se auto-promover a servidor primário, estamos perante o cenário de duplo servidor primário.

Numa aplicação real, os dois servidores teriam uma ligação extra entre eles. Sempre que se deparassem nesta situação, um deles activaria um sinal eléctrico que simplesmente obrigaria o outro a para, o chamado “Shoot the Other Node in the Head” (STONITH).

Naturalmente, não conseguimos reproduzir de forma exacta este mecanismo extra para evitar o cenário referido. Não obstante, criamos um pequeno módulo que simule tal solução.

Basicamente, temos uma *thread*, associada ao *ConnectionWithOtherServerManager* que está sempre activa e à escuta num dado porto TCP. Assim, sempre que um servidor está prestes a ficar com o papel de servidor principal, liga-se a este porto e verifica o resultado. Se tiver causado uma excepção, quer dizer que o outro servidor está mesmo em baixo e podemos assumir sem problemas a entidade de servidor primário. Se contudo, não houver qualquer excepção, quer dizer que o outro servidor está activo e desse modo, não podemos exercer a auto-promoção.

Obviamente, toda esta simulação tem uma falha óbvia. Se o *socket* TCP usar a mesma ligação que o UDP, quando um falhar, é provável que o outro também falhe. Desse modo, tivemos de criar mais um artifício para que a ligação em si não falhe realmente mas apenas na perspectiva dos servidores envolvidos. Simplesmente enviamos mensagens para um porto no qual o outro servidor não está à escuta e desse modo, as mensagens nunca chegaram ao seus destino correcto.

Este segundo passo também possibilita algo que, no nosso caso, acaba por ser muito importante, isto é, simular uma falha na rede estando os dois processos relativos aos dois servidores diferentes a correr na mesma máquina.

# Conclusão

O sistemas distribuídos indiscutivelmente avolumam exponencialmente o número de problemas que encontramos para um sistema fechado e isolado. Existe o problema da sincronização entre todos os intervenientes; ao contrário de um sistema fechado, parte da rede pode falhar sem que a outra saiba do sucedido; a própria ligação entre os intervenientes está sujeita a interferências, o que, no caso de um sistema restringido ao seu espaço de endereçamento, raras vezes acontece sem ser por erros de programação.

Assim, o programador fica impossibilitado de evitar grande parte dos problemas existentes, pois um considerável quantidade deles depende de factores externos e estocásticos. Cabe então aos responsáveis prever quais as dificuldades com as quais as aplicações podem eventualmente confrontar-se e implementar mecanismos que lidem com as adversidades. Foi isso que procurámos fazer, com o máximo de pormenor permitido pelos nossos conhecimentos e experiência na área.

Apesar das soluções que apresentámos ao longo do relatório, claramente estão longe de cobrir todas as possibilidades de falha, especialmente se transportarmos o nosso sistema para o mundo global e, hipoteticamente, permitíssemos milhares de clientes ligados simultaneamente.

Em primeiro lugar, talvez precisássemos de um maior número de servidores secundários, ou mesmo vários servidores que trabalhassem em conjunto como um único servido primário. O protocolo de troca de mensagens e eleição do servidor primário teria de ser repensado, apesar de certamente partilhar muitas das características do protocolo usado na nossa aplicação.

Em segundo lugar, a forma de atender clientes TCP iria certamente explodir com os recursos do sistema. Tendo uma *thread* responsável por cada cliente que se liga, aumentando o número de clientes médios para as casas dos milhares e não das unidades, como temos agora, o número de *threads* que teriam de estar activas no servidor seria incomportável. Desse modo, uma solução possível seria criar uma *pool* de *threads* que permitisse gerir o número máximo de entidades activas no sistema.

Também não podemos esquecer que os nossos ficheiros e mecanismos para implementar a base de dados dos clientes são bastante rudimentares. Tal como é dito no enunciado, partimos do princípio que não existe falha na ligação entre servidor e sistema de suporte à persistência dos dados. Na realidade, como é natural, tal garantia não existe.

Ainda no campo da salvaguarda dos dados, o nosso sistema recorre aos ficheiros sempre que existe qualquer alteração, por mais pequena que seja. Enquanto no nosso pequeno sistema tal *overhead* não será demasiado significativo, no mundo real, estaríamos a escrever e ler de ficheiros um elevado número de vezes num curto espaço de tempo. Para além disso, tudo é guardado em memória física, o que, devido à lentidão de acesso a disco, ainda atrasa mais o sistema.

Contudo, apesar de todos estes pontos negativos, consideramos o trabalho em geral bastante satisfatório, principalmente no que diz respeito à aquisição de conhecimentos e apresentação de problemas que nos ensinem a pensar de uma forma distribuída.

# Bibliografia

**IDE**

Java Eclipse IDE

**Sites Internet**

- Javaspecialists.eu java training: Resetting ObjectOutputStream by Dr. Heinz M. Kabutz. Disponível em: [http://www.javaspecialists.co.za](http://www.javaspecialists.co.za/) Acesso em 23/10/10

-

**Imagens  
 Capa:** BWIN Disponível em: <http://www.bwin.com/> Acesso em 20/10/10

**Diagrama –** Microsof Visio 2010