Universidade de Coimbra

Faculdade de Ciências e Tecnologia

Licenciatura em Engenharia Informática

*2º Trabalho Prático – Aplicação Distribuída ‘Web-based’*

*Sistemas Distribuídos, 1º Semestre*



**RELATÓRIO**

**Trabalho realizado por:**

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

Coimbra, 10 de Dezembro de 2010

**Índice**

[Introdução 2](#_Toc279497681)

[Principais Estruturas do Sistema 3](#_Toc279497682)

[Aplicação *Web-based* 3](#_Toc279497683)

[Servidor Intermédio 4](#_Toc279497684)

[Comunicação Entre Cliente e Servidor 6](#_Toc279497685)

[Esquema de Troca de Mensagens 6](#_Toc279497686)

[Cliente 7](#_Toc279497687)

[*Login*, registo e invalidação de sessões 7](#_Toc279497688)

[Chat 7](#_Toc279497689)

[Realização de uma aposta e actualização dos jogos 7](#_Toc279497690)

[Actualização da lista de clientes *online* e dos créditos 7](#_Toc279497691)

[REST 9](#_Toc279497692)

[Notícias 9](#_Toc279497693)

[Especificação de Testes 10](#_Toc279497694)

[Tabelas de Testes 10](#_Toc279497695)

[Cenário de Duplo Servidor Primário 13](#_Toc279497696)

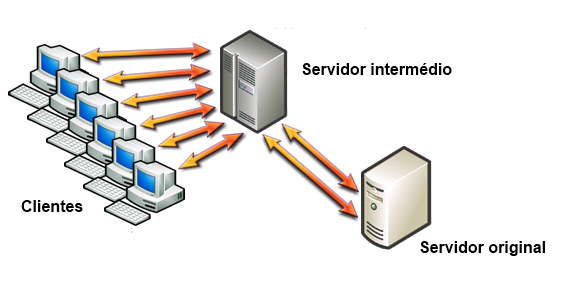
[Conclusão 15](#_Toc279497697)

[Bibliografia 17](#_Toc279497698)

# Introdução

Depois de realizado o primeiro trabalho prático, foi-nos pedido que o estendêssemos para uma aplicação “Web-based”, isto é, criar uma plataforma Web que disponibilizasse as mesmas ferramentas que a aplicação por consola e introduzisse um novo elemento, notícias actualizadas do “Guardian”, recorrendo à tecnologia REST.

Deste modo, não só tivemos de construir um *site* como tivemos de garantir a interoperabilidade entre os diversos elementos (i.e. *site* e clientes TCP e RMI já existentes para o trabalho prático anterior). Por outro lado, deparámo-nos com um modelo *Three Tier*, exemplificado na imagem em baixo.



*Imagem 1 - Three Tier Model para o nosso projecto.*

Do lado esquerdo, temos os clientes que usam as aplicações *Web-based* para aceder aos serviços BetAndUin. No entanto, ao contrário do cenário do primeiro trabalho prático, estes não comunicam directamente com o servidor principal (na figura, o servidor original), mas sim com o servidor *TomCat*, referenciado como servidor intermédio. São assim feitos os pedidos primeiro para o servidor intermédio e este fica encarregue de comunicar, agora sim directamente com o servidor principal.

Este relatório vai seguir um formato em tudo idêntico ao anterior, onde vamos começar por fazer uma exposição geral do trabalho, indo aos poucos aprofundando e revelando os pormenores mais subtis e relevantes.

É ainda de notar que neste trabalho prático não foram solicitadas soluções para o caso de falhas do servidor, uma vez que tal tarefa encontrava-se englobada no âmbito do primeiro trabalho. Deste modo, corremos o servidor principal do primeiro trabalho (que note-se teve de sofrer leves alterações para se adaptar à nova aplicação) e não pomos em causa a sua fiabilidade.

# Principais Estruturas do Sistema

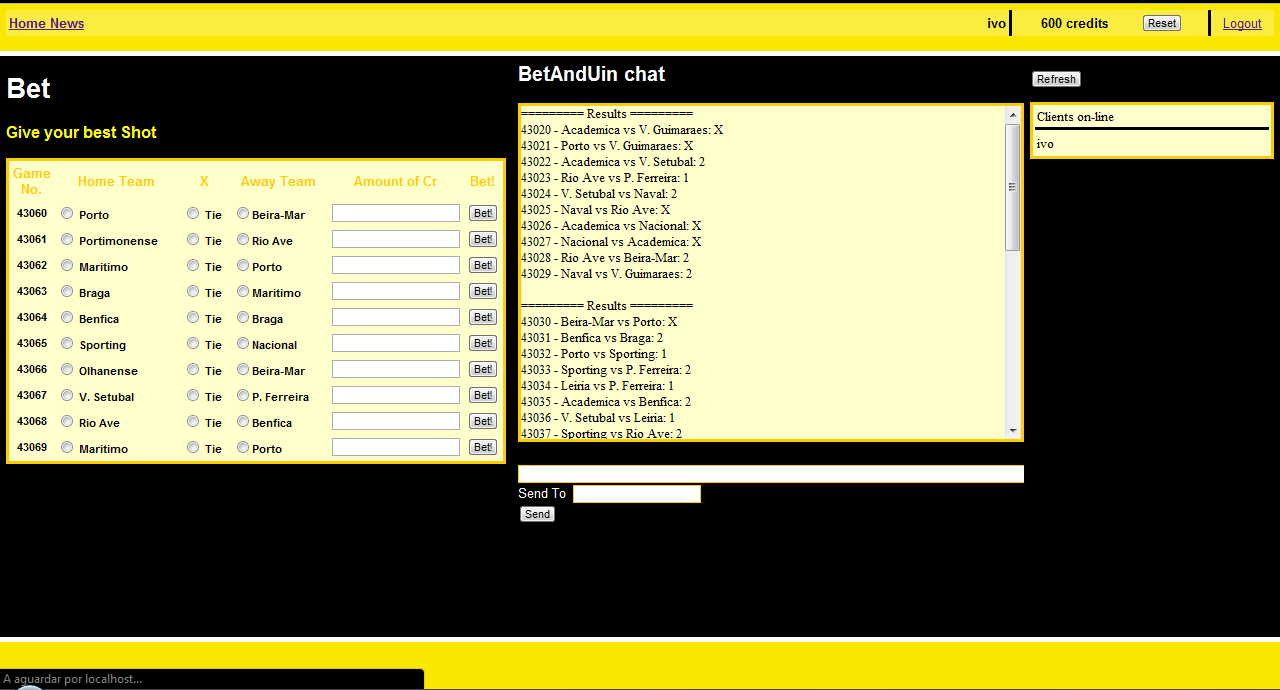
Este trabalho prático envolveu menos elementos do que o anterior, sendo que no entanto é de notar o aumento da complexidade de algumas das partes, especialmente no que se refere à aplicação de interacção com o cliente, já que vimos simplificada a interacção com e entre servidores.

Assim, neste capítulo, iremos iniciar a discussão da aplicação *Web-based* e uma vez observadas todas as suas principais funcionalidades, iremos abordar a arquitectura do servidor intermédio.

## Aplicação *Web-based*

Para aceder à aplicação, precisamos de inicializar todos os servidores de antemão e uma vez concluída esta tarefa, apenas recorremos a um *browser* e acedemos ao seu endereço como se tratasse de um qualquer *site* (estando a correr na nossa máquina, o endereço será *localhost:8080/BetAndUinWeb/Pages/Login.jsp* ).

Uma vez na aplicação, deparámo-nos com a página inicial, onde o utilizador deve inserir um *login* válido para continuar. Caso ainda não tenha um, pode sempre recorrer ao *link* para o registo de novos utilizadores. O texto, na caixa de cor amarela do lado direito, vai dando informações ao utilizador caso haja alguma anomalia com o *login*, como por exemplo nomes de utilizador ou palavras-chave erradas, ou então tentativas de autenticação de utilizadores activos no sistema. Uma vez dentro do sistema, o utilizador é presenteado com uma página que permite realizar todas as actividades de apostas nos servidores BetAndUin.



*Imagem 2 - A página principal do nossa aplicação.*

Do lado esquerdo, temos uma *frame* onde são registados os jogos actuais, assim como campos para a realização de apostas. Ao centro, temos uma janela onde são impressas todas as mensagens do *chat*, sendo que por baixo dessa mesma janela, temos os campos para enviar mensagens.

No topo, encontramos à esquerda um *link* para a página que contêm as notícias mais recentes relacionadas com o campeonato português de futebol, recorrendo à versão *online* do jornal “The Guardian”, tal como é explicado mais à frente. À direita, é impresso o nome de utilizador, o número de créditos desta conta, um botão para realizar o *reset* dos créditos e um *link* final para realizar o *logout*.

Por fim, à direita, temos a coluna onde são apresentados os clientes que também estão activos, sendo que é possível pressionar o seu nome com o ponteiro do rato o seu nome, que este irá aparecer automaticamente no campo ‘*Send To*’ do *chat*, deste modo poupando o utilizador o trabalho de digitar o nome do destinatário sempre que queira enviar mensagens.

Relativamente à página de notícias, esta contêm diversos *links* que uma vez pressionados, apresentam uma pequena mensagem uma imagem revelando o seu conteúdo.

## Servidor Intermédio

Tal como já foi dito na introdução, o servidor *TomCat* fica responsável pela ponte entre os clientes *Web* e o servidor principal BetAndUin, sendo que sempre que existem mensagens para todos os clientes, o servidor principal apenas necessita de envia uma dessas mensagens para o servidor *Tomcat* que este se encarrega de a espalhar pelos clientes *Web* a si ligados (detalhes a serem discutidos no capítulo TODO x). De uma maneira geral, este servidor vai funcionar como mais um cliente RMI, que faz e envia pedidos para o servidor principal.

Este servidor é constituído por três *servlets*, nomeadamente a *WebServlet*, *Logout* e a *BetServlet*. A primeira fica encarregue de aceitar pedidos de *login* e registo, obtendo os dados necessários para validar a sessão, guardar um registo do cliente em questão e ainda transmitir ao servidor principal a intenção deste novo cliente de entrar no sistema.

A *servlet* *Logout* é apenas usada quando o utilizador pressiona o botão de *logout* ou então fecha o *browser*, sendo que em ambos os casos esta estrutura tem a função de invalidar a sessão agora termina e limpar o registo do cliente das listas de clientes activos do servidor principal.

Por fim, a terceira, *BetServlet*, tal como o nome indica, fica encarregue de actualizar a lista de jogos que é impressa no *browser* de cada utilizador, assim como interagir com o servidor principal sempre que um cliente faz um pedido para realizar uma aposta. Por outro lado, esta *servlet* também é invocada sempre que ocorre uma alteração no número de créditos do utilizador e é preciso actualizar o seu número no topo da página.

# Cliente

- - - TODO escrever qualquer coisa

## *Login*, registo e invalidação de sessões

Tanto o *login* como registo de novos cliente processa-se por duas fases. Primeiro, é enviado um GET com os dados obtidos dos formulários de cada página para a *WebServlet*. Esta, faz uma chamada RMI para o servidor principal e fica à espera da resposta. Uma vez obtida, uma dada mensagem (relacionada com o conteúdo da resposta do servidor principal) é registada na sessão do utilizador e neste momento, abrem-se dois cenários:

* Ou a resposta vinda do servidor principal é favorável e o cliente é redireccionado para a página de apostas;
* Ou por outro lado, a resposta é negativa e o utilizador permanece na mesma página. Contudo, nesta situação, ao fazer o *refresh* da página, é detectado uma sessão já existente e será disposta a mensagem justificando o erro encontrado, tal como conta já existente ou *login* inválido, por exemplo.

As invalidações de sessão ocorrem quando o cliente efectua um *logout*, tanto explicitamente (pressionando o *link* no topo direito da página) ou então fechando a página, sendo que esta última situação é detectada pela *ChatServlet*, uma vez que corre *COMET*, e esta toma as medidas adequadas para proceder a um término limpo e livre de problemas.

## *Chat*

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

, 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*.

## Realização de uma aposta e actualização dos jogos

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

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

## Actualização da lista de clientes *online* e dos créditos

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 maneira diferente, também o envios de comandos por clientes TCP ou RMI não seguem a mesma metodologia. 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 ante

# REST

Antes poder haver comunicação entre o cliente e servidor, é preciso autenticar qualquer utilizador que se tente ligar ao sistema. Deste 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 existente. Qualquer que seja a opção, esta troca de mensagens efectua-se como uma normal troca de comandos, descrita no capítulo anterior.

## Notícias

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 rejeitar o pedido, porque a *keyword ‘all’* é usada pelo comando *‘send’* quando queremos enviar uma mensagem para todos os utilizadores.

Se passar o primeiro teste, 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.

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.

## Falta de conecção

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

# Especificação de Testes

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

## Tabelas de Testes

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NOME DO TESTE:** | | Caso de Teste 1: Efectuar o *login* / registo e correr a aplicação | | |
| **DESCRIÇÃO:** | | O utilizador inicia a aplicação, tentando efectuar o *login* caso já tenha uma conta no sistema ou então proceder ao registo caso contrário.  Uma vez dentro do sistema, o utilizador enviará qualquer um dos comandos disponíveis. | | |
| **PRÉ-REQUISITOS:** | | O utilizador introduz toda a informação correctamente. Caso contrário, a aplicação rejeitará os comandos. | | |
|  | | | | |
| **PASSO** | **EVENTO** | | **RESULTADOS ESPERADOS** | **RESULTADOS OBSERVADOS** |
| 1 a). | O utilizador envia um comando de registo. | | Se o nome de utilizador já se encontrar em uso ou então for a *keyword* ‘*all’*, o registo não deve ser efectuado.  Caso ainda esse nome ainda estiver disponível, o registo é realizado com sucesso e a aplicação automaticamente faz o *login* no sistema. | Todos os resultados observados para este caso de teste estão de acordo com o resultados esperados. |
| 1 b). | O utilizador envia um comando de *login*. | | Se o utilizador não estiver registado no sistema, a palavra-chave não estiver correcta ou outro utilizador já está no sistema usando o mesmo nome de utilizador, o cliente é notificado e o *login* não é concluído.  Caso contrário, o utilizador pode começar a utilizar as funcionalidades do programa. |
| 2. | O utilizador envia um dos comandos disponíveis a partir do menu principal. | | O servidor processa o dado, transmitindo, no final da operação, o sucesso ou insucesso da mesma. |
| 2. a) | O comando enviado é um *reset*. | | Se o utilizador tiver no máximo tantos créditos quanto o número por defeito de créditos, o *reset* é efectuado de imediato.  Caso contrário, é perguntado ao utilizador se realmente está disposto a perder créditos, revelando os números desta perda. |
| 2. b) | O comando executado é um *exit*. | | A aplicação termina de uma forma organizada. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NOME DO TESTE:** | | Caso de Teste 2: Um servidor sem clientes falha | | |
| **DESCRIÇÃO:** | | Testamos o caso em que o servidor primário, não tendo nenhum cliente ligado, falha. Este caso de teste também irá servir para o caso em que um cliente se está a ligar pela primeira vez. | | |
| **PRÉ-REQUISITOS:** | | O servidor primário falhou. Partimos do princípio que o secundário já está disponível para tomar o lugar deste. Por outro lado, assumimos que não existem problemas com a rede e qualquer problema com a ligação a um dos servidor se deve ao uma falha fatal no mesmo e não quebras na ligação. | | |
|  | | | | |
| **PASSO** | **EVENTO** | | **RESULTADOS ESPERADOS** | **RESULTADOS OBSERVADOS** |
| 1. | O servidor primário falha. | | Quando ocorrer um *timeout*, o servidor secundário detecta a falha e assume a posição de servidor primário. | Todos os resultados observados para este caso de teste estão de acordo com o resultados esperados. |
| 2 a). | Um cliente tenta ligar-se a um servidor, sendo que o servidor que foi abaixo era o servidor primário por defeito. | | Após a primeira tentativa, verificando que o servidor 1 está em baixo, o cliente irá tentar ligar-se ao servidor 2, sendo que este irá aceitar a ligação.  Nenhuma informação relativa aos clientes e jogos é perdida com responsabilidades do servidor. |
| 2 b). | Um cliente tenta ligar-se a um servidor, sendo que o servidor que foi abaixo não era o servidor primário por defeito. | | O cliente irá imediatamente estabelecer a ligação com o servidor primário.  Nenhuma informação relativa aos clientes e jogos é perdida com responsabilidades do servidor. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NOME DO TESTE:** | | Caso de Teste 3: Um servidor com clientes falha | | |
| **DESCRIÇÃO:** | | Testamos o caso em que o servidor primário, tendo clientes ligados, falha.  Como nota, convém relembrar que a falha é detectada num cliente TCP assim que a ligação através do *socket* for quebrada, enquanto num cliente RMI a falha só é detectada quando o utilizador tentar executar um comando que envolva interacção com o servidor. | | |
| **PRÉ-REQUISITOS:** | | Os mesmos do caso de teste anterior.  O servidor primário falhou. Partimos do princípio que o secundário já está disponível para tomar o lugar deste. Por outro lado, assumimos que não existem problemas com a rede e qualquer problema com a ligação a um dos servidor se deve ao uma falha fatal no mesmo e não quebras na ligação. | | |
|  | | | | |
| **PASSO** | **EVENTO** | | **RESULTADOS ESPERADOS** | **RESULTADOS OBSERVADOS** |
| 1. | O servidor primário falha. | | Quando ocorrer um *timeout*, o servidor secundário detecta a falha e assume a posição de servidor primário. | Todos os resultados observados para este caso de teste estão de acordo com o resultados esperados. |
| 2 | O cliente detecta a falha. | | O cliente irá tentar de imediato tentar ligar-se ao segundo servidor.  Nenhuma informação relativa aos clientes e jogos é perdida com responsabilidades do servidor.  Por outro lado, a aplicação do lado do cliente terá guardado os dados de *login* e procederá à autenticação imediata do utilizador. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NOME DO TESTE:** | | Caso de Teste 4: Ambos os servidores estão em baixo. | | |
| **DESCRIÇÃO:** | | Tal como indica, neste caso de teste, ambos os servidores estão em baixo, mas mesmo assim, pelo menos um cliente irá tentar efectuar uma ligação.  Como nota, podemos ainda dizer que o ponto | | |
| **PRÉ-REQUISITOS:** | | - - - - | | |
|  | | | | |
| **PASSO** | **EVENTO** | | **RESULTADOS ESPERADOS** | **RESULTADOS OBSERVADOS** |
| 1. | Ambos os servidores falham ou não chegam a ser inicializados. | | Não haverá modo de os clientes acederem ao modo de apostas. | Todos os resultados observados para este caso de teste estão de acordo com o resultados esperados. |
| 2. | O cliente tenta ligar-se ao servidor 1. | | Não encontrando nenhuma resposta, desiste de imediato. |
| 3. | O cliente tenta ligar-se ao servidor 2. | | Ao fim de diversas tentativas, o cliente desiste de estabelecer a ligação. |
| 4. | O cliente tenta ligar-se de novo ao servidor 1. | | Dado que apenas tinha feito uma tentativa de ligar-se ao servidor 1, antes de desistir por completo, irá efectuar mais um número definido de tentativas de conseguir estabelecer a ligação.  No final, irá desistir de qualquer tentativa de ligação e a aplicação é encerrada. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NOME DO TESTE:** | | Caso de Teste 5: Dois servidores primários | | |
| **DESCRIÇÃO:** | | A comunicação entre os servidores falha, sendo que o servidor secundário deixa de receber qualquer tipo de mensagens vindas do outro servidor. Irá então ficar também com o estatuto de servidor primário.  Este cenário de teste está descrito com mais detalhe na secção seguinte. | | |
| **PRÉ-REQUISITOS:** | | Tem de haver uma falha na ligação entre servidores (ou outro problema que impossibilite a comunicação normal entre os servidores), sem que a ligação auxiliar usado pelo STONITH seja comprometida.  Ambos os servidores têm de estar operacionais. | | |
|  | | | | |
| **PASSO** | **EVENTO** | | **RESULTADOS ESPERADOS** | **RESULTADOS OBSERVADOS** |
| 1. | A comunicação falha. | | O servidor que era primário continua como primário. O servidor secundário irá tentar auto-promover-se a servidor primário. | Todos os resultados observados para este caso de teste estão de acordo com o resultados esperados. |
| 2. | O servidor agora primário recorre ao canal de STONITH antes de concluir a operação. | | O servidor secundário irá detectar que o servidor primário ainda se encontra activo, concluído que é um problema de ligação entre os dois. Assim, irá manter o seu estado de servidor secundário. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NOME DO TESTE:** | | Caso de Teste 6: Ligação de clientes a servidores secundários | | |
| **DESCRIÇÃO:** | | Neste caso de teste, o servidor primário é o servidor 2, aquele que não é primário por defeito. Estudamos o caso em que um cliente se tenta ligar ao servidor secundário antes de tentar o primário. | | |
| **PRÉ-REQUISITOS:** | | O servidor 2 terá de ter o estatuto de servidor primário, sendo que ambos os servidores necessitam de estar operacionais. | | |
|  | | | | |
| **PASSO** | **EVENTO** | | **RESULTADOS ESPERADOS** | **RESULTADOS OBSERVADOS** |
| 1. | O cliente tenta ligar-se ao servidor secundário. | | Estando consciente do seu estatuto de servidor e não tendo qualquer informação sobre uma falha no servidor primário, o servidor secundário não irá aceitar nenhum tipo de ligação, tanto TCP como RMI. | Todos os resultados observados para este caso de teste estão de acordo com o resultados esperados. |
| 2. | O cliente finalmente tenta ligar-se ao servidor primário. | | O servidor aceita a ligação do cliente e inicia o processo de *login* ou registo. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NOME DO TESTE:** | | Caso de Teste 7: Envio de mensagens | | |
| **DESCRIÇÃO:** | | O cliente envia um mensagem para um grupo de utilizadores. | | |
| **PRÉ-REQUISITOS:** | | O cliente já tem de estar autenticado no sistema. | | |
|  | | | | |
| **PASSO** | **EVENTO** | | **RESULTADOS ESPERADOS** | **RESULTADOS OBSERVADOS** |
| 1. | O cliente envia um mensagem. | | A mensagem é analisada e recorrendo aos métodos correctos, enviada para o servidor. | Todos os resultados observados para este caso de teste estão de acordo com o resultados esperados. |
| 2 a). | O cliente envia um mensagem para todos os utilizadores. | | Todos os utilizadores que estão *online* (exceptuando o emissor) recebem a mensagem, sendo que o emissor é notificado do sucesso do envio. |
| 2 b). | O cliente envia uma mensagem para um utilizador específico. | | Se o destinatário está *offline* ou não estiver registado, o cliente é notificado de tal situação e a mensagem é descartada.  Se o destinatário estiver *online*, a mensagem é entregue e o emissor é notificado. |
| 2 c). | O cliente tenta enviar um mensagem enquanto está *offline*. | | A mensagem enviada pelo utilizador é guardada em memória, até que o *buffer* atinja o seu limite. Mesmo terminando a aplicação, as mensagens são guardadas e lidas de novo para o *buffer* quando o utilizador iniciar de novo o programa.  Quando estabelecida a ligação, as mensagens são enviadas para o servidor. |

## 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. Por seu lado, não implementamos o STONITH original, mas sim uma variante nossa, *intelligent-STONITH (*iSTONITH), pois não é o servidor secundário que termina com o servidor primário, mas sim é o servidor primário, activando a sua ligação iSTONITH, que impede o servidor secundário de se auto-promover.

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. Com esta abordagem, não obrigamos todos os clientes a terem de mudar-se para um novo servidor apenas porque a comunicação entre servidores não funciona.

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.

Para testar esta simulação na aplicação, basta pressionar ‘Enter’ enquanto o servidor está a correr para mudar o porto UDP para o qual se envia e consequentemente, entrar ou sair do cenário iSTONITH.

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 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 serem adequadas a um ambiente de desenvolvimento académico, 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.

Por seu lado, a assumpção de que não há falhas na ligação e processamento de informação pela base de dados também nos aliviou de um outro problema que poderá ser bastante grave. Imaginemos que um cliente realiza uma aposta, e esta é recebida pelo servidor. No entanto, antes de a conseguir processar, o servidor falha e a mensagem perde-se. Aqui, o cliente fica num dilema. Sabe que o servidor recebeu a mensagem, mas, não obtendo resposta, não sabe ao certo se a aposta foi processada ou não.

Como soluções, poderíamos permitir que um cliente apenas pudesse fazer uma aposta para um dado jogo. Assim, em caso de dúvida, voltaria a reenviar a mensagem e caso ela já tivesse sido processada, era apenas substituída. O problema seria, pelo menos no nosso caso, ao usar um vector, que teríamos sempre de percorrer toda a lista de modo a verificar se o cliente já tinha feito uma aposta para o dado jogo. Outras estruturas de dados e funcionamentos teriam de ser analisados. Como exemplo, poderíamos separar as apostas por cliente e dentro dessa separação, ter um campo destinado para cada jogo da ronda, onde era dito se havia ou não apostas já realizadas. Assim, teríamos de fazer duas procuras directas, dado o nome do utilizador e número do jogo em questão.

Contudo, apesar de todos estes pontos negativos que podem impossibilitar a nossa aplicação de crescer para um escala maior no imediato, consideramos o trabalho em muito satisfatório para os cenários propostos e requeridos, 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