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, 22 de Outubro de 2010

**Índice**

**Introdução2**

**Principais Estruturas do Sistema** **3**

Cliente (*sockets* TCP e RMI)3

Servidores4

Suporte à Persistência de Dados5

**Casos de Uso do “Front Office”7**

Alíneas 2.1 .7

Alíneas 2.2. – 2.3. 8

Alíneas 2.4. – 2.5. 9

Alínea 2.6.11

**Conclusão18**

**Bibliografia19**

**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 (sockets 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 três tipos de *threads* (designadamente as classes *TCPClient*, *ClientReadTCP* e *ClientWriteTCP*), cada uma com uma função distinta mas que trabalham em conjunto e dependem do contributo de todas.

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).

Após arrancar, esta *thread* fica encarregue de lançar as outras duas, mantendo-as em espera com recuso a uma quarta classe, *ConnectionLock*, que apenas funciona como um semáforo, informando se a ligação está activa ou não. As duas *threads* já lançadas ficam 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 as outras duas *threads*, acorda-as 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, fica por sua vez à espera que a ligação falhe.
* Se houver problemas com a ligação, então vai ser gerada uma excepção e as duas *threads* filhas continuam adormecidas. 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, *ClientReadTCP* tem como função ler toda a informação do *socket* que é naturalmente enviada pelo servidor, imprimindo mensagens caso seja necessário. (TODO: Se tirarmos esta thread, não esquecer de apagar as menções no relatório a ela).

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.

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.

(TODO: Inserir aqui o esquema)..

Quando o servidor é lançado, podemos observer 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 fomos nomeados servidores, 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.
* 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***

O envio de

**Iniciar uma Nova Sessão**

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*.

***Registar um Novo Cliente***

Como já foi referido anteriormente

***Efectuar o Login***

Como já foi referido anteriormente

**Bibliografia**

1. Java Eclipse
2. Ficheiros disponíveis na WOC sobre princípios básicos do *MATLAB*
3. Imagens: **Da capa:**
   1. http:// http://www.destination360.com/

**Gráficos -** Resultados obtidos pela execução do código elaborado no *MATLAB*.

1. “The Scientist and Engineer’s Guide to Digital Signal Processing” - Smith, Steven. *Edição online*.
2. Sites visitados:
   1. http://new.myfonts.com/
   2. http://www.bwin.com/