Tolerância a faltas em jogos de computador multi-utilizador

M.João Monteiro
mjoao_monteiro@clix.pt

Sandra Teixeira
steixeira@di.fc.ul.pt

Resumo

Actualmente, existe uma grande preocupação em dotar as aplicações distribuídas de mecanismos de tolerância a faltas. Neste artigo é apresentada uma possível integração de um jogo de computador multi-utilizador com um sistema de chamadas a procedimentos remotos a servidores replicados, de forma a tornar o jogo tolerante a faltas.

1 Introdução

Uma parte significativa das aplicações distribuídas baseia-se em chamadas a procedimentos remotos (RPC's). No entanto, este modelo não é tolerante a faltas do servidor, uma vez que se este falhar todo o sistema pára. Uma das formas de aumentar a tolerância a faltas de um sistema é a redundância espacial sob a forma de replicação. Assim, o servidor deixa de ser um único processo e passa a ser um conjunto de processos replicados. Um dos exemplos mais comuns de aplicações distribuídas são os jogos de computador multi-utilizador, daí a necessidade de aumentar a disponibilidade deste tipo de aplicações, recorrendo à replicação.

Neste artigo é proposta uma solução para introduzir mecanismos de tolerância a faltas em jogos de computador multi-utilizador. É utilizado um protótipo de um jogo distribuído que, apesar de bastante simples captura as interacções existentes em muitas concretizações disponíveis actualmente. Este protótipo é composto por um cliente e um servidor sem mecanismos de tolerância a faltas.

Dotou-se esta arquitectura de tais mecanismos, para permitir aos jogadores continuarem a jogar mesmo que uma réplica do servidor falhe. Para tal recorreu-se a um serviço, já desenvolvido, de chamadas a procedimentos remotos a servidores replicados.

Na secção seguinte são apresentados os componentes sobre os quais se baseou o desenvolvimento do sistema. A integração do sistema é definida na secção 3 ao apresentar as alterações feitas aos dois componentes e a arquitectura do sistema final.

Na secção 4 é abordada a forma como os pedidos são tratados pelos servidores e é apresentada uma alternativa.

Em seguida é discutida a recuperação de servidores e a secção 6 apresenta as conclusões.

2 Componentes da arquitectura

2.1 "Sistema de chamadas a procedimentos remotos a servidores replicados" [2]

Este sistema foi desenvolvido sobre o *Appia*[1] [1] e concretiza o tradicional modelo cliente-servidor com chamadas a procedimentos remotos (RPC's), aumentando a tolerância a faltas ao replicar o servidor de forma transparente para o cliente. Esta replicação pode ser activa ou passiva.

Neste sistema cliente-servidor, os clientes comunicam com os servidores usando comunicação ponto-a-ponto com um dos elementos do grupo de réplicas do servidor. Este elemento envia os pedidos totalmente ordenados para os restantes elementos usando comunicação em grupo.

Os clientes possuem uma camada que concretiza a lógica da aplicação (Client), uma camada que concretiza a lógica do RRPC (RRPC) e em baixo destas, uma pilha de protocolos que fornece FIFO e comunicação UDP. Os servidores possuem também uma camada (Server) com a lógica da aplicação e uma com a lógica do RRPC (Coordination). Estas camadas ficam em cima de uma pilha de protocolos que oferece ordem total, comunicação em grupo, FIFO, UDP e comunicação ponto-a-ponto.

2.2 Protótipo de um jogo multi-utilizador

O protótipo do jogo a utilizar consiste numa grelha que mantém o estado do jogo. Esta grelha é actualizada pelo servidor através dos pedidos dos clientes. Estes pedidos podem ser de dois tipos: pedido-movimento, que corresponde a uma actualização de posição ou pedido-disparo, que corresponde a um tiro. Após o processamento dos pedidos e consequente actualização da grelha, o servidor envia a grelha resultante para todos os clientes.

O jogador tem uma pilha com uma camada com a lógica do jogador (Player) e uma camada de comunicação TCP. O servidor tem uma pilha com uma camada com a lógica do servidor (Game), uma camada responsável por gerir os endereços dos clientes e distribuir as grelhas (Distribute) pelos mesmos e uma camada de comunicação TCP.

[1] http://appia.di.fc.ul.pt

3 Concretização

3.1 Alterações aos sistemas integrados

De forma a integrar os dois sistemas apresentados, foi necessário proceder a algumas alterações.

No protótipo do Jogo utilizado, quando um cliente faz um pedido, a grelha resultante é enviada para todos os clientes. Desta forma, o Jogo não funciona por RPC's uma vez que a cada cliente chegam respostas para os quais ele não emitiu pedidos. Para o jogo funcionar unicamente com RPC's, optou-se por alterar a filosofia do jogo. Deste modo cada vez que o servidor executa um pedido, em vez de enviar a grelha resultante para todos os clientes - função executada pela camada Distribute que na solução desenvolvida desaparece - a grelha é enviada unicamente para o cliente que emitiu o pedido. Deste modo a única forma de um cliente receber uma actualização da grelha é através da emissão de um pedido.

Para permitir que o cliente receba actualizações da grelha sem ter executado um movimento ou tiro, foi acrescentado ao jogo um novo pedido (pedidoGrelha) que é efectuado automaticamente. O jogador define, quando começa a jogar, o período de refrescamento que pretende para o seu jogo. De acordo com este período *x,* o sistema executará um pedidoGrelha no servidor. Se um jogador executar movimentos com um intervalo menor que o período de refrescamento, o pedidoGrelha não é efectuado, mas se o jogador for lento e os seus pedidos demorarem mais do que o período definido, então o sistema executará um pedidoGrelha que irá actualizar o estado da grelha no jogador, mesmo sem este fazer qualquer movimento ou tiro. Deste modo o jogador consegue ter um estado do jogo tão actual quanto o desejado.

O protótipo Jogo deixa de ser orientado aos eventos e passa a ser orientado ao tempo, uma vez que o jogador recebe de certeza uma grelha nova no período de tempo definido.

Como um jogo é uma aplicação caracterizada por interacções regulares, uma aproximação orientada ao tempo torna-se mais apropriada uma vez que, por exemplo, cinco grelhas diferentes num segundo não permitiriam ao jogador reagir nesse segundo. Assim, ao jogador interessa apenas receber actualizações de *x* em *x* tempo para que possa reagir. Este *x* tem que ser curto o suficiente para acompanhar a evolução do Jogo e longo o suficiente para o jogador conseguir reagir nesse intervalo [3].

O protótipo do Jogo foi também alterado de forma a retirar a computação não determinista que existia na escolha da posição inicial do jogador, uma vez que esta era feita aleatoriamente. Agora, o jogador é colocado sempre numa posição pre-definida, a menos que esta esteja ocupada. Se isto acontecer, o jogador será colocado na casa seguinte.

// O sistema de Chamadas a Procedimentos Remotos foi também alterado, para suportar pedidos de tipos de leitura e de escrita. Pedidos de leitura de estado não serão enviados para todas as réplicas como os pedidos de escrita, uma vez que não farão nenhuma alteração ao estado. Assim, de acordo com o protótipo do Jogo, pedidos de movimento ou tiros serão pedidos de escrita e o pedidoGrelha será um pedido de leitura. A primeira réplica do servidor a receber um pedido de leitura, executa-o e envia a resposta ao cliente sem o conhecimento das restantes réplicas.

Ao nível da comunicação, as camadas FIFO e UDP do sistema de Chamadas a Procedimentos Remotos foram substituídas pela camada de TCP que é igualmente suportada por este sistema, é a utilizada no protótipo do Jogo e é mais rápida.

3.2 Descrição geral

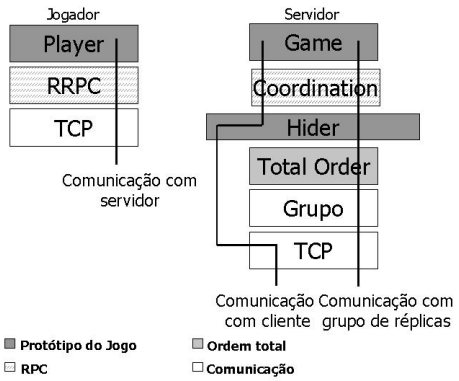

Figura 1. Arquitectura do sistema

A arquitectura desenvolvida, figura 1, consiste na alteração do Jogo para que este passe a tratar todos os pedidos por RPC's, permitindo assim, a integração dos dois sistemas.

Na solução encontrada existem clientes e um grupo de réplicas do servidor.

Aos clientes que, no protótipo do Jogo, eram compostos apenas por duas camadas - a Player e a TCP - foi acrescentada a camada RRPC do sistema de Chamadas a Procedimentos Remotos, que lhe permitirá aceder de uma forma transparente às réplicas do servidor. Esta camada foi inserida entre as duas camadas anteriores.

A composição do servidor no Jogo foi alterada pois a camada Distribute desaparece e, por baixo da camada Game, foi introduzida a Coordination, o protocolo de ordem total e a pilha de comunicação em grupo. Por baixo da camada Coordination é ainda inserida uma nova camada Hider, cujo papel será descrito mais abaixo.

A camada Distribute do protótipo não é necessária, uma vez que a resposta ao pedido vai unicamente para o cliente que o executa - contexto de RPC's.

Para que o sistema funcione é necessário que as camadas que compõem a arquitectura estejam preparadas para comunicarem entre si. Uma vez que ambos os sistemas foram desenvolvidos sobre o *Appia*, as camadas que os compõe comunicam entre si através de eventos específicos. No caso das camadas utilizadas do sistema de Chamadas a Procedimentos Remotos, estas trocam entre si e esperam receber das restantes eventos do tipo RpcEvent. As camadas

Game e Player tiveram que ser alteradas para receberem e enviarem este evento, em vez dos que utilizavam anteriormente.

Na solução desenvolvida, optou-se pela replicação activa, uma vez que a recuperação se torna menos problemática e não existem no protótipo do Jogo computações não deterministas.

Na replicação activa, as várias réplicas do servidor processam os pedidos [3]. Para assegurar a coerência de estado, os pedidos efectuados pelos clientes a uma das réplicas são difundidos pelas restantes, por difusão atómica. Assim, todas as réplicas recebem exactamente os mesmos pedidos pela mesma ordem, garantindo que produzem as mesmas grelhas como resultado, uma vez que são apenas considerados servidores deterministas. Todas as réplicas enviam a grelha para o cliente, mas uma vez que o serviço de replicação é transparente para o cliente, a camada RRPC descarta as respostas duplicadas que lhe chegam.

3.2.1 Camada Hider

A camada Hider foi introduzida no sistema para evitar que todas as réplicas do servidor enviassem respostas para os clientes. Assim, existe apenas uma réplica responsável por enviar as respostas para os clientes, se esta réplica falhar será eleita outra. O servidor apresenta, assim, um comportamento passivo no envio das respostas ao cliente, mantendo o comportamento activa no processamento dos pedidos.

Esta optimização não poderia ser feita ao nível da camada Game uma vez que a Coordination espera receber as respostas aos pedidos que enviou para cima. A optimização tem, então, de ser feita abaixo da Coordination.

Desta forma, a camada Hider apanha a resposta enviada pela camada Coordination e só a envia para o cliente se for a réplica responsável, caso contrário descarta-a. Assim, no cliente é recebida apenas uma resposta. As respostas aos pedidos de leitura não são descartadas na camada Hider, uma vez que apenas uma réplica trata este tipo de pedidos.

4 Disparos atómicos vs não atómicos

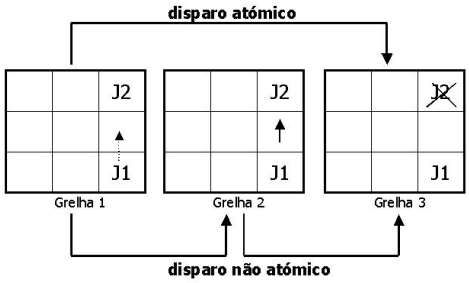

Figura 2. Exemplo de um disparo

Na figura 2 está representada a diferença entre disparos atómicos e disparos não atómicos.

No protótipo do Jogo estão concretizados os disparos atómicos. Este tipo de disparos é processado instantaneamente, ou seja, quando o servidor recebe um pedido-disparo de um cliente - Grelha 1 da figura 2 -, processa de imediato o efeito que esse disparo causa na disposição do jogo - Grelha 3 da figura 2. Assim, os jogadores que hipoteticamente se encontrem, naquela grelha, na direcção do disparo não conseguem actuar em resposta ao disparo.

Este tipo de disparos é uma limitação do protótipo do Jogo que pode ser ultrapassada se considerarmos outro tipo de disparos - os não atómicos.

Os disparos não atómicos permitem que os jogadores se apercebam do movimento dos projécteis e possam desviar-se antes de serem atingidos. Assim, o servidor ao receber um pedido-disparo do cliente - Grelha 1 da figura 2 -, desloca o projéctil apenas uma posição, processa uma grelha neste estado - Grelha 2 da figura 2 - e guarda a direcção do projéctil para continuar o processamento mais tarde. Os jogadores ao receberem esta grelha, por resposta a um pedido, podem decidir deslocar-se, enviando, para tal, um pedido-movimento ao servidor. Este vai processando os pedidos que lhe chegam assim como vai continuando o processamento do projéctil pendente, até este atingir um jogador, uma "parede" ou outro projéctil - Grelha 3 da figura 2.

4.1 Concretização de disparos não atómicos

A concretização dos disparos não atómicos baseou-se na alteração do comportamento do servidor na recepção de pedidos-disparo. Ao receber um pedido de um cliente para processar um disparo, o servidor processa uma grelha com o primeiro movimento do projéctil e activa um alarme para tratar posteriores movimentos. Entretanto, podem chegar novos pedidos ao servidor, inclusivé novos disparos, que são tratados de imediato.

A ideia inicial era gerar um alarme para cada disparo, no entanto esta solução, para uma grelha com vários disparos concorrentes mostrar-se-ia pouco eficiente. Desta forma, optou-se por gerir os disparos todos com um mesmo alarme.

Quando esse alarme dispara, o processamento dos projécteis em questão é retomado, é processada a respectiva grelha e activado novo alarme, caso ainda hajam projécteis na grelha.

4.2 Impacto dos disparos não atómicos no sistema

A concretização sugerida no ponto anterior levanta alguns problemas quando o servidor está replicado de uma forma activa. As réplicas do servidor estão todas a executar pela mesma ordem todos os pedidos. Um pedido-disparo, feito por um cliente, chegará a todas as réplicas exactamente pela mesma ordem mas não no mesmo instante temporal devido a eventuais atrasos da rede ou de processamento.

Este facto poderá originar incoerências entre as diferentes réplicas do servidor. Por exemplo, tendo duas réplicas do servidor, *r1* e *r2*, e um pedido-disparo *pd*, considere-se o possível caso: O cliente envia *pd*. A réplica *r1* recebe

pd antes de *r2*, logo gera o alarme primeiro. Quando o alarme de *r1* dispara, esta usa a última grelha - *g1* - para processar o pedido gerando *g2*. Entretanto, após o disparo do alarme em *r1* e antes do disparo em *r2*, é recebido um pedido-movimento de um jogador. Este pedido é, então, executado em *r1* após o disparo do alarme - grelha *g2* - mas antes do alarme em *r2* - grelha *g1*. Assim, *r1* e *r2* executam o pedido-movimento em grelhas diferentes o que vai originar grelhas resultado diferentes. Estas grelhas podem propagar esta incoerência entre as réplicas aos clientes.

4.2.1 Solução

O problema anterior acontece porque os pedidos do cliente seriam entregues às réplicas por ordem total mas os alarmes, uma vez que são específicos de cada réplica, podiam aparecer em cada réplica em instantes diferentes.

A forma de resolver o problema consistiu em ordenar os alarmes juntamente com os pedidos.

Numa primeira aproximação, quando um alarme disparasse numa réplica, em vez desta proceder à execução imediata dos disparos, enviava uma mensagem para todo o grupo de réplicas informando que o seu alarme tinha disparado. A mensagem seria, então, ordenada juntamente com os pedidos e recebida em todas as réplicas exactamente pela mesma ordem.

Esta solução obrigava a que as réplicas do servidor trocassem mensagens entre si, o que comprometeria a transparência da replicação para o servidor. Mesmo sabendo que o sistema de Chamadas a Procedimentos Remotos não garante a transparência no servidor, optou-se por concretizar a gestão nos alarmes na camada Hider - figura 1- libertando assim a camada Game desta responsabilidade ao mesmo tempo que, não se compromete a transparência que deveria existir.

Assim, quando a camada Game quiser activar o alarme para os disparos envia uma AlarmMessage para a camada Hider - o pedido de activação de um alarme para daqui a *x* tempo. A camada Hider ficará responsável por accionar esse alarme. Quando o alarme disparar, uma das camadas Hider enviará a AlarmReplicasMessage para as Hider's das restantes réplicas - informação que o seu alarme já disparou. A existência de uma Hider mestra para enviar as AlarmReplicasMessage para o grupo evita o excesso de mensagens. A replicação torna-se passiva, não só no envio das respostas ao cliente, como descrito anteriormente, mas também no tratamento dos alarmes.

Na recepção de uma AlarmReplicasMessage, a camada Hider envia, para a camada Game, a informação de que pode processar os disparos.

Cada vez que a camada Game receber uma AlarmMessage, desloca todos os disparos da grelha uma posição e envia nova AlarmMessage, se necessário, para a camada Hider.

5 Reintegração de servidores

Para lidar com a camada Coordination do sistema de Chamadas a Procedimentos Remotos tornou-se necessária a existência de pelo menos duas réplicas do servidor em todas as execuções.

Num contexto de tolerância a faltas impõem-se os problemas de reintegração de réplicas novas e de atribuição de responsabilidades no caso de uma réplica falhar.

Quando uma réplica do servidor falha, a recuperação do sistema é linear, uma vez que se dá a reeleição das réplicas mestras para prevenir o caso em que foi uma das mestras a falhar. As réplicas mestras, quer para o envio das respostas quer para os alarmes, poderão começar a executar de imediato uma vez que a informação necessária se encontra replicada por todas as réplicas - replicação activa.

Quando entra uma nova réplica no grupo é necessário actualizar o seu estado de acordo com o das réplicas existentes. Esta transferência de estado é feita quer a nível da camada Game, com a informação sobre o estado actual da grelha, quer a nível da camada Hider, com a informação sobre o alarme.

6 Conclusões

Neste artigo foi apresentada uma solução para introduzir mecanismos de tolerâncias de faltas num jogo de computador multi-utilizador. Uma das técnicas mais utilizadas para introduzir tolerâncias num sistema distribuído é a replicação - quer activa, quer passiva. Para replicar o servidor do protótipo do Jogo utilizado, recorreu-se a um sistema de Chamadas a Procedimentos Remotos a servidores replicados. Além dos problemos intrínsecos à integração dos dois sistemas, cujas soluções estão descritas neste artigo, surgiram no sistema de Chamadas a Procedimentos Remotos vários erros de concretização, que faziam com que o sistema não fornecesse o serviço previsto.

Referências

[1] H. Miranda A. Pinto, L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. *Proceedings of the 21st International Conference on Distributed Computing Systems*, páginas 707–710, April 2001.

[2] P. Vicente J. Martins. Arquitectura de um sistema de chamadas a procedimentos remotos a servidores replicados. 2001.

[3] P. Veríssimo L. Rodrigues. Distributed systems for systems architects. 2001.