Pong Multiplayer

Ana Cotrim
i28348@alunos.di.fc.ul.pt

Bruno Gonçalves
i26512@alunos.di.fc.ul.pt

João Sequeira
i26546@alunos.di.fc.ul.pt

TFD 009

Abstract

Neste artigo descreve-se uma possível implementação de uma versão multijogador, através de uma rede, do famoso jogo Pong. Para tal implementação iremos usar a linguagem Java e primitivas de sincronismo virtual e comunicação em grupos fornecidas pelo APPIA. Desta forma procuramos aumentar a fiabilidade e disponibilidade, permitindo que um jogador entre e saia do jogo sem qualquer interferência para os outros jogadores.

1. Introdução

Desde os anos 70 quando a Atari criou o pong, na altura uma máquina enorme que possuía 2 rodas que permitiam mover os pads de ambos os lados. Estas máquinas na altura tiveram um sucesso tal que serviram para cobrir os $700.000, que a Atari teve de pagar a uma outra empresa por esta já ter patenteado o conceito dos pads presente no Pong, e ainda obtiveram lucros invejáveis. Mais tarde apareceram outras versões do jogo nomeadamente o quad pong que permitia 4 jogadores na mesma máquina.

Com a massificação da Internet começaram também a aparecer os jogos multijogador usando-a como base de comunicação. Como a Internet está longe de ser um meio de comunicação fiável e com toda a procura por jogos on-line existe uma contínua busca pelas software houses por novos e melhores protocolos para fornecer aos seus utentes.

Nos dias que correm a grande maioria dos jogos on-line usam um servidor centralizado onde reside o"jogo" em si, sendo os clientes meramente terminais gráficos que apresentam o resultado que o servidor lhes indicou. Nesta situação existem alguns problemas, como congestão do servidor, este torna-se no único ponto por onde toda a informação tem que passar, ou seja, se o número de jogadores é elevado, então o servidor terá uma carga adicional a suportar, tornando-se lento; outro problema é que o servidor é o único ponto de falha, quando este pára, então todo o jogo pára, no entanto se um cliente falhar, ninguém se importa.

Neste artigo iremos descrever uma solução que permita que haja falhas de servidores e de jogadores, que no caso se trata do mesmo, já que cada jogador irá ter uma réplica global do sistema. Permitindo assim a entrada e saída de jogadores, quer por desistência ou por "crash".

Com este tipo de arquitectura procuramos aumentar a fiabilidade e disponibilidade dos jogos on-line, já que esta se pode extrapolar para a maioria destes jogos, eliminando assim os servidores centralizados e todas as suas desvantagens.

O jogo que nos propomos a implementar trata-se de uma versão para quatro jogadores do Pong, colocando cada jogador em cada aresta da arena, ou seja ao contrário do pong normal as arestas superiores e inferiores da arena não fazem ricochete automático sendo sim defendidas por um jogador cada uma.

No caso considera-se que um jogador perde um ponto quando deixa que a bola bata na parede que deveria estar a defender.

Existem 2 problemas inerentes a este tipo de arquitecturas, que neste jogo se reflectem no sincronismo da posição da bola e no movimento dos pads dos jogadores, o outro problema refere-se ao consenso das situações de golo.

Estes algoritmos serão implementados em Java, e toda a comunicação entre os vários servidores/jogadores terá como base as primitivas de sincronia virtual e suporte a comunicação em grupo disponibilizados pelo APPIA [1].

A razão da escolha desta combinação Java e APPIA deve-se ao facto de o APPIA permitir a criação e conjunção de microprotocolos tendo a vantagem de já existirem alguns disponibilizados, permitindo-nos dar mais enfŝe à resolução do nosso problema e não a todos os problemas que a comunicaç ão em grupos levanta.

O Java por sua vez permite uma implementação independente da plataforma.

Nos próximos 2 capítulos iremos abordar ambos os problemas descritos acima de forma detalhada apresentando os algoritmos por nós propostos para a sua resolução.

No capítulo 3 iremos abordar os microprotocolos do AP-

PIA usados na implementação.

No capítulo 4 indicaremos os eventos APPIA usados nesta implementação.

2. Eventos

A comunicação entre os vários servidores é feita através de eventos tratados pelo APPIA que atravessam um canal composto por micro-protocolos definidos no próprio APPIA e seleccionados de forma a criar o canal de comunicação necessário à implementação dos protocolos descritos mais à frente.

Os tipos de mensagens são:

Reset, que é enviado por um servidor quando há necessidade de repôr a bola na posição inicial e recomeçar o seu movimento. Não espera por confirmação de todos.

StartEvent, enviado quando se quer dar início ao movimento da bola. Qualquer jogador pode gerar este evento. A bola deve só começar o movimento após a confirmação da parte de todos para garantir que todos receberam o evento sem atrasos. É semelhante ao ResetEvent, mas sem posicionar a bola na sua posição inicial.

Stop é enviado por quem quiser pausar o jogo. A bola só pára quando todos confirmarem que vão parar a bola para garantir que se um servidor receber o evento atrasado, pára a bola na mesma posição que os outros.

Goal, enviado por todos quando detectado um golo. Todos devem confirmar o golo, e quem não o fizer deve ser corrigido pelos outros servidores. Contém um parametro denominado 'status' que contém a decisão do servidor que o emite, o seu valor é TRUE ou FALSE, confirmando ou negando o golo. O servidor que detecta o golo primeiro tem sempre este valor a TRUE.

Update, dá início ao protocolo de ressincronização da posição da bola e dos pads com os outros servidores. Também é usado por um servidor quando este entra em jogo, para ir buscar a posição actual da bola. Contém parâmetros que com as coordenadas da bola e dos pads.

Quit avisa os outros servidores de que o jogador saiu de jogo.

Move, informa os servidores de que o jogador do servidor emissor moveu o seu pad. Contém um parametro que contém as novas coordenadas do pad.

Além dos eventos acima descritos, existem também eventos de confirmação:

UpdateAck, confirmação de actualizações (Update).

StartAck, confirmação de início de jogo (Start).

StopAck, confirmação de paragem do jogo (Stop).

3. Pilha de protocolos usada

O APPIA é constituído por um kernel base a que são adicionados um ou mais micro-protocolos ou Layers, cujo conjunto define o protocolo de comunicação ou quality of service (QoS) usado nas sessões.

3.1 Difusão Atómica

Esta camada do appia vai permitir que haja garantias que se uma mensagem é entregue a um dos membros do grupo então todos os membros correctos desse grupo a recebem, no caso toma-se como membros correctos os que aparecem na vista seguinte, abordaremos o conceito de vistas na secção seguinte.

O método de funcionamento deste protocolo passa pelo reenvio de mensagens até que todos os membros do grupo recebam a mensagem ou então se detecte a falha de um membro e se instale uma nova vista no sistema.

A correcção deste protocolo resume-se às seguintes propriedades:

-Validade: Se um processo envia uma mensagem essa mesma mensagem é entregue aos restantes processos do grupo num tempo indeterminado mas finito.

-Unanimidade: Se uma mensagem é entregue a um processo correcto de um grupo então essa mesma mensagem é entregue aos restantes processos correctos do grupo.

Esta camada vai permitir à aplicação uma abstracção do canal de comunicação por parte da aplicação devido as fortes garantias que fornece acerca da comunicação.

3.2 Sincronia Virtual

A sincronia virtual fornece uma lista de processos que poderão estar correctos numa determinada altura da execução. Essa lista de processos é igual para todos os processos correctos, sendo utilizada na difusão atómica para se saber de quais processos esperar resposta.

A uma lista de processos correctos numa determinada altura dá-se o nome de vista, podendo ser alterada ao longo da execução do algoritmo quando se detecta uma falha num processo ou quando um processo se deseja juntar ao grupo.

Para a implementação iremos usar o suporte que o APPIA fornece à sincronia virtual.

3.3 Ordenação de mensagens FIFO

A garantia de ordenação de mensagens que iremos usar na implementação do protocolo será FIFO (First In First Out).

Este tipo de ordenação não dá garantias nenhumas em relação a mensagens oriundas de máquinas diferentes. No entanto se um servidor envia m1 antes de m2 então todos os processos correctos recebem m1 antes de m2.

Para o tipo de implementação escolhido torna-se irrelevante a ordem de entrega de mensagens vindas de servidores/jogadores diferentes, visto que cada um apenas está

autorizado a mover o pad correspondente, e os pedidos de sincronização são executados simultaneamente por 2 ou mais servidores é irrelevante para o sistema se está a executar o pedido vindo do servidor 1, 2, 3 ou 4.

Esta camada no APPIA também oferece garantia de entrega, foi tomada a decisão de utilizar um protocolo com garantia de entrega devido ao facto de este protocolo ser mais rápido a detectar falhas na rede e fazer as retransmissões.

4 Algoritmos usados para a implementação:

Há dois métodos usados nestes protocolos, o Majority e o Average. O primeiro tenta descobrir o valor que mais aparece dentro de um array, caso não encontre nenhum em menor número, devolve o primeiro da lista. O segundo faz uma média de todos os valores de um array, e é usado em situações em que o Majority não deve funcionar, como por exemplo situações em que constantemente os valores contidos nos arrays são diferentes entre si.

4.1 Algoritmo de sincronização da bola e dos pads:

Este algoritmo é usado quando se quer sincronizar as posições dos elementos do jogo com os restantes servidores.

```
1- Processo p quer sincronizar:
    broadcast Update;

2- Quando recebe Update de q:
    sincronismos[q] =
       Update.coordenadas(bola e pads);
    #sincronismos++;

3- Quando #sincronismos >= #p.correctos:
    posicao(bola) =
          Average(sincronismos(bola));
    posicao(pads) =
          Average(sincronismos(pads));
    broadcast (UpdateAck);
//todos actualizados, podem continuar

4- Quando recebe UpdateAck de q:
    acks[q] = TRUE;

 5- Quando #acks >= #p.correctos:
     //continuar o movimento da bola
```

Os processos correm este algoritmo de 10 em 10 movimentos de bola. Quando um processo entra em jogo, executa este algoritmo antes de movimentar a bola para estabelecer as posições da bola e dos pads actuais. É usado o procedimento Average porque pode ser frequente todos os servidores estarem dessincronizados, o que tornaria o uso do Majority inútil.

4.2 Algoritmo de consenso de Golo:

Este algoritmo é usado quando um golo é detectado por um servidor, ou seja, quando a bola toca na fronteira de um jogador.

```
1- Processo p detecta golo:
     broadcast(Goal+true);

2- Quando p recebe Goal de q:
     if nao fez broadcast de goal goal e nao
     esta em situacao de golo
          broadcast(Goal+false)

    goals[q] =  Goal_status;
     #goals++;

3- Quando #goals >= #correctos:
    if Majority(goals[]) = TRUE;
        broadcast(Reset);
//para repor a bola na posicao inicial
     else
         if goals[p] = TRUE;
            broadcast(Update);
//para  receber a posicao real da bola
         else
//nao fazer nada,
//os que erraram vao fazer update
```

Quando um golo é detectado, o servidor fica à espera que todos os servidores lhe respondam, confirmando se também dectaram o golo, ou se por outro lado, o golo nunca sucedeu. No caso de o golo não se confirmar, então o servidor começa a fase de sincronização, para actualizar a posição da bola e dos pads. A falta de sincronismo é a mais provável razão para que um golo seja mal detectado.

5 Optimizações

A seguir apresentam-se algumas optimizações a aplicar aos algoritmos propostos em 4.1 e 4.2, alterando variáveis como o tráfego na rede ou a precisão de coordenadas.

5.1 Sincronismo adaptativo

No processo de sincronização, em vez de se ter um valor fixo de movimentos até à próxima actualização, poderemos ter um algoritmo adaptativo às condições ambientais.

Começando com um valor inicial de 10 movimentos, no caso de todos se manterem sincronizados durante a fase de sincronização, ou seja, se nenhum tiver que alterar as suas coordenadas, então ele aumenta o número de movimentos até à próxima sincronização, e no caso de os valores apresentarem uma diferença significativa, ou seja, os servidores apresentarem coordenadas diferentes entre si, então reduz-se esse valor.

5.2 Golo predictivo

Um golo é detectado quando a bola atinge uma parede. Nesta situação, se um servidor que esteja adiantado em relação aos outros detecta um golo, e aos outros ainda falta um ou mais pixeis para atingir a parede, o golo não será validado até todos atingirem a parede, ou seja, o servidor adiantado terá que se sincronizar com os outros, recuando a sua posição da bola. Esta situação gera um excesso de tráfego redundante, que poderia ser evitado caso os servidores tivessem uma forma de predizer se o golo é ou não inevitável. Caso concluam que um golo assinalado é de facto inevitável, então aceitam o golo, difundindo o status da mensagem Goal a TRUE em vez de FALSE como seria de esperar segundo o algoritmo base.

6 Conclusão

Esta implementação do Pong apresenta uma ausência de servidor centralizado, tornando-o mais tolerante a faltas. Esta ausência de centralização, contudo, gera um volume de tráfego mais elevado e exige mais processamento local. Apresenta também problemas relacionados com sincronismo (posição dos objectos dispostos no ecrã) e consenso (decisão de golo). De forma a resolver estes problemas, foram propostos dois algoritmos capazes de lidar com estas duas situações. Apesar disto, apresentam alguns problemas de tráfego e falta de adaptação às condições da rede. Devido a isto, foram apresentadas duas optimizações que lidam com cada um destes problemas.

References

[1] Hugo Miranda, Alexandre Pinto, Luís Rodrigues. Documentação do Appia. (http://appia.di.fc.ul.pt/documents.htm) Fevereiro 2001.

[2] Paulo Veríssimo, Luís Rodrigues. *Distributed Systems for System Architects*. Kluwer Academic Publishers, 2001.

[3] Rachid Guerraoui, Luís Rodrigues. *Abstractions for Distributed Programming (Preliminary Draft)* Outubro 2003.

[4] Andrew Tanenbaum, Maarten van Steen *Distributed Systems Principles and Paradigms* Prentice Hall, 2002.