Jogo Multi-Utilizador, Distribuído e Tolerante a Faltas

José Pedro B. Sequeira
i29209@alunos.di.fc.ul.pt

Daniel Neves
i28598@alunos.di.fc.ul.pt

Abstract

O objectivo deste trabalho é criar um jogo multi-utilizador bastante simples, nomeadamente, o Pong. No entanto pretende-se que este jogo, que será distribuído, seja tolerante a faltas, sendo que, a falha de um servidor não obrigue a que o jogo seja terminado. Este artigo descreve uma proposta de concretização tolerante a falhas do jogo descrito.

1. Introdução

Lamport definiu um sistema distribuído da seguinte forma: "A distributed system is a system where I can't get any work done if a machine I've never heard of crashes". Para evitar este tipo de inconvenientes surgiram vários sistemas tolerantes a faltas. Para assegurar que, quando uma máquina falha, o sistema continua operacional, utiliza-se, entre outras técnicas, o modelo de replicação. Assim se uma máquina fica impossibilitada de fornecer um serviço, uma réplica desta máquina "toma o seu lugar", fornecendo o mesmo serviço. Este modelo deve ser completamente transparente para os utilizadores, que nem se apercebem que estas réplicas existem. Como exemplo temos a Internet, na qual os servidores estão replicados (não só para tolerar faltas, mas também para balanceamento de carga) e ninguém dá por isso, não sendo necessário mudar o endereço URL sempre que queremos comunicar com uma réplica diferente. É natural que um dos requisitos em aplicações distribuídas seja a tolerância a faltas. Por esta razão, descrevemos um jogo tolerante a faltas, capaz portanto de continuar a sua execução normal em caso de falha do servidor.

É descrito na secção 2 o funcionamento do jogo, na secção 3 é discutida a arquitectura geral do sistema e na secção 4 uma é feita uma apresentação sucinta da nossa proposta de implementação, focando-se na utilização do Appia.

2. Jogo: Pong

Apresentamos uma versão do clássico jogo de computador Pong, originalmente desenvolvido pela Atari. Na sua versão original participavam apenas dois jogadores. No entanto, o jogo que pretendemos desenvolver será jogado com quatro jogadores. O jogo ocorre num campo quadrado, sendo cada jogador responsável por proteger uma das paredes do campo de uma bola que se move continuamente. Para este efeito, cada jogador controla um cursor (padlle) que pode deslocar paralelamente à sua parede. A bola faz ricochete quando embate num cursor ou nas paredes que delimitam o campo. Se um jogador deixar a bola bater na sua parede soma um ponto. Será vitorioso o jogador que obtiver menos pontos.

A implementação que propomos permite o jogo online, em que cada jogador se encontra numa máquina diferente, e comunica com os outros através de protocolos de rede. Em vez de os jogadores terem de estabelecer ligação directa com os seus adversários(o que implicaria conhecer os seus endereços IP e portos), terão que se registar com um servidor. Assim, durante o decorrer do jogo, os clientes (jogadores) enviaram os seus movimentos do cursor (paddle) ao servidor, que será responsável pela actualização do estado do jogo nos clientes. O servidor serve assim de coordenador do jogo.

3. Arquitectura do Sistema

O Pong será concretizado usando uma arquitectura cliente servidor. Cada jogador regista-se num servidor, ao qual indica os movimentos do seu cursor (paddle). O servidor é responsável por actualizar todos os jogadores de alterações ao estado do jogo (movimentos dos jogadores e da bola). Os clientes não comunicam entre si, para evitar clientes "batoteiros". Deste modo asseguramos, também, que a falha de um cliente não implica que o jogo termina. Será mantida uma lista de espera de clientes, e quando um jogador falha ou desiste outro poderá tomar o seu lugar. Para facilitar

a concretização do sistema, apenas um jogo de Pong poderá estar a decorrer. Num trabalho futuro poder-se-ia estender o sistema para permitir vários jogos em simultâneo. Outra simplificação será que todos os jogadores estarão a falar com o mesmo servidor, não permitindo assim existir balanceamento de carga. Sendo que os servidores devem estar coerentes, isto é, devem conter a mesma representação do jogo que todos os outros, não seria difícil, num trabalho futuro permitir que os clientes comunicassem com servidores arbitrários.

O Appia será usado como middleware. Será o Appia que permitirá a transparência aos clientes, que apenas conhecem um endereço IP e um porto. O Appia também permitirá a comunicação em grupo entre os servidores, que necessitam de trocar informação do estado de jogo. Estes servidores, como já foi referido, devem estar sempre coerentes com os outros para que não ocorram fenómenos estranhos quando um servidor "morre" e há uma troca de servidor principal. Um efeito negativo, se tal não for garantido, seria um salto abrupto no estado do jogo podendo existir até um salto para uma pontuação anterior, proporcionando um grande transtorno aos jogadores.

4. Appia

Um grupo, em sistemas distribuídos, é um conjunto de máquinas que comunicam entre si. O Appia fornece primitivas para a comunicação em grupo, tais como enviar mensagens ao grupo, juntar ou retirar um elemento ao grupo e controlo de vistas (quem é que está no grupo). O Appia concretiza a sincronia na vista ou sincronismo virtual, isto é, ao actualizar a vista V0 envia um bloqueio, para garantir que uma mensagem enviada na vista V0 é recebida apenas pelos elementos dessa vista V0, e que todos os elementos dessa vista V0 que transitem para a próxima vista V1 recebem a mensagem. Esta característica é importante para a concretização da arquitectura proposta na secção anterior, pois ajuda a garantir a coerência dos estados dos servidores.

Outro aspecto a considerar é a ordenação das mensagens. Será necessário, num jogo com estas características, que as mensagens provenientes do servidor sejam entregues por uma ordem bem definida? Pensamos que sim, e o Appia fornece a possibilidade de garantir ordem causal e total. Na aplicação do Pong, vamos garantir que todos os clientes recebam as mensagens pela mesma ordem (para que não ocorram discrepâncias entre o jogo de cada um dos jogadores), usando a ordenação total, baseada em sequenciador, fornecida pelo Appia

Outro requesito para que seja garantida a coerência dos estados dos servidores é que todas as operações tomadas localmente por um servidor (e que afectem o seu estado) sejam deterministas. À primeira vista pode parecer que as únicas alterações ao estado, não-deterministas, desta aplicação, se prendem com o movimento do paddle da parte dos clientes, que são, é claro, imprevisíveis. No entanto, devido à sua estreita ligação com o relógio interno das máquinas, o movimento da bola não é, de facto, determinista. Assim, teremos que escolher uma máquina no sistema para ser responsável por esta operação. Escolhemos um servidor (escolhido arbitráriamente) a que chamamos coordenador, e que será responsável por propagar o seu estado aos outros (replicação passiva).

Assim, o servidor com identificador menor será o servidor com o qual todos os clientes comunicam, e será este que enviará as mensagens e a respectiva ordem aos clientes.

4.1. A Pilha Protocolar

O Appia funciona definindo uma pilha, em que cada camada corresponde a um protocolo, e que fornece uma determinada Qualidade de Serviço (QoS). Para o desenvolvimento deste projecto utilizaremos a seguinte configuração:

TotalAbcastLayer Esta camada é responsável pela ordem de entrega das mensagens aos vários membros do grupo.

VSyncLayer Esta camada é responsável por garantir sincronia na vista. Quando a VSyncSession recebe uma nova vista, envia uma mensagem Block a todos os membros da vista antiga. Neste momento ninguém pode enviar mensagens, até que uma mensagem BlockOk seja recebida. Deste modo é possível sincronizar as vistas, não existindo, num determinado instante, mensagens destinadas a vistas antigas.

LeaveLayer Esta camada oferece a possibilidade de um membro sair do grupo.

StableLayer Esta camada é responsável por oferecer o reliable broadcast [1]. Isto é, garante que todos os membros vivos recebem as mensagens.

HealLayer Esta camada é responsável por detectar vistas concorrentes. Usa o GossipServer (semelhante a um servidor de nomes) para descobrir novos membros.

InterLayer Esta camada permite que várias vistas se juntem numa única. Isto é feito usando o algoritmo flooding consensus [1], que decide qual a ordem das uniões de vistas.

IntraLayer Esta camada executa as mudanças de vista.

SuspectLayer Detecta falhas nos membros.

MergeOutLayer Camada responsável pela comunicação entre as várias vistas que se fundem.

GossipOutLayer Camada responsável pela comunicação com o Gossip.

GroupBottomLayer Interface entre as camadas de comunicação em grupo e a comunicação ponto a ponto.

TCPCompleteLayer Interface entre as camadas de comunicação em grupo e as sockets TCP.

5. Conclusão

Durante o próximo mês será desenvolvida a aplicação Pong tal como foi descrita anteriormente. O objectivo será ganhar familiaridade com aplicações tolerantes a faltas e com o Appia. Esta será concebida para que seja possível a ocorrencia de falhas durante o decorrer do jogo, sem que isso implique que o jogo termine. Concentrar-nos-emos principalmente nas falhas do tipo "crash", em que um servidor (incluido o coordenador) deixa de comunicar com o sistema. A falha será transparente aos clientes, que, eventualmente, poderão aperceber-se que ocorreu o pequeníssimo salto no decorrer do jogo (devido aos timeouts para detectar a falha do servidor).

References

[1] Rachid Guerraoui and Luis Rodrigues, *Introduction to Reliable Distributed Programming,* (Springer, 2005).