Jogo Multi-Utilizador Em Tempo-Real: 4Pong

Tfd06: Fernando Felício 27908 Susana Guedes 27860 Valter Conceição 28011

December 9, 2003

*Abstract: : Existe uma procura crescente de sistemas distribuídos de tempo-real fiáveis por toda a comunidade informática. São necessárias novas técnicas para estas aplicações perpetuarem face a alterações nas condições da rede. Este artigo apresenta uma implementação do clássico jogo Pong da Atari à qual chamámos **4Pong**. Face às suas básicas necessidades de implementação, são apresentados aspectos fundamentais para implementar qualquer software síncrono multi-utilizador em tempo-real. A funcionalidade chave baseia-se na existência de, não dois, mas quatro jogadores em simultâneo.*

1 Introdução

Ao longo dos tempos tem-se verificado um aumento da necessidade de especificar sistemas em termos de requisitos temporais ditados pelo ambiente. Estes são conhecidos como sistemas de tempo-real e a sua computação definida tanto em termos dos seus resultados lógicos como do tempo no qual eles são fornecidos. Há muitos exemplos de sistemas de tempo-real: sistemas de controlo do tráfego aéreo, sistemas de reservas de bilhetes on-line ou até mesmo jogos de computador em tempo-real. [1]

Com o passar do tempo o campo computacional tem mutado a sua antiga faceta da computação pessoal para uma computação multi-pessoal extremamente prometedora. A distribuição tem penetrado na área de tempo-real não só por esse novo desafio como também pela necessidade de descentralizar, equilibrar a carga computacional, replicar e paralelizar a computação. Estas novas necessidades levaram ao aparecimento dos sistemas distribuídos de tempo-real como são hoje conhecidos. Nestes, as garantias temporais têm de ser fornecidas sobre um sistema de máquinas ligadas pela rede. Uma vez conseguido isso, as garantias devem ser associadas a outros atributos, tais como a modularidade, a separação geográfica, a independência das faltas e o balanço da carga, entre outros. [1]

Embora os sistemas distribuídos de tempo-real pareçam uma óptima solução para estes novos requisitos, seria inútil considerar a sua existência apenas em ambientes perfeitos[1] ou em ambientes onde as faltas pudessem ser ignoradas. Como solução a este problema surgiram os sistemas distribuídos de tempo-real fiáveis nos quais se tenta garantir o funcionamento dos sistemas distribuídos de tempo-real mesmo sobre situações faltosas. Um exemplo de um sistema com estas características será um jogo de computador para multi-jogadores em tempo-real. [1]

Todas estas evoluções ao nível das aplicações resultaram num aumento da sua complexidade. De forma a fornecer um desempenho satisfatório, estas aplicações têm-se tornado muito exigentes em termos do suporte à comunicação. Uma aproximação promissora para aliviar a complexidade destes sistemas é confiar em arquitecturas de comunicação configurável capazes de suportar reutilização e composição de componentes. Núcleos de protocolos[2] recentes, como o Ensemble e o Coyote, oferecem um ambiente onde diferentes micro-protocolos podem ser combinados. No entanto, poucos sistemas suportam a coordenação de múltiplos canais. O Appia é um núcleo de protocolo que oferece uma forma limpa e elegante de expressar restrições entre canais. Esta funcionalidade é obtida como uma extensão das funcionalidades dos sistemas correntes. O Appia tem uma boa flexibilidade e um design modular que permite que as pilhas da comunicação sejam compostas e reconfiguradas em tempo de execução. [2]

Com este artigo pretendemos acompanhar a evolução natural dos sistemas considerando uma aplicação antiga, o jogo Pong da Atari[3], e recriando-a numa versão multi-utilizador, distribuída e fiável. O jogo, ao qual chamámos **4Pong** , será desenvolvido utilizando a tecnologia Appia. Mas porquê perder tempo com este jogo? Bom, este jogo pode ser considerado como a aplicação mínima que uma arquitectura para um software síncrono multi-utilizador em tempo real deve ser capaz de implementar. Mas a grande diferença para

[1] Ambientes isentos de faltas, ou ambientes onde estas são ignoradas.

[2] Protocol kernels

[3] O Pong da Atari foi o primeiro jogo de computador a ser desenvolvido e suportava apenas um utilizador.

as implementações já existentes baseia-se na existência de, não dois, mas quatro jogadores em simultâneo, o que se traduz num aumento da complexidade de implementação e no consequente refinamento das técnicas de implementação utilizadas.

O resto do artigo encontra-se organizado da seguinte forma. A Secção 2 descreve os detalhes do nosso jogo, inclui a descrição do jogo, a arquitectura, a gestão do grupo de participantes, o protocolo de mensagens, o modelo de faltas e a divergência de estados. A Secção 3 aborda a implementao efectuada para o jogo contendo uma breve descrição do APPIA, descreve um pouco a noção de grupo, da organização e fiabilidade, da sincronização de pontuações, da nossa estratégia de retardamento e por fim da pilha de protocolos e eventos. A Secção 4 aborda a nossa análise de resultados obtidos, a secção 5 relata o que poderia ser feito no futuro para melhorar a nosso trabalho, a secção 6 aborda o trabalho relacionado e por fim a secção 7 mostra as conclusões.

2 Jogo: 4Pong

2.1 Modelo do Sistema e Suposições

4Pong é jogado num campo quadrado com uma baliza por jogador. Cada jogador possui um cursor que pode deslocar lateralmente, de modo a tentar evitar que a bola entre na sua baliza. Em todos os lados onde não existam jogadores a baliza é apenas uma parede onde a bola faz ricochete.

Todos os jogadores começam com zero pontos. Se a bola passa o paddle de um jogador batendo na sua parede, o último jogador a tocar na bola marca um golo. Após um golo, a bola é reposta em jogo pelo jogador que o sofreu, isto é, parte do seu paddle. Cada golo vale um ponto e, no final do jogo, o jogador com mais golos é declarado vencedor.

O jogo começa assim que existam pelo menos dois jogadores. A bola é lançada do centro do campo com um sentido e uma velocidade bem conhecidos.

Durante o jogo é possível que entrem ou saiam jogadores. Quando um novo jogador entra em jogo é-lhe atribuída a pontuação inicial de zero pontos e pode de imediato começar a jogar. Quando um jogador sai de jogo mas ainda ficam suficientes jogadores em campo para prosseguir o jogo, o seu paddle é removido, a sua baliza passa a ser uma simples parede e a sua pontuação é apagada.

O jogo pode terminar em duas situações. A primeira situação é quando a pontuação de um jogador atinge os 20 golos, sendo declarado o vencedor. A segunda situação é quando já não há suficientes jogadores em campo e, nesse caso, o único jogador em campo é declarado o vencedor.

2.2 Arquitectura

Desde que as arquitecturas centralizadas começaram a evoluir para arquitecturas descentralizadas que se têm debatido as vantagens e desvantagens dessa evolução. Numa arquitectura centralizada existe uma autoridade central coordenadora que organiza e condiciona todos os outros participantes. Como consequência, esta solução pode originar uma performance inaceitável já que toda a carga de computação e difusão de informação se encontra num só ponto do sistema, o coordenador. Numa arquitectura distribuída, cada participante mantém uma cópia dos dados processando tudo o que puder de forma local antes de se actualizar com os outros participantes. Embora a consistência se torne mais difícil de garantir, a performance fica significativamente melhorada. [1]

Neste jogo a arquitectura utilizada será distribuída do tipo cliente-servidor. Nesta arquitectura, um jogador será considerado o servidor e o(s) restante(s) cliente(s). Um jogador é servidor quando a bola se encontra no seu domínio[4] e os papéis vão sendo trocados mediante a posição da bola. O(s) cliente(s) vão executando localmente os cálculos possíveis esperando que o servidor os informe das alterações ao jogo.

2.3 Noção de grupo

Visto que este é um jogo multi-utilizador será utilizado um sistema de gestão dos vários participantes. O sistema Appia oferece a funcionalidade de gestão de grupos com recurso a técnicas de sincronismo virtual [3][4]. Esta técnica é vantajosa já que permite a cada utilizador conhecer os outros participantes e, por outro lado, facilita a gestão de entradas e saídas do grupo de jogo. Esta tecnologia permite ainda detectar falhas nos diversos elementos do grupo (ver secção D).

2.4 Protocolo de mensagens

Nesta aplicação é necessário garantir que todas réplicas apresentem o mesmo estado de jogo. Para isso é necessário não só minimizar o tráfego na rede como mascarar o tempo de latência. Por outro lado é necessário que, mesmo tolerando os atrasos, a aplicação seja rápida o suficiente de forma a fornecer uma boa jogabilidade. é necessário, portanto, fornecer uma interface coerente, que se mantenha rápida mesmo com os diversos atrasos na rede.

Como descrito na secção 2.2, será utilizado o método de replicação das actividades distribuidas, não só porque minimiza o total de informação trocada, mas

[4]Supondo que o campo quadrangular se encontra dividido em quatro zonas triangulares, uma para cada jogador.

também porque a carga de computação está distribuída pelos vários pontos do sistema, o que aumenta a sua tolerância a faltas[5].

Para usar o modelo replicado é necessário definir quais as mensagens que são trocadas pelas diversas réplicas. Vamos dividir as alterações ao estado da aplicação em dois conjuntos, as determináveis e as indetermináveis. Por alterações determináveis referimos aquelas que, para um dado evento, todas réplicas obtêm o mesmo resultado sem que haja troca de mensagens[6]. As alterações indetermináveis são aquelas que a não se conseguem prever, nomeadamente interacção de um utilizador numa das replicas. No nosso caso apenas as alterações indetermináveis devem ser propagadas pela rede.

Existem três tipos de alterações que têm de ser propagadas:

1. Quando o jogador move o paddle.
2. Quando um jogador intercepta a bola. Neste caso deve indicar a nova posiçâo, a velocidade e a direcçâo da bola.
3. Quando um jogador sofre golo. Neste caso a bola vai ser colocada no paddle do jogador em questâo e este recomeça o jogo.

2.5 Modelo de faltas

Sendo esta uma aplicação distribuída é necessário ter em conta as faltas gerais que ocorrem em todos os outros sistemas deste tipo. Existem dois grandes grupos de faltas a analisar nos sistemas distribuídos: as omissivas e as assertivas.

No primeiro grande grupo encontra-se as faltas assertivas. Estas existem no domínio do valor e resultam de uma interacção inesperada por parte de um componente. As faltas assertivas podem ser classificadas como semânticas ou sintácticas.

Nas faltas semânticas os valores dos dados enviados numa mensagem ente dois componentes não fazem sentido, por exemplo, a velocidade da bola ser negativa. Estas faltas podem colocar problemas ao nível da consistência dos dados na máquina receptora e, no pior caso, podem ser propagados desta para as outras máquinas. Para tratarmos deste problema, descartamos todas as mensagens em que sejam detectadas alterações. A detecção será efectuada por um algoritmo que tem como base a gama de valores lógicos para cada campo de dados das mensagens. Sempre que o valor não pertença à gama indicada a mensagem é descartada.

As faltas sintácticas não são mais que um subconjunto das faltas semânticas onde os dados são transmitidos nas mensagens são diferentes daqueles esperados pelo receptor, por exemplo, esperamos a velocidade da bola e recebemos "ab". Como solução para este problema o receptor descarta a mensagem sempre não a consiga ler da forma esperada.

No primeiro grupo encontram-se as faltas omissivas. Estas, ao contrário das assertivas, existem no domínio do tempo e surgem sempre que um componente não efectua uma determinada interacção. As faltas omissivas podem ser de paragem, de omissão ou temporais.

As faltas temporais surgem quando um componente se atrasa numa acção. Estas faltas introduzem a noção de grau de latência que se traduz no valor do atraso. A solução a este problema está descrita no capítulo seguinte sobre divergências.

Nas faltas de omissão um componente omite uma acção. Estas são um subconjunto das faltas temporais onde o grau de latência tende para infinito. O factor grau de omissão exprime o total de faltas omissivas sucessivas. Estas faltas estão mascaradas já que cada participante estima o estado seguinte enquanto espera a actualização. Neste caso a actualização não chega e as estimações prosseguem até à actualização seguinte.

As faltas de paragem ocorrem, tal como o nome indica, quando um componente pára. Estas faltas são um subconjunto das faltas de omissão onde o grau de omissão tende para infinito. Como solução a este problema, sempre que seja detectada a paragem de um componente este é retirado de jogo. Esta detecção é possível graças ao mecanismo de gestão de grupos mencionado na secção B.

2.6 Solução para a divergência de estados

Tal como em qualquer sistema distribuído em tempo-real, o jogo **4Pong** pode originar divergências entre as simulações das diferentes réplicas. Neste capítulo vamos apresentar em pormenor um dos problemas que podem ocorrer, bem como a solução adoptada para diminuir os seus efeitos.

Imaginemos um cenário em que a bola encaminha-se para a zona do jogador B, vinda de A. As outras replicas, vão tentar adivinhar se o jogador B falhou a bola ou se a interceptou, caso em que ficam sem saber qual a nova direcção e velocidade. Mesmo que B tenha enviado a mensagem com a respectiva informação basta que exista alguma latência na rede para que a mensagem se atrase, caso em que cada réplica mostra um estado aproximado que pode ser inconsistente com o

[5] Na solução coordenada o que acontece no caso de uma falha no coordenador?

[6] Por exemplo, dada a posição inicial da bola e o vector velocidade, todas as réplicas deverão ser capazes de calcular e representar a trajectória da bola.

estado real. Quando recebem a mensagem de B com o novo estado têm de corrigir a posição e trajectória da bola, o que, aos olhos do jogador, dá a noção de um salto no jogo e quebra toda a sensação de se estar a jogar em tempo real.

Como solução para o problema da sincronização dos estados das réplicas iremos utilizar um sistema de retardamento da representação gráfica do jogo. Segundo este método, quando uma bola se afasta de um jogador é utilizada uma heurística para representar o seu movimento de forma retardada. Assim, o jogador receptor tem tempo de acertar, ou não, na bola e propagar o estado resultante da sua interacção para os outros participantes. Se o retardamento for o adequado, cada réplica recebe o novo estado da bola mesmo na altura em que ele deve acontecer, o que faz com que a representação gráfica seja a correcta.

Idealmente, a retardação faz com que a simulação chegue ao ponto da alteração indeterminável mesmo quando é recebida a actualização. Dado que este valor é uma aproximação, a função heurística baseia-se numa tabela com os tempos de latência da rede para cada réplica existente. Mas esta solução levanta o problema de quais os tempos de latência aceitáveis. Basta que a latência seja muito elevada para que os atrasos da bola se tornem mais distractivos do que úteis. Isto pode ocorrer sempre que há sobrecarga na rede, sobrecarga do processador, ou ambos. Neste caso, a divergência de estados tem tendência a aumentar incontrolavelmente. Como solução, qualquer máquina que entre num estado de sobrecarga é retirada do jogo podendo, no entanto, voltar a entrar mais tarde.

3 Concretização:APPIA

3.1 Appia

As aplicações têm-se tornado cada vez mais exigentes em termos do suporte à comunicação. Uma aproximação promissora para aliviar a complexidade destes sistemas é confiar em arquitecturas de comunicação configurável capazes de suportar reutilização e composição de componentes.

O Appia é um núcleo de protocolo que tenta balancear a flexibilidade na composição de protocolos com a eficácia em tempo de execução. O Appia não é mais do que uma ferramenta em camadas de suporte à comunicação. A sua missão é definir uma interface base a ser respeitada por todas as camadas e facilitar a comunicação entre elas. O Appia é protocolo-independente, isto é, agrupa as camadas desde que estas respeitem a interface base sem validar o resultado final da composição.

O Appia apresenta uma clara distinção entre a declaração de algo e a sua implementação. Uma Camada é definida pelas propriedades que o protocolo requer e as que fornece. A Sessão é a instância que se executa de um protocolo. As Sessões são sempre criadas tendo em conta a Camada e o seu estado é independente entre instâncias. Uma QoS é a descrição estática de um conjunto ordenado de protocolos. Um Canal é a instanciação dinâmica de uma QoS. As instâncias de protocolos (sessões) comunicam usando o Canal.

A comunicação com o canal e entre sessões são efectuadas usando eventos. O Appia fornece um conjunto predefinido de eventos, cada um com diferentes objectivos, mas estes podem ser estendidos, pelo programador, para um conjunto mais detalhado e específico para a aplicação. Começando pela sessão que o gerou, os eventos percorrem o canal num sentido predefinido. Se a direcção for descendente, os eventos percorrem todo o canal sendo depois propagados para processos remotos que os recebem e propagam no sentido ascendente até à camada aplicacional. é assim que se processa a comunicação com processos remotos.

O Appia oferece uma forma limpa e elegante de expressar restrições entre canais. Esta funcionalidade é obtida como uma extensão das funcionalidades dos sistemas correntes. O Appia tem uma boa flexibilidade e um design modular que permite que as pilhas da comunicação sejam compostas e reconfiguradas em tempo de execução.[2][10]

3.2 Noção de grupo

Visto que este é um jogo multi-utilizador será utilizado um sistema de gestão dos vários participantes. O sistema Appia oferece, através de vários protocolos, a funcionalidade de gestão de grupos com recurso a técnicas de sincronismo virtual [3][4].

A sincronia virtual é um paradigma da comunicação em grupo que estipula que:

- todos os participantes de um grupo vêm a sua constituição em forma de vistas.
- os participantes de um grupo vêm as mesmas mudanças de vista pela mesma ordem.
- todas as mensagens de uma vista são entregues nessa vista
- antes de uma mudança de vista, todas as mensagens entregues a um membro são entregues aos outros.

As mudanças de vista ocorrem porque se espera que os grupos sejam dinâmicos, isto é, que alguns membros falhem ou saiam do grupo enquanto novos membros se juntam ao mesmo.

Esta técnica é vantajosa já que permite a cada utilizador conhecer os outros participantes facilitando a gestão de entradas e saídas do grupo de jogo. Esta tecnologia permite ainda detectar falhas nos diversos elementos do grupo (ver secção D).

No Appia não existe uma operação que permita a um membro entrar num grupo, no entanto existe um mecanismo de junção de vistas. Assim, cada vez que um processo deseje juntar-se a um grupo apenas tem de criar uma vista desse grupo com um único membro, ele próprio, depois o protocolo de junção tratará de unir essa nova vista a uma vista possivelmente já existente o que resultará na entrada do novo membro. O mecanismo de união está dividido em duas partes:

- Detecção da nova vista.
- União das vistas.

A união das vistas está a cargo do protocolo inter implementado pelo InterLayer e correspondente InterSession. Por outro lado, a detecção da nova vista é efectuada pelo protocolo heal, implementado pelo HealLayer e correspondente HealSession. A detecção é efectuada usando um servidor externo, GossipServer que recebe todas as mensagens dos processos retransmitindo-as para todos os processos conhecidos, isto é, todos os processos dos quais já recebeu mensagens. Pode ser visto como um meio para alcançar um pseudo-broadcast mas a verdade é que é a base de todo o sistema de comunicação em grupo porque sem ele não existe união de vistas. Como tal, o GossipServer é um dos maiores pontos de falha de todo este sistema (ver secção de trabalho futuro).

Do ponto de vista prático, quando o jogo é lançado o utilizador pode indicar qual o grupo ao qual se quer juntar. Esta opção permite que se criem salas de jogo privadas entre os diversos utilizadores. O processo começa por criar uma vista consigo próprio (no grupo indicado ou num grupo por predefinido). O GossipServer trata de unir essa vista a uma possível vista já existente desse grupo. Sempre que é recebida uma nova vista é necessário garantir que existe apenas um jogador por paddle (impõe um limite de quatro jogadores). Para isso é necessário criar um evento que contenha a informação de controlo necessária à gestão de processos e de migrações. Sempre que um jogador mais novo num grupo esteja a ocupar um paddle já ocupado este deve migrar para uma nova mesa de jogo.

StatusEvent é o evento gerado pela camada ApplLayer sempre que se recebe uma vista com mais do que um elemento. Este evento tem como objectivo fornecer a informação sobre um membro do grupo. Contém o rank, o paddle, o endereço e o identificador(endpt) do membro no grupo. Contém ainda o total de vistas a que o processo já assistiu num determinado grupo. Este último atributo tem como objectivo ordenar os membros por antiguidade para um mais justo processo de migrações.

Após reunir o StatusEvent de todos os membros procede-se à migração dos processos.

O algoritmo de gestão de grupos está brevemente descrito a seguir:

table=0;

paddle1Players, paddle2Players, paddle3Players, paddle4Players;

totalPaddle1, totalPaddle2, totalPaddle3, totalPaddle4;

1. Criar nova vista do GrupoInicial só com P e enviar evento GroupInit com essa vista

1.1. Se utilizador indicou o nome do grupo de jogo ao qual quer pertencer GrupoInicial = grupo indicado

1.2. Caso Contrário
GrupoInicial = "4Pong table " + table

2. Quando recebe uma nova vista VS Actualiza atributos sobre a vista Envia eventos em espera totalView ++;

2.1. Se tamanhoVS > 1
Enviar evento StatusEvent com (rank, totalView, paddle, endpt, addr)

3. Quando recebe StatusEvent SE

3.1. Se SE.paddle=X

3.1.1. Se SE.totalView > paddleXPlayers[0].totalView Adicionar SE a paddleXPlayers na posicao 0

3.1.2. Caso contrário
Se SE.totalView=paddleXPlayers[0].totalView && SE.rank < paddleXPlayers[0].rank Adicionar SE a paddleXPlayers na posicao 0

3.1.3. Caso Contrário
Adicionar SE a paddleXPlayers na posicao final

3.2. Se recebeu de todos && meuPaddle=X && paddleXPlayers[0].rank=meuRank Leave(grupoActual) leaving=true

3.3. Se leaving=true

3.3.1. Enquanto pelo menos um de paddle1players até paddle4Players não nulo
Agrupar elementos não nulos de paddle1Players até paddle4Players na posição actual

3.3.1.1. Se sou paddle menor do grupo
Table++
Criar novo grupo com esses elementos de nome: "4Pong table "+table break

3.3 Ordenação e fiabilidade

Dois dos factores mais importantes a definir são a ordenação e a fiabilidade para as mensagens entre os vários participantes.

A fiabilidade não é mais do que o factor de ponderação sobre as faltas omissivas, isto é, define se são ou não aceitáveis e em que termos. Tal como foi analisado no modelo de faltas, as faltas omissivas são toleradas pela aplicação já que cada participante estima o estado seguinte enquanto espera a actualização que não chega e as estimações prosseguem até à actualização seguinte. Portanto, o sistema não vai garantir fiabilidade, até porque, garantir a fiabilidade num sistema de tempo real é muitas vezes inútil, já que, a mensagem pode chegar muito depois do momento em que seria necessária tornando-se, portanto, inútil.

A ordenação, ao contrário da fiabilidade, é um factor cuja escolha não é tão óbvia. Em primeiro lugar convém analisar o impacto que uma chegada desordenada de mensagens pode originar na aplicação. Se duas mensagens chegarem por ordem inversa, poderá acontecer que o processo actualize o estado do jogo, pela mensagem que chegou posteriormente. Isto poder fazer com que o processo fique dessincronizado dos outros, porque ao actualizar o estado com uma mensagem antiga leva a que a bola e os paddles estejam em posições antigas. Dentro das camadas APPIA, a Sincronia Virtual, que foi utilizada na realização deste trabalho, garante a entrega e ordenação das mensagens.

3.4 Sincronização de Pontuações

A sincronização das pontuações é efectuada quando um dos jogadores falha a bola, nessa altura o jogador que detectar que sofreu um golo incrementa o resultado do jogador que o marcou e envia a sua tabela de resultados juntamente com a nova posição da bola e sua direcção num MissBallEvent para todos os outros jogadores.

Os jogadores ao receberem um eventodo tipo MissBallEvent vão retirar as novas pontuações e actualizar a sua tabela de pontuações.

3.5 Estratégia de retardamento

No capitulo 2.4 foi mencionado o problema da inconsistência de estados entre as réplicas, este problema é causado pela existência de latência na comunicação entre as mesmas. Para uma melhor compreensão da solução proposta vamos dividir o problema em duas partes, a primeira consiste em calcular o tempo de latência entre as diferentes réplicas , a segunda parte consiste em sincronizar os estados das diferentes réplicas utilizando o valor latência encontrado anteriormente.

A primeira parte do problema foi resolvida criando um protocolo, o *GetGroupDelay*, responsável por calcular o valor máximo de latência entre uma réplica e os restantes elementos do grupo a que pertence. Este deverá ficar situado acima das camadas que proporcionam a *Sincronia Virtual* e abaixo da camada de *Sessão de aplicação*.

Descrição do funcionamento do protocolo GetGroupDelay:

1. Inicialmente a camada de *sessão de aplicação* cria um evento do tipo *CalcDelayInitEvent* que indica á Sessão de *GetGroupDelay* que deve iniciar o calculo de latência no Grupo.

2. Ao receber o evento do tipo *CalcDelayInitEvent* a Sessão de *GetGroupDelay*, coloca o seu atributo *collecting* a verdadeiro e instância um evento *GetGroupDelayEvent* (uma extensão do Evento do tipo *GroupSendableEvent*) e envia-o para todos os elementos da vista actual, guardando o instante de envio deste evento.

3. Todos os elementos da vista actual ao receberem um evento do tipo *GetGroupDelay* que lhes indica que existe uma réplica da vista que está a calcular valores de latência,instanciam um evento do tipo *ReplyDelayEvent*, colocam neste evento o seu rank enviam-no para o emissor do *GetGroupDelayEvent*.

4. A réplica que iniciou o pedido de calculo de latencia, ao receber estas respostas vai calcular o momento de chegada de cada resposta e subtrair-lhe o momento em que foi inicialmente enviado o pedido, obtendo assim um valor aproximado da latencia. Assim que chegarem todas as respostas ao pedido, a *SessãoGetGroupDelay* instancia um evento do tipo *CalcDelayEndEvent* coloca-lhe o máximo dos valores obtidos e envia-o para a camada acima terminado o fim da fase de calculo de latência. é de salientar que caso a Sessão de *GetGroupDelay* receba indicação de mudança de vista toda a fase calculo de latência será cancelada, sendo necessário reiniciar uma nova fase após a entrega da nova vista.

3.5.1 Sincronização do Estado das réplicas.

Como foi indicado na secção 2.4 os pontos de sincronização das réplicas são ou quando um jogador sofre um golo ou quando um jogador defende um golo. Sempre que ocorre umas destas situações a camada de *Sessão de aplicação* instancia um evento e coloca neste a nova posição e velocidade da bola e envia-o para as restantes réplicas do grupo. Aquando da recepção de um destes eventos as réplicas vão efectuar o calculo da nova posição da bola tendo em conta a informação recebida e o valor de latencia da rede.

3.5.2 Calculo da nova posição da bola:

posiçãoX=posiçãoX+(velocidadeX*TempoLatencia)
posiçãoY=posiçãoY+(velocidadeY*TempoLatencia)

3.5.3 Calculo da nova velocidade da bola:

PosiçãoXGolo- Corresponde á coordenada X da linha de golo do jogador para onde a bola se dirige.

PosiçãoYGolo- Corresponde á coordenada Y da linha de golo do jogador para onde a bola se dirige.

velVirtualX=(PtoIntersecçãoX-PosiçãoBolaX)/(tmpIntersecção+latência);

velVirtualX=(PtoIntersecçãoY-PosiçãoBolaY)/(tmpIntersecção+latência);

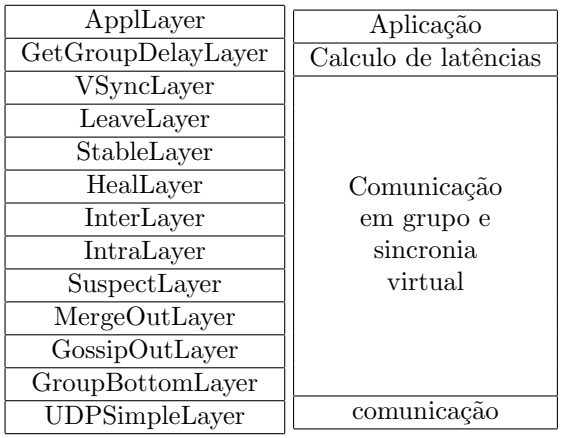
tempototal=(PosiçãoXBola-PosiçãoXgolo) / velocidadeX

Existe num entanto um facto que é necessário ter em conta, a ideia de movimento da bola no jogo é dado por um PeriodicTimer que expira cada 100 milisegundos, e que calcula uma nova posição para a bola. Assim sendo só necessitamos de sincronizar o estado das réplicas quando tivermos uma latencia superior a 100 milisegundos, uma vez que todas as réplicas com que tiverem uma latencia inferior a 100 milisegundos vão puder informar atempadamente as outras a cerca das suas acções .

3.6 Análise global: Pilha de Protocolos e Eventos.

Nos capítulos acima foram descritas as técnicas usadas na implementação do jogo 4Pong, mas, para uma profunda compreensão da mesma, é conveniente efectuar uma ligação geral entre os vários pontos abordados de forma a obter um modelo final dos protocolos e eventos utilizados. é isso que este capítulo pretende abordar.

A pilha de protocolos usada está descrita a seguir:



Todas as camadas à excepção de ApplLayer e GetGroupDelayLayer, são fornecidas pelo Appia. A descrição dessas camadas bem como dos seus eventos pode ser encontrada em [2] e [3].

Para além dos eventos definidos pelos protocolos fornecidas pelo Appia foram ainda criados os eventos descritos a seguir.

ButtonStartEvent é um evento gerado pela interface gráfica sempre que o utilizador pressiona o botão Start. Este evento indica que o utilizador pretende dar inicio ao jogo.

MouseMovedEvent é o outro evento gerado pela interface gráfica que é lançado sempre que o jogador move o rato. Este evento tem como resultado um movimento no paddle correspondente ao movimento do rato recebido.

MoveBallTimer é um timer inicializado pela camada *ApplLayer* aquando do início do jogo com uma periodicidade bastante elevada. Este evento é gerado pelo Appia e indica que deve ser calculada a nova posição da bola e é através da sua impressão periódica que se fornece a noção de movimento.

StatusEvent é o evento gerado pela camada *ApplLayer* sempre que se recebe uma vista com mais do que um elemento. Este evento tem como objectivo fornecer a informação sobre um membro do grupo. Contém o rank, o paddle, o endereço e o identificador(endpt) do membro no grupo. Contém ainda o total de vistas a que o processo já assistiu num determinado grupo. Este último atributo tem como objectivo ordenar os membros por antiguidade para um mais justo processo de migrações.

CalcDelayInitEvent este evento é gerado pela camada *ApplLayer*, sempre que existe a necessidade de calcular a latência entre os elementos do grupo, tem como destino a camada *GetGroupDelayLayer*, inicia nesta uma fase de cálculo de latência.

GetGroupDelayEvent é um evento descende do *GroupSendableEvent*, 'e gerado na camada *GetGroupDelayLayer*, aquando do seu envio para os restantes elementos do grupo é iniciado um contador com o instante do envio.

ReplyDelayEvent este evento é criado pela camada *GetGroupDelayLayer* após a recepção de um evento *GetGroupDelayEvent*, e é enviado para o emissor deste último.

CalcDelayEndEvent este evento é criado pela camada *GetGroupDelayLayer*, marca o fim da fase de cálculo de latências e transporta o valor calculado para a camada superior.

4 Análise dos Resultados Obtidos

O nosso trabalho apresenta uma jogabilidade bastante razoável. Ao que nos foi dado observar, no nosso jogo a bola e os paddles estão perfeitamente sincronizados, excepto quando um jogador começa o jogo, e nesse caso o seu ecrán fica sincronizado com os dos demais jogadores logo após a primeira troca de mensagens. Não nos foram visíveis grandes alterações ao rumo, posição ou velocidade da bola para fazer a sincronização da mesma entre os diferentes processos. Poderiamos melhorar a jogabilidade alterando o período em que a posição da bola é actualizada de 100ms para 40ms, imprimindo para o écran 25 imagens por segundo e não as actuais 10. Isto levaria a que o movimento parecesse mais lenear. De notar que o jogo foi testado numa LAN, que é bastante fiável, mas seria curioso de testar o jogo numa rede de maior escala, para ver se a jogabilidade do jogo se manteria idêntica.

5 Trabalho Futuro

Um dos maiores problemas da implementação apresentada reside claramente na estratégia de gestão de grupos. Em primeiro lugar a solução distribuída e tolerante a faltas para o jogo 4Pong acenta num servidor centralizado para o qual não se toleram faltas: o GossipServer (ver secção 3.2). Esta solução é no mínimo contraditória e deverá, portanto, ser melhorada.

Outro problema que está inerente a este é a sobrecarga decrescente que é dada aos diversos jogadores, isto é, os jogadores da mesa 0 estão constantemente a receber novos jogadores e a tratar das migrações. Os jogadores migram para a mesa 1 que deverá repetir o processo e assim sucessivamente. Esta solução torna o jogo da mesa 0 muito mais lento que o jogo na mesa 1 e assim sucessivamente.

A solução que projectámos passa pelo desenvolvimento de um servidor externo que contém a informação sobre todas as mesas de jogo abertas e gere as filiações. Sempre que um jogador pretende juntar-se ao jogo contacta esse servidor que decidirá qual o grupo ao qual se deve juntar e, se possível, tratará ele mesmo dessa união. O servidor deverá também tratar as saídas das mesas de jogo.

Para contornar o problema desta ser também uma solução centralizada, o servidor deverá ser replicado tornando a solução distribuída. Note-se que esta solução resolve também o problema das carga extra nas primeiras mesas de jogo já que a aproximação não é do tipo tentativa - erro pois cada jogador entra directamente para uma mesa de jogo válida.

6 Trabalho relacionado

A comunicação em grupo através de sincronismo de vistas foi anteriormente estudado por José Pereira, Luís Rodrigues e Rui Oliveira [4]. Um dos maiores problemas dos jogos multi-utilizador distribuídos reside nas diferentes proximidades entre os jogadores. Os jogadores mais próximos do servidor contam com uma injusta vantagem devido aos mais baixos tempos de latência na troca de mensagens. Muitas foram já as soluções adoptadas para este problema. Duas das aproximações já estudadas podem ser encontradas em [6] e [7]. A solução por nós usada: estratégia de retardamento, foi anteriormente utilizada no jogo Ppong! desenvolvido por James Begole e Clifford Shafer em 1999 [5].

7 Conclusões

O jogo 4Pong recria o jogo Pong da Atari numa versão multi-utilizador, distribuída e fiável, sendo considerado como a aplicação mínima que uma arquitectura para um software síncrono multi-utilizador em tempo real deve ser capaz de implementar. Comparado com implementações já existentes, esta permite até quatro jogadores em simultâneo. A implementação segue uma arquitectura cliente-servidor onde cada processo é uma réplica autónoma. Para minimizar o tráfego na rede, apenas os estados indeterminísticos são comunicados. De forma a evitar alterações bruscas de estado foi desenvolvida uma estratégia de retardamento através da qual se tenta fazer com que os updates sejam visualizados na altura em que deveriam ocorrer. Do ponto de vista da implementação é utilizada comunicação em grupo para gerir os grupos de jogo. Os jogadores juntam-se a um grupo escolhido ou ao grupo predefinido e, mediante o estado do grupo podem ser forçados a migrar para outro grupo de forma a manter a coerência dos grupos. Através das técnicas utilizadas, a aplicação garante que na maioria das situações, todas as réplicas apresentam o mesmo estado de jogo. O tráfego na rede foi minimizado e os seus atrasos mascarados o que permite que a aplicação seja rápida o suficiente para fornecer uma boa jogabilidade.

References

[1] *P*aulo Veríssimo e Luís Rodrigues, "Distributed Systems for System Architects", Kluwer Academic Publishers ISBN 0-7923-7266-2

[2] *U*RL: http://appia.di.fc.ul.pt

[3] *P*into, "Appia Group Communication Manual", 2001.

[4] *J*osé Pereira, Luís Rodrigues e Rui Oliveira, "Reducing the Cost of Group Comunication with Semantic View Synchrony*"

[5] *J*ames Begole e Clifford Shaffer, "Internet Based Real-Time Multiuser Simulation: Ppong!", TR-97-02, Virginia Polytecnic Inst. e State University, Department os Computer Science.

[6] *K*atherine Guo, Sarit Mukherjee, Sampath Rangarajan e Sanjoy Paul, "A Fair Message Exchange Framework for Distributed Multi-Player Games", NJ 07733, Center for Networking Research, Bell Laboratories, Holmdel. Link: http://www.bell-labs.com/user/kguo/

[7] *Y*ow-Jian Lin, Katherine Guo e Sanjoy Paul, "Sync-MS: Synchronized Messaging Service for Real-Time Multi-Player Distributed Games" Link: http://www.bell-labs.com/user/kguo/

[8] *M.* Hayden, "The Ensemble System", PhD thesis, Cornell University, Computer Science Department, 1998

[9] *N.* Bhatti, M. Hiltunen, R. Schlichting e W. Chiu, "Coyote: A System for Constructing Fine-Grain Configurable Communication Services", ACM Trans, 16(4):321-366, Novembro 1998.

[10] *H*ugo Miranda, Alexandre Pinto e Luís Rodrigues, "Appia, a Flexible Protocol Kernel Supporting Multiple Coordinated Channels", Universidade de Lisboa.