# FTP - Fault-Tolerant Pong *
# Grupo 08

José Côrte-Real 29037 | Leonel Duarte 25887 | Miguel Figueiredo 28152

5 de Dezembro de 2005

## Resumo

Ao longo do tempo, as aplicações informáticas têm vindo a evoluir de modelos de computação centralizada para modelos de computação distribuída. Tal evolução foi acompanhada por um acréscimo de complexidade na implementação de abstracções e algoritmos relativos às aplicações distribuídas. Neste panorama, é importante a utilização de técnicas de tolerância a faltas para garantir a disponibilidade de um serviço.

No âmbito da disciplina de Tolerância a Faltas Distribuídas foi proposto pelo docente que se concretizasse o jogo Pong para múltiplos jogadores, com tolerância a faltas.

## 1 Introdução

Pong consiste num jogo da autoria de Nolan Bushnell lançado pela Atari em 1972 e que se tornou no primeiro grande sucesso comercial da - naquela altura - emergente indústria dos jogos de computador. O jogo consiste na adaptação de um jogo de ténis de mesa a uma plataforma de jogo de computador:

- A cada jogador é fornecida uma *paddle* ou raquete que usa para proteger a sua parede contra a colisão de uma bola que se encontra em movimento no tabuleiro do jogo;
- A bola do jogo descreve um movimento linear que é alterado aquando de uma colisão com uma parede ou com uma raquete de um jogador;
- Um jogador perde um ponto - ou sofre um golo - sempre que a bola colidir com a parede que protege.

À partida, o jogo apenas suporta dois jogadores num tabuleiro rectangular e o controlo de ambos efectua-se localmente no computador que executa o jogo.

Por razões académicas e de modo a tornar este simples jogo mais interessante e tecnicamente mais complexo, foram propostas duas modificações:

- Suporte para mais de dois jogadores
- Suporte à computação distribuída fiável

O jogo a implementar no âmbito da disciplina de Tolerância a Faltas Distribuídas consiste numa área de jogo quadrangular em que cada jogador controla uma *paddle* que utiliza para impedir que uma bola que se movimenta dentro da área de jogo, embata na *sua* parede. Cada jogador inicia o jogo com 0 pontos e por cada golo (colisão com a sua parede) que sofrer, acrescenta um ponto à sua pontuação (número de golos sofridos).

Este jogo embora seja relativamente simples, captura requisitos essenciais para outras aplicações distribuídas, sendo que as aproximações utilizadas para resolver este problema podem servir de trabalho futuro para problemas mais práticos e complexos.

O resto deste artigo apresenta-se organizado da seguinte forma:

Na secção 2 apresenta-se o problema e a ferramenta a utilizar para o resolver, na secção 3 a relação entre clientes e servidores, a secção 4 refere-se a conceitos importantes para a resolução

*O grupo 08 agradece o apoio do Dr. Nuno Carvalho e do aluno Joao Leitao.

do problema, a secção 5 , ao modelo de faltas relativo a esta aplicação, na 6 apresenta-se a arquitectura utilizada para resolver o problema , na 7 apresenta-se alguns pormenores de implementação , na 8 efectua-se uma avaliação ao projecto do ponto de vista da performance, na secção 9 consideramos que trabalho a realizar no futuro, e finalmente na 10, conclui-se o artigo.

# 2 Descrição do problema

A aplicação a implementar limitar-se-á a uma lotação máxima de 4 jogadores simultaneamente, com uma lista de espera para jogadores adicionais.

O projecto seguirá o modelo cliente-servidor, tendo em conta a preocupação de tornar a aplicação tolerante a faltas.

## 2.1 Appia

O Appia[3] é uma *framework* de suporte ao desenvolvimento e execução de protocolos por camadas, totalmente implementada em Java, que oferece abstracções que implementam inúmeros modelos de computação distribuída. No Appia a arquitectura das aplicações assenta numa pilha de micro-protocolos, cuja implementação toma a forma de um canal[4] Appia. A informação circula neste canal sob a forma de eventos. No trabalho a implementar, pretende-se tomar partido das vantagens do Appia para desenvolver um sistema de comunicação em grupo para os servidores.

# 3 Operações entre componentes

## 3.1 Cliente - Servidor

De modo a participar no *Fault-Tolerant Pong* um cliente (representando um jogador) tem que se conectar a um servidor de jogo.

A interacção entre clientes e servidores deve, mormente, incidir sobre a capacidade de apresentar um estado de jogo ao cliente de modo coerente e eficiente.

Uma vez que na inicialização entre cliente e servidor, o jogador não se encontra efectivamente a jogar, considera-se este momento crucial para que existam trocas de mensagens de auxílio à tolerância a faltas.

Os clientes comunicam ponto-a-ponto com um servidor, a informação é propagada apenas para o servidor, que terá por sua vez de a devolver aos clientes a ele conectados.

## 3.2 Servidor - Servidor

A interacção entre servidores de jogo é fundamental para garantir a tolerância a faltas de um servidor: sem este tipo de interacções torna-se difícil garantir o correcto funcionamento do jogo, tendo em conta os requisitos fundamentais do enunciado proposto[5].

Aos servidores compete apresentar, principalmente, garantias de continuidade de serviço aos seus clientes. O problema reside em conseguir oferecer essas garantias em detrimento de coerência e eficiência do funcionamento do jogo. Ou seja, para o cliente o funcionamento replicado do servidor e a sua recuperação de falhas, devem de ser transparentes.

# 4 Conceitos relevantes

## 4.1 Noção de grupo

Por grupo[1] entende-se um sistema composto por:

- Suporte de filiação que permite acrescentar/retirar elementos de um grupo e saber que elementos pertencem num dado momento a esse mesmo grupo (a vista do grupo);
- Suporte de comunicação que permite que todos os elementos do grupo comuniquem entre si, utilizando comunicação ponto-a-ponto ou multicast[1].

De modo a simplificar as interacções servidor - servidor no *Fault-Tolerant Pong* a utilização de um sistema de comunicação em grupo é vantajosa. Esse sistema é concretizado tomando partido de protocolos fornecidos com a distribuição da *framework* Appia.

## 4.2 Sincronia virtual

Por sincronia virtual[2] entende-se ser um mecanismo baseado em suporte de filiação e comunicação por broadcast fiável que permite saber,

num determinado momento, quais os elementos pertencentes a um grupo - vista do grupo. Sempre que a vista do grupo se altera, a sincronia virtual garante que todas as mensagens a enviar para a vista antiga são entregues, antes de instalar a nova vista.

### 4.3 Noção de coordenador

Frequentemente, de modo a simplificar as interacções entre elementos de um grupo, é eleito um coordenador[2] do grupo ao qual poderão ser atribuídas funcionalidades exclusivas como, por exemplo: prioridade de interacção com entidades exteriores ao grupo, poder de decisão em problemas de consenso, poder de decisão em problemas de replicação, etc.

A utilização de um coordenador na implementação do *Fault-Tolerant Pong* está dependente do modelo arquitectural escolhido para a resolução do problema, nomeadamente o modelo de replicação. Este ponto será abordado mais adiante, na secção 4.5.

### 4.4 Ordenação de mensagens

A ordenação de mensagens assume, principalmente, duas facetas[1, Sec. 2.7]:

- Determinar *a posteriori* a ordem em que as mensagens foram enviadas de modo a assumir ou excluir relações de causa-efeito entre entre elas
- Garantir que as mensagens obedecem a critérios de ordenação estabelecidos por políticas definidas *a priori*

Existe uma série de modelos de ordenação que permitem aplicar uma ou ambas das facetas referidas: ordem FIFO, ordem causal, ordem total, entre outros. Dada a natureza da arquitectura adoptada para resolver o enunciado proposto, apenas a noção de ordenação FIFO[2] interessará para este artigo.

### 4.5 Modelo de replicação

Para além de garantir a continuidade de serviço o grupo de servidores tem de garantir a coerência do mesmo, e tal requisito é garantido através da replicação do estado do jogo pelos elementos (réplicas) do grupo de servidores. Consideramos de seguida três modelos de replicação possíveis para a resolução do problema[1, Sec. 7.6]:

1. Replicação Activa
2. Replicação Semi-Activa
3. Replicação Passiva

No modelo de replicação activa todos os servidores recebem actualizações de estado dos clientes e mantêm um estado de jogo idêntico de servidor para servidor. A disseminação de estados para o cliente é feita por todos os servidores em *simultâneo*. Cabe aos clientes, de seguida, assimilar o estado de jogo. A replicação activa tem algumas vantagens, como por exemplo, a possibilidade de efectuar balanceamento de carga entre as réplicas, e a detecção de erros nas mensagens para o cliente através de um processo de maioria.[1]

O modelo de replicação semi-activa assemelha-se ao modelo de replicação activa pois todos os servidores continuam a receber os estados de jogo, cabendo a um dos servidores - o coordenador - actualizar as réplicas do estado correcto do jogo. Perante a falha do servidor coordenador um novo coordenador é eleito de entre os restantes elementos do grupo.

No modelo de replicação passiva cabe apenas a um servidor coordenador receber actualizações de estado vindas dos clientes; as restantes réplicas encontram-se em espera de actualizações periódicas - *checkpoints* - vindas do servidor coordenador. Perante a falha do servidor coordenador um novo coordenador é eleito de entre os restantes elementos do grupo.

### 4.6 Uniformidade

No contexto da comunicação em grupo, uniformidade caracteriza-se pela seguinte garantia:

- *Dada uma mensagem m enviada para o grupo, ou todos os elementos a recebem ou nenhum a recebe*[2].

[1] e.g: Se o cliente receber 4 estados com o valor 'sim' e 1 com o valor 'não' dos servidores, pode assumir como correcto o valor 'sim' com alguma certeza.

Com o objectivo de alcançar esta propriedade na implementação do enunciado proposto, tomou-se partido da camada *uniform* fornecida pela distribuição da *framework* Appia. A uniformidade voltará a ser discutida na secção 6.1. De notar que para uma arquitectura como a especificada neste artigo, seria ideal uma camada uniforme que esperasse resposta de todos os elementos vivos do grupo num determinado momento. Mas, devido à imprecisão inerente à camada *Suspect*[2][2] do Appia, a camada *uniform*, a ser utilizada, necessita *saber a priori* o número mínimo de servidores necessários para garantir a uniformidade. Logo, se esse número mínimo não for igual ou superior a uma maioria dos servidores do grupo, a uniformidade pode não ser garantida. Se, pelo contrário, o número de servidores do grupo for inferior a esse valor estipulado *a priori*, todas as operações que necessitem de uniformidade deixam de estar operacionais.

### 4.7 Tipos de faltas

Num sistema distribuído é imperativo ter especial cuidado com o tipo de faltas que o sistema *tolera*. Existem dois grupos essenciais de faltas: assertivas e omissivas.

Na classe de faltas assertivas incluem-se as interacções entre componentes em que os valores transmitidos estão errados. Existem duas subclasses de faltas assertivas: semânticas e sintácticas: Uma falta semântica caracteriza-se pelo envio de um valor errado mas possível; uma falta sintáctica pelo envio de um valor *impossível*.

A detecção de faltas sintácticas é, pela própria natureza da linguagem java, normalmente fácil, visto que o envio de um tipo de dados inesperados resultará no levantamento de uma excepção, tornando essa interacção inválida. O envio de um valor errado mas possível, torna a sua detecção virtualmente impraticável na arquitectura proposta (Uma maneira possível de detectar estas falhas semânticas seria através do uso de replicação activa e redundância de operação, onde um pedido a um componente originaria $n$ respostas, através das quais se poderia deduzir um valor correcto com maior probabilidade).

As faltas omissivas situam-se no domínio do tempo, e dividem-se em três subcategorias: faltas por omissão, paragem e temporais.

Quando um componente falha em realizar uma determinada interacção – por exemplo enviar um ACK – considera-se uma falta por omissão, ou seja, um componente omite uma interacção potencialmente crítica para o funcionamento do sistema.

Uma falta por paragem acontece quando um componente *crasha* e, por consequência, deixa de interagir com os restantes componentes.

Uma falta temporal é, essencialmente, um atraso numa comunicação, introduzindo latência na computação do sistema. As faltas de paragem são portanto, falhas temporais com latência superior a um *timeout* do sistema.

## 5 Modelo de Faltas

É impraticável tentar desenvolver um sistema que se considere tolerante a todo o tipo de faltas mencionadas anteriormente. É neste contexto que surge a necessidade de definir um modelo de faltas onde se especifique que faltas se tenciona tolerar.
A aproximação ao problema escolhida, privilegia a tolerância a faltas omissivas, considerando-se que faltas afirmativas não *bizantinas*[1] não deverão ocorrer, e que faltas maliciosas não são consideradas dado contexto do projecto. Nomeadamente pretende-se tolerar faltas de atraso e falha de servidores, e garantir que estas falhas são transparentes para os jogadores e clientes em lista de espera.

Para que o *FTP* tolere faltas omissivas – temporais e de omissão – foi desenvolvida uma camada de emulação na aplicação cliente que esconde este tipo de faltas do utilizador[**?**, ve01]- *Masking*. Na secção 6.3.1 aprofunda-se o conceito de camada de emulação e sua aplicação prática.

As faltas de paragem de um servidor são toleradas através de um mecanismo de replicação passiva com coordenador. Esta solução é a base de toda a arquitectura, e é discutida na secção 5.1.

## 6 Arquitectura

Para resolver o problema proposto, implementou-se a seguinte arquitectura:

- *Arquitectura baseada em coordenador e replicação passiva sobre TCP sem balanceamento*

[2] A camada suspect é responsável pela detecção de falhas de membros do grupo.

*de carga.*

No modelo adoptado os clientes ligam-se a um único servidor – o coordenador – encarregue de actualizar o seu estado de jogo.

O coordenador é um dos elementos de um grupo de $f$ servidores.

Os elementos do grupo de servidores são actualizados pelo coordenador periodicamente – *checkpointing*[1] – ou em situações em que a sua actualização seja imediatamente necessária.[3] Dada as características da replicação passiva, e de modo a reduzir a complexidade da solução, não é feito nenhum balanceamento de carga dos servidores. Ou seja, o coordenador é responsável pela manutenção de um estado de jogo coerente em todas as réplicas, e por disponibilizar aos clientes uma plataforma de jogo.

Seguidamente, apresenta-se a arquitectura em dois passos:

## 6.1 Arquitectura base

Nesta arquitectura, privilegia-se a simplicidade de implementação em detrimento de abordagens ao problema mais sofisticadas.

O funcionamento do sistema baseado nesta arquitectura assenta nos seguintes pressupostos:

- Todos os servidores de jogo fazem parte de um grupo implementado com a tecnologia Appia. Um dos servidores é automaticamente eleito coordenador pelo protocolo de comunicação em grupo;
- Todos os clientes (no caso do *Fault-Tolerant Pong* de 4 jogadores) ligam-se ao coordenador;
- Se o coordenador falhar os clientes ligar-se-ão ao novo coordenador (eleito de entre as réplicas *vivas* no grupo de servidores) caso este exista; caso contrário, não é possível continuar o jogo;
- Durante o funcionamento do jogo o estado do mesmo é disseminado pelas réplicas; e, se ocorrer uma falta no servidor coordenador em tempo de jogo, o cliente continuará a ter acesso ao jogo (cujo estado corresponde ao estado disseminado no último *checkpoint* anterior à falta do servidor) enquanto existir coordenador no grupo de servidores (ou seja, enquanto existirem servidores).
- Em situações críticas, será necessário garantir a uniformidade da informação entre as réplicas, antes de se devolver uma resposta aos clientes (Quando um cliente se regista ou quando sofre um golo).

[3] A ordenação das mensagens é feita tomando partido da ordenação FIFO do protocolo TCP

Relativamente à disseminação do estado do jogo entre o coordenador e as réplicas, esta será feita periodicamente (100ms) ou quando se observarem as seguintes situações:

1. Quando um cliente se regista no coordenador.
2. Quando um cliente sofre um golo.

A disseminação é feita obrigatoriamente nestes casos para garantir que, após um cliente entrar no jogo ou sofrer um golo, se o servidor coordenador falhar o estado com a informação do jogador ou do golo terá sido enviado para as réplicas. Adicionalmente, de modo a garantir a coerência, esta informação não pode ser enviada do coordenador ao cliente antes que o coordenador se certifique que as restantes réplicas receberam a informação: se tal não se verificasse, poderia ocorrer um fenómeno de *rollback* onde o estado do jogo andaria para trás, quebrando a sequência de jogo e prejudicando o seu realismo. (por exemplo um jogador sofrer um golo e, após falha do coordenador e consequente re-conexão com uma réplica que ainda não tinha recebido a informação do golo, esse golo *desaparecia.*) Para garantir esta ultima condição, os tomou-se partido da camada *uniform* fornecida pela *framework*. Sempre que uma nova vista é criada, o suporte de sincronia virtual bloqueará todas as réplicas através do evento BlockOk. Nestes momentos é necessário garantir que todas as mensagens criticas serão guardadas para serem enviadas assim que o servidor seja desbloqueado.

## 6.2 Pilhas protocolares

Tomando partido das características do Appia, foi desenvolvida uma aplicação servidor e uma aplicação cliente que assentam numa pilha protocolar.

Visto que apenas um canal[3] foi utilizado no servidor, apresenta-se uma única pilha protocolar:

| PongServerLayer |
|---|
| UniformLayer |
| LoopbackLayer |
| VSyncLayer |
| LeaveLayer |
| StableLayer |
| HealLayer |
| InterLayer |
| IntraLayer |
| SuspectLayer |
| GossipOutExtLayer |
| GroupBottomLayer |
| TcpCompleteLayer |

Essencialmente, esta pilha, baseada em micro-protocolos [4] fornecidos pela *framework* Appia, oferece comunicação em grupo inter-servidor com sincronia virtual e comunicação por TCP tanto para a comunicação com o grupo bem como para a comunicação com os clientes. Adicionalmente, esta pilha, garante a uniformidade[2] de entrega de mensagens entre os elementos do grupo de servidores.

Para o cliente, foi utilizada a seguinte pilha protocolar para o seu canal:

| PongClientLayer |
|---|
| PongClientCommLayer |
| RemoteViewLayer |
| TcpCompleteLayer |

Esta pilha fornece comunicação por TCP através da camada TcpCompleteLayer e permite ao cliente contactar o *GossipServer* para receber uma vista do grupo – sem ter que fazer parte do grupo – utilizando a camada RemoteViewLayer[3] . Adicionalmente implementaram-se três camadas protocolares para a resolução do problema proposto:

- PongClientCommLayer - Camada de comunicação do cliente, torna a comunicação com os servidores – e respectiva reconexão – transparente para as camadas superiores;
- PongClientLayer - Finalmente, esta camada trata da recepção de estados de jogo e envio do estado da *paddle* do jogador.

## 6.3 Arquitectura optimizada

A arquitectura base apresentada anteriormente não considera a problemática da performance computacional. Assim sendo, apresenta-se a arquitectura optimizada que melhora, do ponto de vista de performance e jogabilidade, a aplicação.

A optimização da arquitectura base assenta em cinco mecanismos: *Camada de emulação no cliente*, *Camada de buffering no cliente*, *Lista de espera de jogadores*, *Uso selectivo de uniformidade* e *Camada RemoteView*.

### 6.3.1 Camada de emulação

De modo a colmatar atrasos de actualização de estado por parte do coordenador, uma camada de emulação do estado do jogo foi colocada na pilha protocolar do cliente:

| PongClientLayer |
|---|
| **PongEmulationLayer** |
| PongClientCommLayer |
| RemoteViewLayer |
| TcpCompleteLayer |

Ao *ver* o primeiro estado de jogo que o cliente recebe, a camada de emulação activa um *periodic timer*[3] responsável por medir atrasos na comunicação do servidor para o cliente. Em caso de atraso a camada envia imediatamente ao cliente um estado *aproximado* do estado do servidor; caso contrário, a camada de emulação encaminha directamente o estado do servidor para a aplicação.

A camada de emulação foi implementada de modo a não ser *activada* numa zona crítica do tabuleiro de jogo - nas imediações de uma parede ou de um *paddle*: é preferível que haja um atraso de um estado coerente do que apresentar um estado potencialmente incoerente que levaria a um comportamento *anormal* da aplicação.

### 6.3.2 Camada de *Buffering*

Conforme o código disponibilizado da aplicação Pong, por cada movimento do *paddle* de um cliente, um evento *MovePaddleEvent* é enviado ao servidor. É fácil perceber que, se os 4 jogadores estiverem a mover constantemente o *paddle*, vai existir uma grande carga sobre o servidor coordenador. De modo a diminuir esta carga no coordenador, introduziu-se uma camada de *Buffering* de eventos *MovePaddleEvent*, onde existe um valor de *threshold* que define de quantos em quantos eventos é que se envia, efectivamente, um *MovePaddleEvent*

ao servidor. Ou seja, se o threshold for, por exemplo, 2, só se envia um *MovePaddleEvent* quando o valor do contador de eventos da camada é 2. Após o envio do evento *MovePaddleEvent* o contador é reinicializado.

| PongClientLayer |
|---|
| **BufferLayer** |
| PongEmulationLayer |
| PongClientCommLayer |
| RemoteViewLayer |
| TcpCompleteLayer |

De notar que um valor muito alto do Threshold, vai diminuir a precisão de movimento do *paddle* por parte do cliente.

Devido à capacidade da *framework* Appia de dividir a estrutura de uma aplicação em diversas camadas, o funcionamento da camada de emulação e de *Buffering*, é completamente transparente à aplicação; ou seja, a camada de aplicação não é capaz de diferenciar estados vindos do coordenador de estados emulados, nem tem de saber que apenas alguns dos seus *MovePaddleEvent* é que são enviados ao servidor.

### 6.3.3 Lista de espera de jogadores

Uma vez que o jogo *FTP* apenas suporta jogos de quatro jogadores em simultâneo, outros jogadores que se tentem ligar ao coordenador não têm oportunidade de jogar se o coordenador já estiver *cheio* i.e. com 4 jogadores ligados.

Para resolver este problema foi implementado um mecanismo de lista de espera de jogadores no servidor coordenador: um cliente que se tente ligar a um coordenador *cheio* fica em lista de espera até que o coordenador lhe conceda um lugar no jogo *FTP*.

Para se manter a coerência do jogo e da lista de espera, esta também é enviada obrigatoriamente para as réplicas passivas a cada *checkpoint*.

### 6.3.4 Uso selectivo de uniformidade

A utilização de comunicação uniforme entre as réplicas torna-se necessária, como já foi explicado na arquitectura base, para impossibilitar situações em que estados críticos do jogo retrocedam. No entanto, a comunicação uniforme introduz um *overhead* nas transacções de rede, devido aos requisitos inerentes ao próprio conceito de uniformidade. Para melhorar esse *overhead*, foi introduzida uma camada na pilha protocolar do servidor cujo propósito é fazer um *bypass* a eventos do coordenador para as réplicas que não necessitam de uniformidade:

| PongServerLayer |
|---|
| UniformLayer |
| LoopbackLayer |
| **BypassLayer** |
| VSyncLayer |
| LeaveLayer |
| StableLayer |
| HealLayer |
| InterLayer |
| IntraLayer |
| SuspectLayer |
| GossipOutExtLayer |
| GroupBottomLayer |
| TcpCompleteLayer |

Uma das particularidades da *framework* Appia consiste na especificação de eventos através do conceito de herança da programação orientada a objectos. Por este motivo a camada de uniform, devido à sua própria semântica de funcionamento, apenas aceita e trata eventos orientados à comunicação em grupo. Todos os outros eventos são despachados pela camada uniform para a camada seguinte (dependendo da direcção do evento: *up* ou *down*). A camada de *bypass* utiliza esta técnica para *forçar* que alguns eventos específicos sejam despachados pela camada uniform sem qualquer tratamento ao encapsular tais eventos específicos num evento que a camada uniform não *aceite*.

### 6.3.5 Camada RemoteView

Um dos problemas implícitos no enunciado proposto trata-se da necessidade do cliente conhecer um primeiro elemento do grupo de servidores, através do qual poderá conhecer os restantes elementos do grupo, incluindo o coordenador. Para resolver esta questão, tomou-se partido da funcionalidade da camada *RemoteView* da *framework* Appia que permite contactar o sistema de suporte de filiação do grupo – *GossipServer* – que retorna uma vista do grupo requisitada por um elemento exterior ao mesmo.

# 7 Detalhes de implementação

Nesta secção explicar-se-á alguns detalhes de implementação, dignos de melhor explicação, devido à sua importância na concretização do projecto, nomeadamente:

## 7.1 No servidor

### 7.1.1 Mecanismo de RequestView

Como complemento ao mecanismo do Appia de obtenção de vistas remotas – RemoteView – implementou-se um sistema que permite propagar a vista do grupo de servidores para os clientes sempre que esta se altera.

O evento RequestView representa o pedido de vista do cliente bem como a resposta do servidor.

**Algoritmo 1**: Server View Broadcast

```
Data: View - The server group view.
upon event ⟨ NewView ⟩ do
foreach p ∈ Players, w ∈ PlayersWaiting
do
    trigger ⟨ RequestView | View, p, w ⟩;
end
```

Também foi implementado um método que permite responder a apenas um cliente caso este o requira.

**Algoritmo 2**: Server reply to Request View

```
Data: View - The server group view.
upon event ⟨ RequestView | p ⟩ do
    trigger ⟨ RequestView | View, p ⟩;
```

### 7.1.2 Mecanismo de *Checkpointing*

De modo a garantir a coerência entre réplicas, utiliza-se um método de sincronização de estados entre o coordenador e as suas réplicas. É utilizado um *periodic timer* – CheckpointTimer – no coordenador que controla o ritmo de sincronização.

Para manter as réplicas *up-to-date*, o coordenador envia o estado do jogo, composto por: (a bola, os jogadores, os paddles e a lista de jogadores em espera, tudo encapsulado num CheckPointPackage[6]). Esta informação vai ser recebida pelas réplicas e assimilada, ficando assim as réplicas com um estado de jogo muito próximo do estado actual do coordenador; A coerência do estado de jogo das réplicas é relativa à frequência de *checkpointing*.

**Algoritmo 3**: Server Checkpointing

```
upon event ⟨ CheckpointTimer ⟩ do
    trigger ⟨ Syncsend | CheckpointInfo ⟩;
```

Considerou-se que o envio de estados periódicos do servidor coordenador para as suas réplicas não necessita de uniformidade.

### 7.1.3 Registo de jogadores

O registo de jogadores no *FTP* processa-se através do envio de um evento de registo por parte do cliente e de uma resposta ao registo por parte do coordenador. No entanto, este mecanismo é susceptível a dois problemas:

- Se o servidor não for coordenador o cliente deve ser reencaminhado para o coordenador;
- Se o coordenador estiver *cheio* o cliente fica em lista de espera e esta tem de ser disseminada para as réplicas.

Apresenta-se, então, o pseudo-código do algoritmo de registo de jogadores; Deve-se ter em conta

**Algoritmo 4**: Server player registration

```
upon event ⟨ RegisterPlayer | p ⟩ do
if coordinator then
    if #Players = MAXPLAYERS then
        PlayersWaiting =
        PlayersWaiting ∪ {p};
    else if p ∉ Players then
        Players = Players ∪ {p};
        trigger ⟨ PlayerInfo | p ⟩;
    end
end
else
    trigger ⟨ RequestView | p ⟩;
end
```

que o algoritmo de registo de jogadores aqui apresentado é utilizado apenas quando um cliente toma a iniciativa de se registar no jogo. Quando um cliente se encontra em lista de espera e abre uma vaga do lado do servidor este deve *notificar* o cliente para se juntar ao jogo.

## 7.2 No cliente

### 7.2.1 Registo de jogador

O cliente coloca na lista de backups os endereços da vista do grupo de servidores. De seguida envia um *RegisterPlayerEvent* para o coordenador.

**Algoritmo 5**: Client register

**Data**: Backups - Set of Backup Servers

**upon event** ⟨ *Init* ⟩ **do**
    **trigger** ⟨ *RemoteView* ⟩;

**upon event** ⟨ *RemoteView* | *View* ⟩ **do**
    $Backups \leftarrow View$;
    $primary \leftarrow Backups[0]$;
    **trigger** ⟨ *RegisterPlayer* | *primary*⟩;

### 7.2.2 Mecanismo de RequestView

O cliente aproveita a funcionalidade *RequestView* implementado no servidor, para requisitar quando necessário uma vista do grupo remotamente. Este tipo de pedidos acontece quando o cliente se re-conecta a um novo servidor, de modo a confirmar que este é o coordenador e para ter uma versão mais actual dos endereços dos servidores de jogo.
Em termos de código , é apenas enviado um evento *RequestViewEvent* , e o servidor utiliza o Mecanismo de RequestView para responder **apenas** a este cliente.

**Algoritmo 6**: Client request view

**Data**: Backups - Set of Backup Servers

**upon event** ⟨ *RequestView* | *View* ⟩ **do**
    $Backups \leftarrow View$;

### 7.2.3 Camada de emulação

O cliente emprega uma camada de emulação, entre a camada de comunicação e a de jogo. O objectivo desta camada é controlar o tempo de chegada entre dois estados – BSL *(Between State Lag)* – através de um *periodic timer* Appia configurável, e caso o BSL entre dois estados seja superior ao *timer* escolhido, esta camada irá utilizar o último estado recebido do servidor para emular um próximo estado aproximado.

**Algoritmo 7**: Emulation Layer Behaviour

**upon event** ⟨ *Init* ⟩ **do**
    $serialNumber \leftarrow 0$;
    $CheckSerialNumber \leftarrow 0$;
    $isTimerActive \leftarrow false$;
    $EmulationState \leftarrow \emptyset$;

**upon event** ⟨ *State* | *ServerState*⟩ **do**
**if** *isTimerActive = false* **then**
    $isTimerActive \leftarrow true$;
    **trigger** ⟨ *DeadlineTimer* | *timeout* ⟩;
**end**
$EmulationState \leftarrow ServerState$;
$serialNumber \leftarrow serialNumber + 1$;
**trigger** ⟨ *State* | *ServerState*⟩;

**upon event** ⟨ *DeadlineTimer* ⟩ **do**
**if** *serialNumber = CheckSerialNumber* **then**
    $CheckSerialNumber \leftarrow serialNumber$;
    EmulationState.update;
    **if** *EmulationState is not critical* **then**
        **trigger** ⟨ *State* | *EmulationState*⟩;
    **end**
**end**
**else**
    $CheckSerialNumber \leftarrow serialNumber$;

De modo a permitir uma melhor tolerância a flutuações de latência na ligação entre o cliente e o servidor, existe uma camada de emulação por baixo da camada de jogo mas por cima da camada de comunicação, que recebe todos os estados vindos do servidor e passa-os para as camadas de cima, controlando a frequência com que chegam. Na camada de emulação, existe um *periodic timer* que, de 75

em 75 ms[4], executa o algoritmo descrito. Quando é executado, este algoritmo compara o número de série do último estado de jogo recebido do servidor com o número de série verificado a ultima vez por este algoritmo. Se forem iguais, significa que em 75ms ainda não foi recebida uma actualização do servidor e, para manter a qualidade de jogo, emula-se a posição da bola e passa-se o evento (O estado emulado) para a camada de cima como se tratasse dum evento proveniente do servidor.

#### 7.2.4 Mecanismo de reconexão

Aquando da falha do servidor coordenador, compete ao cliente deliberar qual é o novo coordenador e conectar-se a este. A reconexão efectua-se escolhendo o primeiro servidor da sua lista de *Backup Servers*.

**Algoritmo 8**: Client re-connect

**Data**: Backups - Set of Backup Servers

**upon event** ⟨ $ServerFailed$ | $s$⟩ **do**
 $Backups \leftarrow View \setminus \{s\}$;
 $primary \leftarrow Backups[0]$;

# 8 Avaliação

## 8.1 Contexto de avaliação

Todas as medições feitas para esta secção foram efectuadas nos computadores do laboratório 1.2.09 da FCUL.

As aplicações foram executadas no software Eclipse sobre o sistema operativo Linux (ubuntu).

O método de *Benchmarking*[5] utilizado consiste em medir o tempo da máquina antes e depois da recepção de um GameState – *BSL* – do servidor. Partindo do pressuposto que, pelo menos de 50 em 50ms (o *periodic timer* do movimento da bola) o servidor envia estados aos clientes (caso estes não movam os *paddles*), é possível ver de que modo diferentes opções de implementação afectam esta medida.

[4]Este valor pode ser alterado.

[5]Medição de desempenho.

## 8.2 Medições

De modo a avaliar o comportamento do *software* implementado apresentam-se, nesta secção, algumas medidas feitas do *BSL* sob diversas condições. Os resultados obtidos são apresentados na figura 1. A condição de teste avaliada foi a falha do servidor coordenador com 4 clientes ligados, e a consequente passagem para um novo coordenador. As variaveis deste teste são: a utilização (ou não) da camada de emulação no cliente, a execução em modo DEBUG[6] versus modo normal, e a alteração do *timer* da camada de emulação. Todos os testes foram efectuados com a camada de *Buffering* do cliente desactivada, e sem detecção de zona crítica.

De modo a reproduzir uma condição de falha realista, os testes de falha do coordenador foram efectuados manualmente, ou seja, a falha não foi programada na execução do sistema, mas sim gerada manualmente através do cancelamento da execução do processo do coordenador, durante o período de *benchmarking*. Os autores favoreceram esta aproximação em detrimento de uma falha programada na execução do sistema, visto que reflecte mais realisticamente um comportamento que o sistema teria na práctica.

- Caso 1 (Linha azul escura) : Leitura do BSL durante 100 recepções de estado, DEBUG ligado, sem emulação, com 4 clientes, 4 servidores, e com o coordenador a falhar e a réplica rank 1 a assumir a posição de coordenador no estado 31 –*server swap* – O *BSL* atinge o valor máximo na mudança de servidor, atingindo o valor de 421ms. De notar que esta latência de 421ms, irá introduzir uma paragem **significativa** no funcionamento do jogo.
- Caso 2 (linha púrpura) : Leitura do BSL durante 100 recepções de estado, DEBUG ligado, emulação com *timer* a 100ms, com 4 clientes, 4 servidores, e com o coordenador a falhar e a réplica rank 1 a assumir a posição de coordenador no estado 29 – *server swap* – o *BSL* atinge o valor máximo na mudança de servidor, atingindo o valor de 195ms, com um estado a anteceder este pico com um BSL de 168ms, e com dois estados posteriores ao pico com 101ms e

[6]Em modo DEBUG a aplicação imprime informação de rastreamento de erros para a consola, acrescentando *overhead* adicional á computação.

133ms. Esta latência é pouco perceptível no funcionamento do jogo.

- Caso 3 (linha amarela) : Leitura do BSL durante 100 recepções de estado, DEBUG ligado, emulação com *timer* a 50ms, com 4 clientes, 4 servidores, e com o coordenador a falhar e a réplica rank 1 a assumir a posição de coordenador no estado 28 – *server swap* – o *BSL* atinge o valor máximo na mudança de servidor, atingindo o valor de 170ms. De notar que na reconexão do servidor observa-se menos flutuações do BSL na periferia do estado de mudança de servidor. Esta latência é pouco perceptível no funcionamento do jogo.
- Caso 4 (linha azul clara) : Leitura do BSL durante 100 recepções de estado, DEBUG desligado, emulação com *timer* a 50ms , com 4 clientes, 4 servidores, e com o coordenador a falhar e a réplica rank 1 a assumir a posição de coordenador no estado 31 – *server swap* – o *BSL* atinge o valor máximo na mudança de servidor, atingindo o valor de 159ms. Comportamento similar ao do caso 3 excepto valor do pico de *BSL*. Esta latência é pouco perceptível no funcionamento do jogo.

Finalmente, a utilização da BufferLayer por parte dos clientes, resulta numa diminuição de carga no servidor proporcional ao valor Threshold da camada. Como tal não foram feitas medições, mas com um Threshold de factor 10, e com 4 clientes, o servidor terá 40 vezes menos mensagens de movimento de *paddle* oriundas dos clientes para processar.

## 8.3 Testes de funcionalidade

Foram feitos testes de funcionalidade da aplicação de modo a determinar possíveis pontos de falha da mesma. Teste com falha de servidores, nomeadamente o coordenador, com 4 clientes e clientes em lista de espera, foram ultrapassados com sucesso pela aplicação. Falhas em pontos críticos do algoritmo, como o envio de mensagens uniformes durante o bloqueio do grupo por parte da sincronia virtual, foram também ultrapassados com sucesso. Finalmente, foi detectado um erro com a utilização da camada *RemoteView* apenas nos laboratórios da faculdade, onde um dos clientes bloqueava. Esta situação não se verificou fora do ambiente dos laboratórios da faculdade, nem foi diagnosticada com sucesso pelos autores.

# 9 Trabalho futuro

Outra arquitectura possível para resolver o enunciado proposto, apesar de potencialmente mais complexa, consiste em:

- *Arquitectura baseada em memória partilhada e registos atómicos e balanceamento de carga.*

Com o objectivo de aplicar os conhecimentos adquiridos na disciplina de Tolerância a Faltas Distribuídas, poder-se-ia desenvolver uma arquitectura mais generalista que potencie a utilização dos recursos disponibilizados ao sistema, ou seja, uma arquitectura que fizesse proveito, activamente, do maior número de réplicas servidor possível e simultaneamente permitisse escalar o número de clientes que o sistema serve. Para concretizar esta arquitectura, a aplicação de um protocolo de balanceamento de carga entre as várias réplicas tem de ser considerado. Assim, é de ponderar a concretização de um sistema com memória partilhada entre as réplicas e com registos atómicos, de modo a que cada réplica tenha a possibilidade de servir clientes concorrentemente com as restantes, e, mesmo assim, mantendo um estado de jogo coerente entre todas as réplicas.

A não consideração desta arquitectura como prioritária deveu-se, essencialmente, a duas questões:

- Para um problema como o proposto, será necessário tanta complexidade? (nomeadamente a implementação de memória partilhada)

E, principalmente:

- Será que uma arquitectura destas apresentará níveis performance suficientes (qualidade de serviço) para satisfazer os requisitos de jogabilidade do *Fault-Tolerant Pong*?

Como tal, foi apenas implementada a arquitectura de replicação passiva com coordenador , no entanto vamos analisar teoricamente esta alternativa nesta secção.

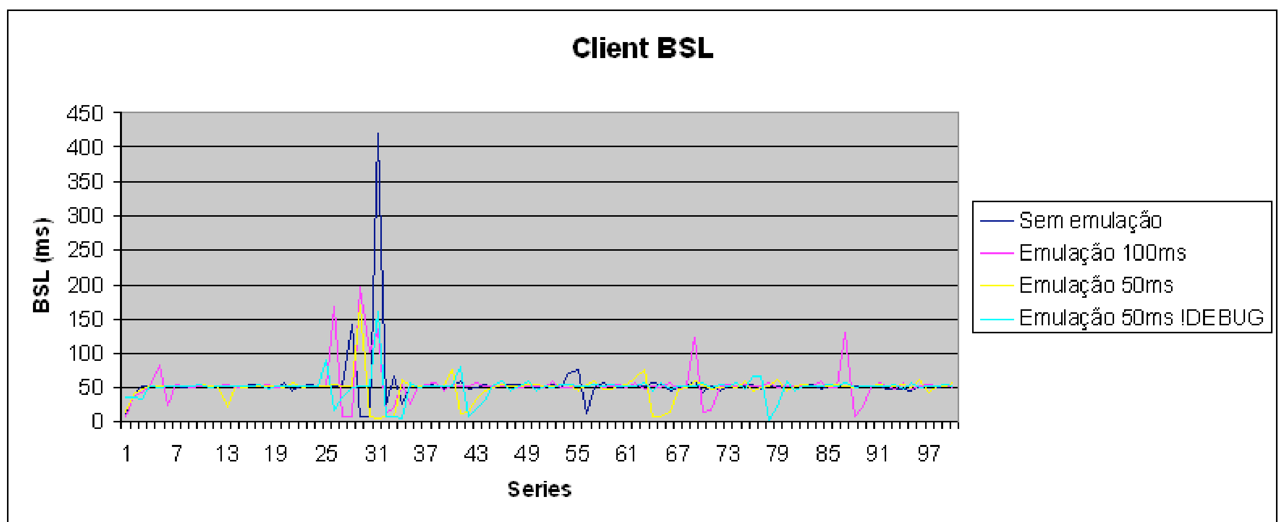


Figura 1: Medida de *Between-State-Lag* no cliente

# 10 Conclusão

A criação de um sistema distribuído permite oferecer garantias mais fortes de funcionamento que um sistema centralizado, como por exemplo: maior disponibilidade de serviço, maior confiabilidade e, por vezes, maiores níveis de performance. No entanto, o advento dos sistemas distribuídos veio acrescentar mais complexidade ao desenvolvimento de sistemas informáticos e novos desafios aos mesmos. É neste contexto que se insere a utilidade de *frameworks* - como o Appia - e a implementação de sistemas distribuídos tolerantes a faltas - como o que foi apresentado.
Neste artigo, apresenta-se uma solução e uma possível implementação de um sistema distribuído tolerante a faltas (nomeadamente, Pong multijogador), tomando partido das características da *framework* Appia. Finalmente mostra-se ainda resultados de testes de performance efectuados , de modo a justificar processos de optimização implementados.

# Referências


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

[2] Rachid Guerraoui, Luís Rodrigues, 2005. *Introduction to Reliable Distributed Programming*, Preliminary Draft, Springer-Verlag.

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

[4] Alexandre Pinto, 2005. *Appia Group Communication*

[5] Enunciado do projecto de TFD - http://www.di.fc.ul.pt/ ler/docencia/tfd/trab.html

[6] JavaDoc do projecto.